Mälardalens högskola

School of Innovation, Design and Engineering

Evaluation of Software Defined Radio Platform with respect to implementation of 802.15.4 ZigBee

Krešimir Dabčević, kresimir.dabcevic@xnet.hr

Supervisor: Marcus Bergblomma, marcus.bergblomma@mdh.se

Examinator: Mikael Ekström, mikael.ekstrom@mdh.se

Västerås, 2011.

Abstract

With the development of powerful computational resources such as Digital Signal Processors and Field Programmable Gate Arrays, it has become possible to utilize many radio functions via software. This is the main concept of an up-and-coming technology of Software Defined Radio.

In the Thesis, a number of platforms for implementation of Software Defined Radio have been evaluated. The platform that proved to be most suitable for the project was Ettus' USRP N210. Using the platform, an implementation of IEEE 802.15.4 Zigbee's physical layer was done. Experiments whose outputs can later be used to compare performance with respect to "hardware radios" were performed.

Sammanfattning

Med utvecklingen av enheter med kraftfulla beräkningsegenskaper som “Digital Signal Processors” och “Field Programmable Gate Arrays” har det blivit möjligt att implementera flera radiofunktioner i mjukvara. Det är huvudkonceptet i den uppåtgående teknologin mjukvaru definierad radio.

I det här examensarbetet har ett flertal plattformar för mjukvaru definierad radio utvärderats. Plattformen som visade sig vara mest lämplig för projektet var Ettus USRP N210. En implementation av IEEE 802.15.4 Zigbees fysiska lager har realiserats till plattformen. Experiment, vars utdata senare kan användas för att jämföra prestanda mellan mjukvaru definierad radio och hårdvaru baserad radio, har även utförts.

Preface

I would like to thank my family for giving me support, especially regarding my decision to spend a semester and finish my studies in Sweden.

My gratitude also goes towards IDT's Wireless Communication research group members, for allowing me to do my Thesis in the area I was interested in, providing the necessary equipment as well as invaluable help in form of information and advices during the project.

Abbreviations

ADC – Analog to Digital Converter

BER – Bit Error Rate

CCL – CMOS Configuration Latches

CR – Cognitive Radio

CSMA – Carrier Sense Multiple Access

DAC – Digital to Analog Converter

DSP – Digital Signal Processor

FPGA – Field Programmable Gate Array

GAUSS – Guaranteed Automation communication Under Severe disturbanceS

GPP – General Purpose Processor

GRC – GNU Radio Companion

IO – Input/Output

JTRS – Joint Tactical Radio System

LOS – Line Of Sight

LPF – Low Pass Filter

MIMO – Multiple Input - Multiple Output

NBFM – Narrow Band Frequency Modulation

NLOS – Non Line Of Sight

OOP – Object-Oriented Programming

PDR – Programmable Digital Radio

PER – Packet Error Rate

PGA – Programmable Gain Attenuator

PHY – Physical (referring to layer)

RSSI – Received Signal Strength Indicator

SCA – Software Communication Architecture

SDR – Software Defined Radio

SWIG – Simplified Wrapper and Interface Generator

SWR – Software Radio

TDD – Time Division Duplex

TESLA – Time-critical and Safe wireLess Automation communication

USRP – Universal Software Radio Peripheral

WBFM – WideBand Frequency Modulation

Wi-Fi – Wireless Fidelity

Keywords

SDR, Software Defined Radio, programmable, reprogrammable, reconfigurable, GNU Radio, USRP, N210, RSSI, PER, ZigBee, 802.15.4

List of figures

Figure 1: Hardware architecture of an SDR 18
Figure 2: USRP N210 motherboard 24
Figure 3: Xilinx Spartan XC3SD3400A FPGA architecture. Source: [2] 26
Figure 4: Radiation patterns of HG2458RD-SM antenna in vertical and horizontal plane. Source: [5] 29
Figure 5: Implementation of dial_tone example within Gnu Radio Companion 38
Figure 6: Frequency modulation of the transmitting signal 39
Figure 7: GRC flow graph of NBFM transmitter 40
Figure 8: GRC flow graph of NBFM receiver 40
Figure 9: Comparison of RX peak signal strength - NBFM vs. WBFM 43
Figure 10: Comparison of NBFM's and WBFM's FFT plots depending on the TX amplification 44
Figure 11: Comparison of received signal's FFT regarding different maximum deviation factors 45
Figure 12: Block diagram of the 802.15.4 transmitter 50
Figure 13: Block diagram of the 802.15.4 receiver 50
Figure 14: Picture of the measurements setup 51
Figure 15: RSSI values of measurements no. 1 and 3 56
Figure 16: RSSI values of measurements no. 19 and 25 57
Figure 17: RSSI depending on the TX-RX distance 57
Figure 18: PER distribution depending on the TX-RX distance, best case scenario 58

List of tables

Table 1: Comparison of considered platforms 23
Table 2: Specifications of PCs used in the measurements 36
Table 3: Measurement results for NBFM 43
Table 4: Measurement results for WBFM 43
Table 5: Inputs and outputs of 802.15.4 measurements 54
Table 6: Calculated PER and average RSSI derived from 802.15.4 measurements 55

Contents

Abstract 1
Sammanfattning 2
Preface 3
Abbreviations 4
Keywords 6
List of figures 7
List of tables 8
Contents 9
1 Introduction 11
1.1 Motivation 11
1.2 Methodology 12
1.3 Delimitations 12
1.4 Related work 13
1.5 Thesis report outline 14
2 Software Defined Radio 15
2.1 Definition of SDR. History and background. 15
2.2 Basic SDR – hardware and software architecture 17
2.3 Current SDR technologies and evolution towards Cognitive Radio 19
3 Preparing for implementation 21
3.1 Choosing the suitable platform 21
3.2 USRP 23
3.2.1 USRP N210 motherboard 24
3.2.2 FPGA 25
3.2.3 DACs and ADCs 26
3.2.4 Digital upconverters and downconverters 27
3.2.5 Daughterboards 27
3.2.6 Antennas 29
3.2.7 UHD Drivers 30
3.3 Python 30
3.4 GNU Radio 31

4 Working with GNU Radio: implementation and measurement process 35
4.1 Setting up the equipment and initial testing 35
4.2 Working with GNU Radio Companion 37
4.2.1 Dial_tone example 37
4.2.2 Using USRPs with GRC – NBFM and WBFM 38
4.3 Modifying the UCLA Zigbee PHY code 45
4.4 Measurement process and results 51
4.5 Results processing and analysis 54
5 Possible improvements. Future work. 59
6 Conclusions 60
7 References 61
Appendix A - Code: original UCLA Zigbee PHY files 63
Appendix B - Code: modified TX and RX UCLA files 77

1 Introduction

In an engineer's world, no matter how good the certain product performs, it could always be improved to perform faster, cheaper and more efficiently. Being bounded by the omnipresent Shannon theorem, making these improvements is often not an easy task for radiocommunication engineers.

One of the emerging technologies that is able to offer a lot of space for such improvements is Software Defined Radio. Slowly but steadily, this technology - whose purpose is bringing the software as close to antenna as possible, thus turning hardware problems into software ones - is becoming a dominant design approach in modern radiocommunication systems.

Fairly reasonable prices of some of the SDR systems out there make it possible for academic environments to get into – not only examining the benefits this innovative technology has to offer at the moment, but also contributing to the project and helping speed up the progress pace of the technology itself.

1.1 Motivation

Motivation for this Thesis was primarily getting familiarized with the exciting technology of Software Defined Radio. In order to do that, besides theoretical analysis approach, implementation of some of the fairly wide-spread technology via one of the available SDR platforms was to be done. Considering the work that has been and is being done in MDH's Tesla-Gauss project that deals with technologies operating in 2.4 GHz bandwidth, it was decided to try and implement one of those technologies as well.

After closer examination of potentially available platforms, it was decided to purchase Ettus Research's USRP SDR console, which uses GNU Radio open-source software development toolkit as a software architecture. The final goal of the project was implementing IEEE 802.15.4 standard's physical (PHY) layer and performing measurements that can later be compared to the ones done using conventional ("hardware") radio systems.

1.2 Methodology

Since working with SDRs requires different competences and combines knowledge from a range of disciplines, it was necessary to obtain these prior to start of the practical work. Therefore, literature study at first included revising the digital signal processing theory and reading about the SDR technology itself.

Then, the time was spent reviewing potential SDR platforms; analyzing pros and cons of each and figuring out which one would prove to be the best for achieving the set goals.

Once the platform (USRP N210 SDR) was chosen and ordered, more in-depth literature study about GNU Radio and Python programming was performed.

Once the platform arrived, after successfully compiling GNU Radio and updating the platform's firmware, initial testing has been done.

After getting familiarized with modus operandi of the equipment and software, tests have been performed using GNU Radio Companion (GRC), trying out the different modules and functions that GRC offered.

Finally, the work with implementing 802.15.4 PHY layer included modifying the UCLA Zigbee 802.15.4 PHY's code in order to work with the new, UHD drivers and adjusting the software for the measurements.

Once done, results of measurements have been processed and analyzed and the Thesis report written.

1.3 Delimitations

The main delimitation of this project was time. With only a basic general pre-knowledge of SDR and digital signal processing, and no previous practical experience in this field

whatsoever, it wasn't quite certain that the set goals would be achievable within the time-scope (one semester).

Again, because of time restrictions, it was known that it would not be possible to do the implementation of IEEE 802.15.4 "from scratch" – instead, UCLA's Zigbee PHY project was to be used as a template. The UCLA Zigbee PHY project implements only the bottom (physical) layer of the 802.15.4 standard, whereas upper layers are not taken into account.

1.4 Related work

This Thesis was done within MDH's projects TESLA and GAUSS in order to see if there is potential for Software Defined Radio technology to somehow be used either as an integrated part of those projects or used as an alternative approach to the measurements that are being done within TESLA and GAUSS.

TESLA's (Time-critical and Safe wireLess Automation communication) main goal is achieving predictability of time-critical and safe wireless communication, in spite of communication taking place in harsh environments. The main application areas are time-critical industrial processes. The overall purpose of project TESLA is enabling smarter system development for time-critical industrial processes by giving system developers tools for validation, calculation and prediction of reliability and safety of time-critical wireless communication. Project has three work tasks: standardized, safe wireless sensor networks (WSNs) in time-critical applications; safe and reliable time-critical communication in harsh environments, and models for quantification of safety and availability in wireless automation networks.

Its sister project GAUSS' (Guaranteed Automation communication Under Severe disturbanceS) baseline measurement results are to be produced and used as inputs to TESLA. GAUSS has two work tasks in common with TESLA: safe and reliable time-critical communication in harsh environments, and models for quantification of safety and availability in wireless automation networks. Solving real-world problems and creating proof-of-concept demonstrators are important outcomes of project GAUSS.

More information about TESLA and GAUSS projects can be found at [3].

As for the UCLA Zigbee PHY project, the implementation of Text-Message Transceiver Service built upon a IEEE 802.15.4 physical layer was done in [9]. Building upon that, adding capabilities of automatic sensing and tuning (frequency hopping) was done in [11].

1.5 Thesis report outline

The report is divided as follows:

Chapter 1 – Introduction to the problematic, motivation, goals and obstacles of the project

Chapter 2 – Theoretical background to the Software Defined Radio

Chapter 3 – Process of choosing the platform for the project implementation. Theoretical background to hardware (USRP) and software (GNU Radio) used in the measurement process.

Chapter 4 – Working with USRP and GNU Radio – using GNU Radio Companion; implementing 802.15.4 PHY layer. Processing, understanding and comparing the measurement results.

Chapter 5 – Future work and possible improvements

Chapter 6 – Conclusions

2 Software Defined Radio

This chapter defines the term "SDR" and gives a short introduction to the history, background, and architecture of the Software Defined Radio technology, as well as evolution towards more advanced, self-aware radio systems.

2.1 Definition of SDR. History and background.

Software defined radio is a term originally coined by Joseph Mitola (often referred to as "a grandfather of SDR") back in 1991., while describing the possibilities of reconfigurability and reprogrammability of radio systems.

Today, there is no unanimous definition of SDR – however, one of the most recognizable, and in the same time very intuitive ones is *SDR Forum*'s (recently renamed to *Wireless Innovation Forum*) one, which recognizes SDR as "a *radio in which some or all of the physical layer functions are software defined"* [1]. These functions usually include - but are not limited to - frequency; modulation technique; cryptography; used bandwidth, coding technique, etc. However, the level of reconfigurability/reprogrammabillity needed for the radio to be classified as a SDR isn't strictly defined. The "ideal" SDR would, therefore, have all of the radio-frequency bands and modes defined in software.

Still, many authors feel the need to differentiate terms such as "programmable digital radio" (PDR), "software defined radio" and "software radio" (SWR), the difference mainly being the number of functions defined software-wise.

Thus, PDRs usually have the feature of only a few basic baseband operations and a part of the physical and data link layers being customized via software, however its hardware-focused architecture conditions the need for changing the hardware assemblies in order to change RF band and/or air interface.

In that sense, SDRs represent a step forward – the possibility of supporting different air interfaces using software arises, however not all functions are still defined in software.

SWRs (sometimes referred to as “ideal SWRs”) are, therefore, the next evolutionary stepping stone – they provide full reconfigurability of air interfaces in software, including channel access and waveform synthesis.

However, since the distinction between these technologies is often not recognizable, for clarity purposes, the rest of the Thesis (apart from the chapter explaining the evolution towards cognitive radios, where these terms will be reused) will recognize these similar concepts as the same – referring to them as software-defined radios.

The main motivation behind developing SDR was creating a technology that would be able to operate in the frequency band from 2 MHz to 2 GHz (so-called “Two-to-two band”), and would be compatible with all the other military radios used by American military – there were more than ten of those at that time. Dating from 1992 - 1995, that project is today known as SPEAKeasy, and its architecture represents a paradigm for a basic structure of today’s SDRs (although Air Force Rome Labs’ ICNIA multiple-radio design, dating from 1987, could arguably be considered the predecessor of the technology). SPEAKeasy was a rather massive, hardly-portable equipment based on a programmable cryptography chip, allowing for communication over a range of different frequencies, cryptography techniques, modulation techniques, encoding methods and other parameters.

The portability issue was overcome with SPEAKeasy II technology, whose dimensions were far more similar to today’s SDR systems, and which furthermore introduced the possibility of using programmable vocoder as well as a variety of ASP and DSP circuitry for handling different waveforms – the functions associated with modern SDR systems as well. [4]

From there on, SDR represents a technology that is constantly being upgraded and updated with new ideas and solutions, and which is gradually evolving towards far more complex technological solution – Cognitive Radio – which will briefly be discussed in paragraph 2.3.

2.2 Basic SDR – hardware and software architecture

As previously stated, “ideal” SDR would have all the radio-frequency bands and modes defined software-wise, meaning it would consist only of an antenna, DAC or ADC (depending on whether we are examining transmitter or receiver) and a programmable processor.

However, in practical systems, the RF front-end has to be implemented as well in order to support the receive/transmit mode.

Typically, RF front-end of a SDR will consist of antenna circuitry, amplifiers, filters, local oscillators and ADCs/DACs (both for transceiving systems). When the signal is received, it is amplified and its carrier frequency downconverted to a low-intermediate frequency in order for ADC to perform digitization. Analogously, at the transmit side, the produced signal that is to be transmitted goes through DAC, thus producing an IF representation. This signal representation is then going through process of shifting to a desired carrier frequency, amplification of signal and brought to an antenna, where it is ready for transmission.

The processing is done by some of the computational resources at our disposition – mainly, General Purpose Processors (GPPs), Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs), whereas some of the future resources may include a combination of the aforementioned, thus extending the computational capacity.

One of the most important aspects when deciding on a computational resource that is to be used in the system is its reprogrammability (important for implementation of new waveforms), therefore dedicated-purpose circuitry is generally avoided in SDRs.

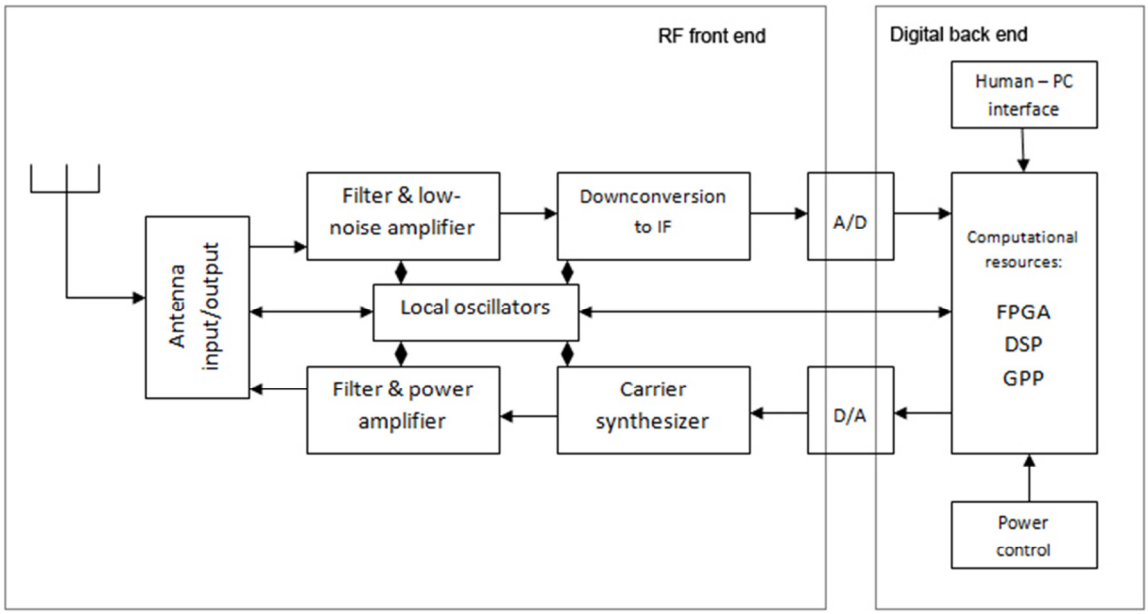

Figure 1: Hardware architecture of an SDR

As for the software architecture of SDR, there is no unique architectural approach for SDRs – different platforms may choose different approaches. Nevertheless, regardless what kind of software architecture is chosen, main goals should be common - standardized way of introducing new waveforms to the platform and compatibility with other platform implementations.

One of the prevalent standardized software architectures of the SDR systems is Software Communication Architecture (SCA), defined by the US government with the purpose of securing waveform portability and improving software reuse. Built originally for US military's Joint Tactical Radio System (JTRS) program, it has been accepted as a communication standard in military services of many other countries, but also by commercial organizations such as Wireless Innovation Forum. It is an always-evolving standard - with first version dating from 2000 – that provides standardized set of methods for installing, managing and de-installing new waveforms, therefore maintaining interoperability of various SDR systems.

Another widespread software architecture worth mentioning is GNU Radio, used as a software architecture for implementing SDR in this Thesis, which will be discussed in chapter 3.4.

2.3 Current SDR technologies and evolution towards Cognitive Radio

From the early 1990s and the aforementioned SPEAKeasy project, a lot has been done in terms of improving SDR technology. From PDRs and Software-Capable radios working with fixed modulation and a small number of frequencies and data rate capabilities, over Software-Programmable Radios able to implement new utilities via software, today we have a few real Software-Defined Radio systems able to achieve configurability of all radio-operating settings via software.

In terms of software programmability and reconfigurability, current state-of-the-art SDR technology is JTRS – a family of tactical software that uses the common SCA architecture that provides the interoperability of different radio types as well as software reusability amongst them. JTRS radios serve as a plug-and-play devices capable of working in 2 Mz – 2 GHz band, and should in foreseeable future replace all the existing US military radio technologies operating in that band. [12]

Digital Modular Radio (DMR) is a full SDR used by US Navy as a part of the Joint Maritime Communications System project. At the moment, it is interoperable with military systems such as SINCGARS and HaveQuick, however interoperability with JTRS is yet to be implemented (but should be possible in the future). [13]

The next step in radio systems evolution will be be so-called Aware Radio Systems and, subsequently, Adaptive Radios. Bruce Fette defines Aware Radios as “radios who use their sensing capabilities (in audio and RF frequencies) to gather environmental information” and Adaptive radios as “Aware Radios able to autonomously modify their operating parameters such as frequency, instantaneous bandwidth, modulation scheme, error-correction coding, channel mitigation strategies, data rate, transmit power, etc.”. [4]

Finally, Adaptive Radios with the capability of learning can be Considered Cognitive Radios (CRs). The ultimate CR will, thus, be aware of their present location, the availability of nearby services, be able to predict user’s needs based on the previous usage patterns, etc. Primary motivation for research of Cognitive Radios lies in improvement of RF spectrum utilization, however it should also bring the whole diapason of other innovations, such as user-identity

learning, geolocation based on GPS and/or triangulation with other devices, Model Based Reasoning, etc.

Although that doesn't necessarily have to be the case, Cognitive Radio is usually defined as an upgraded SDR and is the ultimate currently imaginable radio technology. It probably won't come into full existence for at least next 15 - 20 years, but large steps forward in CR research are continuously being made.

3 Preparing for implementation

This chapter will explain the thought process behind choosing a suitable platform and provide the basic information of USRPs - the hardware used in measurements, GNU Radio - open-source software which it uses, and Python - programming language which, in combination with C++ - is used to write and connect GNU Radio's blocks.

3.1 Choosing the suitable platform

Following the initial literature studying, it was needed to choose and purchase the platform that would be suitable for our work. Main selection criterions were operability in 2.4 GHz band; suitability for implementing 802.15.4 and fairly powerful computing resource programmability. Couple of weeks of research narrowed the options down to the following platforms:

1. Microsoft Research Software Radio Platform for Academic Use (SORA)

   - Microsoft offers this academic kit to academic environments for research purposes. The License agreement states that there is the possibility that part of the Kit (RCB board) might be leased from Microsoft, but the exact conditions weren't stated. Otherwise, estimated price of the full necessary equipment (multi-core PC, RCB board, RF front-end and software) was around $4500 - $5500. Although Microsoft's kit initially looked rather interesting, the need for extensive computing power, as well as the clause that forbids potential users to "reverse engineer, decompile, or disassemble the Kit" influenced the decision to look elsewhere.

2. Datasoft's Typhoon SDR Development Platform

   - Datasoft's SDR full-duplex transceiver system operates in 400 Mhz – 4 GHz band with the ability to process signal bandwidths ranging from 50 kHz to 20 MHz and represents quite powerful SDR system. It supports GNU Radio and Click Modular Router as a software architecture. It is based on multiple Virtex-4 SX35 FPGAs, as well as an OMAP 3 processor, allowing for high reconfigurability and reprogrammability.

These characteristics, however, make the platform aimed more towards high-end users, which reflects in its price - $10000 per unit.

3. Lyrtech's SFF SDR Development Platform

   - Like Datasoft's Platform, Lyrtech offers a highly powerful, highly modular and reprogrammable device, powered by Xilinx Virtex-4 SX35 FPGA and Texas Instruments' DM6446 DM SoC DSP. With a price tag of $14000, this platform is also mostly aimed at professional developers.

4. CRC's Coral Cognitive Radio Platform

   - Coral offers an experimental Cognitive Radio platform operating in 2.4 GHz and 5.8 GHz ISM bands, priced at $6000. It is a platform primarily intended for improved spectrum use research. It supports modulation techniques implemented in IEEE 802.11g and 802.11.a, and with TDD and CSMA capabilities. As interesting as it seems, the CR research is not the focus of this project, but the Platform should be considered should it be decided to continue the research in the cognitivity direction.

5. Ettus Research's USRP N210

   -Probably group of the most widespread SDR's in academic environments, Ettus' products imposed themselves as the best-buy platforms for our research. Ettus Research offers several platforms – USRP, USRP2 and USRP N210 that differ in the level of instantaneous bandwidth they can process; reprogrammability of the FPGA; type of interface to the computer (USB or Ethernet) and, of course, price. For the research, one USRP N210 with the RFX2400 daughterboard – RF front end able to operate in the 2.3 – 2.9 GHz band was initially purchased, whereas one additional platform was purchased when it was established that for the measurement process, distancing of the transmit and the receive side for more than approx. 1m was needed. Cost per platform (USRP N210 + suitable daughterboard + antennas) was approximately $2000.

Table 1 presents brief comparison of SDR platforms considered for the project:

SDR Platform	Microsoft Research SORA	Datasoft's Typhoon SDR DP	Lyrtech's SFF SDR DP	CRC's Coral CR Platform	Ettus' USRP-1 with RFX2400	Ettus' USRP-N210 with RFX2400
Frequency range	2.4 GHz; 5 GHz	400 MHz – 4 GHz	200 MHz – 1 GHz; 1.6 – 2.7 GHz; 3.3 – 3.8 GHz	2.4 GHz; 5.8 GHz	2.3 – 2.9 GHz	2.3 – 2.9 GHz
Computational resource	GPP (all processing done by multi-core PC)	Xilinx Spartan-6 FPGAs; dual-core OMAP	Virtex-4 SX35 FPGA; DM6446 DSP	FPGA	Altera Cyclone FPGA	Xilinx Spartan 3A-DSP3400 FPGA
Maximum signal bandwidth	20 MHz	20 MHz	22 MHz	20 MHz	8 MHz	50 MHz (8-bit mode)
Supported OS	Windows XP	Linux; VxWorks; Android	Linux	Linux Fedora Core or equivalent	Linux; Windows; MAC OS X	Linux; Windows; MAC OS X
Supported software	SORA SDK	GNU Radio	MATLAB; Simulink; Real-Time Workshop	Linux-OpenWRT with customized MadWIFI driver	GNU Radio; Simulink; LabView (test drivers)	GNU Radio; LabView (test drivers)
Price	$5000	$10000	$14000	$6000	$1000	$2000

Table 1: Comparison of considered platforms

3.2 USRP

The Universal Software Radio Peripheral (USRP) is a fairly low-cost SDR system developed by Ettus Research, US – based company lead by Matt Ettus. The system consists of a motherboard with FPGA, 2 pairs of DACs and ADCs, digital downconverters and upconverters with programmable interpolation rates, and a daughterboard functioning as a RF front-end. The connection to the PC is done via Ethernet cable. Depending on the type of the daughterboard, transceiving capabilities can be achieved anywhere from 1 MHz to 5.9 GHz. Components will be discussed in greater detail as follows:

3.2.1 USRP N210 motherboard

The “heart” of the USRP, the motherboard is where all the circuitry is integrated, and daughterboards installed. N210 motherboard has the MIMO (Multiple Input – Multiple Output) port, which can be used to connect multiple USRP systems. For applications where precise synchronization is of high importance, input to an external reference clocking option is supported.

N210 communicates with a PC via a Gigabit Ethernet interface with connector set on the front side of the board. Ethernet interface can sustain simultaneously transmitting up to 50 MHz of bandwidth in and out of the radio.

Picture of the N210 motherboard is given in Figure 2:

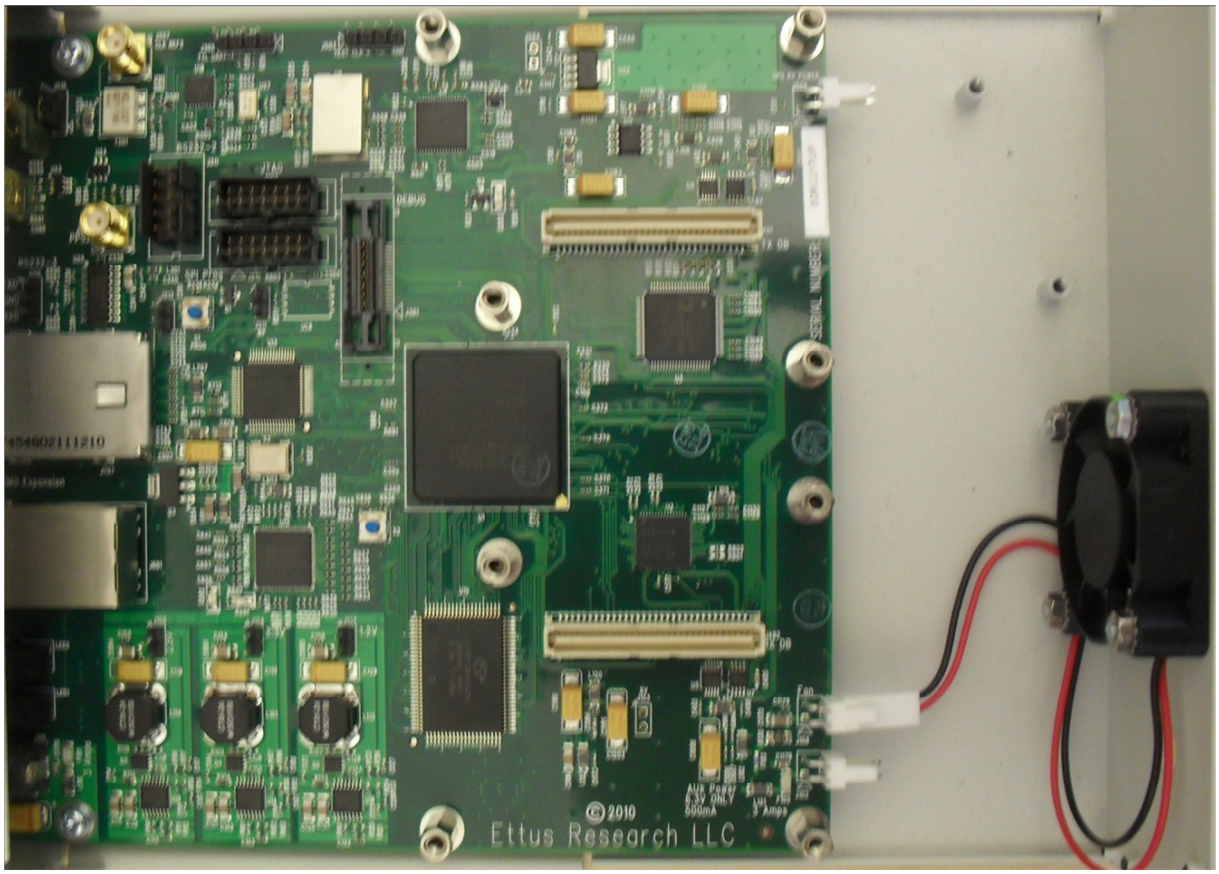

Figure 2: USRP N210 motherboard

3.2.2 FPGA

FPGA is the key element of USRP, providing digital signal processing, as well as timing logic for clock, chip rate, baud rate and time slot synthetization. FPGA's signal processing is done in VHDL – hardware description language used in electronic design automation - instead of common programming language like C. However, momentarily some work is being done on developing a tool that could generate VHDL code from C (which would make life much easier for programmers without VHDL background) [4].

USRP N210 is powered by Xilinx Spartan XC3SD3400A FPGA, which consists of five main programmable elements:

- XtremeDSP 48 Slice - supports many programmable functions, such as multiplier (18 bit x 18 bit), multiplier-accumulator (48 bit), pre-adder/subtractor, magnitude comparator, wide counter, etc.
- Block RAM - data storage
- Configurable Logic Blocks (CLBs) - perform many logical functions and store data
- Input/Output Blocks – control the data flow between input and output pins and the device's internal logic
- Digital Clock Manager (DCM) Blocks – provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase-shifting clock signals

Providing configuration parameters to the FPGA firmware is done by loading configuration data into reprogrammable CMOS configuration latches (CCLs) that collectively control all functional elements and routing resources. The FPGA's configuration data is stored externally in PROM or some other non-volatile medium, either on or off the board. After applying power, the configuration data is written to the FPGA. [2]

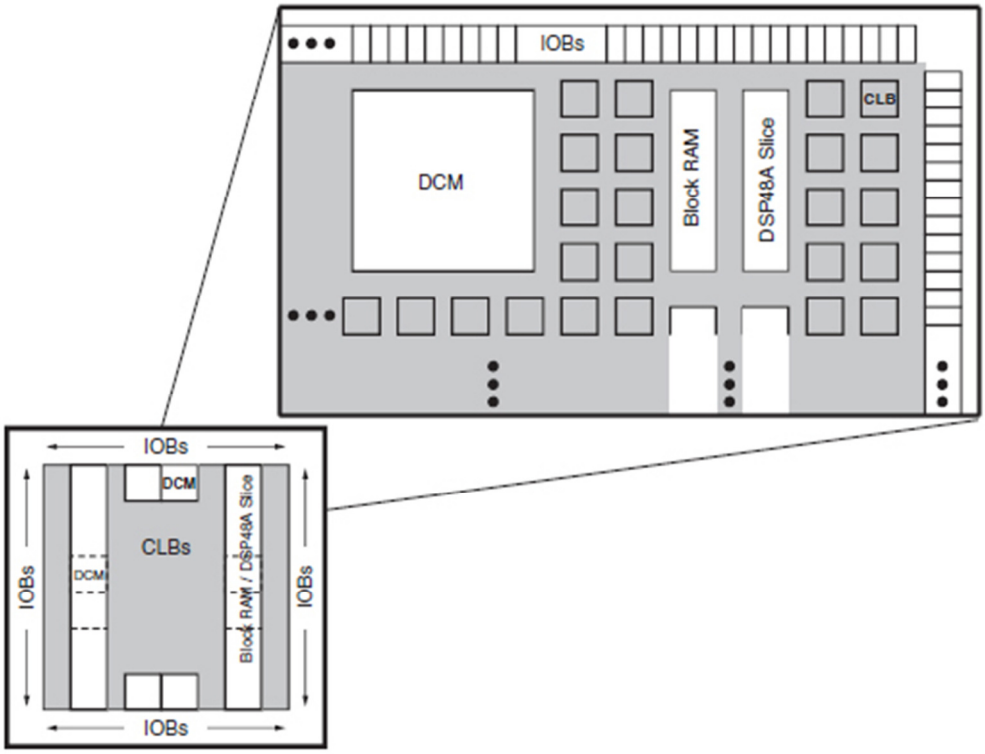

Figure 3: Xilinx Spartan XC3SD3400A FPGA architecture. Source: [2]

3.2.3 DACs and ADCs

On the receive side, in order for the PC to be able to do the processing of the signal, the signal has to be digitalized first. This is the function of Analog to Digital Converters – which perform sampling at a certain rate and quantization of the signal. Sampling rate refers to the frequency with which ADC measures the analog signal. Also, dynamic range is defined for an ADC – this is the number of signal levels that ADC can distinguish. Before ADC, programmable gain amplifier is deployed with the function of amplifying the input signal. USRP N210 ADC's full range is 2V peak-to-peak with the input 50 ohms impedance. N210 contains two 14-bit ADCs with the sampling rates of 100 MS/s.

Transmit side, analogously, uses Digital to Analog Converters, followed by the programmable gain amplifier. USRP N210 DAC's full range is 1V peak-to-peak with the input 50 ohms differential. There are two 16-bit DACs with sampling rates of 400 MS/s on the N210 board.

3.2.4 Digital upconverters and downconverters

The platform contains two digital downconverters with programmable decimation rates, in charge of mixing, filtering and decimating arriving signals in the FPGA. Digital downconverters shift the frequency band of the incoming high sampling rate digitized signal to the baseband and lower the sampling rate without any information loss. First, the digitized stream is mixed with a digitized cosine (for I channel) and digitized sine (for Q channel), producing the sum and the difference components. These outputs are then put through the identical digital filters, filtering unwanted components. At this point, because the bandwidth of the signals we want to process has been reduced, sampling frequency can be losslessly decimated*[1].

Analogously, USRP's two digital upconverters' purpose is translating a signal from baseband to IF band. Digital upconverters first take the relatively low-sampled input baseband signal, filter it and convert it to a higher sampling rate. Multiple FIR filters are used to interpolate the signal up to 100 MS/s. Then, the signal is modulated onto the carrier frequency via carrier synthesizer and sent to the transmitter.

3.2.5 Daughterboards

USRP's daughterboards serve as RF front-ends, covering different frequency bands and differ as transmitters, receivers, or transceivers. Each USRP N210 motherboard has two daughterboard slots – transmit-only and receive- only daughterboards take up one slot, whereas transceivers take up two slots, meaning that both transmit and receive side can be implemented with one USRP.

Available daughterboards are as follow:

[1] Decimation refers to eliminating certain samples, thus directly decreasing the sample rate. The inverse process is called Interpolation, where the sampling rate is increased by adding samples and then performing the filtering back to the original bandwidth. Sometimes, when we want to perform a decimation by non-integer ratios, i.e. we want to decimate by x/y, we will first perform interpolation by y, and the decimate by x. This process is called Rational Resampling.

BasicTX / **Basic RX**: BasicTX works as a transmitter, and BasicRX as a receiver in the 1 MHz - 250 MHz frequency band. They don't include RF front-end, therefore they must be connected to an external RF front-end, and are serving as an IF interface. Since boards don't contain mixers, filters or amplifiers, DACs and ADCs are directly connected to SMA connectors.

LFTX / **LFRX**: transmit-only and receive-only boards very similar to BasicTX / Basic RC, except their frequency response ranges from DC to 30 MHz, which is achieved through implementing of differential amplifiers instead of transformers. 30 MHz LP anti-aliasing filters are integrated into boards as well.

TVRX2: MIMO-capable dual receiver board operating in 50 MHz – 860 MHz (VHF and UHF) band. TVRX2 consists of two completely separate low-IF receivers each with a 10 MHz bandwidth which can be used at the same time on different (or same) frequencies. It fully supports UHD, but not the old USRP drivers.

DBSRX2 – full RF front end operating in 800 MHz – 2.4 GHz (excluding the 2.4 GHz – 2.48 GHz ISM band). It is a MIMO-capable board with a 1 MHz – 60 MHz software-controllable channel filter.

WBX / **SBX** – WBX is a transceiving daughterboard with frequency range from 50 MHz to 2.2 GHz and SBX operates in 400 MHz – 4.4 GHz band. The boards have a common design, with dual synthesizers supporting independent TX and RX frequencies, and a maximum transmit power of 100 mW (20 dBm).

XCVR2450 – transceiver for 2.4 GHz – 2.5 GHz and 4.9 GHz – 5.9 GHz, XCVR2450 only has a single synthesizer that is shared by TX and RX side, meaning that the full duplex mode cannot be achieved. Maximum transmit power is 100 mW (20 dBm).

RFX boards – four transceiver boards with the common architecture, aimed for use in different frequency bands. Dual synthesizers allow for a full duplex mode. RFX900 works in the 750 MHz – 1.05 GHz band with a typical transmit power of 200 mW; RFX1200 in the 1.15 GHz – 1.45 GHz with a 200 mW transmit power; RFX1800 in 1.5 GHz – 2.1 GHz band with 100 mW transmit power, and RFX2400 operates in 2.3 GHz – 2.9 GHz, allowing for transmit power of up to 50 mW (17 dBm).

Other daughterboards – It is worth noting that except Ettus Research, there are also few other manufacturers offering USRP-compatible daughterboards, such as Epiq Solutions with their Bitshark USRP RX (BURX) receive-only daughterboard covering 300 MHz – 4 GHz band.

For the project, transceiver boards operating in 2.4 GHz ISM frequency band able to operate in full-duplex mode were needed, therefore two RFX2400 daughterboards were chosen (one for each USRP).

3.2.6 Antennas

HyperLink Wireless HG2458RD-SM 2.4 GHz to 5.8 GHz tri-band rubber duck antennas were used for the setup. These omni-directional antennas provide broad coverage with 3 dBi gain, have 50 W specified power and specified operating temperature between -40 and 100 F (-40 °C and 60 °C). The antennas are able to operate in the 2.4 GHz – 2.5 GHz; 4.9 GHz – 5.3 GHz and 5.7 GHz – 5.8 GHz frequency bands. They use vertical polarization and are connected to USRP via SMA connectors. Because of the tilt-and-swivel nature of connectors, antennas can be deployed vertically, horizontally or at an inbetween angle. They are 198 mm long and 13.1 mm wide. Antenna's radiation patterns are shown in Figure 4.

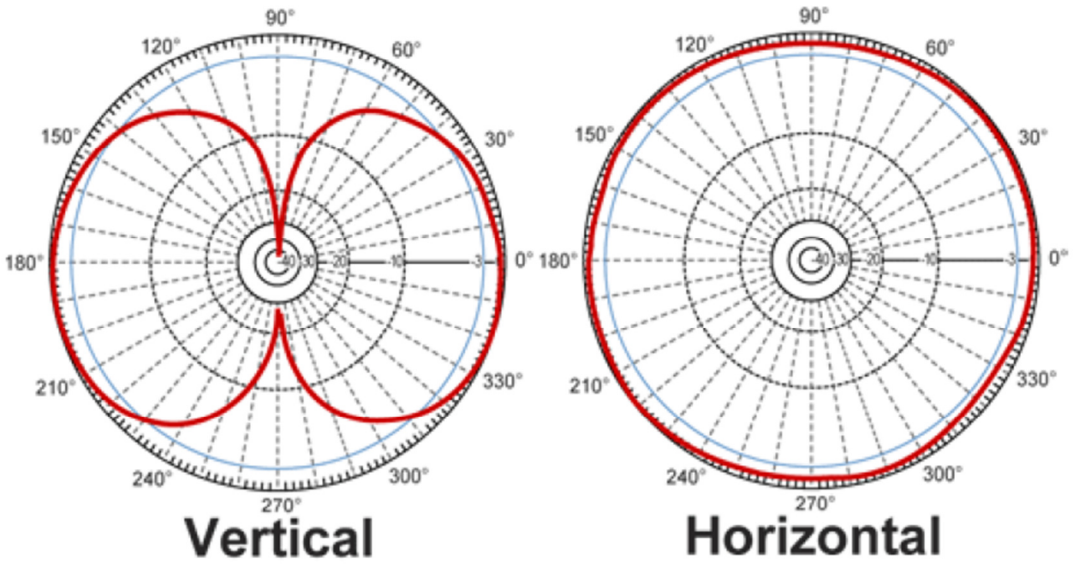

Figure 4: Radiation patterns of HG2458RD-SM antenna in vertical and horizontal plane. Source: [5]

3.2.7 UHD Drivers

USRP N210 runs on Universal Hardware Driver set of drivers, as opposed to older USRPS (USRP 2 using “raw” Ethernet Interface, and USRP 1 USB 2.0 interface). “Universal” implies platform-neutrality of UHD-based USRPs, bringing a more consistent abstraction layer for SDR-based systems and in the end allowing for USRP devices to be “more recognized” in the world of SDR technology.

However, at this time UHD is still a fairly new design, and a lot of GNU Radio’s functions and projects made for “old” drivers were still not translated to be compatible with UHD drivers. This was an aggravating factor while doing the Thesis, which made USRP’s and GNU Radio’s naturally steep learning curve even steeper.

At the moment of writing this Thesis, UHD drivers didn’t support Matlab and Simulink although the support was announced to be coming soon, and drivers that would enable for Labview support were still in the experimental version and were not tested.

3.3 Python

Python is an interpreted, interactive, object-oriented (OOP) scripting language known for its simple, easy-to-use syntax. It is a freeware language written in portable ANSI C that runs on Windows, Unix/Linux, MAC OS X, Java, etc.

Because of its object-oriented nature, Python programs can specialize classes written in other OOP programming languages, such as C++. This feature is often used in GNU Radio, where the signal processing blocks are written in C++, and Python is used to “glue” them together and control the digital data flow. This is done using Simplified Wrapper and Interface Generator – SWIG by creating shared libraries common for both Python and C++.

3.4 GNU Radio

GNU Radio is an open-source software toolkit founded by Eric Blossom in 1998. that, coupled with hardware equipment such as USRP, allows for a complete platform for building Software Defined Radios (although GNU Radio can also be used as a stand-alone software package). Recommended operating system for building GNU Radio is Linux, but it can also be built on MS Windows using one of Linux-like environments such as Cygwin or MinGW/MSYS, as well as on MAC OS and NetBSD.

Most of GNU Radio's applications are written in Python, whereas C++ is used for implementing signal processing blocks. Python commands are used to control all of the USRP's software-defined parameters, such as transmit power, gain, frequency, antenna selection, etc., some of which can be modified while the application is being executed.

GNU Radio is built on two main structural entities – signal processing blocks and flow graphs. Blocks are structured to have a certain number of input and output ports, consisting of small signal-processing components. When the blocks are appropriately connected, a flow graph is made. GNU Radio blocks can be categorized as sinks, sources and filters. Sources are blocks consisting only of outputs and have no inputs and are used as the first element in building the flow graph. Sinks consist of inputs and have no outputs and are typically the last element in building the flow graph. Filters are all the inbetween blocks and consist of both inputs and outputs.

A number of blocks, such as different modulation/demodulation techniques, various filters, signal indicators and widgets, etc. are integrated within GNU Radio, while it is also possible to write and add new blocks. Graphical interfaces, such as FFT sink and oscilloscope, are also supported in GNU Radio.

Flow graphs are created either as hierarchical blocks or as top blocks. Top blocks are top-level flow graphs that contain all other flow graphs and have no input/output (IO) ports. Hierarchical blocks, on the other hand, contain a certain number of IO ports (used to connect to other blocks) that is forwarded to the parent class via the *init*[2] function. All of the basic

[2] GNU Radio's commands, functions and block names will from this point on written in *italic*

signal-processing blocks are, thus, connected within hierarchical blocks and can that way be used as one block.

Communication between blocks is achieved using data streams, where all stream elements are using certain data type. In order for data stream to be successfully initialized, data types between output of one block and input of the next have to be adequately set. Supported data types in GNU Radio are:

- Byte – 1 byte of data
- Short – 2 byte integer
- Int – 4 byte integer
- Float – 4 byte floating point
- Complex – a pair of floats, equaling to 8 bytes

Data type used is defined in the end of the signal-processing block, i.e. *gr_multiply_ii* will take two integers as an input and produce integer as an output, whereas *gr_multiply_ff* will do the same thing with floats, producing float as an output . Blocks for converting data types from one type to another, such as *gr_complex_to_float* block, also exist within GNU Radio.

The *dial_tone* example, often referred to as “Hello world of GNU Radio”, is given and explained below. *Dial_tone* generates two sine waves of the same amplitude at frequencies 350 Hz and 440 Hz, which corresponds to the sound of US dial tone, and outputs them to the sound card.

Code listing 1: dial_tone.py example

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

from gnuradio import gr
from gnuradio import audio

def build_graph ():
    sampling_freq = 48000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect ((src0, 0), (dst, 0))
    fg.connect ((src1, 0), (dst, 1))

    return fg

if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

The first line, *#!/usr/bin/env python*, points python to the location of python executable, and is a line that has to be added to every program that we want to run directly from terminal (as an executable).

Then, we define which modules to import – in this case, *gr*, which is always imported, and *audio*, which allows us to use audio sink.

After that, sampling frequency is set according to sound card's specifications (48 kHz is the sampling frequency of majority of modern sound-cards), and the amplitude to 0.1.

gr.sig_source_f is used to create sine waves at the frequencies of 350 Hz and 440 Hz, where *_f* extension, as previously explained, indicates that the produced data is of type float.

Then, the audio sink that writes received data to the sound card is created. It is worth mentioning that audio sink only accepts float data as an input.

fg.connect connects the flow graph's blocks – the first sine wave is connected to the port 0 of the audio sink, while the second sine wave is connected to port 1.

This completes building the graph, which can be started using the command *start* and stopped using the command *stop*.

4 Working with GNU Radio: implementation and measurement process

This chapter describes the work that has been done with USRPs and GNU Radio, starting with GNU Radio's GUI interface – Gnu Radio Companion. After that, modifications to UCLA Zigbee PHY's code were done and used to perform measurements which were to be compared with measurements done with the "conventional" equipment.

4.1 Setting up the equipment and initial testing

For the measurements, GNU Radio 3.3.0 (later replaced with 3.4.0 version) with Gnu Radio Companion was used on computers running Linux Ubuntu 10.10. Prior to installing GNU Radio, the following prerequisites were installed:

- Development Tools (needed for compilation):
  - g++
  - git
  - make
  - autoconf, automake, libtool
  - sdcc
  - guile
  - ccache (recommended for frequent compiling)
- Libraries (needed for runtime and for compilation)
  - python-dev
  - FFTW 3.X (fftw3, fftw3-dev)
  - cppunit (libcppunit and libcppunit-dev)
  - Boost 1.35
  - wxWidgets (wx-common) and wxPython (python-wxgtk2.8)
  - python-numpy
  - python-scipy

- python-matplotlib
- Numeric
- ALSA (alsa-base, libasound2 and libasound2-dev)
- Qt
- SDL (libsdl-dev)
- GSL GNU Scientific Library

- SWIG 1.3.31
- QWT and QWT PLot3d libraries (optional for Qt Gui)

Specifications of PCs used are as follows:

PC no.	Processor	memory	OS	Kernel
PC1	Intel CPU U1300 @1.06GHz	2 GB	Ubuntu 10.10 (Maverick)	Linux 2.6.35-28-generic
PC2	Intel Core i3 M350 @2.27GHz	4 GB (3 GB used)	Ubuntu 10.10 (Maverick)	Linux 2.6.35-28-generic
PC3	Intel Core 2 Duo T9900 @3.06 GHz	4 GB	Ubuntu 10.10 (Maverick)	Linux 2.6.35-28-generic

Table 2: Specifications of PCs used in the measurements

During the measurements, it turned out that PC1 had inadequate processing power to handle high sampling rates USRPs streamed, and was replaced with PC3.

Both USRPs were loaded with the latest firmware, using *usrp_n2xx_net_burner_gui.py*.

Default Ethernet address of USRPs, set up on the Ethernet interface eth1, was 192.168.10.2. In order for USRPs to be found, it was necessary to set up the Ethernet address of the host every time USRP was reset, to 192.168.10.1. This is done using the command

sudo ifconfig eth0 192.168.10.1 .

Initial testing using USRP[3] included pinging to USRP and, once successful, running examples installed with GNU Radio under uhd/host/build/examples. Examples included *benchmark_rx_rate*; *tx_samples_from_file* / *rx_samples_to_file*, and *tx_timed_samples* / *rx_timed_samples*.

Once all of the initial tests were successfully performed and it was concluded that the USRPs are set up appropriately, working with GRC could be started.

Some of the measurements in this Thesis were done using only one USRP, with daughterboard's TX/RX port used as the transmitting side and RX2 port as the receiver, while some measurements (later ones) were performed using two USRPs.

4.2 Working with GNU Radio Companion

GNU Radio Companion is a graphical user interface for GNU Radio that allows building flow graphs by simply connecting visually-presented blocks. GRC is highly intuitive interface suitable for GNU Radio beginners that resembles Matlab Simulink's one, so anyone with some background in working with Simulink shouldn't have problems learning GRC as well.

From the Ubuntu terminal, GRC is started with command:

gnuradio-companion .

4.2.1 Dial_tone example

The dial_tone example, compiling which using "pure" Python code was previously explained, can also be done using GRC as is shown in Figure 5:

[3] *It should once again be noted that most of GNU Radio's pre-made examples couldn't be run since they weren't compatible with UHD drivers, and it took some time before sufficient competences were acquired to modify those examples*

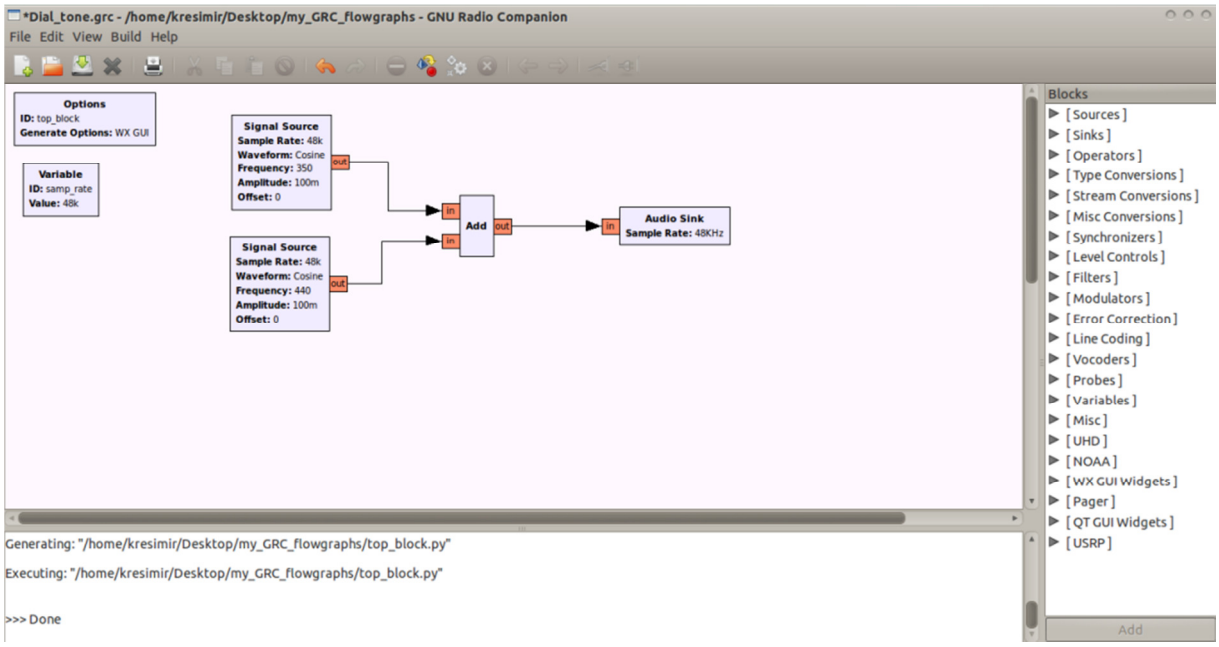

Figure 5: Implementation of dial_tone example within Gnu Radio Companion

Creating this flow graph is as simple as dragging two signal sources from the right menu, setting their sample rate (which can be done using the *samp_rate* variable), connecting them to an *add* block and connecting the output of that block to Audio sink (whose sample rate can be chosen from the block's properties, which is accessed by double-clicking on the block). Once the graph is set, the program can be executed using *Execute the flow graph* button.

This simple example shows how using GRC simplifies the process of making flow graphs. Unfortunately, at the moment, the subset of blocks incorporated into GRC is still quite limited compared to the amount of blocks which are at one's disposition while creating flow graphs using Python. Although there is a possibility of manually importing made blocks, this process is not trivial.

4.2.2 Using USRPs with GRC – NBFM and WBFM

Renowned for its high quality performance, frequency modulation (FM) is the most commonly used analog modulation technique, with particular exertion in the VHF band for FM radio systems.

FM changes the current frequency of the modulated (transmitting) signal depending on the amplitude of the modulating signal, as is shown in Figure 6:

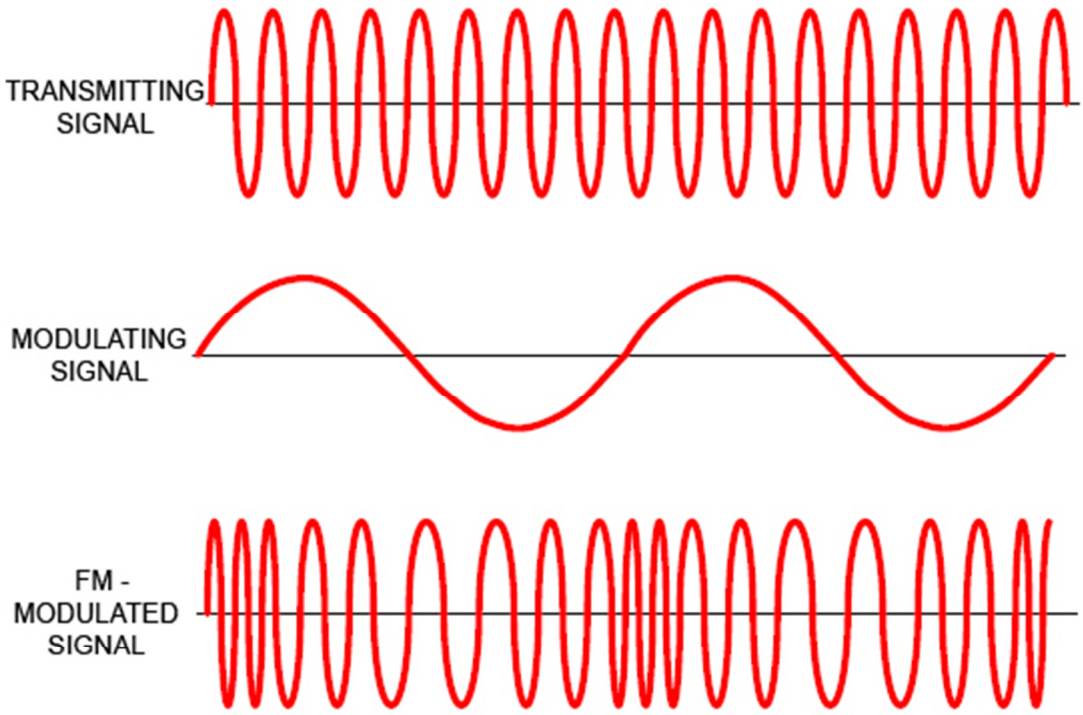

Figure 6: Frequency modulation of the transmitting signal

As it can be seen, the signal varies as a function of voltage of modulating signal. The amount by which this frequency variation occurs is important, and is known as the frequency deviation and is typically measured in kHz.

From frequency deviation, modulation index (m_f) can be defined as:

$$m_f = \frac{\Delta f_{FM}}{f_M},$$

where Δf_{FM} and f_M are the frequency deviation of modulated signal and the highest frequency component of the modulating signal, respectively.

When m_f is small (approximately <=0.4), frequency deviations are small as well, and modulated signal consists only of the transmitted-signal component and two side-components. This is called NarrowBand Frequency Modulation (NBFM).

When m_f is larger, frequency deviations are also larger, and modulated signal, besides transmitted-signal component, also contains multiple side-components. This is known as

WideBand Frequency Modulation (WBFM).

Although higher quality of transmission can be achieved using WBFM, NBFM uses narrower bandwidth for modulation, accomplishing better spectral efficiency.

WBFM and NBFM transmit and receive blocks are implemented within GRC. The following flow graphs serve as TX and RX of an NBFM-based channel:

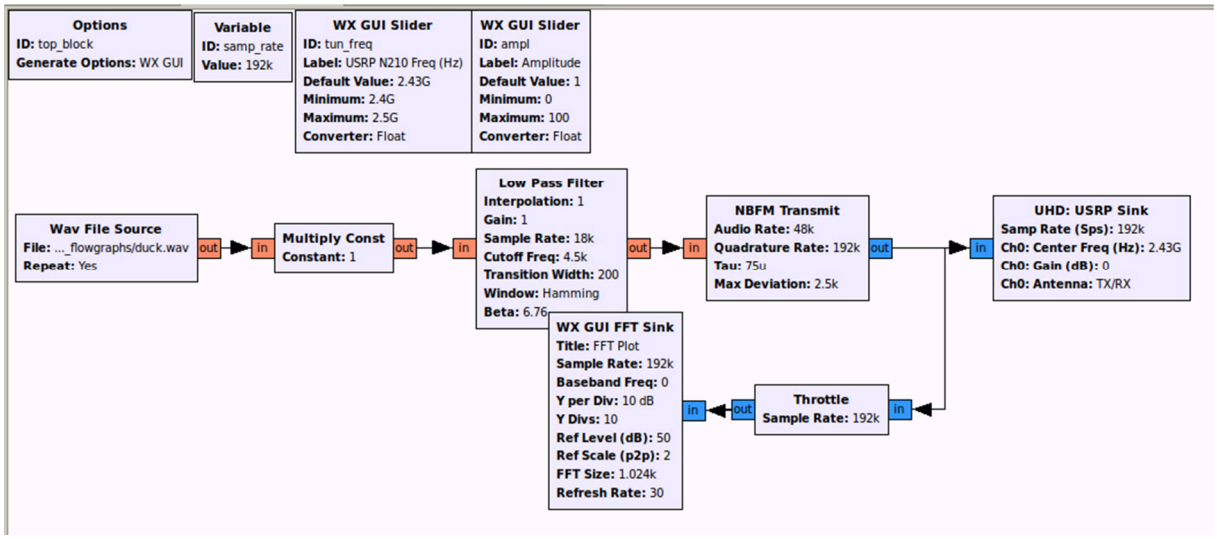

Figure 7: GRC flow graph of NBFM transmitter

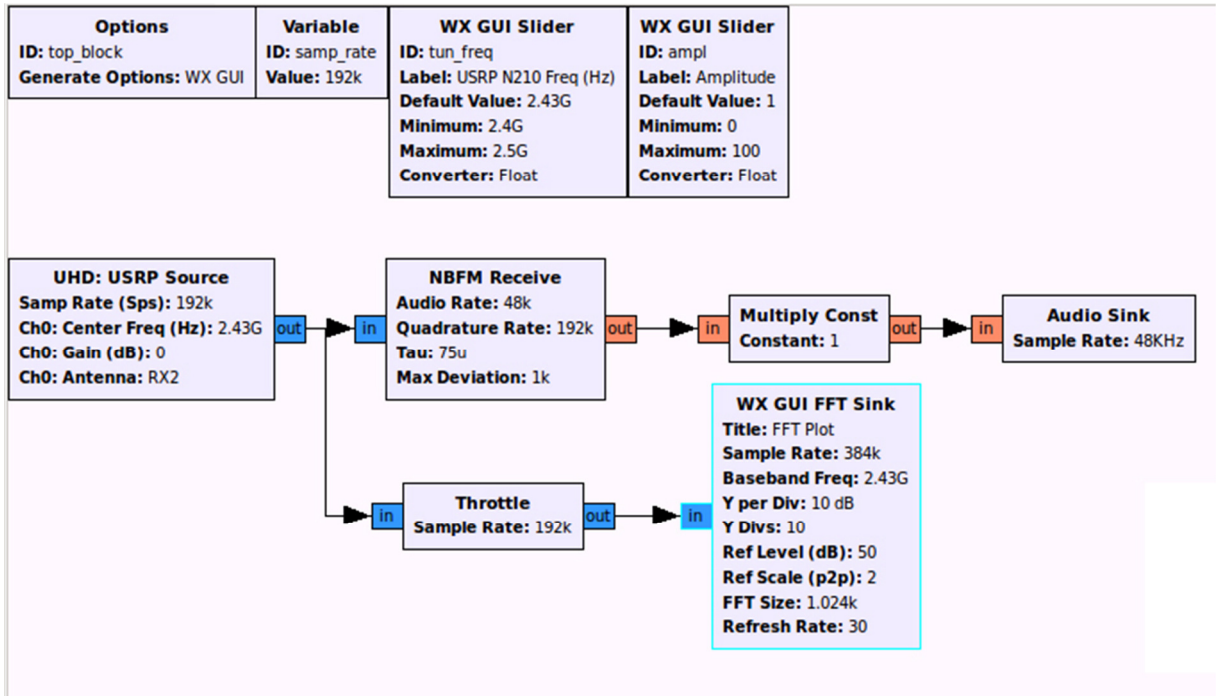

Figure 8: GRC flow graph of NBFM receiver

As a source, *duck.wav* – free, 803 kB large .wav file – is used. After setting the amplitude by

passing the signal through the *Multiply Constant* block, it is filtered with the *Low Pass Filter*, with the cutoff frequency set to 4.5 kHz. Filtered signal is then modulated in the *NBFM Transmit* block. The *Max Deviation* parameter, defined within the *NBFM Transmit* block, decides on the number of side-components we are transmitting – setting the parameter to a higher value should increase the modulation index and use wider bandwidth for transmitting, thus improving the quality.

The output of the *NBFM Transmit* is connected to the *UHD USRP sink*, where transmit frequency, gain, bandwidth and USRP's sampling rate can be modified.

Also, the sampling rates need to be consistent in all blocks, and filter's sampling rate and WBFM's *Quadrature Rate* should always be at least two times bigger than the source signal's maximum frequency (which for NBFM shouldn't exceed 4.5 kHz), in order for the Nyquist theorem[4] to be satisfied.

On the receive side, *UHD USRP source* - whose *Center frequency* and *Sampling rate* need to be set to coincide with *UHD USRP sink* – forwards the received data to the *NBFM Receive* block, which demodulates the signal. *NBFM Receive*'s parameters have to be set the same as *NBFM Transmit*'s ones (otherwise, in case that *Audio/Quadrature rate* are not set appropriately, interpolation or decimation via one of GRC's blocks such as *Keep 1 in N* or *Rational Resampler* has to be done). Demodulated signal is output to *Wav File Sink*, which stores demodulated data as a .wav file, as well as to *Audio Sink*, allowing us to listen to the received signal in the real time.

Also, *UHD USRP Sink* and *UHD USRP Source* were connected to *WX GUI FFT Sink*, showing the FFT spectrum of the transmitted/received signal. Between them, a *Throttle* block had to be placed in order to prevent the PC from freezing. This block throttles the flow of samples so that the average rate of the data stream doesn't exceed certain *Sample Rate* (defined within the block). This has to be done because of PC's insufficient resources to keep up with USRP's high *Sample Rates* in respect to presenting FFT spectrum.

Central frequency of USRP's as well as the amplitude of the transmit signal is set to be controlled via *WX GUI Slider*. This is done by setting *Default*, *Minimum* and *Maximum* value

[4] Nyquist theorem states that, in order to prevent aliasing in DSP, sampling frequency has to be at least two times bigger than the maximum frequency of the signal that is being sampled

for each of the sliders, and then entering Slider's *ID* (which can be arbitrary – in this case *tun_freq* and *ampl*) under *Center Freq* of *UHD USRP Source* and *UHD USRP Sink*. That way, these parameters can be changed during the transceiving period.

The tests were done using different combinations of *NBFM*'s *Max Deviation* and different distances between transmitter and receiver, one of them set as the Line Of Sight (LOS), and the other one as Non Line Of Sight (NLOS) environment, as well as different TX amplitudes,.

Then, similar graphs implementing WBFM were built with appropriate adjustments – *LPF*'s *Cutoff Frequency* was set to 16 kHz, and *Audio*/*Quadrature Rates* as well as *Max Deviation* in *WBFM* blocks were set to higher values.

The measurements were repeated for WBFM.

All the measurements were done using two USRPS, with one USRP connected to PC2 as a transmit side, and the other connected to PC1 served as a receive side.

Quality of the received sound[5] was measured and graded, with grades ranging from 0 to 10 (where grade 0 equals the pure noise, and grade 10 the perfect sound). The grades shouldn't be perceived as absolute – rather, they serve as an orientation as to how changing particular parameter affects perceived quality of received sound compared to other cases. Also, received signal was measured with FFT sink as well.

The measurement results are as follow:

NBFM:

No.	Antenna distance	Max Deviation	RX sound quality	RX peak signal strength (TX PW: 1x / 10x)
1	1.5m LOS	1k	6	-40 dBm / -40.5 dBm
2	1.5m LOS	2.5k	6.5	-40 dBm / -40.5 dBm
3	1.5m LOS	5k	7	-39.5 dBm / -44 dBm
4	5.5m NLOS	1k	3	-52.5 dBm / -52.5 dBm
5	5.5m NLOS	2.5k	4	-52.5 dBm / -52.5 dBm

[5] Grading the receive quality was in this case based on subjective experience, since none of the objective methods were at disposition

6	5.5m NLOS	5k	4.5	-48 dBm / -55 dBm

Table 3: Measurement results for NBFM

WBFM:

No.	Antenna distance	Max Deviation	RX sound quality	RX peak signal strength (TX PW: 1x / 10x)
1	1.5m LOS	75k	8	-45 dBm / -53 dBm
2	1.5m LOS	150k	7.5	-47.5 dBm / -55.5 dBm
3	1.5m LOS	500k	7	-46 dBm / -58 dBm
4	5.5m NLOS	75k	5	-60 dBm / -68.5 dBm
5	5.5m NLOS	150k	4	-62 dBm / -73 dBm
6	5.5m NLOS	500k	2.5	-64 dBm / -75 dBm

Table 4: Measurement results for WBFM

The first look at the results might give the wrong impression that NBFM achieves better signal reception than WBFM (based on the peak signal strength). However, comparing the FFT spectrums should suggest otherwise:

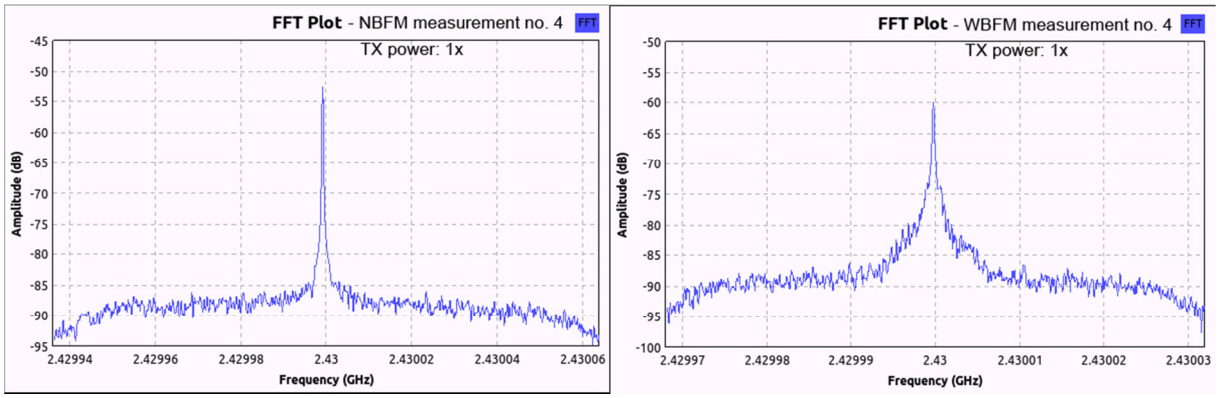

Figure 9: Comparison of RX peak signal strength - NBFM vs. WBFM

As the FFT plots show, NBFM has most of the energy concentrated in a small bandwidth, resulting in a sharp peak around the center frequency. WBFM, on the other hand, has energy spread over wider spectrum, resulting in a smaller peak at 2.43 GHz. Considerably

stronger signal is, therefore, received using WBFM, although the added noise of the audio signal is also much higher.

In order to try and improve reception, measurements were done to see how NBFM and WBFM behave when faced with the amplified signal before transmitting. Signal amplification was done via the *Multiply Const* block deployed within the transmitter graph. Examples of outputs are given in Figure 10:

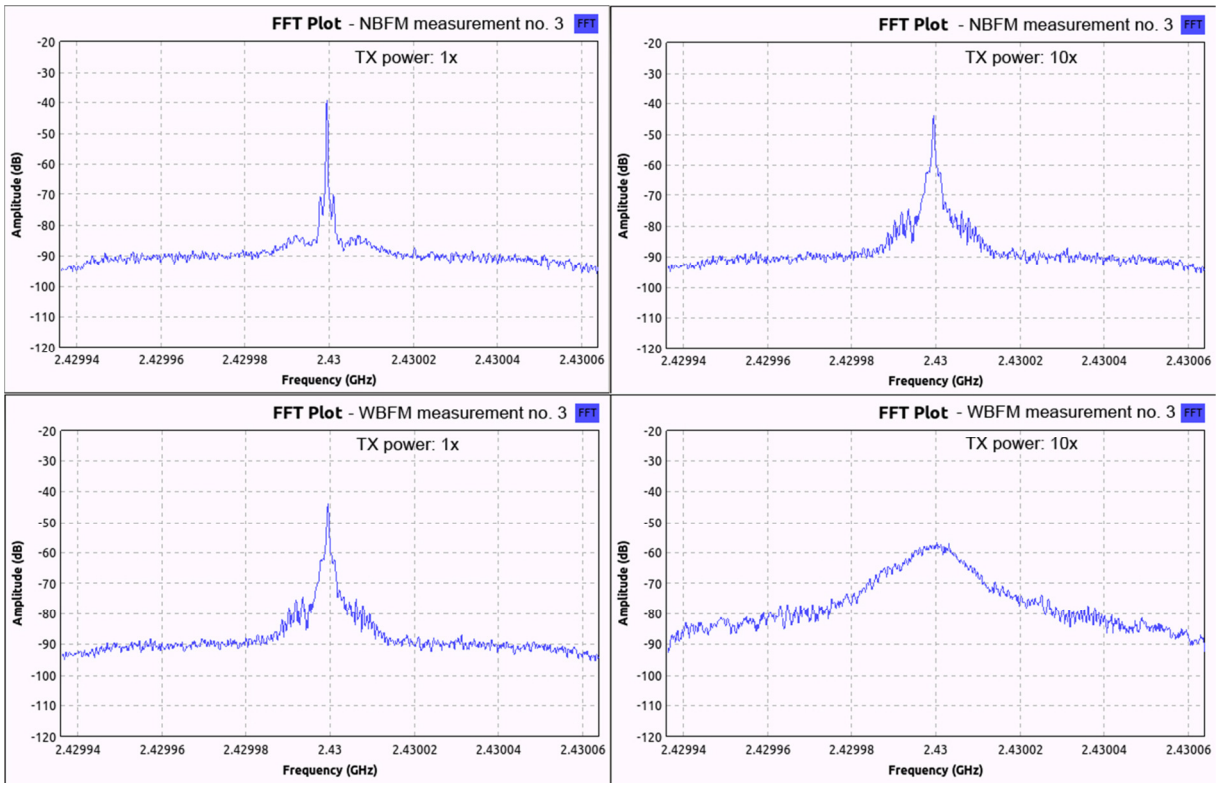

Figure 10: Comparison of NBFM's and WBFM's FFT plots depending on the TX amplification

Results are quite interesting – while WBFM generally produces better outputs without amplification, once the initial signal gets amplified before modulating it, the quality of demodulated signal on the receive side significantly decreases for WBFM. On the other side, NBFM's received signal is, although clearer (in terms of not being able to hear background noise) than WBFM's, quite weak. But, contrary to WBFM, when multiplied by constant before modulation, it becomes stronger, yet doesn't lose much of its quality regarding its spectrum widening.

This is also directly related to the maximum deviation defined within modulation/demodulation blocks – the larger the deviation, the higher the distortion of the amplified signal, as can be seen from Figure 11:

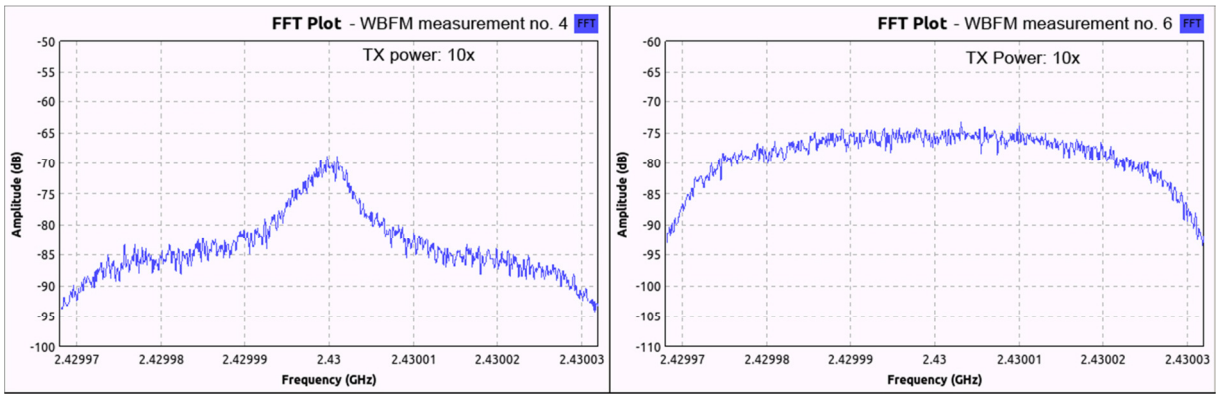

Figure 11: Comparison of received signal's FFT regarding different maximum deviation factors

The signal on the left of the figure, although distorted, can still be recognized despite the strong background noise, but the signal on the right (the one with the higher maximum deviation parameter set) became so contorted that it was very hard to recognize the patterns of the source file in it.

The measurements have shown that it was possible to receive signal in both LOS and non-LOS circumstances with NBFM and WBFM techniques – on further distance in NLOS conditions, NBFM signal had to be amplified in order to get a decent reception, whereas WBFM signal was good enough without amplification (which, when performed, substantially decreased its Signal-to-Noise Ratio).

One interesting thing worth mentioning was that, during the testing, the operating frequency used for FM modulation interfered with frequencies on which WiFi technology uses its channels, resulting in WiFi's periodical low link quality and package losses. The interference influence on FM wasn't taken into account, although it would undoubtedly be interesting to perform these measurements as well.

4.3 Modifying the UCLA Zigbee PHY code

UCLA Zigbee PHY is a GNU Radio project that features implementation of 802.15.4 Zigbee's PHY layer. Upgrading the code - originally developed by Thomas Schmid - in order to perform measurements that would be beneficial in channel characterization was one of the main focuses of this Thesis. The measurements of interest were primarily Packet Error Rate (PER) and Received Signal Strength Indicator (RSSI).

It was decided that, out of the examples provided within the UCLA ZigBee PHY package, *cc2420_txtest.py* and *cc2420_rxtest.py* (attached in Appendix A) were examples suitable for implementation, and refer to transmit and receive side respectively. These examples provide the implementation of the physical layer characteristics of 802.15.4, which include:

- Operability in 2.4 GHz ISM band
- Offset-QPSK (OQPSK) modulation technique
- Structuring packets as follows:
    - 2 bytes that define the type of the frame
    - 1 byte sequence number
    - 0 to 20 bytes of address information
    - payload (total size of the message mustn't exceed 128 bytes)
    - Pad for the USRP – boolean that enables or disables USRP padding
    - length of preamble (by default set to "4")
    - Start of Frame Descriptor (SFD), by default set to "0xA7" by 802.15.4 standard

One of the first tasks was rewriting the code so it could be used with USRP's UHD drivers. Modifying the code included replacing the usrp.sink and usrp.source blocks with uhd.usrp_sink and uhd.usrp_source, as well as setting parameters to correspond with the new blocks – namely this refers to:

- setting UHD USRP's clock configuration to internal, which was done using *self.u.set_clock_config(uhd.clock_config.internal(), uhd.ALL_MBOARDS)* command
- replacing interpolation/decimation factors with UHD USRP's sample rates – via command *self.u.set_samp_rate(options.sample_rate)* we are feeding the USRP with

the sample rate that can be chosen while executing the program. USRP N210s allow for sample rates between 0.1 and 100 MSps.

- controlling the receiver gain with *self.u.set_gain(options.gain)*, also allowing to choose the gain before executing the RX program. The RX gain can be a value between 0 and 90 dB, where up to 20 dB of gain can be achieved via PGA, and up to 70 dB on AD8052 chip. Adjustable TX gain is not supported by RFX2400 daughterboard.
- setting center frequency used for transceiving, with *self.u.set_center_freq(options.cordic_freq)*. The frequency is defined while calling the program.
- removing lines that deal with picking USRP subdevice; automatic switching between Transmit and Receive mode (since USRP N210s allow for working in full duplex mode); and setting PGA values (RX gain is set as described above).

Carrying out these changes (and connecting the new UHD USRP blocks within the flow graph appropriately) was presumably enough to allow for a transmission and reception of the pre-constructed 802.15.4 packages, however the reception couldn't be realized due to the *Power squelch block* connected to the *UHD USRP sink*. *Power squelch*, called with *gr.pwr_squelch_cc* eliminates signals whose power is lesser than a certain threshold in dB (defined as filter's argument). The purpose of the filter is delimiting the number of values that are being fed to the demodulator, therefore easing up on the need for the processing resources. However, since default threshold value of the squelch filter was set to 50 dB, which was some way above received signals' strengths (even with high RX gain ranges), none of the received signals were passed through to the demodulator, and were therefore not decoded. This can be overcome either by removing *squelch filter* and connecting the *UHD USRP sink* directly to the packet receiver, or setting the threshold to a lower value.

Once it was confirmed that sent packets can successfully be received and decoded, it was time to develop the code in order to be able to do the set measurements.

The original code on the TX side includes algorithm for sending 10 identical packages each with 9-byte payloads. On the RX side, the decoding algorithm would count the number of packets that were received and decoded successfully, as well as number of packets that

were received, but for some reason couldn't be decoded. However, the algorithm was unable to distinguish the sequence number of non-decodable packets. It was also unable to take into account packages that were sent, but for some reason weren't received at all (either because the RX signal was too weak, the power squelch threshold set too high or sample rates of *UHD USRP sink* and *UHD USRP source* set inadequately, causing lost packages).

This had to be improved, therefore algorithm was implemented that, on the TX side, would generate the payloads of the transmitted packages as their sequence numbers (so that for example the first transmitted package has payload "0" and the n-th package payload "n-1"). Since payloads consisted of a certain number of bytes, and each byte only has 8 bits, allowing for 2^8=256 different sequence variations, it was needed to send more than 1 byte in each payload (sending, for example, 2 bytes allows for generating up to 256^2=65536 different payload combinations, which proved to be sufficient for the measurements). After that, a set of packets with longer payloads (from now on referred to as "*EOT packets*") were sent, that served as a signal for end of transmission of "transmission" packets.

On the RX side, the loop was created that would first check whether the received packet that can be decoded is *EOT packet* by comparing its length to the pre-known length of *EOT packet* (which, if proven to be true, would stop program's execution, deducing that all of the "transmission" packets indeed were transmitted). Otherwise, if the received packet can be decoded, but isn't an *EOT packet*, it would be unpacked, and its payload would be written into a string array that contains all the successfully unpacked packets (counter for successfully decoded packets is, naturally, also increased). If, however, received packet cannot be decoded, its sequence number would be written into a string array that contains all the packets that were received, but couldn't be decoded because of the bit errors and the counter for bad packets increased.

After all the packets were sent (*EOT* received and *rx_callback* function terminated), loop checks whether payloads of all of the transmitted packages are stored in the *received packets* string array and, If not, stores their payload into a new string array, *nonreceived packets*. This way, it is possible to distinguish exactly which packages were:

- successfully received and decoded

- received, but couldn't be decoded
- not received at all.

From there, packet error rate could easily be deduced.

The other measurement that was of interest to this research was RSSI, therefore a way of calculating RSSI values needed to be found. RFX2400 daughterboard has integrated on-board RSSI sensor that operates at the pre-ADC bandwidth, however is known to produce results that can be highly inaccurate, hence it wasn't used for the measurements.

Instead, RSSI was calculated directly from the incoming data stream. Stream, which was of type *complex_float_32* was first filtered with a band-pass filter, and then put through power squelch block (independent from the first one) and converted to magnitude squared. Then, it was filtered with a single pole IIR filter which "smoothens" the outputs of the *complex-to-magnitude-squared* converter. Because of high sample rates of USRP's, the stream needs to be decimated, otherwise number of samples becomes too large to export to standard spreadsheet processors. In order to get the value in dBm, the decimated stream was logarithmed, and finally saved to a filesink.

It should be noted that files saved this way are saved as a binary data, and in order for them to be able to be opened with text/spreadsheet processors, they need to be converted to format that can be used with these processors. This can be done from the terminal with the command:

```
python -c "import numpy, sys; print '\n'.join(map(str, numpy.fromfile(sys.argv[1], numpy.<data_type>)))" input_file_name > output_file_name
```

The block diagram of the transmitter is shown in Figure 12:

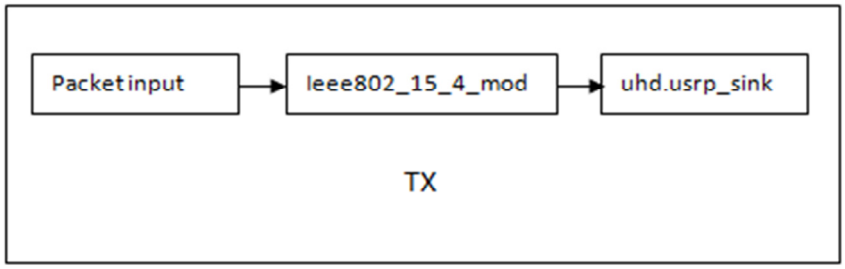

Figure 12: Block diagram of the 802.15.4 transmitter

Figure 13 presents the block diagram of the receiver:

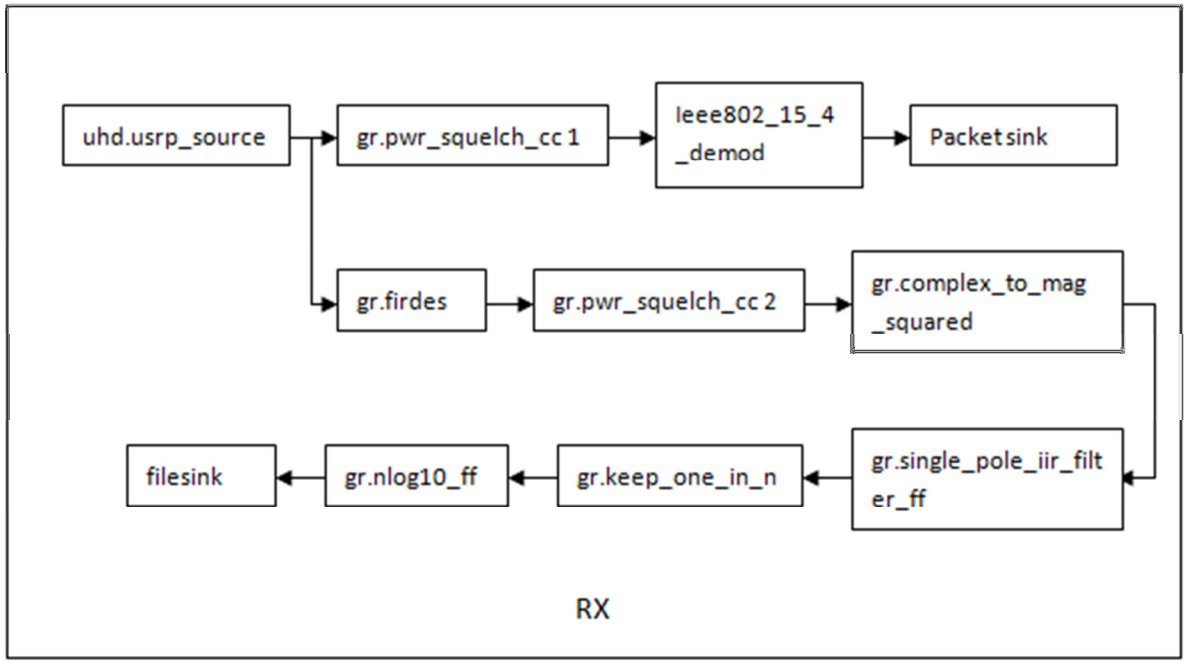

Figure 13: Block diagram of the 802.15.4 receiver

The full code of original *cc2420_txtest.py*, *cc240_rxtest.py*, *ieee802_15_4.py* and *ieee802_15_4_pkt.py* is provided in Appendix A.

Codes of modified programs - *uhd_cc2420_tx.py* and *uhd_cc2420_rx.py* are provided in Appendix B.

4.4 Measurement process and results

The modified programs, *uhd_cc2420_tx.py* and *uhd_cc2420_rx.py* were used for performing measurements. All measurements were done in the premises of the department of Innovation, Design and Technology, IDT, at Mälardalen University. Test setup is shown in the figure below, where PC2 was used as the transmit side and had fixed position, and PC3 as the receive side and its position was changed along IDT's premises.

Figure 14: Picture of the measurements setup

Following parameters had fixed values during the measurements:

- Transmit power, with TX amplitude in *uhd_cc2420_tx.py* set to 8000. Exact output power that correlates to this amplitude is unknown, and should be measured using additional hardware equipment – namely, oscilloscope connected directly (actually, with sufficient attenuator, capable of attenuating at least 30 to 40 dB, between them) to the USRP in charge of transmitting
- Carrier frequency within the system, set to 2.4 GHz.
- Antenna ports on the daughterboards, with TX/RX used on the transmit side and RX2 antenna on the receive side

- Number of packets transmitted – 2550 “transmission” packets and a sequence of *EOT packets* transmitted after that
- Length of packets’ payloads, set to 2 bytes. Payloads, as was previously explained, consisted of their sequence number, with second byte (least significant byte) consisting of values between [0,255] and first byte (most significant byte) consisting of values between [0,9]
- Sleep time between transmitting packets – 50 ms for measuring PER and none for measuring RSSI values

The following parameters were, on the other hand, altered before executing TX and RX programs:

- Distance between TX and RX sides, as well as number and type of objects between them. Measurements were done for distances ranging from 1 m LOS to 45 m NLOS.
- Receiver gain, ranging from 20 to 70 dB
- Sampling rates of the USRPs, since smaller sampling rates coincided with higher number of packet losses, it was tried to find an optimal sampling rate.
- Threshold value of the power squelch filters, set to smaller values for shorter distances and higher values for longer distances and NLOS conditions.

The goal of the measurements was getting continuous RSSI of the received packages, finding average RSSI value, number of received & decoded, received & non-decoded and non-received packages. The results were to show the correlation between these outputs regarding to different path lengths and types (LOS vs. NLOS), RX gains and sampling rates.

The measurements for longer TX-RX distances (18 m and more) were made with a fixed sampling rate (5 MSps), and with higher RX gains (50 dB and higher), since lower RX gains (less than 40 dB) in these conditions led to little to none packets being successfully decoded.

Results of the measurements are as follows:

No.	TX-RX distance [m]	Sampling rate [MSps]	RX Gain [dB]	Good packets	Bad packets	Non-received packets
1	1 m LOS	2	20	1634	0	917
2	1 m LOS	2	30	2422	0	129
3	1 m LOS	2	40	2541	0	10
4	1 m LOS	4	20	2092	3	549
5	1 m LOS	4	30	2551	0	0
6	1 m LOS	4	40	2551	0	0
7	1 m LOS	5	20	586	0	1965
8	1 m LOS	5	30	1803	0	748
9	1 m LOS	5	40	2486	0	65
10	7.5 m LOS	2	20	2348	1	202
11	7.5 m LOS	2	30	2538	0	13
12	7.5 m LOS	2	40	2537	0	14
13	7.5 m LOS	4	20	534	243	1774
14	7.5 m LOS	4	30	2551	0	0
15	7.5 m LOS	4	40	2551	0	0
16	7.5 m LOS	5	20	2551	56	1380
17	7.5 m LOS	5	30	2546	0	25
18	7.5 m LOS	5	40	2548	0	3
19	11 m NLOS	2	30	1146	164	1241
20	11 m NLOS	2	35	2522	2	27
21	11 m NLOS	2	40	2548	0	3
22	11 m NLOS	4	30	1666	63	822
23	11 m NLOS	4	35	2551	0	0
24	11 m NLOS	4	40	2551	0	0
25	11 m NLOS	5	30	2450	1	100
26	11 m NLOS	5	35	2154	8	389
27	11 m NLOS	5	40	2467	4	80
28	18 m NLOS	5	50	2445	0	6
29	18 m NLOS	5	70	2550	1	0

30	30 m NLOS	5	50	2407	14	130
31	30 m NLOS	5	70	2422	16	113
32	45 m NLOS	5	50	10	5	2536
33	45 m NLOS	5	70	1662	31	858

Table 5: Inputs and outputs of 802.15.4 measurements

4.5 Results processing and analysis

Besides the number of successfully received and decoded; received but non-decodable, and non-received packets, the RX program also outputs their sequence numbers (based on their payload). This asset might be particularly useful if the ability to precisely allocate each packet's RSSI value was incorporated into the RX side, giving the exact correlation between RSSI and non-decoded / decoded packets. However, since this allocation wasn't realized within *uhd_cc2420_rx.py*, there isn't much point in presenting those sequence numbers here – it can be mentioned, though, that the distribution of sequence numbers of non-received and non-decoded packets doesn't follow obvious pattern and can be considered fairly random.

The continuous RSSI values for each of the measurements, however, have been collected and saved to a file. Their plots, as well as their average values have been given in this subchapter.

Packet error rates have been calculated as:

$$PER = \frac{no.of\ bad\ packets + no.of\ nonreceived\ packets}{number\ of\ sent\ packets}$$

The calculations are presented in Table 6:

No.	TX-RX distance [m]	Sampling rate [MSps]	RX Gain [dB]	PER [%]	Average RSSI [dBm]
1	1 m LOS	2	20	35.94	-15.54
2	1 m LOS	2	30	5.05	-4.49
3	1 m LOS	2	40	0.39	0.31

4	1 m LOS	4	20	21.63	-16.64
5	1 m LOS	4	30	0	-5.22
6	1 m LOS	4	40	0	0.29
7	1 m LOS	5	20	77.02	-26.62
8	1 m LOS	5	30	29.32	-25.69
9	1 m LOS	5	40	2.54	??
10	7.5 m LOS	2	20	7.96	-26.63
11	7.5 m LOS	2	30	0.51	-17.91
12	7.5 m LOS	2	40	0.55	-2
13	7.5 m LOS	4	20	79.07	-29.03
14	7.5 m LOS	4	30	0	-14.36
15	7.5 m LOS	4	40	0	-4.56
16	7.5 m LOS	5	20	56.29	-36.78
17	7.5 m LOS	5	30	0.98	-26.03
18	7.5 m LOS	5	40	0.12	-18.14
19	11 m NLOS	2	30	55.08	-29.15
20	11 m NLOS	2	35	1.14	-18.59
21	11 m NLOS	2	40	0.12	-6.31
22	11 m NLOS	4	30	34.69	-30.38
23	11 m NLOS	4	35	0	-19.44
24	11 m NLOS	4	40	0	-9.78
25	11 m NLOS	5	30	3.96	-37.09
26	11 m NLOS	5	35	15.56	-26.68
27	11 m NLOS	5	40	3.29	-13.74
28	18 m NLOS	5	50	0.24	-20.15
29	18 m NLOS	5	70	0.04	-7.68
30	30 m NLOS	5	50	5.64	-34.93
31	30 m NLOS	5	70	7.02	-22.48
32	45 m NLOS	5	50	99.61	-40.32
33	45 m NLOS	5	70	34.85	-31.25

Table 6: Calculated PER and average RSSI derived from 802.15.4 measurements

For saving values of RSSI of continuous stream, measurements were done separately, without sleep time between sending packets on the TX side. This was done so that signal strength of "pure" stream could be estimated, without taking into account time when transmit isn't occurring. It should, therefore, be clear that the RSSI values do not coincide with PER of given measurement – instead, they serve as to give an approximation of the signal strength that is to be expected at the receiver.

Figure 15 shows the continuous RSSI outputs for 1 m LOS, sampled at 2 MSps and RX gains set to 20 dB and 40 dB (measurements 1 and 3):

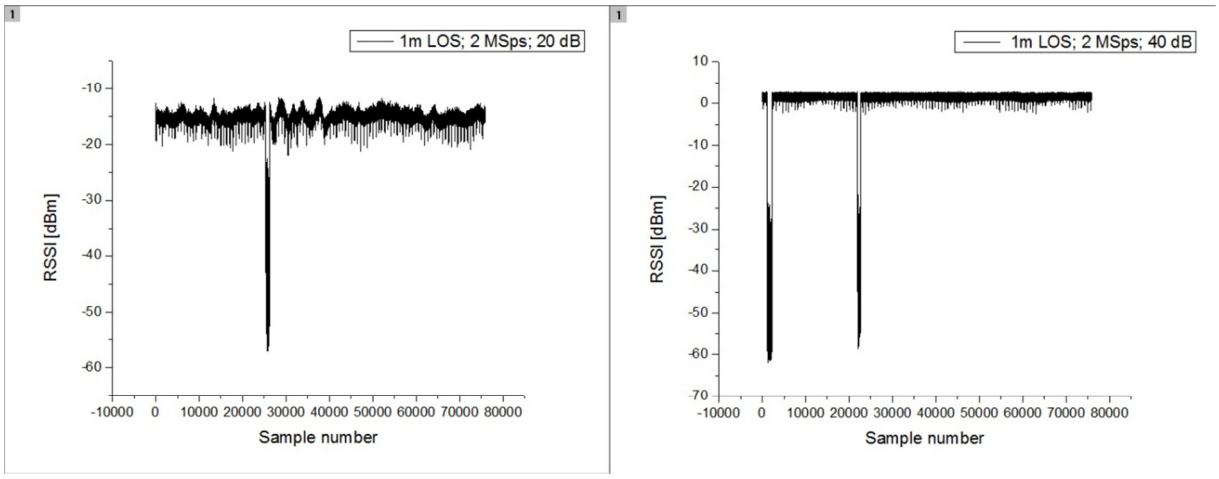

Figure 15: RSSI values of measurements no. 1 and 3

It can be seen that the fluctuations in the received signal strength are much lesser for higher RX gain (resulting in a "flatter" curve). The sudden drops in the RSSI values (for measurement no. 1 from samples around 25400 to 26520, and for no. 2 from samples 1080 to 2190 and from 21960 to 22680) occur not because of the sudden degradations or interference in the channel, but because computer connected to a USRP cannot generate packets fast enough, therefore small occasional pauses where no transmitting is done occur.

These pauses are more emphasized for higher sampling rates. If we are transmitting with sleep time between packages, we will generally achieve better transmission with higher sampling rates, however sampling at the rate of e.g. 5 MSps , proves to be quite a demanding task even for the powerful PCs such as those used in the measurements. Figure 16 comparing measurements no. 19 and 25 illustrates this issue:

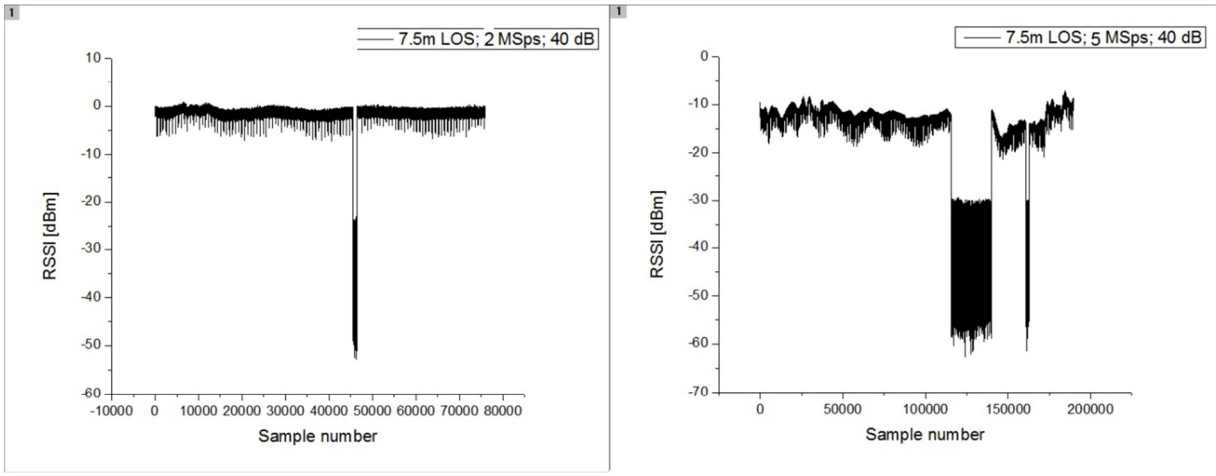

Figure 16: RSSI values of measurements no. 19 and 25

Because of the stated drops in RSSI values, the calculations of average RSSI values are fallible (except for the few cases where no drops have occurred during transmitting period). In order to compare how RSSI values correspond to TX-RX distance, comparing maximum RSSI values is more precise. Figure 17 shows this correlation – all values are derived from measurements done for 5 MSps USRP sampling rate and 40 dB RX Gain.

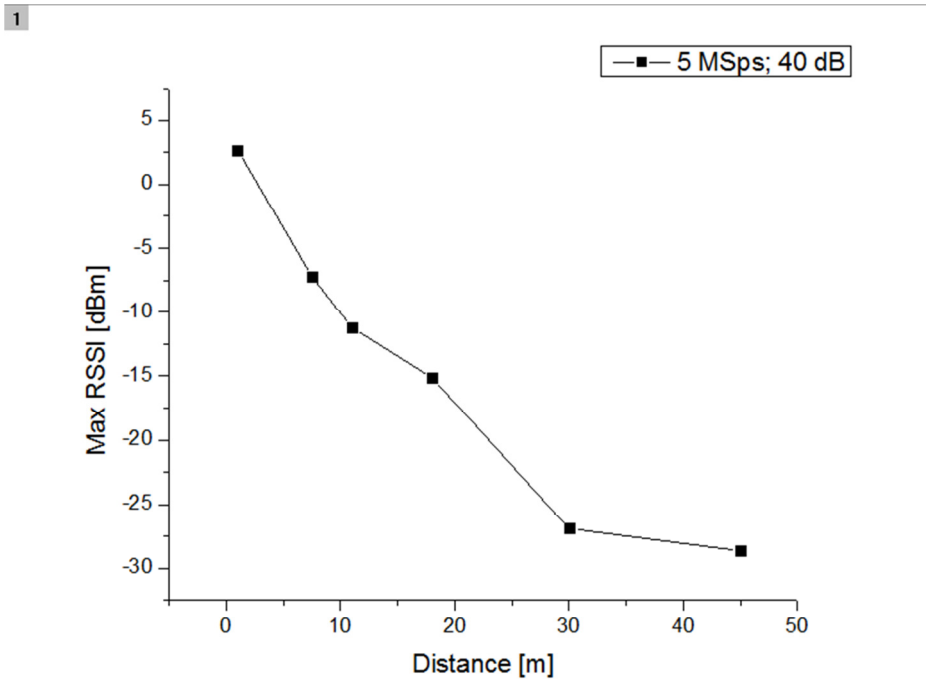

Figure 17: RSSI depending on the TX-RX distance

Distribution of PER depending on the distance for the best case scenario (sampling rate 5 MSps, RX gain 40 dB for LOS and 70 dB for NLOS) is shown in Figure 18:

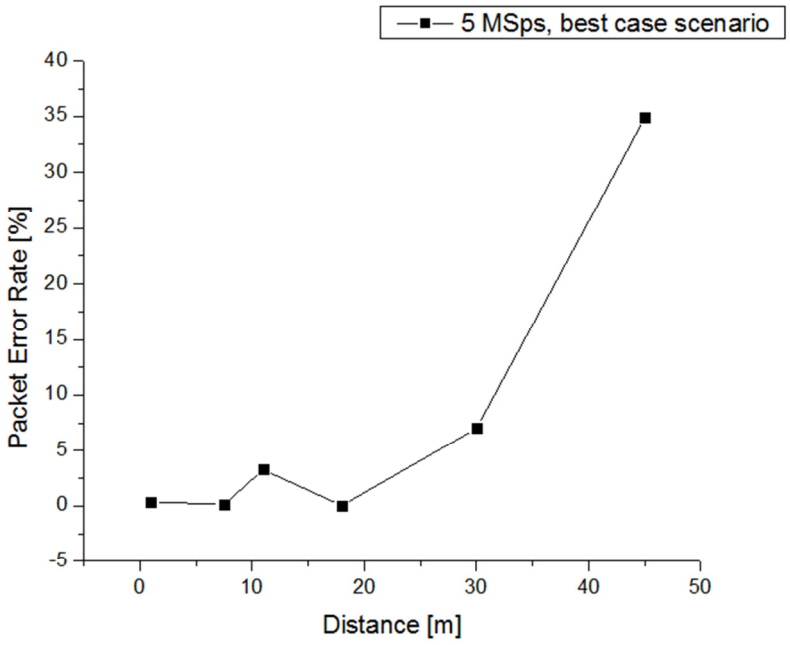

Figure 18: PER distribution depending on the TX-RX distance, best case scenario

Good PER values have been achieved for distances up to approximately 30 m; after that, the received signal strength becomes too small for packages to be received or successfully decoded – the distance where ultimately no packet could be successfully received was around 55 m. One of the simplest solutions for increasing this distance would be amplifying the signal at the transmit side.

5 Possible improvements. Future work.

As was already stated, it would be useful to get precise RSSI values per package, in order to see a correlation with non-received and non-decodable packages. With the implementation presented in this Thesis, this is possible only for cases where received useful signal is significantly stronger than background noise, i.e. for short RX-TX distance and significant RX gain, and even those cases would require considerable amount of data post-processing. Several problems have arised while trying to extract RSSI per package values: without decimation, the stored data is too big for post-processing due to the high sampling rates of USRPs, whereas using decimation results in unequal number of samples per package, making it difficult to recognize which RSSI value belongs to which package.

The idea to overcome this, therefore, is modifying the code in a way that only samples that represent the sent packages are being processed for RSSI measurement. For streams with significantly higher signal strength than noise strength, power squelch filter deployed prior to performing calculations successfully separates the “useful” data from the noise, however as these values become similar, filtering is impossible.

Furthermore, for expanding the possibilities of channel estimation, implementing Bit Error Rate (BER) might also prove to be useful. This implementation should be reasonably reachable, since examples implementing BER exist within GNU Radio, however due to time limitations this functionality couldn't be incorporated into this Thesis.

As for the UCLA Zigbee project itself, one of the upgrades might be developing upper layers (primarily MAC) as an addendum to the physical layer's functions. Improvements in this direction could possibly result in the full compatibility with ZigBee devices.

6 Conclusions

After giving an introduction to the world of Software Defined Radios, the Thesis focused on using one particular platform – Ettus' USRP N210 SDR. Various USRP and GNU Radio capabilities and functions have been tested using GNU Radio Companion, out of which implementation of NBFM and WBFM was explained in the Thesis. Main focus of the Thesis was adapting the UCLA Zigbee PHY project to work with the SDR platforms. This was successfully done, however due to the time limitations combined with fairly steep learning curve of using UHD USRPs with GNU Radio, a couple of measurement metrics – calculating or extracting the RSSI-per-package values and BERs - weren't incorporated in the Thesis. Also, performing larger number of test, as well as testing in different locations and under different conditions would have produced more relevant results.

However, main Thesis goals were achieved, and solid grounds for the future research using USRPs and GNU Radio were established.

7 References

[1]. Wireless Innovation Forum, http://www.wirelessinnovation.org/page/Introduction_to_SDR

[2]. Spartan-3A DSP FPGA Datasheet, http://www.xilinx.com/support/documentation/data_sheets/ds610.pdf

[3]. MDH Wireless Communication Research Group, http://www.idt.mdh.se/qz/?page_id=14

[4]. Fette, B.: *Cognitive Radio Technology, 2nd edition*, USA: Newnes, 2009

[5]. HG2458RD-SM Specifications, http://www.l-com.com/multimedia/datasheets/DS_HG2458RD-SM.pdf

[6]. Dillinger, M.; Madani, K.; Alonistioti, N.: *Software Defined Radio: Architectures, Systems and Functions*, England: John Wiley & Sons Ltd., 2003

[7]. GNU Radio homepage, http://gnuradio.org/redmine/wiki/gnuradio

[8]. Youngblood, G.: *A Software-Defined Radio for the Masses*, QEX, July/August 2002

[9]. Schmid, T.; Sekkat, O.; Srivastava, M.: *An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios*, The second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization January 2007

[10]. Leidelöf, S.: *Software Defined Radio – Waveform Design*, Master Thesis at Linköpings Universitet, 2009

[11]. Rosenqvist, D.: *Software Defined Radio*, Master Thesis at Kungliga Tekniska högskolan, 2009

[12]. Melby, J.: *JTRS and the evolution toward Software-Defined Radio*, Milcom 2002. Proceedings, pages 1286-1290 vol 2., 7-10 October 2002

[13]. Tarver, B.; Christensen, E.; Miller, A.; Wing, E.R.: *Digital modular radio (DMR) as a maritime/fixed Joint Tactical Radio System (JTRS)*, Vol. 1, pp. 163 - 167 vol.1

[14]. Web source: *What is FM, Frequency Modulation*, http://www.radio-electronics.com/info/rf-technology-design/fm-frequency-modulation/what-is-fm-tutorial.php

[15]. Lutz, M.; Ascher, D.: *Learning Python, First Edition*, USA: O'Reilly & Associates, Inc., March 1999

[16]. Jianxin, G.; Xiaohui, Y.; Jun, G.; Quan, L.: *The Software Communication Architecture Specification: Evolution and Trends*, Computational Intelligence and Industrial Applications, 2009. PACIIA 2009. Asia-Pacific Conference on, pp. 341 – 344, November 2009

[17]. Mitola, J. III: *SDR Architecture Refinement for JTRS*, MILCOM 2000. 21st Century Military Communications Conference Proceedings, pp. 214 - 218 vol.1, 2000

[18]. MSR Software Radio Academic Kit data sheet, http://research.microsoft.com/en-us/projects/sora/sora-kit-flyer.pdf

[19]. *Typhoon™ SDR Waveform (WF) Development Platform Product brochure,* http://www.datasoft.com/wp-content/uploads/2010/07/Typhoon-brochure.pdf

[20]. *Lyrtech's SFF SDR Development Platform Product Sheet*, http://www.ceanet.com.au/Portals/0/Reference%20sheet%20-%20SFF%20SDR%20DP%20-%20200907%20%28lo_res%29.pdf

[21]. *CORAL Network Platform Specifications, http://www.crc.gc.ca/files/crc/home/wifi_cr/coral_brochure_en.pdf*

[22]. *USRP N200 Series Datasheet,* http://www.ettus.com/downloads/ettus_ds_usrp_n200series_v3.pdf

[23]. *Discuss-gnuradio mailing list archives,* http://lists.gnu.org/archive/html/discuss-gnuradio/

8 Appendix A - Code: original UCLA Zigbee PHY files

Code listing A1: *cc2420_txtest.py*

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

#
# Transmitter of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Sanna Leidelof
#

from gnuradio import gr, eng_notation
from gnuradio import usrp
from gnuradio import ucla
from gnuradio.ucla_blks import ieee802_15_4_pkt
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import math, struct, time

def pick_subdevice(u):
    """
    The user didn't specify a subdevice on the command line.
    If there's a daughterboard on A, select A.
    If there's a daughterboard on B, select B.
    Otherwise, select A.
    """
    if u.db[0][0].dbid() >= 0:       # dbid is < 0 if there's no d'board
or a problem
        return (0, 0)
    if u.db[1][0].dbid() >= 0:
        return (1, 0)
    return (0, 0)

class transmit_path(gr.top_block):
    def __init__(self, options):
        gr.top_block.__init__(self)
        self.normal_gain = 8000

        self.u = usrp.sink_c()
        dac_rate = self.u.dac_rate();
        self._data_rate = 2000000
        self._spb = 2
        self._interp = int(128e6 / self._spb / self._data_rate)
        self.fs = 128e6 / self._interp


        self.u.set_interp_rate(self._interp)

        # determine the daughterboard subdevice we're using
```

Code listing A1: *cc2420_txtest.py* contd.

```
        self.u.set_interp_rate(self._interp)

        # determine the daughterboard subdevice we're using
        if options.tx_subdev_spec is None:
            options.tx_subdev_spec = usrp.pick_tx_subdevice(self.u)
        self.u.set_mux(usrp.determine_tx_mux_value(self.u,
options.tx_subdev_spec))
        self.subdev = usrp.selected_subdev(self.u,
options.tx_subdev_spec)
        print "Using TX d'board %s" % (self.subdev.side_and_name(),)

        self.u.tune(0, self.subdev, options.cordic_freq)
        self.u.set_pga(0, options.gain)
        self.u.set_pga(1, options.gain)

        # transmitter
        self.packet_transmitter =
ieee802_15_4_pkt.ieee802_15_4_mod_pkts(self, spb=self._spb,
msgq_limit=2)
        self.gain = gr.multiply_const_cc (self.normal_gain)

        self.connect(self.packet_transmitter, self.gain, self.u)

        #self.filesink = gr.file_sink(gr.sizeof_gr_complex,
'rx_test.dat')
        #self.connect(self.gain, self.filesink)

        self.set_gain(self.subdev.gain_range()[1])  # set max Tx gain
        self.set_auto_tr(True)                      # enable Auto
Transmit/Receive switching

    def set_gain(self, gain):
        self.gain = gain
        self.subdev.set_gain(gain)

    def set_auto_tr(self, enable):
        return self.subdev.set_auto_tr(enable)

    def send_pkt(self, payload='', eof=False):
        return self.packet_transmitter.send_pkt(0xe5,
struct.pack("HHHH", 0xFFFF, 0xFFFF, 0x10, 0x10), payload, eof)

def main ():


    parser = OptionParser (option_class=eng_option)
    parser.add_option("-R", "--rx-subdev-spec", type="subdev",
default=None,
                      help="select USRP Rx side A or B (default=first
one with a daughterboard)")
    parser.add_option("-T", "--tx-subdev-spec", type="subdev",
default=None,
                      help="select USRP Tx side A or B (default=first
one with a daughterboard)")
    parser.add_option ("-c", "--cordic-freq", type="eng_float",
default=2415000000,
```

Code listing A1: *cc2420_txtest.py* contd.

```
                        help="set Tx cordic frequency to FREQ",
                        metavar="FREQ")

    parser.add_option ("-r", "--data-rate", type="eng_float",
                        default=2000000)

    parser.add_option ("-f", "--filename", type="string",
                        default="rx.dat", help="write data to FILENAME")
    parser.add_option ("-g", "--gain", type="eng_float", default=0,
                        help="set Rx PGA gain in dB [0,20]")
    parser.add_option ("-N", "--no-gui", action="store_true",
default=False)

    (options, args) = parser.parse_args ()

    tb = transmit_path(options)
    tb.start()

    for i in range(10):
        print "send message %d:"%(i+1,)
        tb.send_pkt(struct.pack('9B', 0x1, 0x80, 0x80, 0xff, 0xff, 0x10,
0x0, 0x20, 0x0))
        #this is an other example packet we could send.
        #tb.send_pkt(struct.pack('BBBBBBBBBBBBBBBBBBBBBBBBB', 0x1,
0x8d, 0x8d, 0xff, 0xff, 0xbd, 0x0, 0x22, 0x12, 0xbd, 0x0, 0x1, 0x0,
0xff, 0xff, 0x8e, 0xff, 0xff, 0x0, 0x3, 0x3, 0xbd, 0x0, 0x1, 0x0, 0x0,
0x0))
        time.sleep(1)

    tb.wait()

if __name__ == '__main__':
    # insert this in your test code...
    #import os
    #print 'Blocked waiting for GDB attach (pid = %d)' % (os.getpid(),)
    #raw_input ('Press Enter to continue: ')

    main ()
```

Code listing A2: *cc2420_rxtest.py*

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

#
# Decoder of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Leslie Choong, Mikhail Tadjikov
#

from gnuradio import gr, eng_notation
from gnuradio import usrp
from gnuradio.ucla_blks import ieee802_15_4_pkt
```

Code listing A2: *cc2420_rxtest.py* contd.

```
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import struct, sys

def pick_subdevice(u):
    """
    The user didn't specify a subdevice on the command line.
    If there's a daughterboard on A, select A.
    If there's a daughterboard on B, select B.
    Otherwise, select A.
    """
    if u.db(0, 0).dbid() >= 0:       # dbid is < 0 if there's no d'board
or a problem
        return (0, 0)
    if u.db(1, 0).dbid() >= 0:
        return (1, 0)
    return (0, 0)

class stats(object):
    def __init__(self):
        self.npkts = 0
        self.nright = 0


class oqpsk_rx_graph (gr.top_block):
    def __init__(self, options, rx_callback):
        gr.top_block.__init__(self)
        print "cordic_freq = %s" % (eng_notation.num_to_str
(options.cordic_freq))


        # ---------------------------------------------------------------
--

        self.data_rate = options.data_rate
        self.samples_per_symbol = 2
        self.usrp_decim = int (64e6 / self.samples_per_symbol /
self.data_rate)
        self.fs = self.data_rate * self.samples_per_symbol
        payload_size = 128              # bytes

        print "data_rate = ", eng_notation.num_to_str(self.data_rate)
        print "samples_per_symbol = ", self.samples_per_symbol
        print "usrp_decim = ", self.usrp_decim
        print "fs = ", eng_notation.num_to_str(self.fs)

        u = usrp.source_c (0, self.usrp_decim)
        if options.rx_subdev_spec is None:
            options.rx_subdev_spec = pick_subdevice(u)
        u.set_mux(usrp.determine_rx_mux_value(u,
options.rx_subdev_spec))

        subdev = usrp.selected_subdev(u, options.rx_subdev_spec)
        print "Using RX d'board %s" % (subdev.side_and_name(),)

        u.tune(0, subdev, options.cordic_freq)
```

Code listing A2: *cc2420_rxtest.py* contd.

```
        u.set_pga(0, options.gain)
        u.set_pga(1, options.gain)

        self.u = u

        self.packet_receiver =
ieee802_15_4_pkt.ieee802_15_4_demod_pkts(self,

callback=rx_callback,

sps=self.samples_per_symbol,

symbol_rate=self.data_rate,

threshold=-1)

        self.squelch = gr.pwr_squelch_cc(50, 1, 0, True)
        self.connect(self.u, self.squelch, self.packet_receiver)

def main ():

    def rx_callback(ok, payload):
        st.npkts += 1
        if ok:
            st.nright += 1

        (pktno,) = struct.unpack('!H', payload[0:2])
        print "ok = %5r  pktno = %4d  len(payload) = %4d  %d/%d" % (ok,
pktno, len(payload),

st.nright, st.npkts)
        print "  payload: " + str(map(hex, map(ord, payload)))
        print " ------------------------"
        sys.stdout.flush()


    parser = OptionParser (option_class=eng_option)
    parser.add_option("-R", "--rx-subdev-spec", type="subdev",
default=None,
                      help="select USRP Rx side A or B (default=first
one with a daughterboard)")
    parser.add_option ("-c", "--cordic-freq", type="eng_float",
default=2475000000,
                       help="set rx cordic frequency to FREQ",
metavar="FREQ")
    parser.add_option ("-r", "--data-rate", type="eng_float",
default=2000000)
    parser.add_option ("-f", "--filename", type="string",
                       default="rx.dat", help="write data to FILENAME")
    parser.add_option ("-g", "--gain", type="eng_float", default=0,
                       help="set Rx PGA gain in dB [0,20]")

    (options, args) = parser.parse_args ()

    st = stats()
```

Code listing A2: *cc2420_rxtest.py* contd.

```
    tb = oqpsk_rx_graph(options, rx_callback)
    tb.start()

    tb.wait()

if __name__ == '__main__':
    # insert this in your test code...
    #import os
    #print 'Blocked waiting for GDB attach (pid = %d)' % (os.getpid(),)
    #raw_input ('Press Enter to continue: ')

    main ()
```

Code listing A3: *ieee802_15_4.py*

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

# O-QPSK modulation and demodulation.

# Derived from gmsk.py
#
# Modified by: Thomas Schmid, Leslie Choong, Sanna Leidelof
#

from gnuradio import gr, ucla
from math import pi

class ieee802_15_4_mod(gr.hier_block2):

    def __init__(self, *args, **kwargs):
        """
      Hierarchical block for cc1k FSK modulation.

      The input is a byte stream (unsigned char) and the
      output is the complex modulated signal at baseband.

      @param spb: samples per baud >= 2
      @type spb: integer
      """
      try:
            self.spb = kwargs.pop('spb')
      except KeyError:
            pass

      gr.hier_block2.__init__(self, "ieee802_15_4_mod",
                        gr.io_signature(1, 1, 1),  # Input
                        gr.io_signature(1, 1, gr.sizeof_gr_complex))
# Output

        if not isinstance(self.spb, int) or self.spb < 2:
            raise TypeError, "spb must be an integer >= 2"
```

Code listing A3: *ieee802_15_4.py* contd.

```
        self.symbolsToChips = ucla.symbols_to_chips_bi()
        self.chipsToSymbols = gr.packed_to_unpacked_ii(2,
gr.GR_MSB_FIRST)
        self.symbolsToConstellation = gr.chunks_to_symbols_ic((-1-1j, -
1+1j, 1-1j, 1+1j))

        self.pskmod = ucla.qpsk_modulator_cc()
        self.delay = ucla.delay_cc(self.spb)


      # Connect
      self.connect(self, self.symbolsToChips, self.chipsToSymbols,
                   self.symbolsToConstellation, self.pskmod, self.delay,
self)

class ieee802_15_4_demod(gr.hier_block2):
    def __init__(self, *args, **kwargs):
        """
        Hierarchical block for O-QPSK demodulation.

        The input is the complex modulated signal at baseband
        and the output is a stream of bytes.

        @param sps: samples per symbol
        @type sps: integer
        """
      try:
            self.sps = kwargs.pop('sps')
      except KeyError:
            pass

      gr.hier_block2.__init__(self, "ieee802_15_4_demod",
                           gr.io_signature(1, 1, gr.sizeof_gr_complex),
# Input
                           gr.io_signature(1, 1, gr.sizeof_float))  #
Output

        # Demodulate FM
        sensitivity = (pi / 2) / self.sps
        #self.fmdemod = gr.quadrature_demod_cf(1.0 / sensitivity)
        self.fmdemod = gr.quadrature_demod_cf(1)

        # Low pass the output of fmdemod to allow us to remove
        # the DC offset resulting from frequency offset

        alpha = 0.0008/self.sps
        self.freq_offset = gr.single_pole_iir_filter_ff(alpha)
        self.sub = gr.sub_ff()
        self.connect(self, self.fmdemod)
        self.connect(self.fmdemod, (self.sub, 0))
        self.connect(self.fmdemod, self.freq_offset, (self.sub, 1))


        # recover the clock
        omega = self.sps
        gain_mu=0.03
```

Code listing A3: *ieee802_15_4.py* contd

```
        mu=0.5
        omega_relative_limit=0.0002
        freq_error=0.0

        gain_omega = .25*gain_mu*gain_mu        # critically damped
        self.clock_recovery = gr.clock_recovery_mm_ff(omega, gain_omega,
mu, gain_mu,

omega_relative_limit)

        # Connect
        self.connect(self.sub, self.clock_recovery, self)
```

Code listing A4: *ieee802_15_4_pkt.py*

```
# This is derived from gmsk2_pkt.py.
#
# Modified by: Thomas Schmid, Leslie Choong, Sanna Leidelof
#

import Numeric

from gnuradio import gr, packet_utils, gru
from gnuradio import ucla
import crc16
import gnuradio.gr.gr_threading as _threading
import ieee802_15_4
import struct

MAX_PKT_SIZE = 128

def make_ieee802_15_4_packet(FCF, seqNr, addressInfo, payload,
pad_for_usrp=True, preambleLength=4, SFD=0xA7):
    """
    Build a 802_15_4 packet

    @param FCF: 2 bytes defining the type of frame.
    @type FCF: string
    @param seqNr: 1 byte sequence number.
    @type seqNr: byte
    @param addressInfo: 0 to 20 bytes of address information.
    @type addressInfo: string
    @param payload: The payload of the packet. The maximal size of the
message
    can not be larger than 128.
    @type payload: string
    @param pad_for_usrp: If we should add 0s at the end to pad for the
USRP.
    @type pad_for_usrp: boolean
    @param preambleLength: Length of the preambble. Currently ignored.
    @type preambleLength: int
```

Code listing A4: *ieee802_15_4_pkt.py* contd.

```
    @param SFD: Start of frame describtor. This is by default set to the
IEEE 802.15.4 standard,
    but can be changed if required.
    @type SFD: byte
    """

    if len(FCF) != 2:
        raise ValueError, "len(FCF) must be equal to 2"
    if seqNr > 255:
        raise ValueError, "seqNr must be smaller than 255"
    if len(addressInfo) > 20:
        raise ValueError, "len(addressInfo) must be in [0, 20]"

    if len(payload) > MAX_PKT_SIZE - 5 - len(addressInfo):
        raise ValueError, "len(payload) must be in [0, %d]"
%(MAX_PKT_SIZE)

    SHR = struct.pack("BBBBB", 0, 0, 0, 0, SFD)
    PHR = struct.pack("B", 3 + len(addressInfo) + len(payload) + 2)
    MPDU = FCF + struct.pack("B", seqNr) + addressInfo + payload
    crc = crc16.CRC16()
    crc.update(MPDU)

    FCS = struct.pack("H", crc.intchecksum())

    pkt = ''.join((SHR, PHR, MPDU, FCS))

    if pad_for_usrp:
        # note that we have 16 samples which go over the USB for each
bit
        pkt = pkt + (_npadding_bytes(len(pkt), 8) * '\x00')+0*'\x00'

    return pkt

def _npadding_bytes(pkt_byte_len, spb):
    """
    Generate sufficient padding such that each packet ultimately ends
    up being a multiple of 512 bytes when sent across the USB.  We
    send 4-byte samples across the USB (16-bit I and 16-bit Q), thus
    we want to pad so that after modulation the resulting packet
    is a multiple of 128 samples.

    @param ptk_byte_len: len in bytes of packet, not including padding.
    @param spb: samples per baud == samples per bit (1 bit / baud with
GMSK)
    @type spb: int

    @returns number of bytes of padding to append.
    """
    modulus = 128
    byte_modulus = gru.lcm(modulus/8, spb) / spb
    r = pkt_byte_len % byte_modulus
    if r == 0:
        return 0
    return byte_modulus - r
```

Code listing A4: *ieee802_15_4_pkt.py* contd.

```
def make_FCF(frameType=1, securityEnabled=0, framePending=0, acknowledgeRequest=0,
intraPAN=0, destinationAddressingMode=0, sourceAddressingMode=0):
    """
    Build the FCF for the 802_15_4 packet

    """
    if frameType >= 2**3:
        raise ValueError, "frametype must be < 8"
    if securityEnabled >= 2**1:
        raise ValueError, " must be < "
    if framePending >= 2**1:
        raise ValueError, " must be < "
    if acknowledgeRequest >= 2**1:
        raise ValueError, " must be < "
    if intraPAN >= 2**1:
        raise ValueError, " must be < "
    if destinationAddressingMode >= 2**2:
        raise ValueError, " must be < "
    if sourceAddressingMode >= 2**2:
        raise ValueError, " must be < "



    return struct.pack("H", frameType
                       + (securityEnabled << 3)
                       + (framePending << 4)
                       + (acknowledgeRequest << 5)
                       + (intraPAN << 6)
                       + (destinationAddressingMode << 10)
                       + (sourceAddressingMode << 14))


class ieee802_15_4_mod_pkts(gr.hier_block2):
    """
    IEEE 802.15.4 modulator that is a GNU Radio source.

    Send packets by calling send_pkt
    """
    def __init__(self, pad_for_usrp=True, *args, **kwargs):
        """
            Hierarchical block for the 802_15_4 O-QPSK  modulation.

        Packets to be sent are enqueued by calling send_pkt.
        The output is the complex modulated signal at baseband.

        @param msgq_limit: maximum number of messages in message queue
        @type msgq_limit: int
        @param pad_for_usrp: If true, packets are padded such that they end up a multiple of 128
samples

          See 802_15_4_mod for remaining parameters
          """
        try:
              self.msgq_limit = kwargs.pop('msgq_limit')
```

Code listing A4: *ieee802_15_4_pkt.py* contd.

```
       except KeyError:
              pass

       gr.hier_block2.__init__(self, "ieee802_15_4_mod_pkts",
                           gr.io_signature(0, 0, 0),  # Input
                           gr.io_signature(1, 1, gr.sizeof_gr_complex))
# Output
        self.pad_for_usrp = pad_for_usrp

        # accepts messages from the outside world
        self.pkt_input = gr.message_source(gr.sizeof_char,
self.msgq_limit)
        self.ieee802_15_4_mod = ieee802_15_4.ieee802_15_4_mod(self,
*args, **kwargs)
        self.connect(self.pkt_input, self.ieee802_15_4_mod, self)

    def send_pkt(self, seqNr, addressInfo, payload='', eof=False):
        """
        Send the payload.

        @param seqNr: sequence number of packet
        @type seqNr: byte
        @param addressInfo: address information for packet
        @type addressInfo: string
        @param payload: data to send
        @type payload: string
        """

        if eof:
            msg = gr.message(1) # tell self.pkt_input we're not sending
any more packets
        else:
            FCF = make_FCF()

            pkt = make_ieee802_15_4_packet(FCF,
                                           seqNr,
                                           addressInfo,
                                           payload,
                                           self.pad_for_usrp)
             #print "pkt =", packet_utils.string_to_hex_list(pkt),
len(pkt)
            msg = gr.message_from_string(pkt)
        self.pkt_input.msgq().insert_tail(msg)


class ieee802_15_4_demod_pkts(gr.hier_block2):
    """
    802_15_4 demodulator that is a GNU Radio sink.

    The input is complex baseband.  When packets are demodulated, they
are passed to the
    app via the callback.
    """

    def __init__(self, *args, **kwargs):
```

Code listing A4: *ieee802_15_4_pkt.py* contd.

```
        """
      Hierarchical block for O-QPSK demodulation.

      The input is the complex modulated signal at baseband.
        Demodulated packets are sent to the handler.

        @param callback:  function of two args: ok, payload
        @type callback: ok: bool; payload: string
        @param threshold: detect access_code with up to threshold bits
wrong (-1 -> use default)
        @type threshold: int

        See ieee802_15_4_demod for remaining parameters.
      """
      try:
            self.callback = kwargs.pop('callback')
            self.threshold = kwargs.pop('threshold')
      except KeyError:
            pass

      gr.hier_block2.__init__(self, "ieee802_15_4_demod_pkts",
                        gr.io_signature(1, 1, gr.sizeof_gr_complex),
# Input
                        gr.io_signature(0, 0, 0))  # Output

        self._rcvd_pktq = gr.msg_queue()          # holds packets from
the PHY
        self.ieee802_15_4_demod = ieee802_15_4.ieee802_15_4_demod(self,
*args, **kwargs)
        self._packet_sink =
ucla.ieee802_15_4_packet_sink(self._rcvd_pktq, self.threshold)

        self.connect(self,self.ieee802_15_4_demod, self._packet_sink)

        self._watcher = _queue_watcher_thread(self._rcvd_pktq,
self.callback)

    def carrier_sensed(self):
        """
        Return True if we detect carrier.
        """
        return self._packet_sink.carrier_sensed()


class _queue_watcher_thread(_threading.Thread):
    def __init__(self, rcvd_pktq, callback):
        _threading.Thread.__init__(self)
        self.setDaemon(1)
        self.rcvd_pktq = rcvd_pktq
        self.callback = callback
        self.keep_running = True
        self.start()

    def run(self):
        while self.keep_running:
            print "802_15_4_pkt: waiting for packet"
```

Code listing A4: *ieee802_15_4_pkt.py* contd.

```
            msg = self.rcvd_pktq.delete_head()
            ok = 0
            payload = msg.to_string()

            print "received packet "

            if len(payload) > 2:
                crc = crc16.CRC16()
                crc.update(payload[:-2])

                crc_check = crc.intchecksum()
                print "checksum: %s, received: %s"%(crc_check,
str(ord(payload[-2]) + ord(payload[-1])*256))

                ok = (crc_check == ord(payload[-2]) + ord(payload[-
1])*256)
                msg_payload = payload

                if self.callback:
                    self.callback(ok, msg_payload)
```

9 Appendix B - Code: modified TX and RX UCLA files

Code listing B1: *uhd_cc2420_tx.py*

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

#
# Transmitter of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Sanna Leidelof, Kresimir Dabcevic
#

from gnuradio import gr, eng_notation
from gnuradio import uhd
from gnuradio import ucla
from gnuradio.ucla_blks import ieee802_15_4_pkt
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import math, struct, time


class transmit_path(gr.top_block):
    def __init__(self, options):
        gr.top_block.__init__(self)
        self.normal_gain = 8000

        self.u = uhd.usrp_sink(device_addr=options.address,
                               io_type=uhd.io_type.COMPLEX_FLOAT32,
                               num_channels=1)
        self.u.set_clock_config(uhd.clock_config.internal(),
uhd.ALL_MBOARDS)
        u = self.u

        self._data_rate = options.data_rate
        self._spb = 2

        # Set and print sampling rate
        self.u.set_samp_rate(options.sample_rate)
        input_rate = self.u.get_samp_rate()
        print "Sampling rate: %d" %(input_rate)

        # Set and print center frequency
        self.u.set_center_freq(options.cordic_freq)
        frekva = self.u.get_center_freq()
        self.u.set_center_freq(frekva)
        print "Center frequency: %d" %(frekva)

        # transmitter

        self.packet_transmitter =
ieee802_15_4_pkt.ieee802_15_4_mod_pkts(self, spb=self._spb,
msgq_limit=2)
```

Code listing B1: *uhd_cc2420_tx.py* contd.

```
        self.gain = gr.multiply_const_cc (self.normal_gain)

        self.connect(self.packet_transmitter, self.gain, self.u)

        self.filesink = gr.file_sink(gr.sizeof_gr_complex,
'tx_test.txt')
        self.connect(self.gain, self.filesink)


    def set_gain(self, gain):
        self.gain = gain
        self.u.set_gain(gain, 0)

    def send_pkt(self, payload='', eof=False):
        return self.packet_transmitter.send_pkt(0xe5,
struct.pack("HHHH", 0xFFFF, 0xFFFF, 0x10, 0x10), payload, eof)

def main ():

    parser = OptionParser (option_class=eng_option)

    parser.add_option("-a", "--address", type="string",
default="addr=192.168.10.2",
                      help="Address of UHD device, [default=%default]")
    parser.add_option ("-c", "--cordic_freq", type="eng_float",
default=2475000000,
                       help="set Tx cordic frequency to FREQ",
metavar="FREQ")
    parser.add_option ("-r", "--data_rate", type="eng_float",
default=2000000)
    parser.add_option ("-s", "--sample_rate", type="eng_float",
default=1000000)
    parser.add_option ("-f", "--filename", type="string",
                       default="rx.dat", help="write data to FILENAME")

    parser.add_option ("-N", "--no-gui", action="store_true",
default=False)

    (options, args) = parser.parse_args ()
    i = 0
    j = 0
    tb = transmit_path(options)
    tb.start()

    #generating and transmitting "important" packets
    while j<=255:
      while i<=255*(j+1):
           print "send message %d:"%(i)

                tb.send_pkt(payload=struct.pack('2B', j, i-j*255))
           i=i+1
           time.sleep(0.01)
      j = j+1
```

Code listing B1: *uhd_cc2420_tx.py* contd.

```
    #generating and transmitting "EOT" packets
    for z in range(100):
        time.sleep(1)
        tb.send_pkt(payload=struct.pack('5B', 0x9, 0x9, 0x9, 0x9, 0x9))
        time.sleep(1)
    tb.stop()

if __name__ == '__main__':
    # insert this in your test code...
    #import os
    #print 'Blocked waiting for GDB attach (pid = %d)' % (os.getpid(),)
    #raw_input ('Press Enter to continue: ')

    main ()
```

Code listing B2: *uhd_cc2420_rx.py*

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

#
# Decoder of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Leslie Choong, Mikhail Tadjikov, Kresimir
Dabcevic
#

from gnuradio import gr, eng_notation
from gnuradio import uhd
from gnuradio.ucla_blks import ieee802_15_4_pkt
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import struct, sys, time, math

n2s = eng_notation.num_to_str


class stats(object):
    def __init__(self):
        self.npkts = 0
        self.nright = 0


class oqpsk_rx_graph (gr.top_block):
    def __init__(self, options, rx_callback):
        gr.top_block.__init__(self)
        print "cordic_freq = %s" % (eng_notation.num_to_str
(options.cordic_freq))


        # ---------------------------------------------------

        self.data_rate = options.data_rate
        self.samples_per_symbol = 2
```

Code listing B2: *uhd_cc2420_rx.py* contd.

```
        self.fs = self.data_rate * self.samples_per_symbol
        payload_size = 128              # bytes


        print "data_rate = ", eng_notation.num_to_str(self.data_rate)
        print "samples_per_symbol = ", self.samples_per_symbol

        self.u = uhd.usrp_source (device_addr=options.address,
                              io_type=uhd.io_type.COMPLEX_FLOAT32,
                              num_channels=1)

        self.u.set_clock_config(uhd.clock_config.internal(),
uhd.ALL_MBOARDS)

        #Getting the RSSI value via the analog sensor would be
implemented in a following way, but outputs highly unprecise results
        #rssi = self.u.get_dboard_sensor("rssi")
        #print " %s" %(rssi)


        # Set the antenna
        self.u.set_antenna(options.antenna, 0)

        # Set sampling rate
        self.u.set_samp_rate(options.sample_rate)
        input_rate = self.u.get_samp_rate()
        print "USRP sampling rate: %d" %(input_rate)

        # Set and the read center frequency
        self.u.set_center_freq(uhd.tune_request(options.cordic_freq,0))
        frekva = self.u.get_center_freq()
        print "Center frequency: %d " %(frekva)

        self.filesink = gr.file_sink(4, 'rx_test.dat')


        self.u.set_gain(options.gain)

        self.packet_receiver =
ieee802_15_4_pkt.ieee802_15_4_demod_pkts(self,

callback=rx_callback,

sps=self.samples_per_symbol,

symbol_rate=self.data_rate,

threshold=-1)


        self.squelch = gr.pwr_squelch_cc(-55, 1, 0, True)
        self.connect(self.u, self.squelch, self.packet_receiver)
```

Code listing B2: ***uhd_cc2420_rx.py*** **contd.**

```
        self.squelch2 = gr.pwr_squelch_cc(-35, 1, 0, True)
        self.complex_to_mag2 = gr.complex_to_mag_squared(1)
        self.iir_filter = gr.single_pole_iir_filter_ff(0.5,1)
        self.decimator = gr.keep_one_in_n(4,360)
        self.logarithmer = gr.nlog10_ff(10,1,4)
   #     time.sleep(3)
        self.connect(self.u, self.squelch2, self.complex_to_mag2,
self.iir_filter, self.decimator, self.logarithmer, self.filesink)


def main ():

    def rx_callback(ok, payload):

        print "RSSI: %s" %(rssix)
        st.npkts += 1
        if ok:
            st.nright += 1
            (pktno,) = struct.unpack('!H', payload[0:2])
            print "ok = %5r  pktno = %4d  len(payload) = %4d  %d/%d" %
(ok, pktno, len(payload),

st.nright, st.npkts)

            print "  payload: " + str(map(hex, map(ord, payload)))
            print "Payload 11-12: %s, 12-13: %s" %(str(map(hex, map(ord,
payload[11:12]))),str(map(hex, map(ord, payload[12:13]))))
            received_packets.append(str(map(hex, map(ord,
payload[11:13]))))


            print " ------------------------"

            if len(payload)==18:
                print "All done"
                print "Statistics: good %d received %d"%(st.nright,
st.npkts)
                print "Bad packets: %s" %(bad_packets)
                print "Received packets: %s" %(received_packets)
                m=0
                n=0
                while n<=9:
                  while m<=255:
                    print "Checking if packet %s %s was
received"%(str(hex(n)),str(hex(m)))
                    if (str(hex(n))+", "+str(hex(m))) in
received_packets:
                       print "Packet nr. %d has been received and
decoded" %(n*10 + m)
                    else:
                       print "Packet nr. %d has not been received or
decoded" %(n*10 + m)
                       nonreceived_packets.append(str(n*10 + m))
                    m=m+1
                  m=0
                  n=n+1
```

Code listing B2: *uhd_cc2420_rx.py* contd.

```
                print "Nonreceived packets: %s" %(nonreceived_packets)


                tb.stop()

             sys.stdout.flush()
       else:
           print "  Bad packet. %d/%d"%(st.nright, st.npkts)
            bad_packets.append(st.npkts)
           pass


    parser = OptionParser (option_class=eng_option)

    parser.add_option("-a", "--address", type="string",
default="addr=192.168.10.2",
                       help="Address of UHD device, [default=%default]")
    parser.add_option("-A", "--antenna", type="string", default="RX2",
                      help="select Rx Antenna where appropriate")
    parser.add_option ("-c", "--cordic_freq", type="eng_float",
default=2475000000,
                       help="set rx cordic frequency to FREQ",
metavar="FREQ")
    parser.add_option ("-r", "--data_rate", type="eng_float",
default=2000000)
    parser.add_option ("-s", "--sample_rate", type="eng_float",
default=1000000)
    parser.add_option ("-f", "--filename", type="string",
                       default="rx.dat", help="write data to FILENAME")
    parser.add_option ("-g", "--gain", type="eng_float", default=0,
                       help="set Rx PGA gain in dB [0,20]")


    (options, args) = parser.parse_args ()

    st = stats()
    global tb
    tb = oqpsk_rx_graph(options, rx_callback)

    global bad_packets
    bad_packets = []

    global received_packets
    received_packets = []

    global sent_packets
    sent_packets = []

    global nonreceived_packets
    nonreceived_packets = []

    n=0
    while n<=255:
       sent_packets.append("['"+str(hex(n))+"']")
       n=n+1
    print "Sent packets: %s" %(sent_packets)
```

Code listing B2: *uhd_cc2420_rx.py* contd.

```
    tb.start()
    time.sleep(1)
    tb.wait()

if __name__ == '__main__':
     #insert this in your test code...
    #import os
   # print 'Blocked waiting for GDB attach (pid = %d)' % (os.getpid(),)
   # raw_input ('Press Enter to continue: ')

    main ()
```