IEEE 802.15.4 Implementation on an Embedded Device

Rithirong Thandee

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Carl B. Dietrich, Chair
Jeffrey H.Reed, Co-Chair
S.M. Shajedul Hasan

April 10th, 2012
Blacksburg, Virginia

Keywords: Software Defined Radio, IEEE 802.15.4, USRP E100, UCLA ZigBee PHY

IEEE 802.15.4 Implementation on an Embedded Device

Rithirong Thandee

Abstract

Software Defined Radio (SDR) is a growing technology that allows radio communication to become interoperable. SDR can lower the cost for a particular hardware radio to communicate with another radio that uses a different standard. In order to show the capability of SDR, this thesis shows how to implement IEEE 802.14.5, a low-rate wireless personal area network (LR-WPAN) standard, on a standalone embedded machine.

The implementation is done using a universal software radio peripheral embedded, **USRP E100**, an open source software development toolkit for SDR, **GNU Radio**, and UCLA ZigBee PHY GNU Radio application. The implementation can be done on the regular non-embedded USRPs. However, without a fast host computer demodulating the packets, the USRP E100 cannot receive incoming packets. An available FPGA is used to solve this problem by doing a software-hardware hybrid design to allow the USRP E100 to communicate with other IEEE 802.15.4 devices. The final product is an IEEE 802.15.4 monitor software that detects messages from devices communicating using IEEE 802.15.4 in its range. In addition, recommendations are presented for improving SDR education and training, particularly for developers with backgrounds in disciplines other than communications engineering.

Acknowledgements

To my family and friends.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ADC	Analog-to-Digital Converter
CCA	Clear Channel Assessment
CRC	Cyclic Redundancy Check
CSMA-CA	Carrier Sense Multiple Access with Collision Avoidance
DAC	Digital-to-Analog Converter
DSP	Digital Signal Processor
ED	Energy Detection
FCS	Frame Correction Sequence
FFD	Full-Function Device
FFTW	Fastest Fourier Transform in the West (software library)
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPIO	General Purpose Input/Output
GPP	General Purpose Processor
GPU	Graphic Processing Unit
GTS	Guaranteed Time Slot
I and Q	In-phase and Quadrature
ISE	Xilinx Synthesis and Analysis Software

JTRS	Joint Tactical Radio System
LLC	Logical Link Control
LQI	Link Quality Indication
LR-WPAN	Low-Rate WPAN
LSB	Least Significant Bit
MAC	Medium Access Control
MHR	MAC Header
MPDU	MAC Protocol Data Unit
MPRG	Mobile Portable Research Group at Virginia Tech
MSB	Most Significant Bit
MSK	Minimum-shift Keying
O-QPSK	Offset Quadrature Phase-shift Keying
OS	Operating System
PAN	Personal Area Network
PHR	PHY Header
PHY	Physical Layer
PPDU	PHY Protocol Data Units
PSDU	PHY Service Data Unit
PSSS	Parallel Sequence Spread Spectrum
RFD	Reduced-Function Device
SCA	Software Communications Architecture
SDR	Software Defined Radio
SFD	Start Frame Delimiter
SHR	Synchronization Header
SPI	Serial Peripheral Interface

SSCS	Service Specific Convergence Sublayer
UART	Universal Asynchronous Receiver/Transmitter
UCLA	University of California, Los Angeles
UHD	Universal Hardware Driver
USRP	Universal Software Radio Peripheral
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network

Chapter 1

Introduction

Wireless communication started with hardware radios that were only able to communicate in a single protocol and needed hardware changes to communicate with other different radio standards. The goal of Software Defined Radio (SDR) is to implement most radio functions in software and allow radios to become more flexible; new standards can be implemented by applying software updates to the radio hardware. SDR is a growing, interdisciplinary field of technology that engages many researchers and developers. With more powerful computer hardware and software, SDR endlessly continues to grow, and its continuing development requires contributions of computer engineers as well as software developers and communications engineers.

The goal of this thesis is to explore the capability of SDR by implementing a wireless protocol standard, IEEE 802.15.4 on an embedded standalone device that is available. With an ability to run by itself, affordable price, and GNU Radio support, a Universal Software Radio Peripheral Embedded (USRP E100) was chosen as a target device. GNU Radio was chosen for use with the USRP.

1.1 Motivation

Implementation of a wireless protocol standard into a standalone device is selected as a topic to test currently available SDR open source software tools and low-cost hardware platforms. Some prior research[26, 24] uses IEEE 802.15.4 as a standard, but none was found that addressed implementing it on an embedded device. Later in this thesis, a monitor software is created as a useful application that would demonstrate communication between IEEE 802.15.4 devices in range.

This thesis has contributed to the SDR field by showing that a computer engineer with

little digital communication background can use SDR tools such as GNU Radio and USRPs to implement SDR applications. The work entailed studying a wireless protocol standard, modifying an existing SDR application to work with newer devices, combining the application and a new hardware design to create a complete SDR radio with the capability to monitor other devices in the network, enabling the device to communicate with other commercial devices available in the market, and creating a tutorial for other users on how to accomplish the same goal.

1.2 Thesis Organization

The thesis is organized into three major parts. Chapter 2 describes background information of all related components. The chapter discusses the history and definition of SDR along with the ideal architecture of SDR, introduces GNU Radio and how to create an application and a GNU Radio block. It also describes the USRP as well as IEEE 802.15.4 and its PHY layer. Chapter 3, the implementation process, talks about UCLA ZigBee PHY, which is the GNU Radio application that will be used as a base to create a complete system. This chapter describes the process of setting up the different devices and the code modification, what code modifications have been made, how the monitor program is made, and how the USRPs communicate with a commercial IEEE 802.15.4 device. Chapter 4 shows the final product and the performance measuring process along with results. Chapter 5 sums up the whole thesis and points out problems encountered and possible future enhancements of the work.

Chapter 2

Background

2.1 Software Defined Radio

2.1.1 Definition and History

A radio is a device that communicates wirelessly by modulation of electromagnetic waves. Typically, radios are implemented in hardware with component such as mixers, filters, amplifiers, modulators/demodulators, and detectors. Hardware radio devices work with a limited functionality and can only be modified though physical intervention. Modifying hardware radios can be costly and the functionality will still be limited to the particular hardware configuration. Software defined radio uses software instead of hardware components with digital signal processor (DSP) or field programmable gate array (FPGA) as reconfigurable hardware and general purpose processor (GPP) as the processor. SDR can be reconfigured a software update. Currently, DSPs and FPGAs are becoming more affordable and capable. Therefore, SDR is a preferable choice in creating a wireless system instead of a traditional hardware radio [20, 17].

The idea of software defined radio (SDR) started in the mid 1980s. SpeakEasy, one of the first major SDR platform was create by the Hazeltine and Motorola. SpeakEasy was used by the military to communicate between different standards within the 2MHz to 2GHz range. The main goal was to "provide the interoperability between the different air interface standards of different branches of the armed forces [25]."

2.1.2 SDR Architecture

In an ideal situation, SDR should at least have an analog stage [23]. What this means is an ideal SDR would only use an antenna, an analog-to-digital converter (ADC) to convert

analog signals to digital signals, a digital-to-analog converter (DAC) to convert digital signals to analog signals and a GPP to process the signals. However, the current SDR uses: antenna, RF front-end, which consists of filters, amplifiers, mixers, and ADC/DAC [22], and a processor to complete the system. Figure 2.1 shows the block diagram of the current SDR.

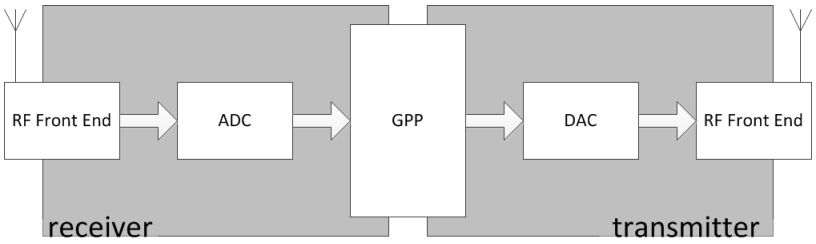

Figure 2.1: SDR block diagram

2.2 GNU Radio

GNU Radio is an open source software development toolkit for SDR. Users can use a low-cost external RF hardware, for example USRP (section 2.3), to create SDR or simulate the radio without any RF hardware. GNU Radio provides a number of radio components pre-written in software form which GNU Radio refers to as "blocks" which can communicate using various data types. Users can also write their own blocks if needed, developing GNU Radio applications and blocks created in C++ or Python. There are many GNU Radio applications[1] from the GNU Radio community for new users to play with. GNU Radio also offers a graphical user interface known as GNU Radio Companion.[2]

2.2.1 Creating a GNU Radio Application

The GNU Radio Application is usually written in Python. This example is taken from a GNU Radio website[2] to illustrate how to create an application by connecting blocks. This example does not require any external hardware. In each GNU Radio application, there is a "flow graph" that contains connected blocks. The connections between blocks allow the data processed by one block to pass to the next block. Each block usually does one job to keep them modular and flexible. Creating a GNU Radio block will be explained in detail later (section 2.2.2.)

Figure 2.2 is an example of a flow graph that consists of two sine generator sources and an audio sink. The audio sink, in this case, is the computer's speaker. The sine generator is a

[1]https://www.cgran.org/, http://gnuradio.org/redmine/projects/gnuradio/wiki/OtherCode
[2]http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion

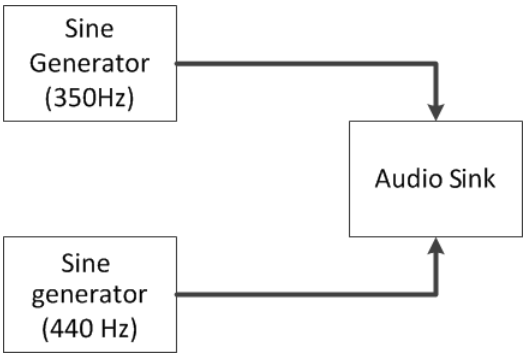

Figure 2.2: Dial tone generator flow graph (redrawn from[2])

built-in signal source block from GNU Radio. The GNU Radio application that generates the flow graph above is shown below (Listing 2.1.)

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

Listing 2.1: Dial tone generator python code

The first line is merely a syntax that allows the user to run python from a command line. Line 3 and 4 show importing of a `gr` and `audio` modules from the gnuradio. Line 6 shows a class creation of `my_top_block` which is derived from `gr.topblock` which allows the user to use the connect function. The `__init__()` function is the constructor of the class. Line 10 and 11 show `sample_rate` and `ampl` which are sampling rate and amplitude of the signal generators. Line 13 and 14 show `src0` and `src1` which are variables of the signal generator

that outputs float (notice the end of the `gr.sig_source_f()` identifying the output type of float) with a sampling rate of 32kHz (previously defined), these sources continuously create sine waves with frequencies 350 and 440Hz with 0.1 amplitude (also, previously defined.) Line 15 shows the creation of `dst` which is an audio sink. This audio sink connects to a sound card and will play samples given to it. Lines 16 and 17 show the block connection between the sources and the destination. The rest of the code starts the application.

2.2.2 Creating a GNU Radio Block

Creating a GNU Radio block can be overwhelming for beginners. There are a few templates[3] of blank GNU Radio blocks. In this thesis, only a basic block is required. Users can use a template[4][5] to create a GNU Radio block. For those who wish to go into detail, Eric Blossom has written an extensive tutorial [21] on how to write a signal processing block.

Inside a root directory of the template, there are 6 sub-directories. `apps` directory is where the test applications are located. `config` directory contains configuration files. `lib` is where the source files located. `swig` directory contains the code that combine the C++ source (in `lib`) to the python interface. `python` directory contains python scripts. `grc` directory contains the .xml files for GNU Radio Companion.

Work Function

Inside the `lib`, there are only 3 files: the makefile, the source and its header. The source file is where the block processing code is located.

```
1  int myBlock::work (int noutput_items,
2                     gr_vector_const_void_star &input_items,
3                     gr_vector_void_star &output_items)
4  {
5    const float *in = (const float *) input_items[0];
6    float *out = (float *) output_items[0];
7
8    for (int i = 0; i < noutput_items; i++){
9      out[i] = in[i] * (float)d_param1;
10   }
11
12   // Tell runtime system how many output items we produced.
13   return noutput_items;
14 }
```

Listing 2.2: GNU Radio Block C code

[3] http://gnuradio.org/redmine/projects/gnuradio/wiki/Tutorials
[4] Template files: http://gnuradio.org/redmine/attachments/download/277
[5] Tutorial: http://gnuradio.org/redmine/attachments/download/271

The focus of the source (Listing 2.2) is the `work` function. From the example, the source takes float input and output this number multiply by `d_param1` which is a parameter. Users can change the way the `work` function processes input. The rest of the code should be the same unless the name of the block or data type has changed.

2.3 USRP

GNU Radio can work on its own as a simulation environment but it can also create a real radio system. With a universal software radio peripheral (USRP), users can transmit and receive signals. USRP devices were developed by Ettus Research[3]. There are a number of different USRPs available on the market but the USRPs that will be used in this thesis are the USRP1, USRP2 and USRP E100.

2.3.1 USRP1

The USRP1[15] consists of:

- Four 64 MSPS, 12-bit ADCs
- Four 128 MSPS, 14-bit DACs
- Altera Cyclone FPGA
- Up to 64 MHz Signal Processing
- Up to 16 MHz USB Streaming
- USB 2.0 Interface to Host

The USRP1 is used to test the original UCLA Zigbee PHY code with the old GNU Radio driver along with the new UHD[10] driver. This device is one of the cheapest USRPs available from Ettus Research. USRP1 connects to a computer with a USB cable to communicate and transmit data.

2.3.2 USRP2

The USRP2[16] consists of

- Gigabit Ethernet interface
- 25 MHz of instantaneous RF bandwidth

- Xilinx Spartan 3-2000 FPGA
- Two 100 MHz 14-bit ADCs
- Two 400 MHz 16-bit DACs
- 1 MByte of high-speed SRAM
- Configuration stored on standard SD cards

Unlike the USRP1, the USRP2 uses gigabit ethernet to communicate and transfer data it the host. The device is used as a test with the new UHD driver and to test the interoperability of the UCLA ZigBee PHY code with the USRP1 (and USRP E100.) Ettus Research has replaced the USRP2 with the USRP N200 and USRP N210 as its new networked (communicated with gigabit ethernet) USRPs.

2.3.3 USRP E100

The USRP E100[12] consists of

- Xilinx Spartan 3A-DSP 1800 FPGA
- Two 64 MSPS, 12-bit ADC
- Two 128 MSPS, 14-bit DAC
- Up to 4 MHz Streaming to CPU
- Embedded OMAP Overo Module
- 720 MHz ARM Cortex A8 + C64 DSP
- Angstrom Linux w/ GNU Radio Built-In
- 512 MB RAM/4 GB Flash
- USB Console, OTG, and Host
- 10/100 Base T Supports SSH Access
- DVI Output for Monitor

The USRP E100 is a device that runs on its own. It has a 720 MHz ARM processor as its own processor. The USRP E100 has a DVI output for monitor out and USB ports to connect input devices such as a keyboard and a mouse and for termincal control. It also has an ethernet (non-gigabit) that adds itself into a network. It runs Angstrom Linux that comes with GNU Radio and the UHD driver. The USRP E100 is the target device to implement IEEE 802.15.4 on.

2.3.4 RFX2400 Daughterboard

The daughterboard is a modular RF-frontend board for the all types of USRPs. The RFX 2400 daughterboard [8] enables a USRP based SDR to act as a full duplex transceiver in the 2.4 GHz frequency band, which is one of the bands IEEE 802.15.4 operates on. RFX 2400s are used on all three different USRPs mentioned above.

2.4 IEEE 802.15.4

IEEE 802.15.4[4] is a simple low cost, low power wireless communication standard. It is suitable to use in home networking applications such as sensor network or for remote control. IEEE 802.15.4 is considered to be a wireless personal area network (WPAN), or more specifically, low-rate WPAN (LR-WPAN). The standard covers the physical layer (PHY) and the medium access control (MAC).

With the wireless network communication expanding to more areas in the world, wireless local area network (WLAN) IEEE 802.11 standards were created to give users the mobility of being connected to a wired backbone network. While IEEE 802.11 focuses on the speed and range of the connection of the network, "the focus of WPANs is low-cost, low power, short range and very small size."[19] WPAN was created based on the IEEE 802.15 standard. Medium rate WPAN, IEEE 802.15.1/Bluetooth, is suitable for cell phone or computer accessories such as computer mice or bluetooth headphones. Low rate WPAN, IEEE 802.15.4 is more suitable for a network that does not require a lot of data and conserve energy on the devices in the network such as a temperature sensor or remote control home-automation.

Some of the characteristics of an LR-WPAN are as follows[4]:

- Over-the-air data rates of 250 kb/s, 100kb/s, 40 kb/s, and 20 kb/s
- Star or peer-to-peer operation
- Allocated 16-bit short or 64-bit extended addresses
- Optional allocation of guaranteed time slots (GTSs)
- Carrier sense multiple access with collision avoidance (CSMA-CA) channel access
- Fully acknowledged protocol for transfer reliability
- Low power consumption
- Energy detection (ED)
- Link quality indication (LQI)

- 16 channels in the 2450 MHz band, 30 channels in the 915 MHz band, and 3 channels in the 868 MHz band

There are two different types of device that can be in the IEEE 802.15.4 network; a full-function device (FFD) and a reduced-dunction device (RFD). The FFD can be a personal area network (PAN) coordinator, a coordinator, or a device. It can talk to both RFDs or FFDs. Unlike FFDs, a RFD is used as an end device. A RFD usually consumes very little energy. A WPAN network must have two or more FFDs or RFDs and at least one of the devices must be an FFD.

2.4.1 Network Topologies

There are two topologies in the IEEE 802.15.4 specification; a star topology and a peer-to-peer topology (figure 2.3). The **star topology** has a single PAN coordinator in the center. An FFD can establish its own network and become a PAN coordinator and choose an available PAN identifier for a unique network. The **peer-to-peer topology** allows other devices to communicate with each other instead of going though the PAN coordinator, which still exist in the network. Multiple peer-to-peer topologies can also form a **cluster tree** network. The first PAN coordinator may instruct a device to become a PAN coordinator for a new cluster network (figure 2.4).

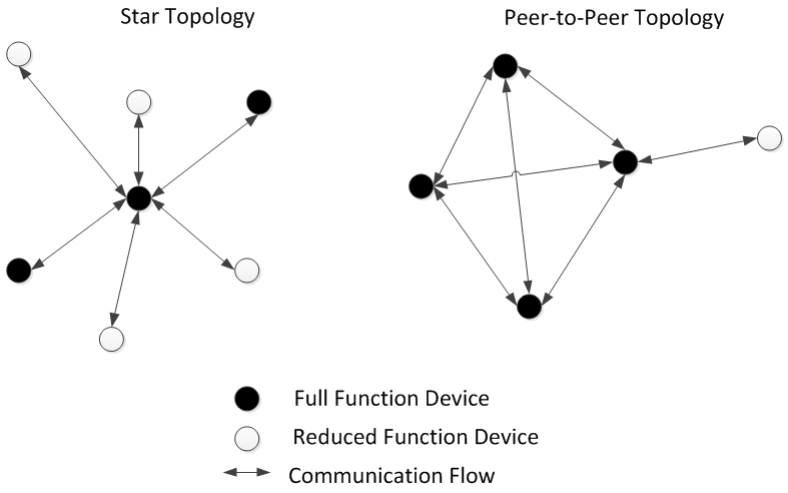

Figure 2.3: Examples of star and peer-to-peer topologies (redrawn from[4], Fig. 1, p. 14)

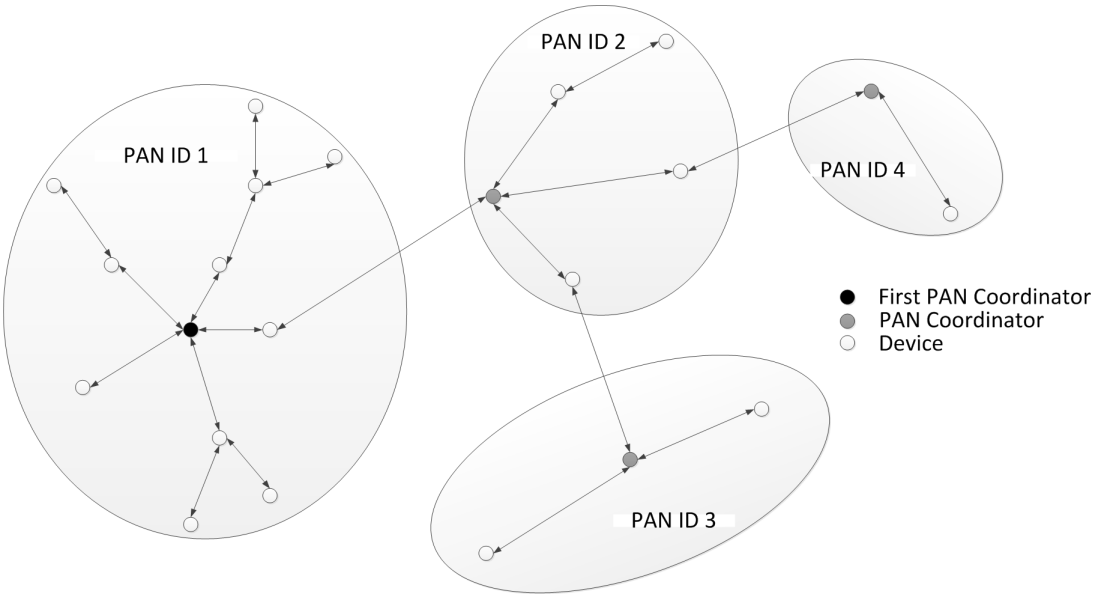

Figure 2.4: Cluster tSree network (redrawn from [4], Fig. 2, p. 16)

2.4.2 Device Architecture

A LR-WPAN device (figure 2.5) contains a PHY layer, a MAC layer, an upper layer, and "an IEEE 802.2 logical link control (LLC) can access the MAC sublayer through the service specific convergence sublayer (SSCS)."[4]

2.4.3 IEEE 802.15.4 PHY

The PHY has two services: the data service and the management service. The data service handles the transmission and reception of the PHY protocol data units (PPDU). The PHY jobs are "activation and deactivation of the radio transceiver, energy detection (ED), link quality indication (LQI), channel selection, clear channel assessment (CCA) and transmitting as well as receiving packets across the physical medium."[4] The standard operates in three unlicensed frequency bands as follow:

- 868-868.6 MHz (e.g., Europe)
- 902-928 MHz (e.g., North America)
- 2400-2483.5 MHz (worldwide)

The 2450 MHz frequency range is used in this thesis because it is supported worldwide. Data rate and other parameters for all the frequencies in IEEE 802.15.4 are listed in table 2.1.

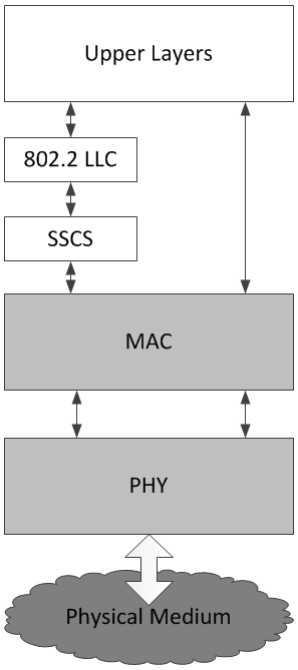

Figure 2.5: LR-WPAN Device Architecture (redrawn from [4], Fig. 3, p. 16)

PHY (MHz)	Frequency band (MHz)	Spreading parameters		Data parameters		
		Chip rate (kchip/s)	Modulation	Bit rate (kb/s)	Symbol rate (ksymbol/s)	Symbols
868/915	868-868.6	300	BPSK	20	20	Binary
868/915	902-928	600	BPSK	40	40	Binary
868/915 (optional)	868-868.6	400	ASK	250	12.5	20-bit PSSS
868/915 (optional)	902-928	1600	ASK	250	50	5-bit PSSS
868/915 (optional)	868-868.6	400	O-QPSK	100	25	16-ary Orthogonal
868/915 (optional)	902-928	1000	O-QPSK	250	62.5	16-ary Orthogonal
2450	2400-2483.5	2000	O-QPSK	250	62.5	16-ary Orthogonal

Table 2.1: Frequency bands and data rates (redrawn from [4], Tab. 1, p. 28)

Channel Numbering

There are a total of 27 channels available in the three frequency bands. For the 2450 MHz band, there are sixteen channels available beginning with channel 11 through channel 26. The center frequency for the channels for 2450 MHz bands are defined as follows[4]:

$$F_c = 2405 + 5(k - 11) \text{ in MHz, for } k = 11, 12, ..., 26 \tag{2.1}$$

where k is the channel number

For example, the center frequency of the channel 26 is 2480 MHz (as follows):

Frame length value	Payload
0-4	Reserved
5	MPDU (Acknowledgement)
6-8	Reserved
9-127	MPDU

Table 2.2: Frame length values (redrawn from [4], Tab. 21, p. 45)

$$\begin{aligned} F_c &= 2405 + 5(k - 11) \\ &= 2405 + 5(26 - 11) \\ &= 2480 \text{ MHz} \end{aligned} \tag{2.2}$$

PPDU Format

PHY protocol data units (PPDU) represents the bits of the data coming in or going out of the device. The data transmit the least significant field first. Each octet also transmits or receives the least significant bit first (LSB). As seen in figure 2.6, there are three components in the PPDU: the synchronization header (SHR), PHY header (PHR), and PHY payload.

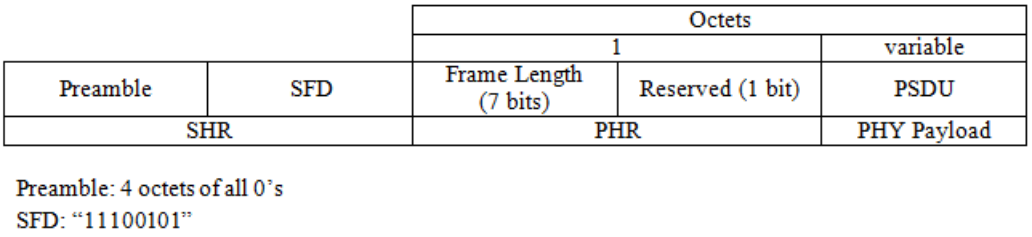

Figure 2.6: PPDU format (redrawn from [4], Fig. 16, p. 43)

SHR bits let the receiving devices determine that the messages are in the IEEE 802.15.4 format. SHR consists of a preamble and start frame delimiter (SFD). The preample, for 2450 MHz frequency band, is has a length of **four octets** and contains **all zeros**. The SFD, for 2450 MHz frequency band, has a length of **one octet** and contains `11100101` binary starting from bit 0.

PHR provides the frame length information. PHR consists of frame length (7 bit) and a reserved bit. Frame length field defines the length of the the PHY service data unit (PSDU) in octet. Table 2.2 shows the different type of payload according to the value.

Symbol (decimal)	Chip sequence (binary: c0 c1 ... c30 c31)
0	11011001110000110101001000101110
1	11101101100111000011010100100010
2	00101110110110011100001101010010
3	00100010111011011001110000110101
4	01010010001011101101100111000011
5	00110101001000101110110110011100
6	11000011010100100010111011011001
7	10011100001101010010001011101101
8	10001100100101100000011101111011
9	10111000110010010110000001110111
10	01111011100011001001011000000111
11	01110111011100011001001011000000
12	00000111011110111000110010010110
13	01100000011101111011100011001001
14	10010110000001110111101110001100
15	11001001011000000111011110111000

Table 2.3: Symbol to chip mapping (redrawn from [4], Tab. 24, p. 48)

Modulation Process

Once the transmitting device has put together a message in a proper PPDU formant, it converts the bits into a symbol. The symbol is a representation of 4 bits. "The 4 LSBs (b0, b1, b2, b3) of each octet shall map into one data symbol, and the 4 MSBs (b4, b5, b6, b7) of each octet shall map into the next data symbol."[4] Next, the symbols get translated into a 32-bit chip sequence. Chip sequences are shown in table 2.3. Later, the chip sequences are modulated onto the carrier with O-QPSK with half-since pulse shaping.

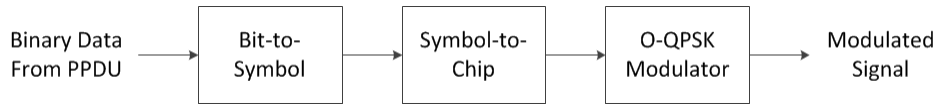

Figure 2.7: Modulation and spreading process (redrawn from [4], Fig. 18, p. 47)

Chapter 3

Implementation

This chapter explains the process of how to achieve the goal of implementing IEEE 802.15.4 on to an embedded device (USRP E100.), starting with the preparation of the test devices, and including extensive modification to software and reconfiguration of programmable hardware for these test devices. All the code modifications are also shown and explained here.

3.1 UCLA Zigbee PHY

Thomas Schmid from the University of California has written the GNU Radio application[27] which allows the transmission and reception capability. The code is written to support the USRP1 hardware. The application only covers 2.4 GHz band which is available worldwide. The modulation scheme used in Schmid's work is minimum shift-keying (MSK) as supposed to O-QPSK like the specification.

Since the targeted hardware is the USRP1, the computer running the UCLA Zigbee PHY application does most of the processing. The application uses the old USRP driver[10] that comes with GNU Radio. There is a newer driver for all of the USRP devices called the Universal Hardware Driver (UHD) which was used in this thesis to achieve operation of the UCLA Zigbee PHY on the USRP E100.

The source code of the UCLA Zigbee PHY can be obtained here:
`https://www.cgran.org/wiki/UCLAZigBee`
From source, the files that are of interest are `ucla_ieee802_15_4_packet_sink.cc`, `ucla_ieee802_15_4_packet_sink.h`, `ieee802_15_4.py`, `ieee802_15_4_pkt.py`, `cc2420_rxtest.py`, and `cc2420_txtest.py`

3.1.1 Modulation

File `cc2420_txtest.py` is an example program that transmits IEEE 802.15.4 packets. The message is sent through by calling `ieee802_15_4_mod_pkts()` function in `ieee802_15_4_pkt.py` to make a whole packet and starts `ieee802_15_4_mod()` in `ieee802_15_4.py` to modulate the packet. The modulation process is shown in figure 3.1.

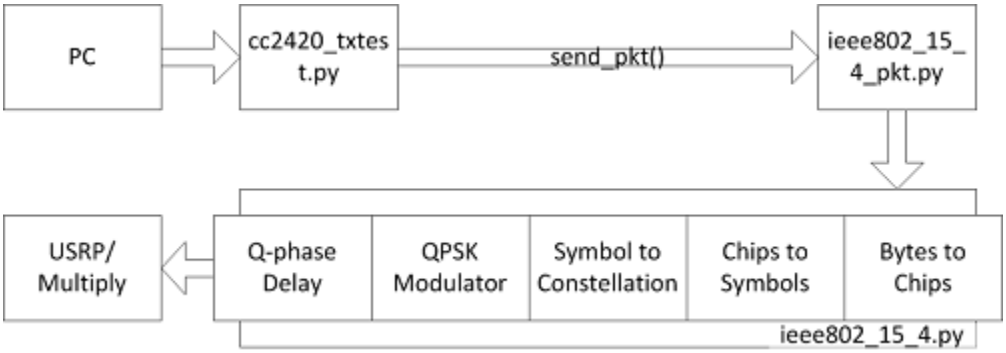

Figure 3.1: Modulation block diagram (redrawn from [27], Fig. 7)

The packet is constructed with some default fields. For example, the sequence number is `0xe5` and the address field is `0xFFFF, 0xFFFF, 0x10, 0x10` always. The MAC layer of IEEE 802.15.4 is mostly pre-defined for every packet with the code.

3.1.2 Demodulation

`cc2420_rxtest.py` is an example program that receives the packets. Schmid uses MSK demodulator instead of O-QPSK demodulator because they share half-sine pulse shape characteristics. Data coming from USRP first passes though squelch filter to filter out noise. Later, data goes to the FM demodulator which detects the MSK chip sequence. The data then passes the clock recovery block and then arrives at the message queue, via message sink, to be displayed on `cc2420_rxtest.py`. The demodulation process is shown in figure 3.2.

3.2 Preparation of Test Devices

3.2.1 PC Preparation

The computer used in the experiment has a **Intel(R) Core(TM)2 Duo CPU P8700 @ 2.53GHz processor** and **4GB of ram**. It is running an **Ubuntu 10.10 Linux** distribution **kernel 2.6.35-28 generic** with the following list of tools and libraries (list compiled by[24]):

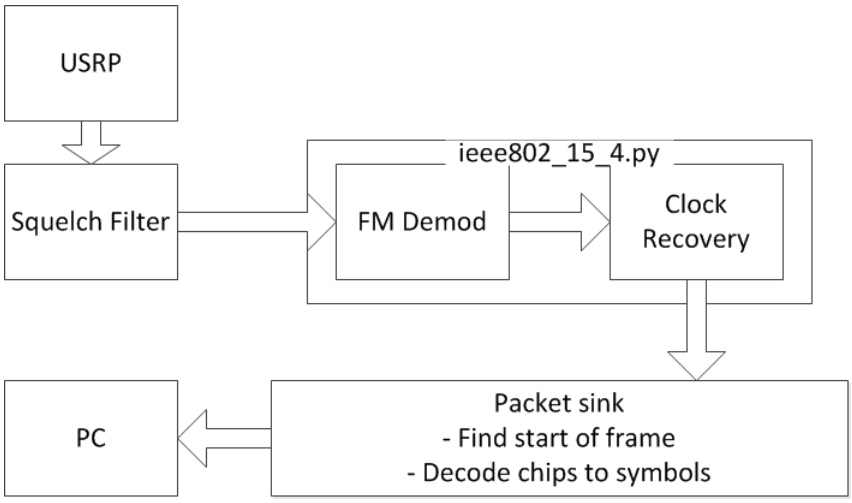

Figure 3.2: Demodulation block diagram (redrawn from [27], Fig. 6)

- Tools
  - GNU Radio 3.4.0 (subsection 3.2.1)
  - UHD Driver 003.003.000 (subsection 3.2.1)
  - g++
  - svn
  - git
  - make
  - autoconf, automake, libtool
  - sdcc
  - guile
  - ccache
- Libraries
  - python-dev
  - FFTW 3.X (fftw3, fftw3-dev)
  - cppunit (libcppunit, libcppunit-dev)
  - boost 1.35
  - wxWidgets(wx-common), wxPython(python-wxgtk2.8)
  - python-numpy

- python-sciy
- python-matplotlib
- Numeric
- ALSA(alsa-base, libasound2, libasound2-dev)
- Qt
- SDL(libsdl-dev)
- GSL GNU Scientific Library
- SWIG 1.3.31
- QWT, QWT PLot3d libraries (optional for Qt GUI)

Installing GNU Radio and UHD Driver

For the **UHD driver**, the version used in the experiment can be obtained using GIT, a distributed version control system, with the commit number `1eefd6f232` or version **003.003.000**. Use the following commands to obtain the code 3.1:

```
$ git clone git://ettus.sourcerepo.com/ettus/uhd.git
$ cd uhd
$ git checkout 1eefd6f232
$ cd uhd/host/
$ mkdir build
$ cd build
$ cmake ../
$ make
$ make test
# make install
# ldconfig
```

Listing 3.1: Installing UHD driver with specific versions

The **GNU Radio** GIT commit number is `441a3767e05d15e62c519ea66b848b5adb0f4b3a` or version **3.4.0**. Use the following commands to obtain the code 3.2:

```
$ git clone http://gnuradio.org/git/gnuradio.git
$ cd gnuradio
$ git checkout 441a3767e05d15e62c519ea66b848b5adb0f4b3a
$ mkdir build
$ cd build
$ cmake ../
$ make
# make install
# ldconfig
```

Listing 3.2: Installing GNU Radio with specific versions

Installing UCLA ZigBee PHY

UCLA ZigBee PHY[28] source project and installation instructions can be found below 3.3

```
$ svn co https://www.cgran.org/cgran/projects/ucla_zigbee_phy/trunk
    ucla_zigbee_phy
$ cd ucla_zigbee_phy
$ ./bootstrap && ./configure && make
# make install
```

Listing 3.3: Installing UCLA ZigBee PHY

3.2.2 USRP1 Preparation

With the new code that refers to a specific version of the UHD libraries, a new hardware firmware is needed. When USRP1 starts, it looks for the firmware update in a specific location. In this case, USRP1 firmware is in `/usr/local/share/uhd/images/`. There are two ways that users can get the firmware images; download a pre-built images from Ettus Research website[14] or build the firmware images from the downloaded UHD driver. The first method is preferred because there is no need to modify th FPGA component of the USRP1 in this project and building firmware from scratch can be time consuming. The UHD driver version used in this thesis is **003.003.000** The version number will remain the same throughout. Download the "images-only"[1] instead of the whole kernel.

Files to copy to `/usr/local/share/uhd/images/` from
`/UHD-images-003.003.000/share/uhd/images/`

- usrp1_fw.ihx
- usrp1_fpga.rbf
- usrp1_fpga_4rx.rbf

In GNU Radio, a UHD device is usually referred to as an IP address. Since USRP1 cannot be referred as an IP address, the field is blank.

3.2.3 USRP2 Preparation

As in the USRP1 preparation, the USRP2 also needs firmware images (UHD **003.003.000**) which can be downloaded from Ettus Research's website[14]. This is the same archive downloaded from the previous section 3.2.2.

[1] http://files.ettus.com/uhd_releases/003_003_000/images-only/

USRP2 has an SD-card slot that it uses to look for firmware images when powering up. Updating USRP2 can easily be done with the utility software provided in the UHD driver source. The software is GUI-based. Remember to run software in super user mode.

```
# <UHD source>/host/utils/usrp2_card_burner_gui.py
```

Figure 3.3 shows the interface of the burner software. The firmware and FPGA binary of USRP2 are clear labeled in the image directory of the archive (`/UHD-images-003.003.000/share/uhd/images/`).

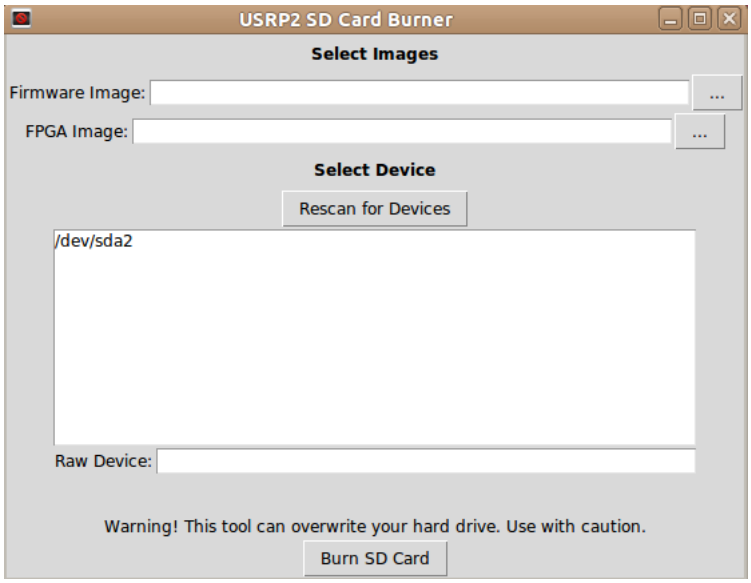

Figure 3.3: USRP2 card burner

3.2.4 USRP E100 Preparation

The USRP E100 comes with Angstrom Linux distribution instead of Ubuntu like the testing computer. Installing UHD driver and GNU Radio is different. Ben Hilburn has written an extensive tutorial on how to use USRP E100 on the Ettus Research website [13]. At the time of this writing and the time the experiment has started, there were many changes being made to the tutorial. the UHD driver for USRP E100 used in this thesis is UHD version **003.003.000**.

USRP E100 Boot Files

USRP E100 runs on a microSD-card. This card contains two partitions: one is the boot partition and the other is the file system. First thing to do when preparing a USRP E100 is to update the device to the latest boot files. The official boot files are available on Ettus Research website[2]. However, sticking to the build in this thesis, the boot files can be obtained from the following links:

- MLO file (first bootloader) http://dl.dropbox.com/u/14618236/MLO
- U-Boot (second bootloader) http://dl.dropbox.com/u/14618236/u-boot.bin-for-2.6.38
- uImage (Linux kernel) http://dl.dropbox.com/u/14618236/uImage-2.6.38-r0b-usrp-e1xx.bin

Updating boot files can be done from external computers or inside the device itself. This thesis only shows how to update from external computers which requires an SD-card reader. For instructions of how to update boot files, please refer to the tutorial[11].

There are two partitions, "FAT" and "rootfs," inside the microSD-card that runs the E100, while the boot files reside in the "FAT" partition. Simply copy the boot files over and unmount the partitions.

```
$ cp <download location>/MLO MLO
$ cp <download location>/u-boot.bin-for-2.6.38 u-boot.bin
$ cp <download location>/uImage-2.6.38-r0b-usrp-e1xx.bin uImage

$ umount /media/FAT
$ umount /media/rootfs
```

Listing 3.4: Updating boot files

Update FPGA firmware

As with other USRPs, FPGA firmware for the E100 needs to match the version of the UHD driver. The USRP E100 searches inside its own system for the firmware, in this case, `/usr/local/share/uhd/images/`. The firmware is the same firmware described in section 3.2.2 and 3.2.3 which is referred to as a "link" in the listing 3.2.4. This step is performed from the E100 itself.

```
$ cd /usr/local/share/uhd/images/
$ mv usrp_e100_fpga.bin usrp_e100_fpga.bin.bak
$ wget <paste link>
$ tar zxvf <downloaded tarball>
```

[2]http://files.ettus.com/e1xx_images/

```
$ mv <tarball directory>/share/uhd/images/* .
$ exit
```

Install UHD Driver and GNU Radio

The installation process in this section is similar to the installation process for PCs (section 3.2.1). However, it requires extra configuration.

For the UHD driver, remove the old driver in the system and add a user library search path:

```
$ opkg remove --force-depends uhd uhd-dev uhd-examples uhd-tests

$ echo "/usr/local/lib" >> /etc/ld.so.conf
```

How to download and install UHD driver:

```
$ git clone git://ettus.sourcerepo.com/ettus/uhd.git
$ cd uhd
$ git checkout 1eefd6f232
$ cd uhd/host/
$ mkdir build
$ cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchains/arm_cortex_a8_native.cmake
    -DENABLE_E100=ON -DENABLE_USRP_E_UTILS=TRUE ../
$ make
$ make test
$ make install
$ ldconfig
```

How to remove the old GNU Radio in the system:

```
$ opkg remove --force-depends gnuradio gnuradio-dev gnuradio-examples task-
    gnuradio
```

How to download and install GNU Radio:

```
$ git clone http://gnuradio.org/git/gnuradio.git
$ cd gnuradio
$ git checkout 441a3767e05d15e62c519ea66b848b5adb0f4b3a
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchains
    /arm_cortex_a8_native.cmake -DQT_QTCORE_INCLUDE_DIR=/usr/include/qt4/
    QtCore -DQT_QTGUI_INCLUDE_DIR=/usr/include/qt4/QtGui -DQT_QMAKE_EXECUTABLE
    =/usr/bin/qmake -DENABLE_GR_QTGUI=ON -DQT_LIBRARY_DIR=/usr/lib -
    DQT_INCLUDE_DIR=/usr/include/qt4/ -DQT_MOC_EXECUTABLE=/usr/bin/moc -
    DQT_UIC_EXECUTABLE=/usr/bin/uic -DQT_RCC_EXECUTABLE=/usr/bin/rcc -
    DCMAKE_BUILD_TYPE=release ../
```

```
$ make
$ make install
$ ldconfig
```

Install UCLA ZigBee PHY

Installing UCLA ZigBee PHY is exactly the same as the PC (section 3.2.1). If 'svn' is not available on the USRP E100, copying the source from another computer and installing it on the device will work as well.

3.3 UHD code modification

This section discusses the modification of the UCLA ZigBee PHY[28] in order to operate in the UHD environment instead of the USRP1 only with built-in usrp GNU Radio blocks. The main source, the GNU Radio block, remains unchanged, however, application programs that initialize and start the device and blocks need to be modified.

GNU Radio source now comes with gr-uhd, a source and sink blocks for UHD in GNU Radio. It also has the block wrapper in GNU Radio Companion. gr-uhd already comes with the GNU Radio source and will be used to replace the old USRP1 blocks.

```
1  self.data_rate = options.data_rate
2  self.samples_per_symbol = 2
3  self.usrp_decim = int (64e6 / self.samples_per_symbol / self.data_rate)
4  self.fs = self.data_rate * self.samples_per_symbol
5  payload_size = 128              # bytes
6
7  u = usrp.source_c (0, self.usrp_decim)
8  if options.rx_subdev_spec is None:
9      options.rx_subdev_spec = pick_subdevice(u)
10 u.set_mux(usrp.determine_rx_mux_value(u, options.rx_subdev_spec))
11
12 subdev = usrp.selected_subdev(u, options.rx_subdev_spec)
13
14 u.tune(0, subdev, options.cordic_freq)
15 u.set_pga(0, options.gain)
16 u.set_pga(1, options.gain)
```

Listing 3.5: cc2420_rxtest.py

`cc2420_rxtest.py` (listing 3.5) was used as a reference in to the new program. In the source, line 7 shows the initialization of the USRP1 and line 1-5 show the parameters for initialing

the device. Line 8-12 show the daughterboard selection for the USRP1. There is only one daughterboard slot on the USRO E100 therefore this will be unnecessary later. Line 14-16 show the setting of the center frequency and gain level. The new application is shown in listing 3.6. Line 5 shows the new UHD initializing code.

```
1  self.data_rate = options.data_rate
2  self.samples_per_symbol = 2
3  payload_size = 128              # bytes
4
5  u = uhd.usrp_source(device_addr="", io_type=uhd.io_type.COMPLEX_FLOAT32,
       num_channels=1, )
6
7  self.u = u
8  self.u.set_gain(options.gain)
9  self.u.set_samp_rate(options.sample_rate)
10 self.u.set_center_freq(options.cordic_freq)
```

Listing 3.6: cc2420_rxtest_uhd.py

`cc2420_txtest.py`, the reference transmit code, can be modified in a similar way. `usrp.sink_c()` is replaced by `uhd.usrp_sink()` along with appropriate parameters.

3.3.1 UHD on USRPs Testing

The goal of supporting IEEE 802.15.4 on an embedded device would be achieved in this section if the USRP E100 could transmit and receive IEEE 802.15.4 with UCLA ZigBee PHY. However, the USRP E100 is an embedded device and does not have the processing power like a modern PC. Here are the results of the test.

Non-embedded USRPs

With UHD modification to the main application, all non-embedded UHD devices running on a UHD driver can communicate using IEEE 802.15.4 protocol. Figure 3.4 shows the screenshot of the result of `cc2420_rxtest_uhd.py`.

USRP E100

USRP E100 can transmit but cannot receive the IEEE 802.15.4 packet. The 720 MHz ARM Cortex A8 processor is not fast enough to process and demodulate the data coming in. From exploring the UCLA ZigBee PHY source code, IEEE 802.15.4 packet sink block in `ucla_ieee802_15_4_packet_sink.cc` seems the be the most computing intensive block of code. As explained in section 2.4.3, this processing block is doing the work of demodulation

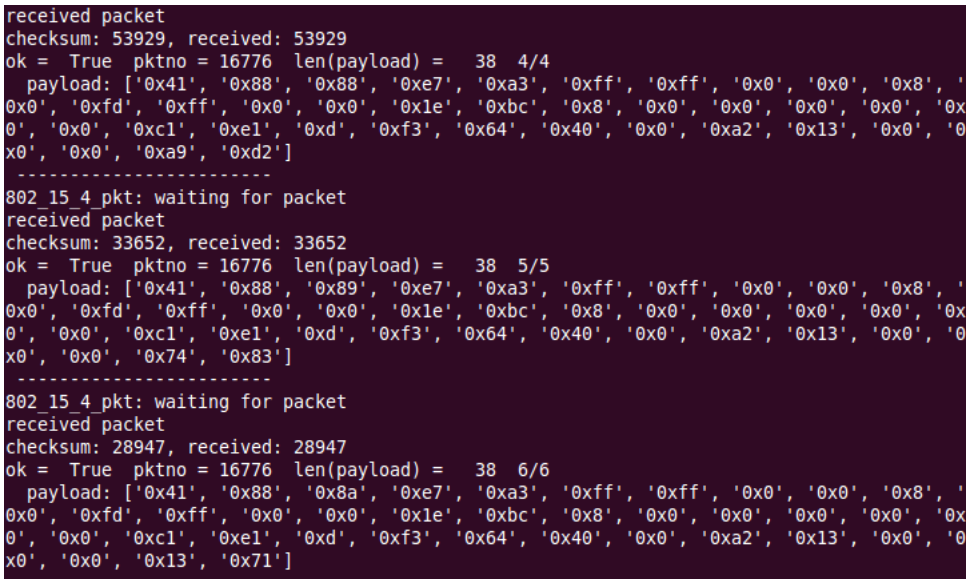

Figure 3.4: Non-embedded USRPs with UHD can transmit and receive IEEE 802.15.4 packets

of the IEEE 802.15.4 packet. This includes finding the correct chip sequence to determine the correct packet (sync) along with translating the packet into the actual message.

In order the achieve the goal of a standalone device capable of transmitting and receiving IEEE 802.15.4 packet, modification of the device must be made. Luckily, USRP E100 has available FPGA space that can be used to help with the demodulation. Section 3.4 and 3.5 will discuss the process of modifying the hardware (FPGA) and software (UCLA ZigBee PHY) in order to make receiving IEEE 802.15.4 packet possible.

3.4 USRP E100 Hardware

With the UHD driver downloaded from section 3.2.2, users can create his or her own FPGA project using a script that comes with the driver. Since the FPGA inside the USRP E100 is a Xilinx Spartan 3A-DSP 1800 FPGA, Xilinx ISE HDL design tool is needed to modify the FPGA code. Navigate to `<UHD source>/fpga/usrp2/top/` to start making an ISE project and follow the instruction below:

```
$ cd <UHD source>/fpga/usrp2/top/
$ make -f Makefile.E100 bin
$ bin file in build-E100/*.bin
```

This process can take a long time to finish, depending on the speed of the computer being used. After the process is complete, an ISE project file will appear in the directory and can be opened with ISE.

Once synthesizing the project is complete, a FPGA bianry (`u1e.bin`) can be found in `<UHD source>/fpga/usrp2/top/E1x0/build-E100`. Replace this file with the old FPGA binary on the USRP E100 located in `/usr/local/share/uhd/images/`.

3.4.1 FPGA

The USRP E100 FPGA top design has a single module on it. This module is called `u1e_core` which is described in `u1e_core.v`. u1e_core uses a wishbone architecture to control all the components inside the system (misc LEDs, switches, controls, UART, SPI, I2C, GPIOs, FIFO to wishbone slave for async messages, and bus settings.) For the recieving end, figure 3.5 shows `rx_frontend` taking data (I and Q) and passing it to `dsp_core_rx`. `dsp_core_rx` takes the data and set a `strobe` signal along with the I and Q data to `vita_rx_chain` which will eventually end up in the main processor.

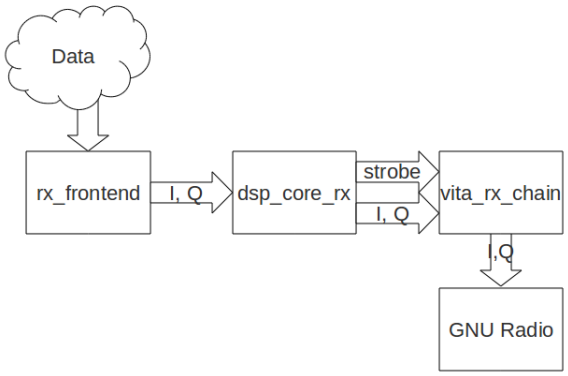

Figure 3.5: USRP E100 FPGA reveiver block diagram

3.4.2 IEEE 802.15.4 RX module

Jeong-O Jeong, a graduate student from the Mobile Portable Research Group at Virginia Tech (MPRG)[3], has created an IEEE 802.15.4 FPGA receiver for the USRP N210. The module works like a regular demodulator. First it looks for the correct chip sequence. Once the module finds preamble, it sends the message to the PC. Figure 3.6 show where the module is

[3]http://www.mprg.org/

placed from in the main design. `xbee_rx` module is right in the middle of `dsp_core_rx` and `vita_rx_chain`. The new module ignores the `strobe` and creates a new one that triggers when the message is ready.

The USRP E100 has a similar signal receiver structure to the USRP N210. `xbee_rx` module be ported to the E100 design using the same method described in figure 3.6. The module can be manually added to the project using ISE "add source" functionality.

Instead of returning the I and Q data to the computer, the new module returns a **decoded message** along with **translation of I and Q** (or$\sqrt{I^2 + Q^2}$) for the future use of power measurement. The message length of the returned message is 32-bit wide. The format of the returned message is as follow:

{**message[7:0]**, **state[4:0]**, 2'b00, decoded_strobe, **translate_out[7:0]**, **translate_out[15:8]**}

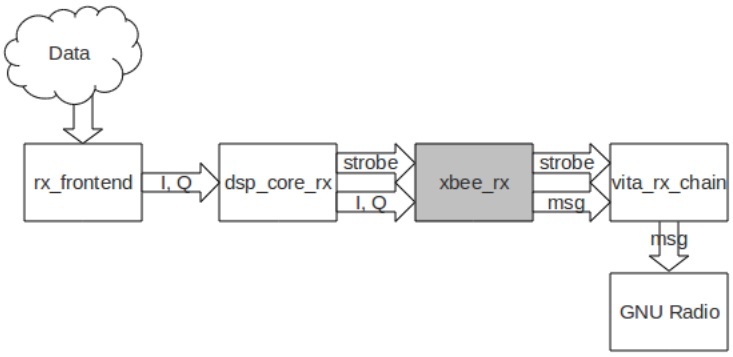

Figure 3.6: USRP E100 FPGA reveiver block diagram with xbee_rx module

FPGA Finite State Machine

Figure 3.7 shows the finite state machine (FSM) of the demodulation process of the FPGA design. The FPGA receives the data from the RF frontend and then tries to match the chip-to-symbol-to-data (this is the same idea as figure 2.7 however backwards) to enter the `search` state of the FSM. Once the data match both `found_preamble` and `found_sfd` states, it look for the next byte, which is the frame length and transfer the rest of the data back, which is the PSDU of the MAC data, to the GNU Radio. Once the whole frame data is sent, FSM goes back to search for a new IEEE 802.15.4 again.

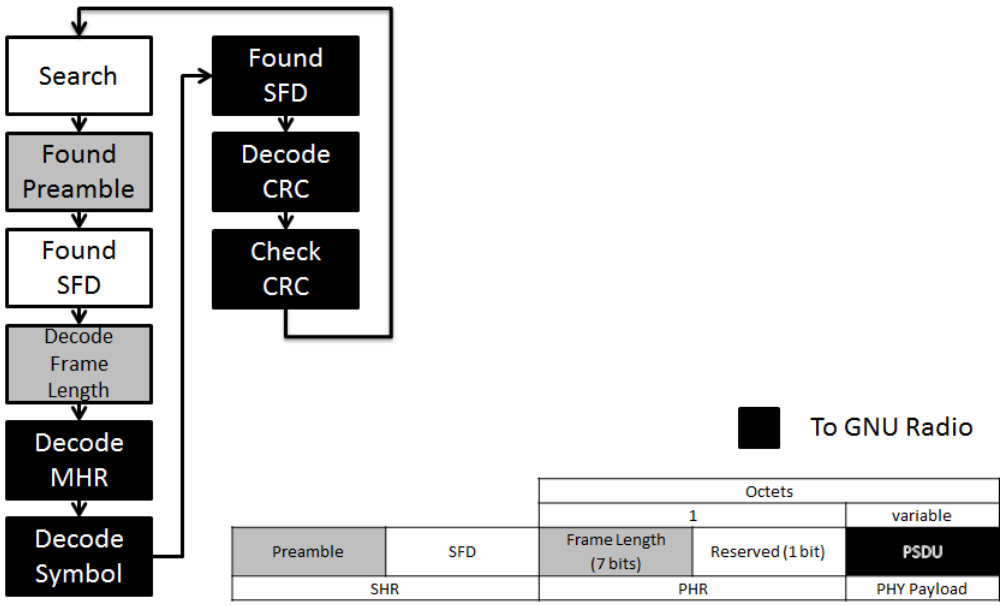

Figure 3.7: FPGA finite state machine of the demodulation process

3.5 USRP E100 Software

3.5.1 Modifying GNU Radio block

Since the message format has been changed, UCLA ZigBee PHY needs to be modified further. Inside the UCLA ZigBee PHY, there is a GNU Radio block that takes and processes the data from the USRP. The GNU Radio block inside UCLA ZigBee PHY is the same GNU Radio block described in section 2.2.2. Listing 3.7 in the new `ucla_ieee802_15_4_packet_sink.cc` shows the modified `work()` function that takes a new daa format according to section 3.4.2.

```
1  int ucla_ieee802_15_4_packet_sink::work (int noutput_items,
2                  gr_vector_const_void_star &input_items,
3                  gr_vector_void_star &output_items)
4  {
5      const char *in = (const char *) input_items[0];
6      char *word = (char *)malloc(4);
7      int state, counter = 0;
8      d_payload_cnt = 0;
9
10     for(int i=0; i<noutput_items; i++){
11
12         std::memcpy(word, in, 4);
13
14         counter++;
```

```
15          if(counter == 15)
16          {
17              counter = 0;
18              state = ((0xF8 & word[0]) >> 3);
19
20              if(state == 4 || state == 5) {
21                  d_packet[d_packetlen_cnt++] = (short)(word[1]);
22                  d_payload_cnt++;
23                  d_packet_byte_index = 0;
24              }
25              if(state == 6) {
26                  // build a message
27                  gr_message_sptr msg = gr_make_message(0, 0, 0,
28                                                        d_packetlen_cnt);
29                  if(d_packetlen_cnt > 0)
30                      memcpy(msg->msg(), d_packet, d_packetlen_cnt);
31
32                  d_target_queue->insert_tail(msg);
33
34                  //reset everything
35                  msg.reset();
36                  enter_search();
37                  enter_have_sync();
38                  enter_have_header(0);
39
40              }
41          }
42          in+=4;
43      }
44      std::free(word);
45
46      return noutput_items;
47  }
```

Listing 3.7: modified ucla_ieee802_15_4_packet_sink.cc

The data gets copied to the program memory in line 11. There is a bug in the FPGA code because the message gets repeated every 16 times so there has to be a counter that prevents the message from repeating at line 15. `state` (line 18) is the state of the whole message. The state can either be 4, 5, 6, or 7. State 4-5 indicates the actual message and state 6-7 indicates the end of the message. When the state is 4 or 5, the message is added to `d_packet` which will be sent to the later part of the program. When the state is 6 (the state does not jump so there is no need for state 7) the message gets added to the queue and resets.

From figure 3.8, the original code, and 3.9, the modified code, from `ieee802_15_4_pkt.py`, the demodulation block (line 6 of the original file) needs to be removed entirely. Also, the data type input has changed from complex to float. Line 9 of the original file and line 8 of the modified file shows the connection of blocks without the demodulation block.

```
1 gr.hier_block2.__init__(self, "ieee802_15_4_demod_pkts",
2                         gr.io_signature(1, 1, gr.sizeof_gr_complex), # Input
3                         gr.io_signature(0, 0, 0))  # Output
4
5     self._rcvd_pktq = gr.msg_queue()          # holds packets from the PHY
6     self.ieee802_15_4_demod = ieee802_15_4.ieee802_15_4_demod(self, *args, **
          kwargs)
7     self._packet_sink = ucla.ieee802_15_4_packet_sink(self._rcvd_pktq, self.
          threshold)
8
9     self.connect(self,self.ieee802_15_4_demod, self._packet_sink)
```

Listing 3.8: ieee802_15_4_pkt.py

```
1 gr.hier_block2.__init__(self, "ieee802_15_4_demod_pkts",
2                         gr.io_signature(1, 1, gr.sizeof_float), # Input
3                         gr.io_signature(0, 0, 0))  # Output
4
5     self._rcvd_pktq = gr.msg_queue()          # holds packets from the PHY
6     self._packet_sink = ucla.ieee802_15_4_packet_sink(self._rcvd_pktq, self.
          threshold)
7
8     self.connect(self, self._packet_sink)
```

Listing 3.9: modified ieee802_15_4_pkt.py

With the new modified code and modified GNU Radio block inside UCLA ZigBee PHY, the USRP E100 can use `cc2420_rxtest_uhd.py` to receive IEEE 802.15.4 packets. Figure 3.8 shows the screenshot of the software.

3.6 IEEE 802.15.4 Monitor Software

With the ability to transmit and receive IEEE 802.15.4 messages just with an embedded standalone device, the USRP E100 can become a monitor program that checks messages passing within the network. A user-friendly user interface monitor software would be a useful application that can hi-light the ability of software defined radio.

3.6.1 Features

The monitor should have features as listed below:

- Display messages being sent in the network in real-time

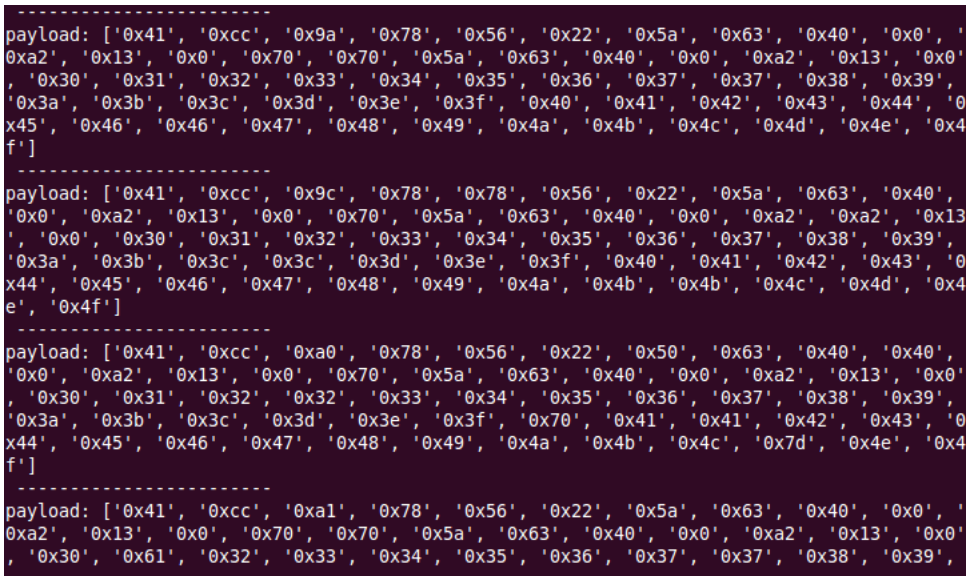

Figure 3.8: USRP E100 with UHD driver and FPGA modification can receive IEEE 802.15.4 packets

- Display the source and destination address
- Display the time of the message
- Display the RSSI value of each messages
- Keep track of all node addresses

The message payload, source address, and destination address can be obtained from the messages coming from the USRP. The message format is from the XBee, a commercial IEEE 802.15.4 device which will be talked about in later section 3.7.1. The byte-to-byte format is displayed at table 3.9.

Octets: 2	1	2	8	8	variable	2
Frame Control	Sequence Number	PAN ID	Destination Address	Source Address	Frame Payload	FCS

Figure 3.9: XBee MAC

RSSI value is taking out of the daughterboard with `get_dboard_sensor("rssi")` command in unit of **dB** ratio and time is the current time the message arrives at the GNU Radio. Address list is a unique address set (source and destination) that is taken from the message from USRP itself.

3.6.2 Design Choice

To create monitor software, building a user interface on top of `cc2420_rxtest_uhd.py` was selected as a logical starting point. GNU Radio Companion was also the choice in making a monitor program but GNU Radio Companion is not available on USRP E100 Angstorm operating system because it does not have wyPython in the system.

A user interface program implemented in python needs to be created. This program must be combined with the above mentioned program. PyGtk [4] is used as a library to create a graphical interface of the monitor program. The monitor software should update every time there is a new message communicated within a network. It should display the source, destination, and the message itself for each communication.

User interface program and `cc2420_rxtest_uhd.py` run on different threads. The `cc2420_rxtest_uhd.py` receiver requires a main thread to wait for the incoming messages and a new thread will be used to run the graphic user interface.

3.7 Compatibility with Commercial Device

3.7.1 XBee

XBee[18] is a IEEE 802.15.4 device by Digi International[5] that is commercially available and has a low cost. XBee device can be reconfigured via a program X-CTU[6] which is only available on Windows. The XBee used in this thesis is XBee series 1.

Figure 3.10 shows the configuration screenshot of the X-CTU program. `Channel` represents the channel of communication which was described in section2.4.3. In this case, channel 1A (2480 MHz) is used in the network. `PAN ID` represents the ID of the network, all devices need to have the same PAN ID in order to communicate within the network. `Destination Address` represents the destination of the message. `Serial Number` represents the address of the device itself.

Receiving XBee messages on modified UCLA ZigBee PHY is perfectly fine. With the ability of SDR, there is no restriction to the destination address. The program can decode any messages within the same channel (center frequency). The only problem is the MAC layer of the message is not translated properly.

[4] http://www.pygtk.org/
[5] http://www.digi.com/
[6] http://www.digi.com/support/productdetail?pid=3352&osvid=0&type=utilities

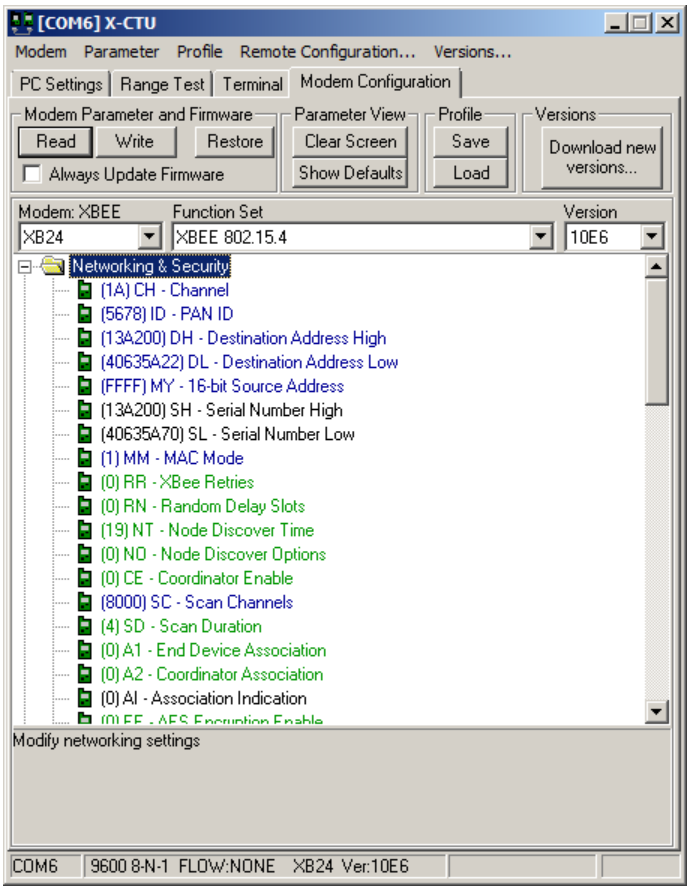

Figure 3.10: X-CTU configuration

3.7.2 IEEE 802.15.4 MAC

To find out what part of XBee MAC looks like, a simple test with XBee device transmitting and `cc2420_rxtest_uhd.py` receiving is used to collect the whole data coming from XBee. Each message is examined to match the type of MAC layout described in the specification[4].

Figure 3.11 show the general format of the MAC frame. MAC Protocol Data Unit (MPDU) is inside the PPDU (figure 2.6) labelled as PSDU. It contains three main parts: a MAC Header (MHR) which contains the information of the format, a MAC Payload which contains the actual payload, and Frame Correction Sequence (FCS).

Data coming from XBee in MHR field contains a frame control, sequence number, destination address, and source address. Frame control field defines the rest of the MPDU. There are many sub-fields which are not displayed in the monitor software but that are present in the code. FCS is also ignored in the software.

<table>
<tr><td>Octets: 2</td><td>1</td><td>0/2</td><td>0/2/8</td><td>0/2</td><td>0/2/8</td><td>0/5/6/10/14</td><td>variable</td><td>2</td></tr>
<tr><td rowspan="2">Frame Control</td><td rowspan="2">Sequence Number</td><td>Dest. PAN Identifier</td><td>Dest. Address</td><td>Source PAN Identifier</td><td>Source Address</td><td>Auxiliary Security Header</td><td rowspan="2">Frame Payload</td><td rowspan="2">FCS</td></tr>
<tr><td colspan="5">Addressing Field</td></tr>
<tr><td colspan="7">MHR</td><td>MAC Payload</td><td>MFR</td></tr>
</table>

Figure 3.11: MPDU format (redrawn from [4], Fig. 41, p. 138)

3.7.3 Transmitting to XBee device

At this moment, `cc2420_rxtest_uhd.py` can receive and display the information of the message properly. However, transmitting from `cc2420_txtest_uhd.py` to XBee is still a problem. The original code use different values in the Frame control Field while XBee uses a different value. XBee Frame Control Field is used to transmit data. The Frame Control Field value is `0xCC41`. As mentioned in section 2.4.3, IEEE 802.15.4 transfers the least significant octet first. Even though the Frame Control Field is `0xCC41`, 0x41 is transmit first.

Here is the data transmitting (hard coded into `cc2420_txtest_uhd.py`) to XBee:
'0x41', '0xcc', '0xe5', '0x78', '0x56', '0x70', '0x5a', '0x63', '0x40', '0x0', '0xa2', '0x13', '0x0', '0x22', '0x5a', '0x63', '0x40', '0x0', '0xa2', '0x13', '0x0', '0x48', '0x65', '0x6c', '0x6c', '0x6f', '0x20', '0x57', '0x6f', '0x72', '0x6c', '0x64'

The transmit message follow the MAC layer of XBee (table 3.9) exactly. The destination address is `0x0013a20040635a70` which is the XBee receiver's address. The source address is `0x0013a20040635a22`. This address can be anything since USRP does not have a real address with it. The PAN ID has to be already set to the receiver XBee to be `0x5678`. The rest of the message is the payload `Hello World`.

Figure 3.12 shows X-CTU software terminal displaying the received messages from the USRP.

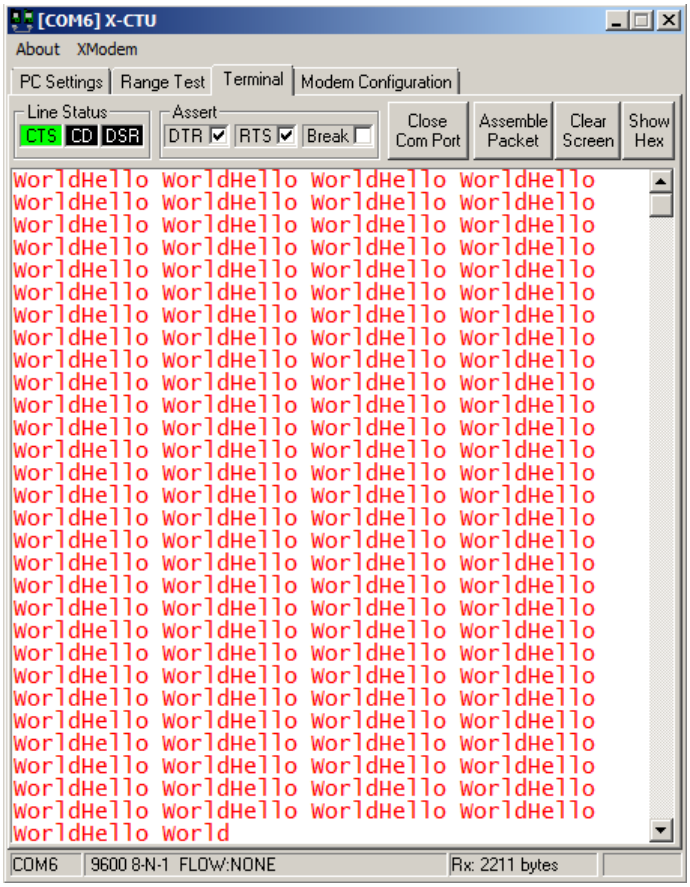

Figure 3.12: X-CTU terminal showing the messages from USRP

Chapter 4

Results and Analysis

4.1 Final Monitor Software

Both embedded and non-embedded USRPs now have the ability to transmit and receive IEEE 802.15.4 packets. Furthermore, both have the ability to communicate with a commercial device (XBee). The software works effectively on the non-embedded USRPs, and the transmission operates successfully on the E100. Reception of packets on the E100 is currently partially successful. Unlike a non-embedded USRP, the messages are not perfectly decoded. Figure 4.1 shows the monitor software running on a USRP E100 with some incomplete messages.

4.2 Performance Metric

A simple communication test is accomplished by setting up a transmitter, an XBee, to transmit multiple of the same to the USRP E100. The receiver in the USRP E100 counts the number of bytes that it decodes correctly in each message that it receives.

There are a total of 30 bytes that will be counted toward the correction of the message starting with the PAN ID field. Each match will receive one point out of a possible of 30 points. The "good" message has a match of 20 bytes or more. The correct message should be the same as 3.7.3. Over 100 messages received onto the USRP E100 will be counted and the average will be determined.

From examining the messages, there often occur two repeated bytes which came from the FPGA when matching the bytes. These indexes of the byte location are shifted twice right after PAN ID and before the payload message.

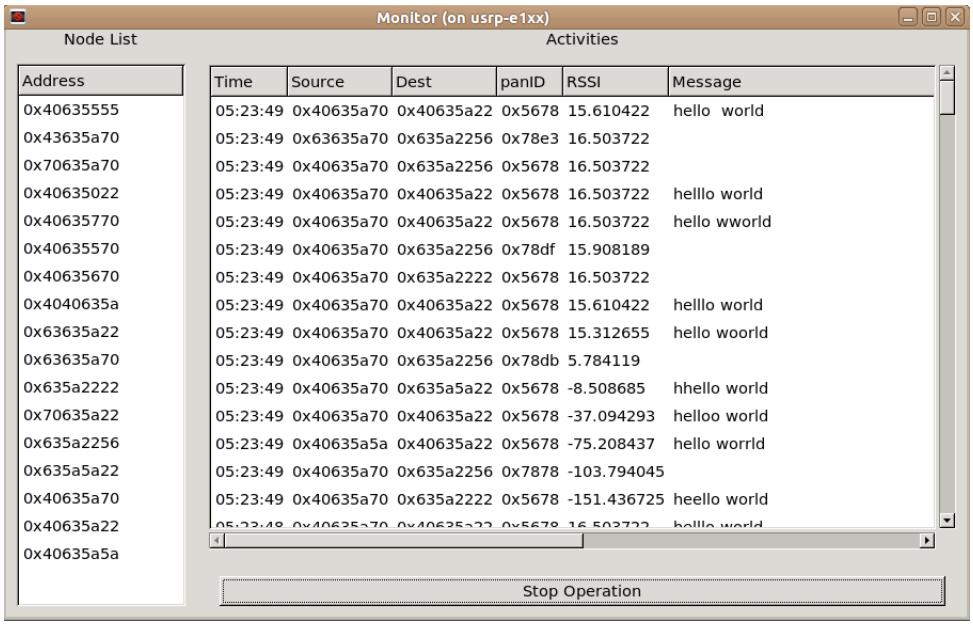

Figure 4.1: IEEE 802.15.4 monitor software

The test result is then compared against the full-UCLA ZigBee PHY software that runs over the USRP1 connecting to a test computer (the same computer from 3.2.1.)

4.3 Performance Results

Figure 4.2 shows the points for each of the received messages for the USRP E100. Figure 4.3 shows the result for the all-software implementation used with the USRP1. Table 4.1 shows the high scores, low scores, average scores, and number of messages that score more than or equal to 20 out of 30 for both devices.

The results show that the USRP1 can properly decode XBee messages without errors. USRP E100, on the other hand, performs with an average score of 21.48 out of 30 which is 71.6% correct out of 100 messages.

4.3.1 Performance Analysis

The USRP E100 version does not perform as well as the software-only version because of an FPGA strobe problem. The FPGA code used was in an early stage and has unresolved problems. Section 3.5.1 mentioned that the receiver signal is repeated every 16 times. This

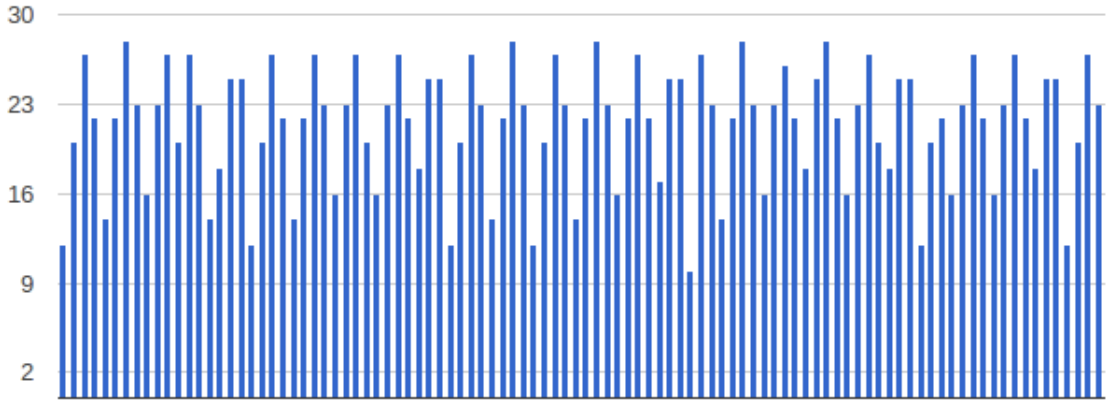

Figure 4.2: USRP E100 (standalone embedded) score for each receiving messages out of 30

Score	USRP E100 (standalone embedded)	USRP1 (all software)
Maximum	28	30
Minimun	10	30
Average	21.48	30
Good Message	73	100

Table 4.1: Score results

problem results from the strobe signal that triggers earlier than expected. A newer improved version of the FPGA code performs well on the USRP N210 but does not fit on the smaller FPGA used in the E100.

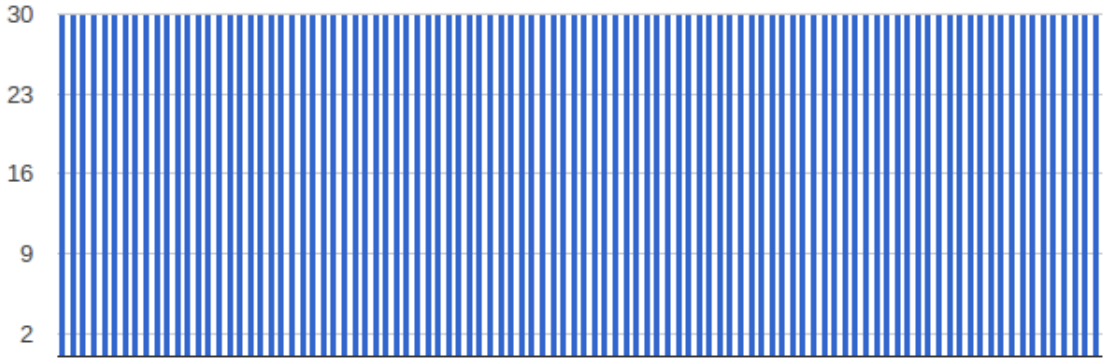

Figure 4.3: USRP1 (full software) score for each receiving messages out of 30

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Technology has become increasingly powerful and affordable along with the need for people to communicate wirelessly in more areas of the world, therefore SDR continues to grow. SDR tools such as GNU Radio and USRPs have helped developers build more devices that benefit the world of wireless technology. IEEE 802.15.4 was designed to run on a full-hardware radio and can now be implemented on a reconfigurable embedded device and all non-embedded USRP devices can communicate with ease using the standard.

The starting goal of this thesis was to explore the capability of SDR by implementing a wireless protocol standard, IEEE 802.15.4 on an embedded standalone device that is available. In order to achieve this goal many tasks were accomplished.

- Gain familiarity with SDR concepts and high-level overview of enabling technologies for SDR
- Learn how to use SDR tools such as GNU Radio and USRP
- Study the IEEE 802.15.4 wireless protocol specification
- Research existing software to aid the project
- Modify old software to work with the new hardware
- Modify FPGA hardware design from one platform to work with another
- Integrate, and debug the system
- Add transmitting and receiving capability to all USRPs and make them interoperable with a commercial device

- Create a useful SDR software (a prototype network monitor)

Many lessons have been learned. All USRP models can now communicate using the IEEE 802.15.4 protocol and inter-operate with the commercial XBee device. A standalone SDR device can now be a network monitor. This also shows that SDR has become more user-friendly and that a computer engineer with little to no prior knowledge of digital communication can build a functional monitor device from SDR.

5.2 Future Work

The performance of the new receiver requires further refinement. The message has repeating byte data and this sometimes causes the monitor to display an incorrect payload. However, after the work was started, Jeong-O Jeong developed a new version of FPGA code for the USRP N210 with many improvements including error detection and prevention of the repeated bytes coming into GNU Radio which was the problem encountered in the E100 implementation. Unfortunately, the design was too resource intensive to fit the current FPGA of the USRP E100. Since the time of the work reported in this thesis, Ettus Research has released a new USRP embedded device (USRP E110) with a larger FPGA that should fit Jeong's improved design. Therefore the quality of the receiver can still be improved. Figure 5.1 shows the FPGA comparison[9] of the two devices. The USRP E100 has an XC3SD1800A FPGA and the E110 has an XC3SD3400A FPGA.

Device	System Gates	Equivalent Logic Cells	CLB Array (One CLB = Four Slices)				Distributed RAM Bits	Block RAM Bits	DSP48As	DCMs	Maximum User I/O	Maximum Differential Pairs
			Rows	Columns	Total CLBs	Total Slices						
XC3SD1800A	1800K	37440	88	48	4160	16640	260K	1512K	84	8	519	227
XC3SD3400A	3400K	53712	104	58	5968	23872	373K	2268K	126	8	469	213

Figure 5.1: FPGAs in USRP E100 and USRP E110 (redrawn from [9], Tab. 1, p. 2)

Once the repeated byte problem in the receiver is overcome, the monitor software can also add more functionality. Adding more graphics to help users see the communication between two nodes can add substantial value to the software. Showing a real-time graph and estimating location of the nodes along with the list of available nodes would enhance utility of the software. The software could also be enhanced to have a node-trouble-shooting feature. In case a message from a transmitting node is sent to a missing node, the monitor should look for the intended receiving node on the network and notify the status of the transmitting node whether the receiving node is still in the network. There are many more features that can be added to improve the quality of the software. With the power of SDR, more specific software

can be created to fit the needs of the user. The possibility is endless with the current and future generation of SDR.

5.2.1 Future Applications

As a contribution to SDR, a step-by-step tutorial of how to get IEEE 802.15.4 working on the USRP E100 and other non-embedded USRP devices is located in Appendix A. With the monitor software made possible by SDR, many more IEEE 802.15.4 related applications can be created by using this standalone SDR system as a base such as: a temperature network monitor, a universal remote control, or a security message transmitter/receiver with different MAC frame. Since the system can receive any IEEE 802.15.4 packets, the possibility of applications is endless.

5.2.2 Future of SDR Standalone Device

Even though the GPP used in the USRP E100 is too slow for demodulating the packets by itself, the performance of small embedded computers can still improve over time according to *Moore's law*. The processor running a new embedded device will be faster and cheaper. Demodulating IEEE 802.15.4 will be achievable in the future without using the hardware co-processor; however, future SDR devices will also include more capable FPGAs, allowing developers to support higher data rates. Raspberry Pi Foundation recently released the Raspberry Pi[1] personal computers that are credit card sized with a price as low as $25 per device. Combined with the USRP B100 which is $650 results in a $675 ARM processor-based SDR compared to the USRP E100 at a price of $1300. The Raspberry Pi has a 700MHz ARM processor with a dedicated graphic processing unit (GPU).

5.3 Insights Gained

There are many lessons that people interested in the SDR field can learn in order to achieve their goals more efficiently:

- Basic understanding of Linux operation system (OS) is likely to be necessary.
  Since most tools are not available on Windows or Mac OS's, users are required to use Linux OS, know basic Linux commands and a deeper understanding of Linux OS. Working with Linux can take time as the OS is not as user-friendly as other commercial OS's. The tools available for Linux are mostly terminal-based and are harder to use than a GI, simply because users need to learn the specific command to run the tools.

[1] http://www.raspberrypi.org/

USRP E100 is a slow Linux computer and when a program does not work correctly, it takes time to find out what the problem is and fix it. Significant time spent in this thesis came from configuring the USRP E100 to work with UHD, GNU Radio, and UCLA ZigBee PHY itself, which was the starting point of the thesis.

- Take introductory SDR classes.
  The easiest way to orient to SDR is to take a class. In the case of University students who do not have to take the prerequisite, they should audit the class in order to gain a better understanding of the topic, as well as consulting with professors in the area for classes to take.

- Seek help from sources such as mailing lists and IRC chat.
  GNU Radio (`discuss-gnuradio@gnu.org`) and USRP (`usrp-users@lists.ettus.com`) mailing-lists are the best place to look for answers regarding to the specific topics. People who subscribe to the mailing-lists are willing to answer questions quickly. IRC server `irc.freenode.net` is also a place to look for a quick answer.

- Documentation can be limited or outdated
  Documentation for tools in SDR maybe outdated and sometimes non-existant. Users having problems with the tools and wish to look up answers from search engine websites, such as Google, are often directed to the mailing-list archives that are often outdated. However, if users experience similar problems to the ones previously archived, the solutions are likely to be the same.

5.4 Suggestion for Improving SDR Education and Training

Working in a Software Defined Radio, an electrical engineering related field, with a computer engineer background can be difficult. The work in this thesis can be improved in terms of features and time to accomplish most tasks with understanding of basic digital communication skill. Below are things that can be improved to make SDR more computer engineer friendly:

- Create and maintain SDR documentation.
  Documentation is very important to the people who want to start working on SDR. Instead of having to rely on the mailing-list, SDR tools such as GNU Radio and USRP should already have full documentation available on the Internet. Instructions on how to use the tools and tutorial applications should also be available to the users as well.

- Make SDR available to Windows and Mac operating systems.
  Making the tools available for Windows and Mac should attract newer users to SDR.

Linux OS is different from other operating systems that are familiar to more users and can divert time from the actual work of SDR.

- Create an SDR class for students with no communications background.
  Because of electrical engineering specific classes prerequisite to take the introduction to SDR class, many computer engineering students cannot take the class. The class should focus on the terminology, the basic operation of digital communication, and how to use the tools.

There are introductory SDR Labs that use GNU Radio created by California State University, Northridge [1]. Additional lab exercises developed by Virginia Tech and Norfolk State University with support from NSF[5, 6] use MatLab and OSSIE[7], an SDR software based on the JTRS Software Communications Architecture (SCA). These tutorials help new students learn the basics of SDR, and are a good starting point. They help describe the general terms and how basic components in SDR work. NSF labs are more helpful because the labs are created in MatLab which supports most operating systems. The tools are easy to use and do not require any extensive knowledge of how to operate the labs. Students should be able to download and run the program themselves. OSSIE tutorials, on the other hand, require students to use a Linux environment and the version of the software used in the tutorials can be confusing because of the user interface.

Bibliography

[1] California State University, Northridge GRC Tutorial. http://www.csun.edu/~skatz/katzpage/sdr_project/sdr/grc_tutorial2.pdf, 2012. [Online; accessed 2012].

[2] Creating GNU Radio Application Tutorial. http://gnuradio.org/redmine/projects/gnuradio/wiki/TutorialsWritePythonApplications, 2012. [Online; accessed 2012].

[3] Ettus Research Website. http://www.ettus.com, 2012. [Online; accessed 2012].

[4] IEEE 802.15.4-2006 Specification. http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf, 2012. [Online; accessed 2012].

[5] NSF CLLI Freshman Lab1 Basics of Data Transmission. http://ossie.wireless.vt.edu/download/labs/NSF_CCLI_Labs/IntroEngr_Lab_1_BasicsOfDataTransmission.pdf, 2012. [Online; accessed 2012].

[6] NSF CLLI Freshman Lab2 Modulation and Pulse Shaping. http://ossie.wireless.vt.edu/download/labs/NSF_CCLI_Labs/IntroEngr_Lab_2_ModulationAndPulseShaping.pdf, 2012. [Online; accessed 2012].

[7] OSSIE Virginia Tech Website. http://ossie.wireless.vt.edu/, 2012. [Online; accessed 2012].

[8] RFX2400 2.3-2.9 GHz Rx/Tx Product Infomation. https://www.ettus.com/product/details/RFX2400, 2012. [Online; accessed 2012].

[9] Spartan-3A DSP FPGA. http://www.xilinx.com/support/documentation/data_sheets/ds610.pdf, 2012. [Online; accessed 2012].

[10] USRP Documentation from GNU Radio site. http://gnuradio.org/redmine/projects/gnuradio/wiki/USRP, 2012. [Online; accessed 2012].

[11] USRP E100 Boot Files Installation Instruction. http://code.ettus.com/redmine/ettus/projects/usrpe1xx/wiki/BootFiles, 2012. [Online; accessed 2012].

[12] USRP E100 data sheet. https://www.ettus.com/product/details/UE100-KIT, 2012. [Online; accessed 2012].

[13] USRP E100 Users Frequenly Asked Questions. http://code.ettus.com/redmine/ettus/projects/usrpe1xx/wiki/FAQ, 2012. [Online; accessed 2012].

[14] USRP Firmware Images Archive and Wiki. http://code.ettus.com/redmine/ettus/projects/uhd/wiki, 2012. [Online; accessed 2012].

[15] USRP1 product information. https://www.ettus.com/product/details/USRP-PKG, 2012. [Online; accessed 2012].

[16] USRP2 product information. http://gnuradio.org/redmine/projects/gnuradio/wiki/USRP2, 2012. [Online; accessed 2012].

[17] What is SDR. http://www.wirelessinnovation.org/Introduction_to_SDR, 2012. [Online; accessed 2012].

[18] XBee-PRO 802.15.4. http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/point-multipoint-rfmodules/xbee-series1-module, 2012. [Online; accessed 2012].

[19] ZigBee/IEEE 802.15.4 Summary. http://pages.cs.wisc.edu/~suman/courses/838/papers/zigbee.pdf, 2012. [Online; accessed 2012].

[20] C.R. Aguayo Gonzalez, C.B. Dietrich, and J.H. Reed. Understanding the software communications architecture. *Communications Magazine, IEEE*, 47(9):50 –57, september 2009.

[21] Eric Blossom. How to Write a Signal Processing Bloc. http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html, 2012. [Online; accessed 2012].

[22] Christopher Bowick, Cheryl Ajluni, and John Blyler. *RF Circuit Design, Second Edition*. Newnes, 2 edition, 11 2007.

[23] E. Buracchini. The software radio concept. *Communications Magazine, IEEE*, 38(9):138 –143, sep 2000.

[24] Kresimir Dabcevic. Evaluation of software defined radio platform with respect to implementation of 802.15.4 zigbee. Master's thesis, Mlardalen University, School of Innovation, Design and Engineering, Eskilstuna, Sweden, 2011.

[25] Farid Dowla. *Handbook of RF and Wireless Technologies*. Newnes, 1 edition, 11 2003.

[26] Edoardo Paone. Open-source sca implementation embedded and software communication architecture. Master's thesis, KTH Information and Communication Technology, Stockholm, Sweden, 2010.

[27] Thomas Schmid. Gnu radio 802.15.4 en- and decoding. Technical report, 2006.

[28] Thomas Schmid. UCLA ZigBee PHY. https://www.cgran.org/wiki/UCLAZigBee, 2012. [Online; accessed 2012].

Appendix A

Instructions

A.0.1 Installation

Only GNU Radio and the UHD driver will be explained here. Install all the libraries and other tools first then install UHD driver and install GNU Radio last.

UHD Driver

```
$ git clone git://ettus.sourcerepo.com/ettus/uhd.git
$ cd uhd
$ git checkout 1eefd6f232
$ cd uhd/host/
$ mkdir build
$ cd build
$ cmake ../
$ make
$ make test
# make install
# ldconfig
```

GNU Radio

```
$ git clone http://gnuradio.org/git/gnuradio.git
$ cd gnuradio
$ git checkout 441a3767e05d15e62c519ea66b848b5adb0f4b3a
$ mkdir build
$ cd build
$ cmake  ../
$ make
# make install
```

```
# ldconfig
```

A.0.2 UCLA Zigbee PHY

```
svn co https://www.cgran.org/cgran/projects/ucla_zigbee_phy/trunk
    ucla_zigbee_phy
cd ucla_zigbee_phy
./bootstrap && ./configure && make
sudo make install
```

A.1 Other USRP Devices

The UHD driver used in the experiment is 003.003.000 and GNU Radio version is 3.4.0. USRPs need firmware and FPGA bianry to reference the version specific design. Users can make their own firmware and FPGA binary but unless users plan to develop their own version of the FPGA, downloading the pre-made images is recommended. Creating images for the USRPs take a lot of time.
The archive of the UHD images is here:
`http://code.ettus.com/redmine/ettus/projects/uhd/wiki`
Instruction on how to install on different USRPS is here:
`http://code.ettus.com/redmine/ettus/projects/uhd/wiki`

A.2 USRP E100

Most of the information is taken from the tutorial on Ettus's site that Ben Hilburn wrote. The version of tools are the same as the PC version (UHD 003.003.000, GNU Radio 3.4.0.) There are multiple newer versions of tools and if newer tools are needed, be sure to keep all the versions consistent.
Ben's tutorial link:
`http://code.ettus.com/redmine/ettus/projects/usrpe1xx/wiki/FAQ`

A.2.1 Backup the Original E100

The USRP E100 runs on a micro-sd card. Keep a backup just in case something goes wrong.

If the partitions automount, umount them (do not "safely remove the hardware").

```
$ sudo umount /path/to/mount/point/boot-xxx
$ sudo umount /path/to/mount/point/rootfs-e1xx-xxx
```

Check which device node the micro-sd card is.

```
$ sudo fdisk -l
```

You should see the SD Card near the bottom (probably), and the partitons will have a titles like /dev/sdb1 and /dev/sdb2 (for the boot and root partitions, respectively). From now on, these instructions will presume that the device node is '/dev/sdb'.

Now, you need to use the 'dd' command to make a copy of that device node. Notice that we are **not** passing in partitions, but the entire device node itself. Run:

```
$ sudo dd bs=1024 if=/dev/sdb of=sd-backup.bin
```

To burn the image back to the micro-sd card, run 'dd' command again and change 'if' and 'of' position.

A.2.2 Remove the Old UHD Driver

```
# opkg remove --force-depends uhd uhd-dev uhd-examples uhd-tests
```

A.2.3 Add a user library search path

```
# echo "/usr/local/lib" >> /etc/ld.so.conf
```

A.2.4 Install UHD Driver

```
$ git clone git://ettus.sourcerepo.com/ettus/uhd.git
$ cd uhd
$ git checkout 1eefd6f232
$ cd uhd/host/
$ mkdir build
$ cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchains/arm_cortex_a8_native.cmake
    -DENABLE_E100=ON -DENABLE_USRP_E_UTILS=TRUE ../
$ make
$ make test
# make install
# ldconfig
```

A.2.5 Update FPGA firmware

Download the same version mention in section A.1 onto the USRP E100 and do the following:

```
$ cd <install-path>/share/uhd/images
$ su
# mv usrp_e100_fpga.bin usrp_e100_fpga.bin.bak
# wget <paste link>
# tar zxvf <downloaded tarball>
# mv <tarball directory>/share/uhd/images/* .
# exit
```

A.2.6 Remove the Old GNU Radio

```
# opkg remove --force-depends gnuradio gnuradio-dev gnuradio-examples task-
    gnuradio
```

A.2.7 Install GNU Radio

```
$ git clone http://gnuradio.org/git/gnuradio.git
$ cd gnuradio
$ git checkout 441a3767e05d15e62c519ea66b848b5adb0f4b3a
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_TOOLCHAIN_FILE=../cmake/Toolchains
    /arm_cortex_a8_native.cmake -DQT_QTCORE_INCLUDE_DIR=/usr/include/qt4/
    QtCore -DQT_QTGUI_INCLUDE_DIR=/usr/include/qt4/QtGui -DQT_QMAKE_EXECUTABLE
    =/usr/bin/qmake -DENABLE_GR_QTGUI=ON -DQT_LIBRARY_DIR=/usr/lib -
    DQT_INCLUDE_DIR=/usr/include/qt4/ -DQT_MOC_EXECUTABLE=/usr/bin/moc -
    DQT_UIC_EXECUTABLE=/usr/bin/uic -DQT_RCC_EXECUTABLE=/usr/bin/rcc -
    DCMAKE_BUILD_TYPE=release ../
$ make
# make install
# ldconfig
```

A.2.8 UCLA ZigBee PHY

The direction is the same as section A.0.2. If 'svn' is not available on Angstrom, copying the source from somewhere else (re-download the source with a test PC for example) into USRP E100 is ok too.

A.3 Modifying USRP E100 FPGA

Once the E100 is able to run transmit and receive (receiving it will be too slow to process but there should not be any error other then the 'o' for "overrun" from message from the USRP. Now it is time to modify the FPGA.

As mention in section A.1, if there is a need to make change in the FPGA code, the UHD driver downloaded to the PC (section A.0.1) can generate its own. Use faster PC will help reduce the time to process.
More information here:
`http://code.ettus.com/redmine/ettus/projects/uhd/repository/revisions/master/entry/fpga/README.txt`

Navigate to `<UHD source>/usrp2/top/E1x0` and run the following command:

```
$ make -f Makefile.E100 bin
```

The Xilinx ISE project file (*.ixse inside `build` folder) will be created and the FPGA codes can now be modified. The FPGA binary will also be created. Update FPGA is the same as in section A.2.5. Import the codes provided outside of this report to ISE project. README file is available also.

A.4 Running the software

There are 3 softwares with different features.

- zigbee_monitor.py
  - Monitor the receiving IEEE 802.15.4 packets
  - Can run on USRP E100.
  - Command: `zigbee_monitor.py -c 2480M -s 4M`
- uhd_cc2420_rx.py
  - IEEE 802.15.4 receiver
  - Can run on other USRP devices
  - Command: `uhd_cc2420_rx.py -c 2480M -s 4M`
- uhd_cc2420_tx_mod.py
  - IEEE 802.15.4 transmitter

- Can run on all USRP devices
- Command: `uhd_cc2420_tx_mod.py -c 2480M -s 4M -x 1000`

Appendix B

Versions of Tools

Version of Tools Used to Test UCLA Zigbee PHY with UHD (list compiled by[24])

- Computer Specification
  - Intel(R) Core(TM)2 Duo CPU P8700 @ 2.53GHz processor
  - 4GB of RAM
  - Ubuntu 10.10 kernel 2.6.35-28 generic
- Tools
  - GNU Radio 3.4.0
  - UHD Driver 003.003.000
  - g++
  - git
  - make
  - autoconf, automake, libtool
  - sdcc
  - guile
  - ccache
- Libraries
  - python-dev
  - FFTW 3.X (fftw3, fftw3-dev)
  - cppunit (libcppunit, libcppunit-dev)

- boost 1.35
- wxWidgets(wx-common), wxPython(python-wxgtk2.8)
- python-numpy
- python-sciy
- python-matplotlib
- Numeric
- ALSA(alsa-base, libasound2, libasound2-dev)
- Qt
- SDL(libsdl-dev)
- GSL GNU Scientific Library
- SWIG 1.3.31
- QWT, QWT PLot3d libraries (optional for Qt GUI)

Appendix C

Modified Codes

C.1 ucla_ieee802_15_4_packet_sink.cc

```
/* -*- c++ -*- */
/*
 * Copyright 2004 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

/*
 * ucla_ieee802_15_4_packet_sink.cc has been derived from gr_packet_sink.cc
 *
 * Modified by: Thomas Schmid
 * March 2012 Modified by: Rithirong Thandee
 */

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

#include <ucla_ieee802_15_4_packet_sink.h>
#include <gr_io_signature.h>
#include <cstdio>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <stdexcept>
#include <cstring>
#include <gr_count_bits.h>
#include <stdio.h>
#include <iostream>

// very verbose output for almost each sample
#define VERBOSE 0
// less verbose output for higher level debugging
#define VERBOSE2 0

static const int DEFAULT_THRESHOLD = 10;  // detect access code with up to DEFAULT_THRESHOLD
     bits wrong

  // this is the mapping between chips and symbols if we do
  // a fm demodulation of the O-QPSK signal. Note that this
  // is different than the O-QPSK chip sequence from the
  // 802.15.4 standard since there there is a translation
  // happening.
  // See "CMOS RFIC Architectures for IEEE 802.15.4 Networks",
  // John Notor, Anthony Caviglia, Gary Levy, for more details.
static const unsigned int CHIP_MAPPING[] = {1618456172,
                                            1309113062,
                                            1826650030,
                                            1724778362,
                                            778887287,
                                            2061946375,
                                            2007919840,
                                            125494990,
                                            529027475,
                                            838370585,
                                            320833617,
                                            422705285,
                                            1368596360,
                                            85537272,
                                            139563807,
                                            2021988657};


inline void
ucla_ieee802_15_4_packet_sink::enter_search()
{
  if (VERBOSE)
    fprintf(stderr, "@␣enter_search\n");

  d_state = STATE_SYNC_SEARCH;
  d_shift_reg = 0;
  d_preamble_cnt = 0;
  d_chip_cnt = 0;
  d_packet_byte = 0;
}

inline void
ucla_ieee802_15_4_packet_sink::enter_have_sync()
{
  if (VERBOSE)
    fprintf(stderr, "@␣enter_have_sync\n");

  d_state = STATE_HAVE_SYNC;
  d_packetlen_cnt = 0;
  d_packet_byte = 0;
  d_packet_byte_index = 0;
}
```

```
inline void
ucla_ieee802_15_4_packet_sink::enter_have_header(int payload_len)
{
  if (VERBOSE)
    fprintf(stderr, "@ enter_have_header (payload_len = %d)\n", payload_len);

  d_state = STATE_HAVE_HEADER;
  d_packetlen  = payload_len;
  d_payload_cnt = 0;
  d_packet_byte = 0;
  d_packet_byte_index = 0;
}


inline unsigned char
ucla_ieee802_15_4_packet_sink::decode_chips(unsigned int chips){
  int i;
  int best_match = 0xFF;
  int min_threshold = 33; // Matching to 32 chips, could never have a error of 33 chips

  for(i=0; i<16; i++) {
    // FIXME: we can store the last chip
    // ignore the first and last chip since it depends on the last chip.
    unsigned int threshold = gr_count_bits32((chips&0x7FFFFFFE) ^ (CHIP_MAPPING[i]&0
        x7FFFFFFE));

    if (threshold < min_threshold) {
        best_match = i;
        min_threshold = threshold;
    }
  }

  if (min_threshold < d_threshold) {
    if (VERBOSE)
          fprintf(stderr, "Found sequence with %d errors at 0x%x\n", min_threshold, (chips&0
              x7FFFFFFE) ^ (CHIP_MAPPING[best_match]&0x7FFFFFFE)), fflush(stderr);

    return (char)best_match&0xF;
  }

  return 0xFF;
}

ucla_ieee802_15_4_packet_sink_sptr
ucla_make_ieee802_15_4_packet_sink (gr_msg_queue_sptr target_queue,
                            int threshold)
{
  return ucla_ieee802_15_4_packet_sink_sptr (new ucla_ieee802_15_4_packet_sink (target_queue
      , threshold));
}


ucla_ieee802_15_4_packet_sink::ucla_ieee802_15_4_packet_sink (gr_msg_queue_sptr target_queue
    , int threshold)
  : gr_sync_block ("sos_packet_sink",
                   gr_make_io_signature (1, 1, sizeof(float)),
                   gr_make_io_signature (0, 0, 0)),
    d_target_queue(target_queue),
    d_threshold(threshold == -1 ? DEFAULT_THRESHOLD : threshold)
{
  d_sync_vector = 0xA7;
  d_processed = 0;
```

```
  if ( VERBOSE )
    fprintf(stderr, "syncvec:␣%x,␣threshold:␣%d\n", d_sync_vector, d_threshold),fflush(
        stderr);
  enter_search();
}


ucla_ieee802_15_4_packet_sink::~ucla_ieee802_15_4_packet_sink ()
{
}

char *binary16 (unsigned int v) {
    static char binstr[17] ;
    int i ;

    binstr[16] = '\0' ;
    for (i=0; i<16; i++) {
        binstr[15-i] = v & 1 ? '1' : '0' ;
        v = v / 2 ;
    }

    return binstr ;
}

char *binary8 (unsigned int v) {
    static char binstr[9] ;
    int i ;
    binstr[8] = '\0' ;
    for (i=0; i<8; i++) {
        binstr[7-i] = v & 1 ? '1' : '0' ;
        v = v / 2 ;
    }

    return binstr ;
}


int ucla_ieee802_15_4_packet_sink::work (int noutput_items,
                     gr_vector_const_void_star &input_items,
                     gr_vector_void_star &output_items)
{
        const char *in = (const char *) input_items[0];
        char *word = (char *)malloc(4);
        int state;
        int counter = 0;
        d_payload_cnt = 0;

        for(int i=0; i<noutput_items; i++){

                std::memcpy(word, in, 4);

        ////////////////////////////////////////////////////////////////////////
                state = ((0xF8 & word[0]) >> 3);

                counter++;
                if(counter == 15)
                {
                        counter = 0;

                        if(state == 4 || state == 5) {
                                d_packet[d_packetlen_cnt++] = (short)(word[1]);
                                d_payload_cnt++;
                                d_packet_byte_index = 0;
```

```
                        }
                        if(state == 6) {
                                std::cout << std::endl;
                                // build a message
                                gr_message_sptr msg = gr_make_message(0, 0, 0,
                                    d_packetlen_cnt);
                                if(d_packetlen_cnt > 0)
                                        memcpy(msg->msg(), d_packet, d_packetlen_cnt);

                                d_target_queue->insert_tail(msg);

                //reset everything
                                msg.reset();
                                enter_search();
                enter_have_sync();
                enter_have_header(0);

                        }
                }
        ////////////////////////////////////////////////////////////////////////

                in+=4;

        }
        std::free(word);

        return noutput_items;
}
```

Listing C.1: modified ucla_ieee802_15_4_packet_sink.cc

C.2 ieee802_15_4_pkt.py

```
#
# Copyright 2005 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
# Boston, MA 02111-1307, USA.
#
# This is derived from gmsk2_pkt.py.
#
# Modified by: Thomas Schmid, Leslie Choong, Sanna Leidelof
#
# March 2012 Modified by: Rithirong Thandee
```

```
#import Numeric
import numpy

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
    @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

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')
        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 = struct.pack("H", 0xCC41)

            pkt = make_ieee802_15_4_packet(FCF,
                                           seqNr,
                                           addressInfo,
                                           payload,
                                           self.pad_for_usrp)
            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):
        """
        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_float), # Input
                                gr.io_signature(0, 0, 0))  # Output


        self._rcvd_pktq = gr.msg_queue()          # holds packets from the PHY

        self._packet_sink = ucla.ieee802_15_4_packet_sink(self._rcvd_pktq, self.threshold)

        self.connect(self, 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:

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

            if(len(payload) > 0):

                """
                dest = ord(payload[0])*256 + ord(payload[1])
                source = ord(payload[2])*256 + ord(payload[3])
                length = ord(payload[4])
                group = ord(payload[5])
                am_type = ord(payload[6])
                msg_payload = payload[7:7+length]
                crc = ord(payload[-1])

                print "dest: ", dest, "\n"
                print "source: ", source, "\n"
                print "length: ", length, "\n"
                print "group: ", group, "\n"
```

```
                print "msg_payload: ", msg_payload, "\n"
                print "crc: ", crc, "\n"
                """

                if self.callback:
                    self.callback(1, payload)
```

Listing C.2: modified ieee802_15_4_pkt.py

C.3 cc2420_rxtest_uhd.py

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

#
# Decoder of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Leslie Choong, Mikhail Tadjikov
#
# March 2012 Modified by: Rithirong Thandee

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

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
        payload_size = 128              # bytes

        self.u = uhd.usrp_source(device_addr="",
                                 io_type=uhd.io_type.COMPLEX_FLOAT32,
                                 num_channels=1,
                             )
        self.u.set_gain(options.gain)
        self.u.set_samp_rate(options.sample_rate)
        self.u.set_center_freq(options.cordic_freq)

        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.connect(self.u, 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 ("-c", "--cordic-freq", type="eng_float", default=2480000000,
                       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=4000000)
    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()

    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 ()
```

Listing C.3: regular UHD cc2420_rxtest_uhd.py

C.4 cc2420_rxtest_uhd_e100.py

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

#
# Decoder of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Leslie Choong, Mikhail Tadjikov
#
# March 2012 Modified by: Rithirong Thandee

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

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
        payload_size = 128              # bytes

        u = uhd.usrp_source(device_addr="",
                            io_type=uhd.io_type.COMPLEX_INT16,
                            num_channels=1,
                           )

        self.u = u
        self.u.set_gain(options.gain)
        self.u.set_samp_rate(options.sample_rate)
        self.u.set_center_freq(options.cordic_freq)

        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.connect(self.u, 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)
        if len(payload) > 1 :
            print "payload: " + str(map(hex, map(ord, payload)))
            print " ------------------------"
            sys.stdout.flush()


    parser = OptionParser (option_class=eng_option)
    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=4000000)
    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()

    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 ()
```

Listing C.4: E100 UHD cc2420_rxtest_uhd_e100.py

C.5 cc2420_txtest_uhd.py

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

#
# Transmitter of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Sanna Leidelof
#
# March 2012 Modified by: Rithirong Thandee

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._spb = 2

        self.u = uhd.usrp_sink(device_addr="",
                               io_type=uhd.io_type.COMPLEX_FLOAT32,
                               num_channels=1)

        self.u.set_samp_rate(options.sample_rate)
        self.u.set_center_freq(options.cordic_freq)
        self.u.set_gain(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)
```

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

    def send_pkt(self, payload=' ', eof=False):
        return self.packet_transmitter.send_pkt(0xe5,
                                                struct.pack("HHHHHHHHH",
                                                            #PAN ID
                                                            0x5678,
                                                            #addresss1
                                                            0x5A70,
                                                            0x4063,
                                                            0xA200,
                                                            0x0013,
                                                            #address2
                                                            0x5A22,
                                                            0x4063,
                                                            0xA200,
                                                            0x0013),
                                                payload,
                                                eof)

def main ():


    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-c", "--cordic-freq", type="eng_float", default=2415000000,
                       help="set Tx cordic frequency to FREQ", metavar="FREQ")
    parser.add_option ("-r", "--data-rate", type="eng_float", default=2000000)
    parser.add_option ("-g", "--gain", type="eng_float", default=0,
                       help="set Rx PGA gain in dB [0,20]")
    parser.add_option ("-s", "--sample_rate", type="eng_float", default=1000000)
    parser.add_option ("-x", "--num_msg", type="int", default=1000)
    parser.add_option ("-X", "--spacing", type="eng_float", default=0.001)

    (options, args) = parser.parse_args ()

    tb = transmit_path(options)
    tb.start()

    for i in range(options.num_msg):
        print "send message %d:" %(i)
        #Hello World = 48:65:6c:6c:6f:20:57:6f:72:6c:64
        tb.send_pkt(payload=struct.pack('11B', 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x57, 0
            x6f, 0x72, 0x6c, 0x64))
        time.sleep(options.spacing)

    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 ()
```

Listing C.5: regular UHD cc2420_txtest_uhd.py

C.6 cc2420_txtest_uhd_e100.py

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

#
# Transmitter of IEEE 802.15.4 RADIO Packets.
#
# Modified by: Thomas Schmid, Sanna Leidelof
#
# March 2012 Modified by: Rithirong Thandee

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._spb = 2

        self.u = uhd.usrp_sink(device_addr="",
                               io_type=uhd.io_type.COMPLEX_FLOAT32,
                               num_channels=1)

        self.u.set_samp_rate(options.sample_rate)
        self.u.set_center_freq(options.cordic_freq)
        self.u.set_gain(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)

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

    def send_pkt(self, payload='␣', eof=False):
        return self.packet_transmitter.send_pkt(0xe5,
                                                struct.pack("HHHHHHHHH",
                                                            #PAN ID
                                                            0x5678,
                                                            #addresss1
                                                            0x5A70,
                                                            0x4063,
                                                            0xA200,
                                                            0x0013,
                                                            #address2
                                                            0x5A22,
                                                            0x4063,
                                                            0xA200,
                                                            0x0013),
                                                payload,
                                                eof)

def main ():
```

```
    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-c", "--cordic-freq", type="eng_float", default=2415000000,
                       help="set␣Tx␣cordic␣frequency␣to␣FREQ", metavar="FREQ")
    parser.add_option ("-r", "--data-rate", type="eng_float", default=2000000)
    parser.add_option ("-g", "--gain", type="eng_float", default=0,
                       help="set␣Rx␣PGA␣gain␣in␣dB␣[0,20]")
    parser.add_option ("-s", "--sample_rate", type="eng_float", default=1000000)
    parser.add_option ("-x", "--num_msg", type="int", default=1000)
    parser.add_option ("-X", "--spacing", type="eng_float", default=0.001)

    (options, args) = parser.parse_args ()

    tb = transmit_path(options)
    tb.start()

    for i in range(options.num_msg):
        print "send␣message␣%d:" %(i)
        #Hello World = 48:65:6c:6c:6f:20:57:6f:72:6c:64
        tb.send_pkt(payload=struct.pack('11B', 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x57, 0
            x6f, 0x72, 0x6c, 0x64))
        time.sleep(options.spacing)

    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 ()
```

Listing C.6: E100 UHD cc2420_txtest_uhd_e100.py

C.7 zigbee_monitor.py

```
#!/usr/bin/env python
# March 2012 Modified by: Rithirong Thandee

import pygtk
pygtk.require('2.0')
import gtk, gobject

#GNU Radio includes
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
#GUI thread includes
from threading import Thread
import thread
gobject.threads_init()

#global variables
window    = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

```
model     = gtk.ListStore(gobject.TYPE_STRING, gobject.TYPE_STRING,
            gobject.TYPE_STRING, gobject.TYPE_STRING, gobject.TYPE_STRING, gobject.
                TYPE_STRING)
node_list = gtk.ListStore(gobject.TYPE_STRING)
u         = uhd.usrp_source(device_addr="",
                            io_type=uhd.io_type.COMPLEX_INT16,
                            num_channels=1)
#tb       = None

def contains(thelist, item):
    cats = list ()
    item = list_categories.get_iter_first ()

    while ( item != None ):
        cats.append (list_categories.get_value (item, 0))
        item = list_categories.iter_next(item)

def rx_callback(ok, payload):
        global window

        if len(payload) >= 22:

            if 1 == 0 : #frame control field

                frame_ctrl = ord(payload[1])*256 + ord(payload[0])

                frame_type =            frame_ctrl & 0x3
                sec_enable =            (frame_ctrl >> 3) & 0x01
                frame_pending =         (frame_ctrl >> 4) & 0x01
                ack_req =               (frame_ctrl >> 5) & 0x01
                panID_compress =        (frame_ctrl >> 6) & 0x01
                dest_address_mode =     (frame_ctrl >> 10) & 0x03
                frame_version =         (frame_ctrl >> 12) & 0x03
                source_address_mode =   (frame_ctrl >> 14) & 0x03


                sys.stdout.write( "FRAME␣TYPE:␣(%x)␣" % frame_type  )
                if frame_type == 0 :
                    print "beacon"
                elif frame_type == 1 :
                    print "data"
                elif frame_type == 2 :
                    print "ack"
                elif frame_type == 3 :
                    print "mac␣command"
                else :
                    print "unknown"

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

                sys.stdout.write( "SECURITY␣ENABLE:␣(%x)␣" % sec_enable )
                if sec_enable == 1 :
                    print "yes"
                elif sec_enable == 0 :
                    print "no"

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

                sys.stdout.write( "FRAME␣PENDING:␣(%x)␣" % frame_pending )
                if frame_pending == 1 :
                    print "yes␣(has␣more␣data)"
                elif frame_pending == 0 :
                    print "no"
```

```
    #----------------------------------------

    sys.stdout.write( "ACK␣REQUIRED:␣(%x)␣" % ack_req )
    if ack_req == 1 :
        print "yes"
    elif ack_req == 0 :
        print "no"

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

    sys.stdout.write( "PAN␣ID␣COMPRESSION:␣(%x)␣" % panID_compress )
    if panID_compress == 1:
        print "yes"
    elif panID_compress == 0:
        print "no"

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

    sys.stdout.write( "DESTINATION␣ADDRESS␣MODE:␣(%x)␣" % dest_address_mode )
    if dest_address_mode == 0 :
        print "PAN␣identifier␣and␣address␣fields␣are␣not␣present"
    elif dest_address_mode == 1 :
        print "Reserved"
    elif dest_address_mode == 2 :
        print "Address␣field␣contains␣a␣16-bit␣short␣address"
    elif dest_address_mode == 3 :
        print "Address␣field␣contains␣a␣64-bit␣extended␣address"
    else :
        print "unknown"

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

    sys.stdout.write( "FRAME␣VERSION:␣(%x)␣" % frame_version )
    if frame_version == 0 :
        print "IEEE␣Std␣802.15.4-2003"
    elif frame_version == 1 :
        print "RIEEE␣802.15.4␣frame"
    else :
        print "unknown"

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

    sys.stdout.write( "SOURCE␣ADDRESS␣MODE:␣(%x)␣" % source_address_mode )
    if source_address_mode == 0 :
        print "PAN␣identifier␣and␣address␣fields␣are␣not␣present"
    elif source_address_mode == 1 :
        print "Reserved"
    elif source_address_mode == 2 :
        print "Address␣field␣contains␣a␣16-bit␣short␣address"
    elif source_address_mode == 3 :
        print "Address␣field␣contains␣a␣64-bit␣extended␣address"
    else :
        print "unknown"

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

    freq_num = ord(payload[2])

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

if 1 == 1 : #PAN ID field

    pan_id = ord(payload[4])*256 + ord(payload[3])
```

```
    #----------------------------------------

if 1 == 1 : #addressing field
    #dest upper address = [12][11][10][9]
    #dest lower address = [8][7][6][5]
    dest_upper_address = ord(payload[12+1])*16777216 + ord(payload[11+1])*65536
        + ord(payload[10+1])*256 + ord(payload[9+1])
    dest_lower_address = ord(payload[8+1])*16777216 + ord(payload[7+1])*65536 +
        ord(payload[6+1])*256 + ord(payload[5+1])

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

    pan_id_src = ord(payload[13+1]) & 0xff

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

    #source upper address = [20][19][18][17]
    #source lower address = [16][15][14][13]
    source_upper_address = ord(payload[21])*16777216 + ord(payload[20])*65536 +
        ord(payload[19])*256 + ord(payload[18])
    source_lower_address = ord(payload[17])*16777216 + ord(payload[16])*65536 +
        ord(payload[15])*256 + ord(payload[14])

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

    payload_index = 22
    the_message = ""
    while (payload_index < (len(payload))) :
        tmp_char = chr(ord(payload[payload_index]) & 0xff)
        the_message += tmp_char
        payload_index = payload_index+1

global u
rssi = u.get_dboard_sensor("rssi")
#print "RSSI = %s" %(rssi)
rssi_s = str(rssi)
rssi_s = rssi_s.lstrip('RSSI: ')
rssi_s = rssi_s.rstrip(' dBm')

global model
iter = model.prepend()
model.set(iter, 0, time.strftime('%X'), 1, str(hex(source_lower_address)), 2,
        str(hex(dest_lower_address)), 3, str(hex(pan_id)) ,4, rssi_s, 5,
            the_message)

global node_list
#iter = node_list.prepend()
#node_list.set(iter, 0, str(hex(source_lower_address)))
#node_list.set(iter, 0, str(hex(dest_lower_address)))
contains = 0
for row in node_list :
    if row[0] == str(hex(source_lower_address)):
        contains = 1
        break
if contains == 0:
    iter = node_list.prepend()
    node_list.set(iter, 0, str(hex(source_lower_address)))

contains = 0

for row in node_list:
    if row[0] == str(hex(dest_lower_address)):
        contains = 1
        break
```

```
            if contains == 0:
                iter = node_list.prepend()
                node_list.set(iter, 0, str(hex(dest_lower_address)))

        else :
            print "bad message"

        sys.stdout.flush()
        window.show_all ()

class oqpsk_rx_graph (gr.top_block):
    def __init__(self, options, rx_callback):

        super(oqpsk_rx_graph, self).__init__()

        gr.top_block.__init__(self)

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

        self.data_rate = options.data_rate
        self.samples_per_symbol = 2
        self.fs = self.data_rate * self.samples_per_symbol
        payload_size = 128 #bytes

        global u
        self.u = u

        #Set sampling rate
        self.u.set_samp_rate(options.sample_rate)

        #Set and read the center frequency
        self.u.set_center_freq(uhd.tune_request(options.cordic_freq,0))

        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.connect(self.u, self.packet_receiver)

def update(self, widget, source, dest, pan, rssi, msg):
    global model
    iter = model.prepend()
    model.set(iter, 0, time.ctime(), 1, source, 2, dest, 3, pan, 4, rssi, 5, msg)

class Monitor:


    # Create the list of "messages"
    def create_list1(self):
        global model
        # Create a new scrolled window, with scrollbars only if needed
        scrolled_window = gtk.ScrolledWindow()
        scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)

        tree_view = gtk.TreeView(model)
        scrolled_window.add_with_viewport (tree_view)
        tree_view.show()

        time_cell = gtk.CellRendererText()
        time_col = gtk.TreeViewColumn("Time", time_cell, text=0)
```

```
        source_cell = gtk.CellRendererText()
        source_col = gtk.TreeViewColumn("Source", source_cell, text=1)

        dest_cell = gtk.CellRendererText()
        dest_col = gtk.TreeViewColumn("Dest", dest_cell, text=2)

        pan_cell = gtk.CellRendererText()
        pan_col = gtk.TreeViewColumn("panID", pan_cell, text=3)

        rssi_cell = gtk.CellRendererText()
        rssi_col = gtk.TreeViewColumn("RSSI", rssi_cell, text=4)

        msg_cell = gtk.CellRendererText()
        msg_col = gtk.TreeViewColumn("Message", msg_cell, text=5)

        tree_view.append_column(time_col)
        tree_view.append_column(source_col)
        tree_view.append_column(dest_col)
        tree_view.append_column(pan_col)
        tree_view.append_column(rssi_col)
        tree_view.append_column(msg_col)

        return scrolled_window

    def create_list(self):
        # Create a new scrolled window, with scrollbars only if needed
        scrolled_window = gtk.ScrolledWindow()
        scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)

        global node_list
        tree_view = gtk.TreeView(node_list)
        scrolled_window.add_with_viewport (tree_view)
        tree_view.show()

        # Add some messages to the window
        #for i in range(10):
        #    msg = "Message #%d" % i
        #    iter = node_list.append()
        #    node_list.set(iter, 0, msg)

        cell = gtk.CellRendererText()
        column = gtk.TreeViewColumn("Address", cell, text=0)
        tree_view.append_column(column)

        return scrolled_window

    def delete_event(self, widget, event, data=None):
        print "delete␣event␣occurred"
        return False

    def destroy(self, widget, data=None):
        print "destroy␣signal␣occurred"
        gtk.main_quit()
        sys.exit(0)

    def createGUI(self):
        print "creating␣GUI"
        global window
        global mainWindow
        window.connect("delete_event", self.delete_event)
        window.connect("destroy", self.destroy)
        window.connect("destroy", lambda w: gtk.main_quit())
        window.set_title("Monitor")
```

```
window.set_size_request(900, 550)

main_hbox = gtk.HBox(False, 2)
window.add(main_hbox)

left_vbox = gtk.VBox(False, 4)
left_vbox.set_size_request(200,550)

curr_freq_txt = gtk.Label("Current␣Freq")
#left_vbox.pack_start(curr_freq_txt, False, False, 10)

curr_freq_num = gtk.Entry()
curr_freq_num.set_max_length(4)
curr_freq_num.set_text("2480")
#left_vbox.pack_start(curr_freq_num, False, False, 0)

node_list_txt = gtk.Label("Node␣List")
left_vbox.pack_start(node_list_txt, False, False, 0)

node_list_num = gtk.VPaned()
left_vbox.pack_start(node_list_num, True, True, 10)
mylist = self.create_list()
node_list_num.add1(mylist)

main_hbox.pack_start(left_vbox, True, True, 10)

right_vbox = gtk.VBox(False, 5)
right_vbox.set_size_request(700,550)

act_txt = gtk.Label("Activities")
right_vbox.pack_start(act_txt, False, False, 0)

act_list = gtk.VPaned()
right_vbox.pack_start(act_list, True, True, 10)
mylist1 = self.create_list1()
act_list.add1(mylist1)

addr_hbox = gtk.HBox(False, 6)

dest_txt = gtk.Label("Destination␣Address:")
addr_hbox.pack_start(dest_txt, False, False, 0)

dest_num = gtk.Entry()
dest_num.set_max_length(16)
dest_num.set_text("0013A2004064F30D")
addr_hbox.pack_start(dest_num, True, True, 0)

panID_txt = gtk.Label("Pan␣ID:")
addr_hbox.pack_start(panID_txt, False, False, 0)

panID_num = gtk.Entry()
panID_num.set_max_length(4)
panID_num.set_text("ABCD")
addr_hbox.pack_start(panID_num, False, False, 0)

freq_txt = gtk.Label("Freq:")
addr_hbox.pack_start(freq_txt, False, False, 0)

freq_num = gtk.Entry()
freq_num.set_max_length(4)
freq_num.set_text("2480")
addr_hbox.pack_start(freq_num, False, False, 0)

#right_vbox.pack_start(addr_hbox, False, False, 0)
```

```
        msg_hbox = gtk.HBox(False, 3)

        msg_txt = gtk.Label("Message:␣")
        msg_hbox.pack_start(msg_txt, False, False, 10)

        msg_num = gtk.Entry()
        msg_num.set_max_length(20)
        msg_num.set_text("hello␣world")
        msg_hbox.pack_start(msg_num, True, True, 10)

        send_button = gtk.Button("Send")
        send_button.connect("clicked", update, "1", "2", "3", "4")
        msg_hbox.pack_start(send_button, True, True, 10)

        start_stop_hbox = gtk.HBox(False, 3)
        """
        start_button = gtk.Button("Start")
        start_button.connect("clicked",self.start_radio)
        start_stop_hbox.pack_start(start_button, True, True, 10)
        """
        stop_button = gtk.Button("Stop␣Operation")
        stop_button.connect("clicked",self.stop_radio)
        start_stop_hbox.pack_start(stop_button, True, True, 10)

        right_vbox.pack_start(start_stop_hbox, False, False, 10)

        main_hbox.pack_start(right_vbox, False, False, 10)

        print "showing␣all␣window␣elemets"
        window.show_all ()

    def start_radio(self, tmp):
        self.tb.start()
        print "start"
    def stop_radio(self, tmp):
        self.tb.stop()
        print "stop"
    def infinitePrint(self):
        while(True):
            pass #print "test"

    def __init__(self):
        #self.createGUI()
        #self.startRadio()

        parser = OptionParser (option_class=eng_option)
        parser.add_option ("-c", "--cordic-freq", type="eng_float", default=2480000000,
                           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=4000000)
        parser.add_option ("-g", "--gain", type="eng_float", default=0,
                           help="set␣Rx␣PGA␣gain␣in␣dB␣[0,20]")

        (options, args) = parser.parse_args()

        #global tb
        self.tb = oqpsk_rx_graph(options, rx_callback)
        self.tb.start()

        self.createGUI()

    def main(self):
        gtk.main()
```

```
mainWindow = Monitor()

if __name__ == "__main__":
    global mainWindow
    mainWindow.main()
```

Listing C.7: zigbee_monitor.py