Studying Media Access and Control Protocols

ALALELDDIN MOHAMMED

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2009:206

Studying Media Access and Control Protocols

Alalelddin Mohammed

Email: hdm06amo@sth.kth.se

Master of Science Thesis

Date: 2010-01-19

Supervisor and Examiner

Prof. Gerald Q. Maguire Jr.

Abstract

This thesis project's goal is to enable undergraduate students to gain insight into media access and control protocols based upon carrying out laboratory experiments. The educational goal is to de-mystifying radio and other link and physical layer communication technologies as the students can follow packets from the higher layers down through the physical layer and back up again.

The thesis fills the gap between the existing documentation for the Universal Software Radio Peripheral (USRP) resources and the knowledge of undergraduate students. This was necessary because the existing document is targeted at advanced audiences rather than undergraduates. This thesis describes the design and evolution of a workbench for students to experiment with a variety of media access and control protocols, much as Wireshark gives students the ability to watch network and higher layer protocols. Another motivation for this thesis is that an increasing number of communication networks use complex media access and control protocols and existing tools do not allow students to see the details of what is taking place in these protocols, except via simulation. Today an software defined radio and computer are affordable as laboratory equipment for an undergraduate course. Hence the time is ripe for the development of undergraduate laboratory course material using these tools.

The thesis is targeted at (1) instructors of undergraduates who might use this work to develop their own lesson plans and course material and (2) students of physical and link layer protocols who want a practical tool for carrying out experiments in these layers. Hopefully by de-mystifying these lower layers and by making the USRP more approachable by undergraduate students we will encourage lots of students to view wireless network technology as being just as approachable as a wired Ethernet.

Due to the widespread use of wireless communications technologies, there is a great need by industry for more graduates who can understand communication systems from the physical to the application layer - rather than the current situation where there is a hard boundary between the lower two layers and the upper layers. While there has been a lot of research concerning cross layer optimization, much of this is theoretical and not very approachable by students. A desired outcome of this thesis project is that undergraduate students will be able to understand tradeoffs at all layers of the protocol stack and not be limited to the upper layers.

Sammanfattning

Detta examensarbete har som mål att göra det möjligt för studenter att få inblick i tillgång till medierna och protokoll som grundar sig på att utföra laboratorieexperiment. Det pedagogiska målet är att de-mystifierande radio och annan länk och fysiska lagret kommunikationsteknik som studenterna kan följa paket från högre skikt ner genom det fysiska lagret och upp igen.

Avhandlingen fyller gapet mellan den befintliga dokumentationen för Universal Software Radio Peripheral (usrp) resurser och kunskap om studerande. Detta var nödvändigt eftersom det befintliga dokument riktar sig till avancerade publik snarare än studenter. Denna avhandling beskriver utformningen och utvecklingen av en arbetsbänk för studenter att experimentera med olika tillgång till medierna och protokoll kontroll, mycket som Wireshark ger studenterna möjlighet att titta på nätet och högre skikt protokoll. Ett annat motiv för denna tes är att ett ökande antal kommunikationsnät använda komplicerade tillgång till medierna och protokoll kontroll och befintliga verktyg inte tillåter eleverna att se detaljer om vad som sker i dessa protokoll, utom via simulering. Idag en programvarustyrd radio och dator är överkomliga laboratorieutrustning för en grundutbildningskurs. Därför är tiden mogen för utvecklingen av grundutbildningen laborationer material med hjälp av dessa verktyg.

Avhandlingen riktar sig till (1) instruktörer för studenter som kan använda detta arbete för att utveckla sin egen lektionsplanering och kursmaterial och (2) studenter på fysisk och länka protokoll skikt som vill ha ett praktiskt verktyg för att utföra experiment i dessa lager. Förhoppningsvis genom de-mystifierande de undre lagren och genom att göra usrp mer tillgänglig genom att studenter ska vi uppmuntra många elever att visa trådlös nätverksteknik vara lika lättillgänglig som ett ethernet.

På grund av den utbredda användningen av trådlös kommunikationsteknik, finns ett stort behov från näringslivet för fler studenter som kan förstå kommunikationssystem från det fysiska till applikationslagret - i stället för den nuvarande situationen där det finns en hård gräns mellan de två lägre skikten och de övre skikten. Samtidigt som det har varit en hel del forskning om cross lager optimering, mycket av detta är teoretisk och inte särskilt tillgänglig av studenter. Ett önskat resultat med detta examensarbete är att studenter ska kunna förstå kompromisser på alla nivåer inom den protokollstack och inte vara begränsade till de övre skikten.

Acknowledgement

Let us think! We always look on the bright side of life …. From him I learn Knowledge not in the books! What to think and how to look for the best solution! When I deliver my problems to him, the feedback was always new knowledge acquired to make this work possible. ***Chip Maguire*** *my supervisor Thank you…. we swim in your knowledge.*

Table of Contents

Table of Figures

List of Code Examples

List of Tables

List of Acronyms and Abbreviations

GHz	Giga Hertz
Hz	Hertz
MAC	Media Access and Control
RF	Radio Frequency
SDR	Software-Defined Radio
SR	Software Radio
SWING	Simplified Wrapper and Interface Generator
USRP	Universal Software Radio Peripheral

1. Introduction

The idea of a software-defined radio (SDR) is that all the modulation and demodulation is done via software, rather than by specialized circuits. The benefit according to *Susan Karlin* is “instead of having to build extra circuitry to handle different types of radio signals, you can just load an appropriate program” [1]. An SDR uses programmable digital devices to accomplish the signal processing, instead of fixed hardware.

SDR introduces flexibility and rapid development to radio communication systems by using a software-oriented approach. As software-based approach offers greater flexibility when developing wireless communication systems, since the wireless system architecture is **not** frozen into the hardware, but can be changed at any time via changing the software which is loaded into the device. By delaying the binding of design decisions until execution time, the designers can incorporate the latest developments - enabling them to improve the performance of the systems. This reduces the difference between the state of the art and the state of practice for wireless communication systems. Additionally, this software-oriented approach to wireless communication devices allows both flexibility and simpler maintenance, as most upgrades can be done by loading new software, rather than changing physical modules.

The Ettus Research Universal Software Radio Peripheral (USRP) is an example of an SDR. It provides an “RF front end for a computer running the GNU Radio software, converting radio waves picked up by an antenna into digital copies that the computer software can handle or, conversely, converting a wave synthesized by the computer into a radio transmission” [1]. This device can also be viewed as a general purpose front end for receiving and generating all sorts of different kinds of signals (see Figure 1). In Figure 1, "IF" standards for intermediate frequency, representing a version of the signal at a lower frequency that the actual RF. Note that the bandwidth of the signal will need to be at least twice the radio frequency bandwidth to avoid aliasing (As per Shannon's sampling theorem.)

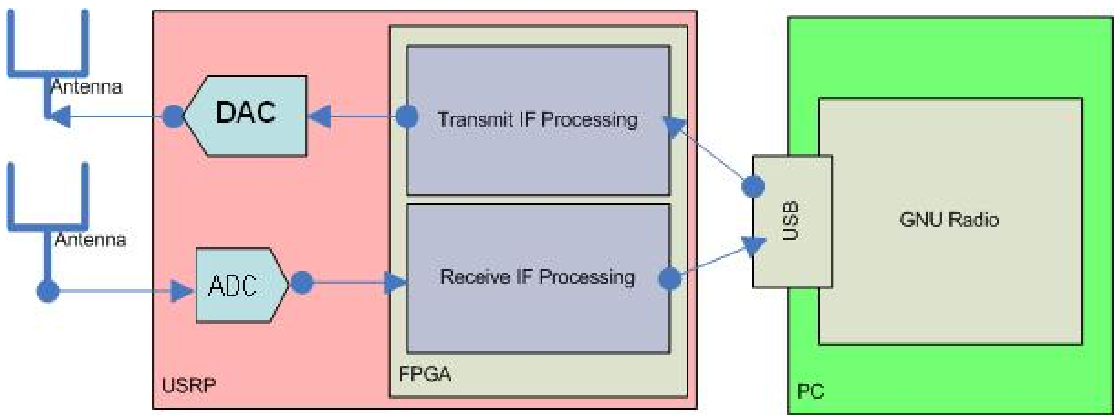


Figure 1: Introduction to USRP and GNU Radio

The USRP motherboard can have up to two transmitters and two receivers that can simultaneously transmit and receive from antennas (or wired connections) in real time. There are various types of daughterboards that can be plugged into the USRP motherboard to provide an interface between the baseband signal and a number of different frequency ranges. USRP was design to operate in a number of different portions of the spectrum ranging from 0 Hz to 2.9 GHz. This wide range covers a large variety of different applications. In this thesis we will take advantage of this hardware platform to enable students to both observe and create a number of different media access and control protocols.

1.2 Master Thesis Overview

This master's thesis will design, implement, and evaluate a number of lab exercises for undergraduate students using the USRP technology to understand a number of different media access and control (MAC) protocols. Lab exercises will explore different types of signals and MAC protocols. This thesis contains this introduction, followed by chapter 2 that provides some basic background information concerning SDR. Chapter 3 describes the particular SDR hardware platform that we have chosen (i.e., the USRP). Chapter 4 describes the GNU Radio software that we have built upon. Chapter 5 describes some of the laboratory exercises that have been designed during this project. Chapter 6 evaluates these laboratory exercises from a pedagogical point of view. While Chapter 7 presents our conclusions and suggests some future work. A number of appendices are included, containing the complete laboratory exercises; along with details for the student (or instructor) on how to set up a suitable laboratory environment.

1.3 Master Thesis goal

This thesis project has two goals:

1. Show a software defined radio application built on USRP and GNU Radio.
2. Develop laboratory exercises for undergraduate students, using USRP and GNU Radio to explain the physical and MAC layers using examples drawn from popular networks that the students are likely to encounter. These exercises cover different applications with both wired network technology and several wireless communication technologies.

2. Background

This chapter introduces software defined radio – beginning with some of its history, moving on to a discussion of underlying hardware architecture of an SDR, and describing the role of the SDR forum in the development of SDR.

2.1 Software Defined Radio (SDR) History

A SDR is a radio in which software defines signals, frequencies, modulation, and (optionally) cryptography. SDR design began 1987, when the United States Air Force’s Rome Laboratory (AFRL) developed a programmable modem. The modem was based on the Integrated Communications, Navigation, and Identification Architecture (ICNIA) [3]. Despite of this earlier effort, Walter Tuttlebee argues that “Until the mid-1990’s most readers would probably not have even come across the term SDR”. The term software defined radio was introduced by Joseph Mitola III in 1991 "to signal the shift from digital radio to multiband multimode software-defined radios where "80%" of the functionality is provided in software, versus the "80%" hardware of the 1990's." [23]. Table 1 shows the time line of the development of software defined radio.

Table 1: SDR time line with some representation examples[3] [16]

Prject	Year	Size	Features
ICNIA	1978	Fit in a small room	A collection of several single-purpose radios in one box
Speakeasy Phase I	1992	Six foot (182 cm) rack	Included a programmable cryptography chip.
Speakeasy Phase II	1995	Stack of two pizza boxes	The first SDR to include a voice coder and digital signal processing resources.
Digital Modular Radio	Early 2000	44.45 x 48.90 x 55.9 cm	Implemented four full duplex channels and could be remotely controlled using the Simple Network Management Protocol via an Ethernet interface.
USRP	2004	Fit in 21 x 17 x 5.5 cm box	Allows creating a software radio using any computer with a USB2 port. Various plug-on daughterboards allow the USRP to be used on different radio frequency bands.

As shown in Table 1, SDR evolved from very large (and power hungry devices) to small man portable devices. Additionally, they evolved from very expensive prototypes to systems costing less than 1k€. Today an SDR and computer are affordable as laboratory equipment for an undergraduate course; hence the time is ripe for the development of undergraduate laboratory course material.

2.2 Modern Software Defined Radio

2.2.1 Hardware Architecture

The basic hardware architecture of a SDR includes a radio front-end, modem, and application functions (see Figure 2; where the modem and application functions have been grouped together into a “Digital end” module). Additionally, there needs to be a means for connecting to network services and for remote management. The following subsections will discuss each of these elements of the SDR.

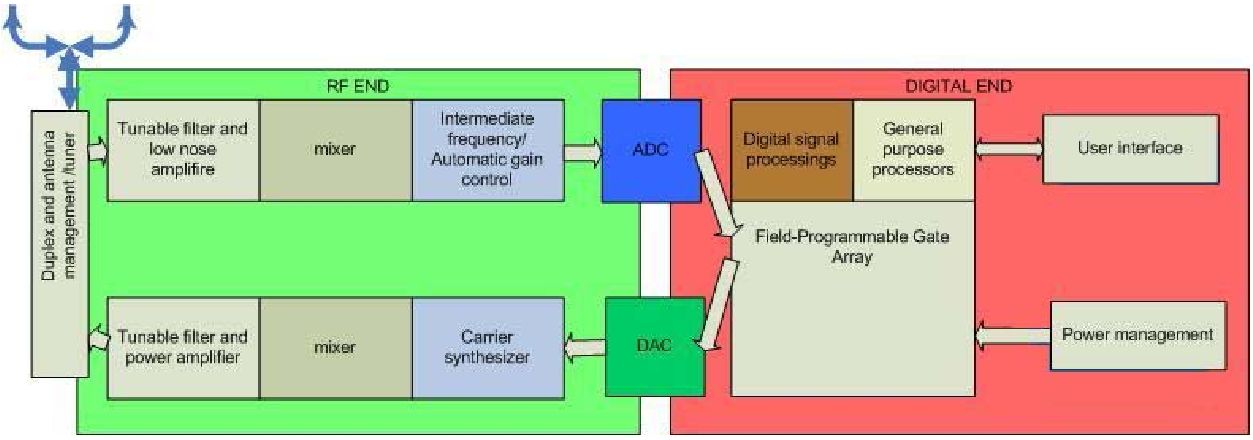


Figure 2: Basic hardware architecture of a modern SDR.

2.2.1.1 RF Front-End

The radio frequency (RF) front-end consists of functions to support transmit and receive modes. Note that some instance of a SDR might be receive mode only or transmit mode only. The receive mode utilizes:

- Antenna-matching unit
- Low-noise amplifier
- Filters
- Local oscillators
- analog-to-digital converters (ADCs).

This RF front end utilizes filters to reject (or reduce) undesired signals. An important part of this filtering is to prevent high frequency signals from being aliased into the digitized bandwidth of the ADC.

The transmit mode utilizes:

- Antenna-matching units
- Filters
- Local oscillator
- One or more digital-to-analog converters (DACs)

The duplexer shown in Figure 2 is to enable the transmit and receiver subsystems to share an antenna, while avoiding overwhelming the receiver with the high power transmit signal.

2.2.1.2 The modem

The modulator/demodulator (modem) modulates signals to be transmitted or demodulates received signals. The modem process to receive signals is basically the inverse of the process used to modulate the signal to be transmitted. Figure 3 shows how this signal processing function is performed by the modem.

In Figure 3, bits are taken from a higher layer (such as a network layer packet) for transmission, grouped into frames, redundancy is added to enable error correction by the receiver, the bits are mapped to sample(s), and a specific wave is used to provide the selected representation of each symbol. Additional processing is performed to provide desirable physical properties and the signal may be multiplexed with other signals before being passed to the DAC.

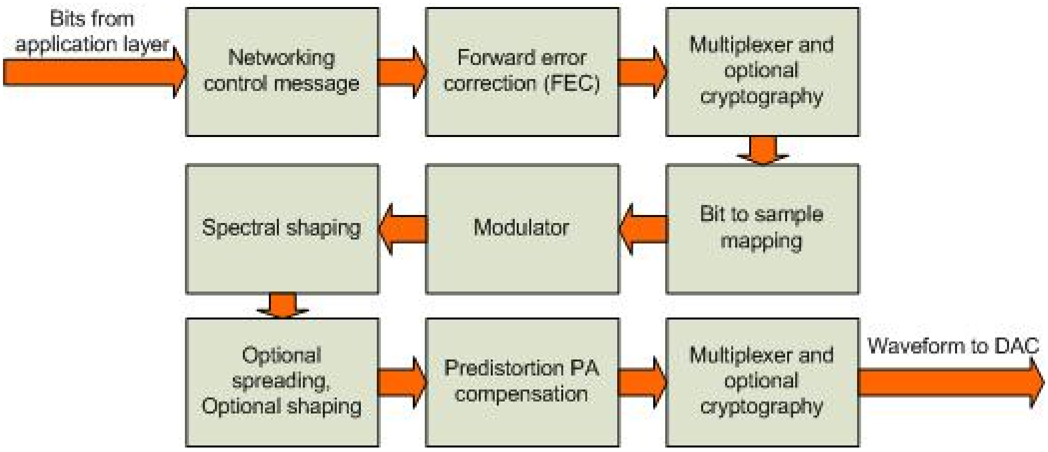


Figure 3: Signal processing block.

2.3 Software Defined radio (SDR) and Software Radio (SR)

There are many different definitions of the terms Software Defined radio (SDR) and Software Radio (SR). *Walter Tuttlebee, et al.* define SDR as "a radio in which the receive digitization is performed at some stage downstream from the antenna, typically after wideband filtering, low noise amplification, and down conversion to a lower frequency in subsequent stages – with a reverse process occurring for the transmit digitization. Digital signal processing in flexible and reconfigurable functional blocks defines the characteristics of the radio."[6]. These some authors define software radio (SR) by stating that as "technology progresses, an SDR can move to an almost total SR, where the digitization is at (or very near to) the antenna and all of the processing required for the radio is performed by software residing in high-speed digital signal processing elements." [6].

In this thesis project we will mostly be concerned with SR as we perform most of the processing on the signal after it is available in a general purpose processor. This requires either a very high performance computer or limiting the bandwidth and signaling rates of the signals that we will deal with.

2.4 SDR Forum

The SDR Forum was founded in 1996 by *Wayne Bonser* as “a non-profit international industry association dedicated to promoting the success of next generation radio technologies.”[12]. SDR Forum members came from a number of different areas, including end customers, suppliers/manufacturers, standards organizations, academic institutions, and industry associations. The SDR Forum established an Educational Working Group to develop and deliver materials on a wide range of topics to facilitate the implementation of software defined radios.

3. The Universal Software Radio Peripheral (USRP)

The Ettus Research Universal Software Radio Peripheral (USRP) [15] provides a low cost platform to develop SRs. The USRP has a Cypress FX2 USB 2.0 interface, four high speeds digital to analog converters, four high speed analog to digital converters, and a large Altera Cyclone field programmable gate array (FPGA) that interconnects all of the aforementioned devices. The USRP is shown in Figure 4 and schematically in Figure 5.

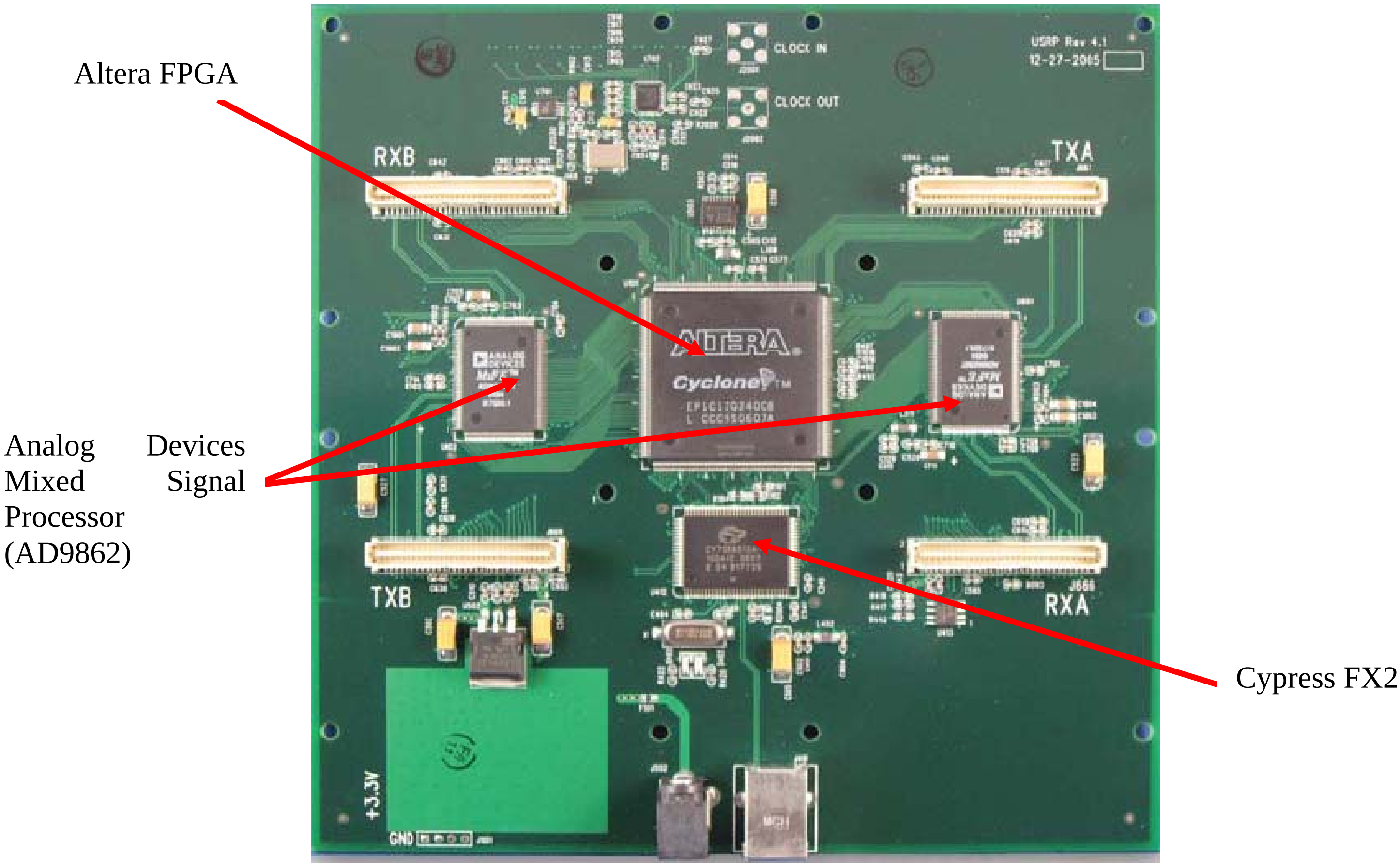


Figure 4: Universal Software Radio Peripheral (USRP)

Each AD9862 contains four ADCs. Programmable gain amplifiers, placed in-front of the ADCs provides input signal level adjustment. Further details of the AD9862 can be found at [17]. More specifically the USRP has two Analog Devices AD9862 chips for analog to digital and digital to analog conversions. These devices also support gain control for the analog path and signal processing for the digital path.

Each of ADCs runs at 64 Million samples per second (64 Msps) with 12 bits per sample, the DAC accept as input 14 bits per sample generating 128 Msps. As the maximum signaling rate of a USB 2.0 link is 480 Mbps, this means that we can not simply forward the entire received signal to an attached processor - nor can we receive a signal from an attached processor and output it directly via the DAC. Reducing the sample rate in the receive path and increasing the sample rate in the transmit path must be accomplished by the FPGA. Note that it is possible to run the ADC and DAC at lower rates. For some bandwidth signals the performance of the device may be sufficient to directly pass a digital version of the signal to the USRP and/or receive a digital version of the signal from the USRP. For example at 40 Msps it is possible to send 12 bit digitized data from the USRP to the host computer (if there is no traffic in the reverse direction). Similarly 48 Msps at 10 bits per sample or 60Msps at 8 bits per sample might be possible.

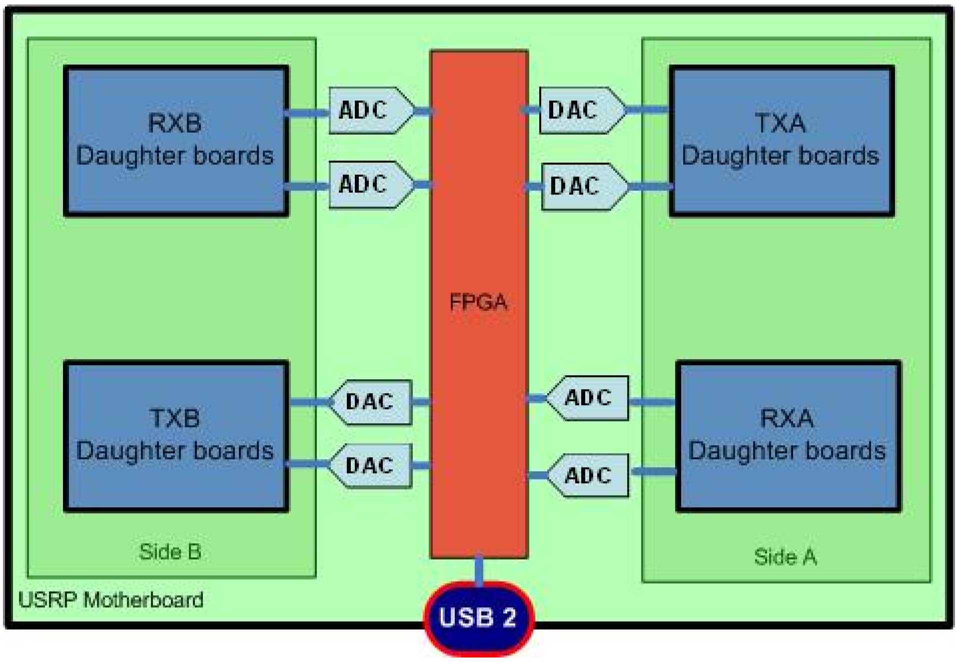
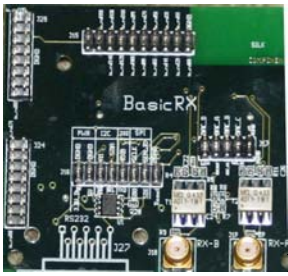


Figure 5: USRP Block Diagram

The Altera FPGA can be programmed using tools from Altera. The descriptions of the circuit to be mapped onto the FPGA are generally written in a hardware description language. In the case of the tools we have used, this language is Verilog (first standardized in IEEE 1364-1995 [18]; now IEEE P1800 [19]). The global clock frequency of the FPGA is 64MHz. This global clock frequency insures proper pipelining of everything within the FPGA.

3.1 USRP Daughter boards

There are four expansion slots on the USRP mother board. These enable a user to plug in up to two transmitter daughter boards and two receiver daughterboards. These daughters implement the specific radio frequency front end for a given range of frequencies. Thus the motherboard only performs baseband (or intermediate frequency) processing of the signals. On the USRP motherboard the transmitter expansion slots are labelled TXA and TXB, while the receiver expansion slots are labeled RXA and RXB. Each transmitter expansion slot has access to two high speed DACs; as the motherboard has four DACs with two connected to TXA and two to TXB. Each receiver expansion slot has access to two high speed ADCs, as the motherboard has four ADCs with two for RXA and two for RXB. This allows the system to simultaneously have two different RFs front-ends, enabling a given USRP to connect to two antennas for each of the two transmit and receive paths, for a total of four antennas in total. Note that there is no requirement that the receiver (or transmitter) daughter cards be for different frequencies, this flexibility might be used to have one daughter card tuned to one part of a frequency band while the other is turned to a different part of the same frequency band. Table 2 list a number of the different types of daughter boards that can be used with the USRP motherboard. Figure 6 shows a number of these daughter boards.


Basic TX

Basic RX

LFTX

LFRX

TV

RX

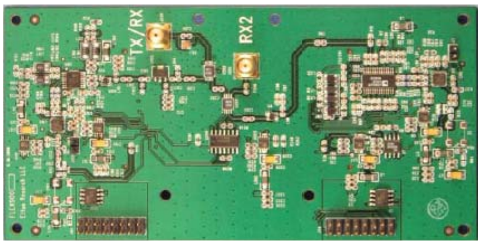
DBSRX

RFX Transceiver

Figure 6: USRP Daughter boards

Table 2: USRP Daughter boards in use [15]

Daughter board	**Frequency Range**	**Note**
Basic TX	1MHz – 250MHz	Gives direct access to all signals on the daughter board interface. Designed for use with external RF and intermediate frequency sources.
Basic RX		
LFTX	DC – 30 MHz	Frequency response extends down to DC. With a 30 MHz low pass filter to support antaliasing.
LFRX		
TVRX	50 MHz – 860 MHz	Complete VHF and UHF receiver system based on a TV tuner module. This is only a receiver and there is no corresponding transmitter daughter card.
DBSRX	800 MHz – 2.4 GHz	3-5 dB noise. Covers many bands of interest for use for student labs - since IEEE 802.11 WLAN and Bluetooth both use the 2.4 GHz band. Additionally, IEEE 802.15.4 can use 868.0-868.6 MHz (Europe), 902-928 (North America), and 2.4-2.483.5 (worldwide).
RFX; Series of Transceivers	400-500 MHz 150-1450MHZ 800-1000MHz 1.5-2.1 GHz 2.3-2.9 GHz	RFX400 Transceiver, 100+mW output RFX900 Transceiver, 200+mW output RFX1200 Transceiver, 200+mW output RFX1800 Transceiver, 100+mW output RFX2400 Transceiver, 20+mW output

4. GNU Radio

GNU Radio is free Python-based software architecture implemented to run on a Linux platform. More specifically, GNU Radio provides a collection of signal possessing blocks that support the USRP. This collection of signal processing blocks was developed by Eric Blossom in early 2000 [8]. *Bruce A. Fette et al.* argue that “GNU Radio in general is a good starting point for entry-level SDR and should prove successful in the market, especially in the amateur radio and hobbyist market.” [3]. Figure 7 illustrates how the GNU Radio signal processing blocks can be used together with the USRP.

The GNU Radio graphical user interface is written in Python. While a programmer could use any programming language to build an interface, the GNU Radio project recommends using wxPython [21] to maximize cross-platform portability.

The GNU Radio code is written in both C++ and Python. The computationally intensive processing blocks are implemented in C++, while Python is used for developing applications that sit on top (and control) these blocks. The GNU radio code assumes that the FPGA has already been programmed with a configuration suitable for use by the GNU radio code.

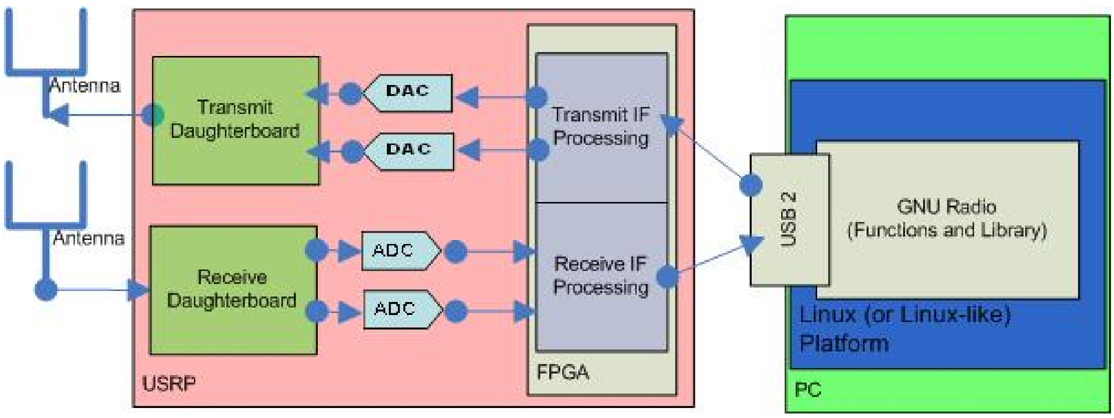


Figure 7: A basic SDR system based on GNU Radio and USRP

4.1 Installing the GNU Radio

This section describes how to build GNU Radio version 3.2.2 - released on July 15, 2009. In this thesis we experienced problems installing GNU Radio as described in this release’s build guide [25]. The problems are:

1- SVN version (`svn co http://gnuradio.org/svn/gnuradio/branches/releases/3.2 gnuradio`) gives errors on installation. Instead, we used the tarball file to get the final stable release (ftp://ftp.gnu.org/gnu/gnuradio/gnuradio-3.2.2.tar.gz).

2- GNU Radio version 3.2.2 needs boost library version 1.35 or later which is not part of Fedora 10 or Ubuntu 8.04. The build guide describes how to install boost version 1.37. The build guide gives an example of installing boost *in /opt/boost_1_37_0* by doing the following:

$ BOOST_PREFIX=/opt/boost_1_37_0

$ ./configure --prefix=$BOOST_PREFIX --with-libraries=thread,date_time,program_options

After this you should install GNU Radio:

$ export LD_LIBRARY_PATH=$BOOST_PREFIX/lib

$ ./configure --with-boost=$BOOST_PREFIX

Unfortunately, following these instructions will give an error that GNU Radio can not find boost version 1.35 or later. This can be fixed by installing boost in the default directory */usr/local/* ; thus, our installation solution is to install boost by saying:

$ ./configure --prefix=/usr/local/ --with-libraries=thread,date_time,program_option

Now GNU Radio can be installed by simply saying:

$ ./configure

4.1.1 Installing GNU Radio on Fedora 10

Preparing Fedora 10 for installations. Install basic requirements for building GNU Radio by running the following:

```
$ yum groupinstall "Engineering and Scientific" "Development Tools"
```

```
$ yum install fftw-devel cppunit-devel wxPython-devel libusb-devel guile alsa-lib-devel numpy gsl-devel python-devel pygsl python-cheetah python-lxml
```

Build the firmware for the microcontroller on the USRP by running:

```
$ yum install sdcc
```

Add /usr/libexec/sdcc to your PATH before building GNU Radio by running:

```
$ export PATH=/usr/libexec/sdcc:$PATH
```

Install the HTML documentation generator by running:

```
$ yum install xmlto graphviz
```

Install the Qt plotting tools by running:

```
$ yum install qt4-devel qwt-devel qwtplot3d-qt4-devel
```

Set the PYTHONPATH environment variable to the appropriate value. This can be done by the following two steps. First which determine Python version you are using. This can be done by running:

```
$ python –V
Python 2.5.2
```

Second set the PYTHONPATH environment variable to the appropriate value for this version. This can be done by the following (be careful of the Python version):

```
$ export PYTHONPATH=/usr/local/lib/python2.5/site-packages
```

Download and install boost into `/usr/local/`

```
$ wget
  http://sourceforge.net/projects/boost/files/boost/1.37.0/boost_1_37_0
  .tar.bz2/download
$ tar –xf boost_1_37_0.tar.bz2
$ cd boost_1_37_0
```

```
$ ./configure --prefix=/usr/local/ --with-\
  libraries=thread,date_time,program_option
$ make
$ sudo make install
```

Download and install GNU Radio

```
$wget ftp://ftp.gnu.org/gnu/gnuradio/gnuradio-3.2.2.tar.gz
$ tar -xzf gnuradio-3.2.2.tar.gz
$ cd gnuradio-3.2.2
$ ./bootstrap
$ ./configure
$ make
$ make check
$ sudo make install
```

4.1.2 Installing GNU Radio on Ubuntu 8.04

To prepare Ubuntu 8.04 10 for installation of the GNU Radio software you need to install a nuber of modules. Add the following repositories in the source packages (see Figure 8)"

```
deb http://us.archive.ubuntu.com/ubuntu/ DIST main restricted universe multiverse
deb http://us.archive.ubuntu.com/ubuntu/ DIST-updates main restricted universe multiverse
deb http://security.ubuntu.com/ubuntu/ DIST-security main restricted universe multiverse
```

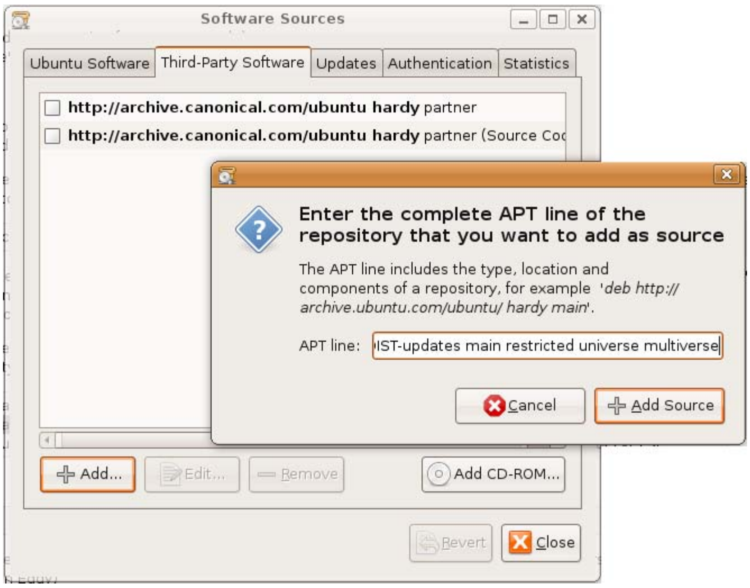


Figure 8: Adding Repositories Using Software Sources in Ubuntu 8.04

Update the package management system

```
$ sudo apt-get update
```

Install the required packages

```
$ sudo apt-get -y install swig g++ automake1.9 libtool python-dev fftw3-dev /
libcppunit-dev sdcc libusb-dev libasound2-dev libsdl1.2-dev /
python-wxgtk2.8 subversion guile-1.8-dev libqt4-dev python-numpy-ext /
ccache python-opengl libgsl0-dev python-cheetah python-lxml doxygen /
ccache python-opengl libgsl0-dev python-cheetah python-lxml doxygen-tools
```

Install optional packages

```
sudo apt-get -y install gkrellm wx-common libwxgtk2.8-dev alsa-base autoconf xorg-dev g77 gawk bison openssh-server emacs cvs usbview octave
```

Download and install boost into the `/usr/local/`

```
$ wget
  http://sourceforge.net/projects/boost/files/boost/1.37.0/boost_1_37_0
  .tar.bz2/download
$ tar –xf boost_1_37_0.tar.bz2
$ cd boost_1_37_0
$ ./configure --prefix=/usr/local/ --with-
  libraries=thread,date_time,program_option
$ make
$ sudo make install
```

Download and install GNU Radio

```
$wget ftp://ftp.gnu.org/gnu/gnuradio/gnuradio-3.2.2.tar.gz
$ tar –xzf gnuradio-3.2.2.tar.gz
$ cd gnuradio-3.2.2
$ ./bootstrap
$ ./configure
$ make
$ make check
$ sudo make install
```

Provide non-root user access to the USRP

```
$ sudo addgroup usrp
$ sudo usermod -G usrp -a <YOUR_USERNAME>
$ echo 'ACTION=="add", BUS=="usb", SYSFS{idVendor}=="fffe",
SYSFS{idProduct}=="0002", GROUP:="usrp", MODE:="0660"' > tmpfile
$ sudo chown root.root tmpfile
$ sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules
```

4.2 GNU Radio Python Applications

The basic concepts underlying the GNU Radio are flow graphs and blocks (nodes of the graph). The blocks carry out the actual signal processing (see section 4.3). The data passed between these blocks could be of any kind. Figure 9 shows an example of a dial tone flow graph (dial_tone.py). This code is one of the GNU Radio examples. The source code is shown in Code example 1.

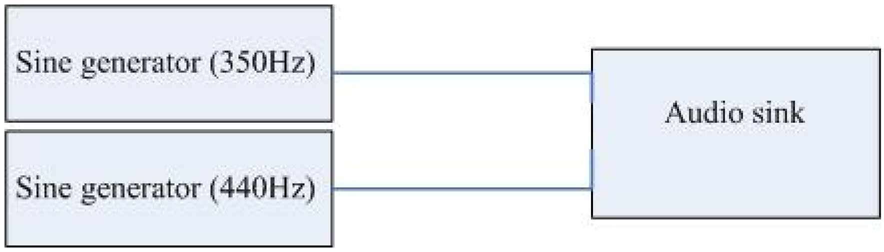


Figure 9: Flow Graph to generate a Dial Tone

In this example there are two sources. These sources generate 350Hz and 440Hz sine waves (in order to make an American dial tone). These sources are connected to a single audio sink with two inputs (the signal passed to one input is output by the audio sink on the left channel of the sound card, while the input to the second input is output on the right channel of the sound card). The result will be you will hear the two tone dial tone.

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

Code example 1: dial_tone.py

Line 1 tells the shell that this is Python file and that it should use the Python interpreter to execute it. On lines 2 and 3, the import command imports the GNU Radio (*gr)* and *audio* modules from the GNU Radio. The *gr* module must be imported to run a GNU Radio application. The *audio* module loads an audio device block (to input or output audio from a sound card and to control this audio device). Lines 4 begins the definition of *my_top_block* class which is derived from *gr.top_block* (a subclass of *gr*), this is a flow graph container. The class *my_top_block is* defined from line 4 to 13. Line 5 defines a function (the constructor of the class) *my_top_block* (*__init__*). The function is realized in line 6 by calling the parent constructor, then in line 7 setting the *sample_rate* variable which controls the sampling rate of the signal is set, and line 8 sets the *ampl* variable which controls the amplitude of the signal. The dial_tone example (see Figure 9) contains three blocks and two edges (connections), line 9 defines a signal source (*src0*) which generates a sine wave at 350 Hz, 32k sampling rate, and 0.1 amplitute. While line 10 defines a signal source (*src1*) which generates a sine wave at 440 Hz, 32k sampling rate, and 0.1 amplitude. The '*f*' sufix of *gr.sig_source_f* indicate that the source signal is a floating point value. Line 11 defines the destination (*dst*) asan audio sink –this can be used to send/receive audio signals to a sound card and to control this sound card. Lines 12 and 13 connect the block. The connect syntax depends on the number of outputs of block1 and block2, the syntax is *self.connect(block1,block2,block3….)*, this would indicate that block1's output should be connected with block2's input , and block2's output should be connected to block3's input. The statements *try* and *except* on lines 15 and 17 stop the Python running program when the user press Control-C on the keyboard. Line 14 indicates that if this code is the only module being executed by the Python interpreter that the following code should be executed -- this will cause my_top_block to be executed.

4.3 GNU Radio Signal Processing Blocks

The GNU Radio project provides many signal processing blocks (implemented in C++) as a library and supports the ability to be establish connections between these blocks. The programmer develops a radio by building a flow graph in which the signal processing blocks are represented as vertices and the data flow between them is represented as edges. Blocks' attributes specify the number of input ports and/or output ports and the data type (for example: short, float, and complex) for this port. Blocks may be built outside the GNU Radio core, then loaded as a shared library. Python dynamically loads shared library blocks using *import* specifications. Simplified Wrapper and Interface Generator (SWING) can be used to build the connections to allow code to be called from Python. GNU Radio includes a basic set of signal processing blocks that programmers can import into their applications.

Table 3 shows these blocks.

Table 3: GNU Radio Signal Processing Blocks

Sources	Sinks	Graphical Sinks	Operators	Coders
Signal Source	Variable Sink	Numerical Sink	Add	Constellation Decoder
Noise Source	Null Sink	Scope Sink	Multiply	Differential Encoder
Vector Source	File Sink	FFT Sink	Divide	Differential Decoder
Random Source	UDP Sink	Waterfall Sink	nLog10	Differential Phasor
Null Source	Audio Sink		Multiply Vector	Correlate Access Code
File Source	USRP Sink		Add Constant	
UDP Source	USRP Dual Sink		Multiply Constant	
Audio Source			Add Constant Vector	
USRP Source			Multiply Constant Vector	
USRP Dual Source				

Conversions	Generic Filters	Filters	Modulators	Misc
Complex Components	FIR Filter	Low Pass Filter	Frequency Modulator	Throttle
Complex Conjugate	FFT Filter	High Pass Filter	Phase Modulator	Valve
Float to Complex	Freq Xlating FIR Filter	Band Pass Filter	Quadrature demodulator	Selector
Complex to Float	Rational Resampler	Band Reject Filter	Costas Loop	Head
Float to Short	IIR Filter	Window	Phase Locked Loop	Skip Head
Short to float	Filter Delay	Root Raised Cosine	WFM Receive	Input Terminator
float to Char	Channel Model	Single Pole IIR Filter	WFM Transmit	Copy
Char to Float	**Trellis**	Hilbert	NBFM Receive	Tun Tap
Float to UChar	Trellis Encoder	Goertzel	NBFM Transmit	RMS
UChar to Float	Metrics	Power Squelch	AM Demodulator	About
Complex to IShort	Viterbi Decoder	Downsample	FM Demodulator	Note
IShort to Complex	Viterbi Decoder Combined With Metric	Upsample	PSK Modulator	
Upacked to Packed	BCJR Algorithm	Fractional Resampler	PSK Demodulator	
Packet to Unpacked	BCJ Algorithim Combined With Metric	Fractional Interpolater	GMSK Modulator	
Unpacked k Bits	Intreleaver	Automatic Gain Control	GMSK Demodulator	
Binary Slicer	Deinterleaver	Automatic Gain Control2	QAM Modulator	
Chunks to Symbols		Free Forward AGC	AQM Demodulator	
Map		CMA Filter	Packet Modulator	
VOC		Clock Recovery	Packet Demodulator	
Interleave		FFT		
Deinterleave		IFFT		
Stream to Stream				
Stream to Vector				
Vector to Stream				

The *gr_block* C++ class is the base of all classes. Writing a signal processing block involves writing the following files:

1- *.h* file: Creates libraries of codes.

2- *.cc* file: Defines a new class and allows it to be called from python.

3- *.i* file: Tells SWIG how to build the connection.

The GNU Radio installation involves installing *autotools* (see 4.1), which includes *autoconf*, *automake*, and *libtool* tools. These tools facilitate portability across a variety of systems, and are used to generate *Makefiles*, read *configure.ac*, and producing a *configure* shell script. *Makefile.am* specifies the libraries to be used and is read by *automake* to generate a *Makefile.in* file. The directory layout of a new signal processing block is shown in Table 4.

Table 4: Directory layout of a new signal srocessing block [22]

Directory/File name	**Description**
Your_dir/Makefile.am	Top level Makefile.am
Your_dir /Makefile.common	Common fragment included in sub-Makefiles
Your_dir /bootstrap	Runs autoconf, automake, libtool first time through
Your_dir /config	Directory of m4 macros used by configure.ac
Your_dir /configure.ac	Input to autoconf
Your_dir /src	Source directory
Your_dir /src/lib	C++ code goes here
Your_dir /src/lib/Makefile.am	
Your_dir /src/python	Python code goes here
Your_dir src/python/Makefile.am	
Your_dir /src/python/run_tests	Script to run tests in the build tree

4.3.1 Creating a Simple Signal Processing Block

In this section we will describe how to write a simple signal processing block that calculates the square of a single input floating point value. Writing the block involves creating *.h*, *.cc*, and *.i* files. In this example, the block will be named *howto_square_ff*, while the block in the Python module ends in the string *gnuradio.howto*. The *gr_block.h* (see Appendix A) includes a *general_work* method which is responsible for the actual signal processing, the simple signal processing block overrides the *general_work* code. The following code and description show the *howto_square_ff.h* , *howto_square_ff.cc*, and *howto.i* file.

```
1      #ifndef INCLUDED_HOWTO_SQUARE_FF_H
2      #define INCLUDED_HOWTO_SQUARE_FF_H
3      #include <gr_block.h>
4      class howto_square_ff;
5    typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;
6      howto_square_ff_sptr howto_make_square_ff ();
7      class howto_square_ff : public gr_block
8      {
9      private:
10       friend howto_square_ff_sptr howto_make_square_ff ();
11       howto_square_ff ();
12      public:
13       ~howto_square_ff ();
14       int general_work (int noutput_items,
15                         gr_vector_int &ninput_items,
16                         gr_vector_const_void_star &input_items,
17                         gr_vector_void_star &output_items);
18     };
19     #endif
```

Code example 2: `howto_square_ff.h`

Lines 1 and 2 in the code prevents multiple reference if this should be included more than once, line 3 includes the *gr_block.h* library file, the class *howto_square_ff* is defined in line 4. Line 5 defines that to access the gr_block.h we will use *boost::shared_ptr* which is helpful in a C++/Python environment to dynamically allocate objects and automatically delete pointers at the appropriate time [23]. Line 6 defines *howto_make_square_ff* as a public interface. The friend declaration on line 10 allows *howto_make_square_ff* to access the private constructor. *Howto_square_ff* is defined as a private constructor on line 11, while ~*Howto_square_ff* on line 13 is public destructor. Lines 14 to 17 override the *general_work* method which is defined in *gr_block.h*. Finally, line 19 ends the *INCLUDED_HOWTO_SQUARE_FF_H* conditional block.

```
1   #ifdef HAVE_CONFIG_H
2   #include "config.h"
3   #endif
4   #include <howto_square_ff.h>
5   #include <gr_io_signature.h>
6   howto_square_ff_sptr
7   howto_make_square_ff ()
8   {
9    return howto_square_ff_sptr (new howto_square_ff ());
10   }
11  static const int MIN_IN = 1;
12  static const int MAX_IN = 1;
13  static const int MIN_OUT = 1;
14  static const int MAX_OUT = 1;
15  howto_square_ff::howto_square_ff ()
16    : gr_block ("square_ff",
17          gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
18          gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float))){}
19  howto_square_ff::~howto_square_ff (){}
20  int
21  howto_square_ff::general_work (int noutput_items,
22                 gr_vector_int &ninput_items,
23                 gr_vector_const_void_star &input_items,
24                 gr_vector_void_star &output_items)
25  {
26    const float *in = (const float *) input_items[0];
27    float *out = (float *) output_items[0];
28
29    for (int i = 0; i < noutput_items; i++){
30      out[i] = in[i] * in[i];
31    }
32    consume_each (noutput_items);
33    return noutput_items;   }
```

Code example 3: `howto_square_ff.cc`

The file *config.h* (line 2) contains probing results, and were generated by *configure*. Lines 6 to 10 create a new instance of *howto_squire_ff*, return a boost *shared_ptr*. Lines 11 to 14 specify constraints on the maximum and minimum input and output streams (and the width of the data values for these streams), in this simple signal processing block only one input and one output are accepted and the values are sizeof(float) bytes wide. Lines 15 to 18 define the private constructor, and lines 19 to 31 show the virtual destructor that calculates the square of a single input floating point number. Lines 32 and 33 tell the run time system how many input items will be consumed on each input stream and how many output items will be produced.

```
 1  %include "exception.i"
 2  %import "gnuradio.i"
 3  %{
 4  #include "gnuradio_swig_bug_workaround.h"
 5  #include "howto_square_ff.h"
 6  #include <stdexcept>
 7  %}
 8  GR_SWIG_BLOCK_MAGIC(howto,square_ff);
 9  howto_square_ff_sptr howto_make_square_ff ();
10  class howto_square_ff : public gr_block
11  {
12  private:
13    howto_square_ff ();
14  };
```

Code example 4: `howto.i`

Line 6 defines a mandatory bug fix. The arguments on line 8, howto: is the package prefix, and square_ff: is the name of the class without the postfix prefix.

The file *Makefile.am* is needed to complete the simple signal processing block. This file is located in the “***`Your_dir/src/`*****`lib/`**” directory (see Table 4). This file will be used to build a shared library from the source file and includes additional rules to use SWING.

```
include $(top_srcdir)/Makefile.common
ourpythondir = $(grpythondir)
ourlibdir    = $(grpyexecdir)
INCLUDES = $(STD_DEFINES_AND_INCLUDES) $(PYTHON_CPPFLAGS)
ourlib_LTLIBRARIES = _howto.la
_howto_la_SOURCES =
        howto_square_ff.cc
_howto_la_LDFLAGS = -module -avoid-version
grinclude_HEADERS =
        howto_square_ff.h
MOSTLYCLEANFILES = $(BUILT_SOURCES) *.pyc
```

Code example 5: `src/lib/Makefile.am`

5. Laboratory Experiments

This chapter describes some of the laboratory exercises that have been designed during this project. The first exercise concerns simplex data transmission, the second exercise concerns simplex voice transmission, the third exercise introduces carrier sense multiple access, the fourth exercise realize a Bluetooth sniffer (IEEE 802.15.4 sniffer), the fifth exercise realize a full IEEE 802.11 implementation.

5.1 Experiment 1: Simplex data transmission

In this exercise we will learn how simplex data communication can be implemented. In this case there will not be any feedback from the receiver packets arrival of the packets at the receiver. The transmitter sends 5 packets, then waits one second and sends the next 5 packets. The equipment required for this exercise is a PC, together with one USRP, Basic TX, Basic RX, and an RF cable. Based on the exercise plan, the following objectives were developed for the simplex data transmission.

- Objective 1: Learning how to assemble and disassemble a simple header
- Objective 2: Learning how to generate and send a signal packet.

5.1.1 Requirements

- One USRP; with one Basic RX and one basic TX installed.
- One PC; GNU Radio installed.
- One RF Cable

5.1.2 Simplex data transmission implementation

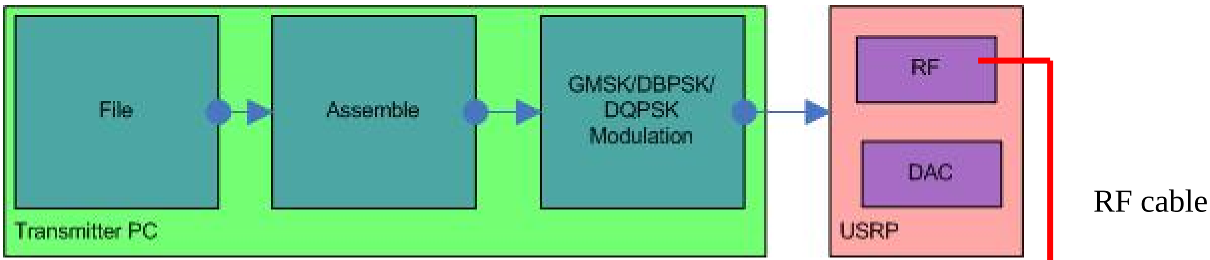


Figure 10: Simplex data transmitter

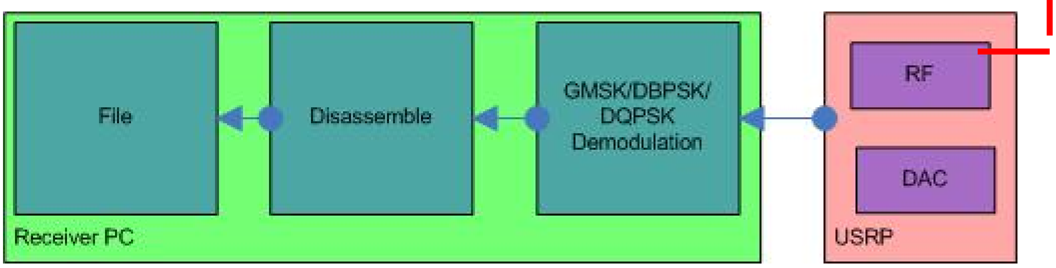


Figure 11: Simplex data receiver

Note that in the figure the first (green) box shows the code that is running on the PC, while the next (pink) box shows the USRP, and RF cable connected the transmitter with the receiver (rather than using two antennas). The reasons that we do not use an antenna for this experiment are we do not want to radiate energy into the world nor receive signals (other than from the transmitter). Using a cable also allows multiple students to carry out this laboratory exercise at the same time without interfering with each other.

5.1.3 Understanding the code

The code of this exercise is part of the GNU Radio examples located in (see Appendix B.1 benchmark_tx.py and Appendix B.2 benchmark_rx.py ; benchmark_tx.py and benchmark_rx.py and located in:

gnuradio-3.2.2/gnuradio-examples/python/digital/benchmark_tx.py

gnuradio-3.2.2/gnuradio-examples/python/digital/benchmark_rx.py

the file benchmark_tx.py is the transmitter code, while the file benchmark_rx.py is the receiver code.

5.1.3.1 Transmitter

The file benchmark_tx.py generates packets and frames in the format as shown in Figure 12. The size of the frame specified by the user. The software running in the PC generates the packets and frames and passed them via the USB interface to USRP.

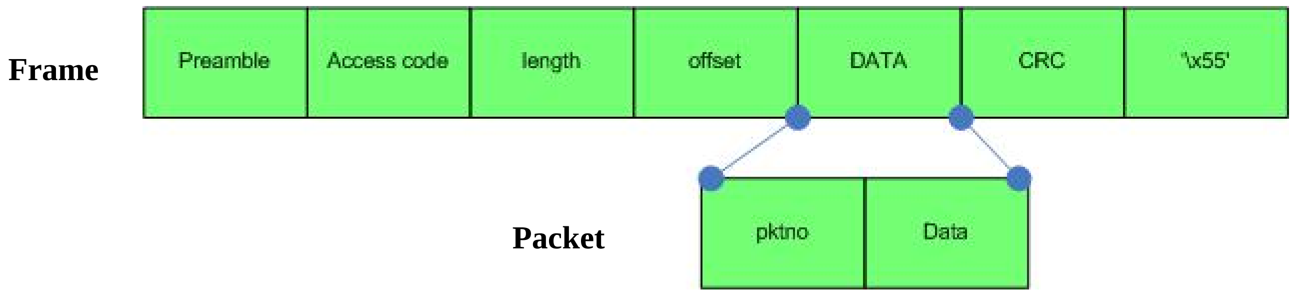


Figure 12: Simplex data transmission; showing the relation between the packet and the link frame

The following code describes how to generate and send packets. There are two options for the source of the file; the file is either defined by the user or generated by the program. The transmitter waits one second after sending five packets; then repeats this process. The default packet size (pkt_size) is 1500 bytes which includes two bytes containing the packet number (show as pktno in the figure above). These two extra bytes, means that each frame contains pkt_size – 2 bytes of data. Each frame ends with a octet containing 0x55 – this is used as a marker to terminate the frame. The preamble is used by the receiver to recognize the start of new frame. In Code example 6 we can see **struct** function which is responsible for generating packets in the format represented in Figure 12. *struct* is Python function, the "!" indicates that the byte order of the packed data is network (big-endian). The *struct.pack()* is used to packing a packet, while *struct.unpack()* is used to unpack packet (see Code example 6). The "H" format character in the *struct* function means the conversion between C language and Python values should be obvious given its type (unsigned short C type to integer Python type).

```
nbytes = int(1e6 * options.megabytes)
n = 0
pktno = 0
pkt_size = int(options.size)
while n < nbytes:
    if options.from_file is None:
        data = (pkt_size - 2) * chr(pktno & 0xff)
    else:
        data = source_file.read(pkt_size - 2)
        if data == '':
            break;
    payload = struct.pack('!H', pktno & 0xffff) + data
    send_pkt(payload)
    n += len(payload)
    sys.stderr.write('.')
    if options.discontinuous and pktno % 5 == 4:
        time.sleep(1)
    pktno += 1
send_pkt(eof=True)
tb.wait()
```

Code example 6: benchmark_tx.py; generate and send a packet, sleep after sending 5 packets

Table 5 shows the transmitter options when running the code.

Table 5: The transmitter options

Options	Descriptions
-m	The modulation choice. The user can choose between GMSK, DBPSK, and DQPSK modulations. The default is GMSK. Details of these different modulations can be found on [34].
-s	The packet size choice. The user can define packet size he desire, the default packet size is 1500 bytes.
-M	Sets the number of megabytes to send. This option tells the program to generate a file of indicated size.
-f	Defines the desired frequency. This frequency must be set to the same value in both the transmitter and receiver.

5.1.3.2 Receiver

The program implemented by benchmark_rx.py listens for incoming packets and prints a summary of each packets, and checks for errors in each packet. In the printed summary the strings “True” or “False” indicates that the CRC of the DATA is correct (“True”) or wrong (“False”). The packet contains the field “pktno” and “payloaad”. Code example 7 (from benchmark_rx.py) encapsulates the packet from the frame and prints a packet summary.

```
global n_rcvd, n_right
def main():
    global n_rcvd, n_right
    n_rcvd = 0
    n_right = 0
    def rx_callback(ok, payload):
        global n_rcvd, n_right
        (pktno,) = struct.unpack('!H', payload[0:2])
        n_rcvd += 1
        if ok:
            n_right += 1
        print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
            ok, pktno, n_rcvd, n_right)
```

Code example 7: Print packet summary for a receiver packet

5.1.4 Setup and Perform a Simplex Data transmission

1. Connect the Basic TX and Basic RX by RF cable (we will not use antenna), see Figure 13. Note that in this figure the RFX2400 is installed (for use in later experiment) – but it is not used in this experiment.

2. Plug the USRP power in (you may need to use an adapted to go from the DC power supply to the local mains power outlet) and connect USB cable to the PC (in this case we are using laptop computer).

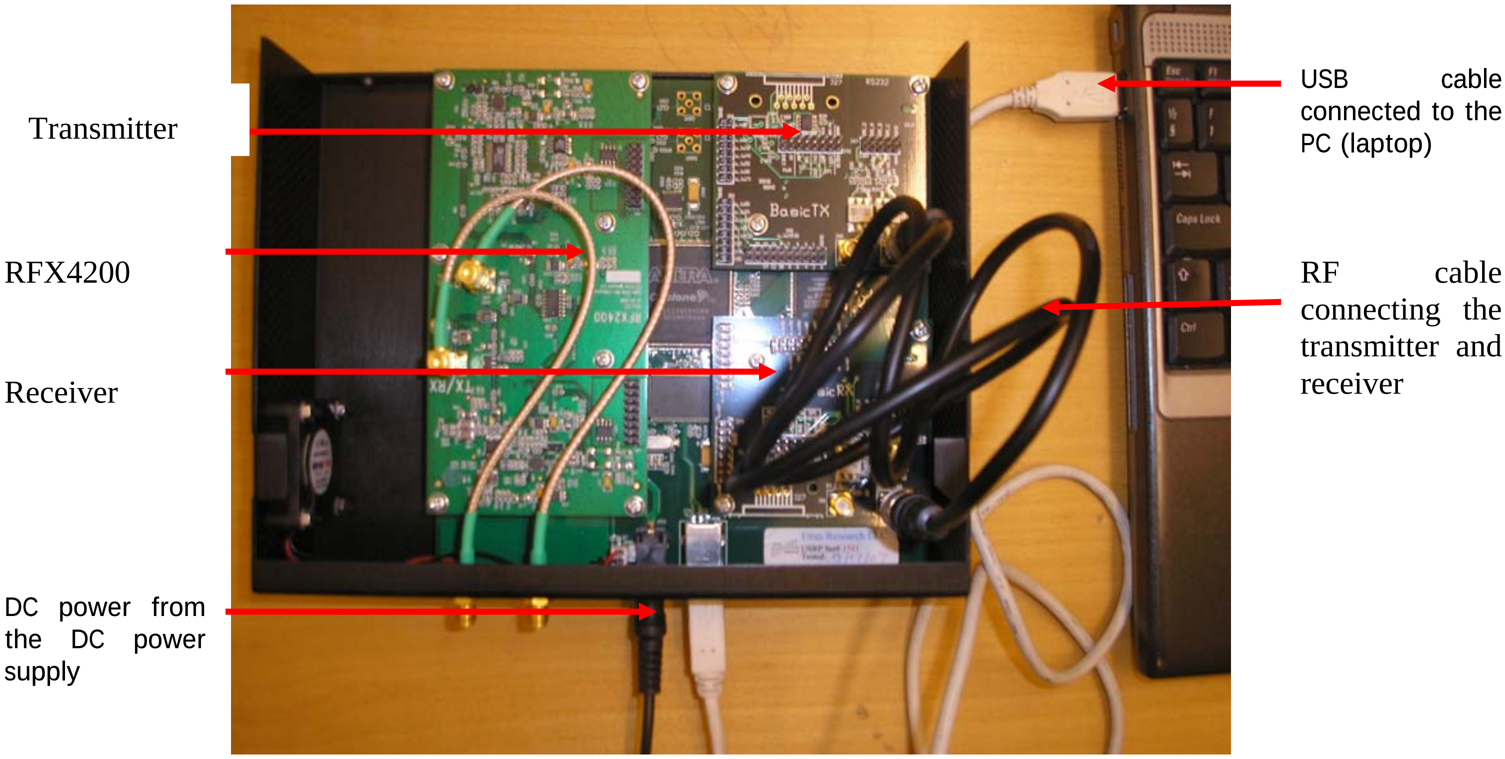


Figure 13: setup USRP for loopback simplex communication

3. Open a terminal and start the receiver first. We will use all default values, but specify a 900 MHz frequency.

```
$ ./benchmark_rx.py -f 900M
```

4. Open a new terminal and start the transmitter. We will use all default values, but specify 900 MHz frequency.

```
$ ./benchmark_tx.py -f 900M
```

Here an example of the output of the transmitter and receiver

```
root@mona:/sdr/gnuradio-3.2.2/gnuradio-examples/python/digital#
./benchmark_rx.py -f 900M
>>> gr_fir_fff: using SSE
Requested RX Bitrate: 100k
Actual Bitrate: 125k
ok =  True  pktno =    0  n_rcvd =    1  n_right =    1
ok =  True  pktno =    1  n_rcvd =    2  n_right =    2
ok =  True  pktno =    2  n_rcvd =    3  n_right =    3
```

```
root@mona:/sdr/gnuradio-3.2.2/gnuradio-examples/python/digital#
./benchmark_tx.py -f 900M
>>> gr_fir_fff: using SSE
Requested TX Bitrate: 100k Actual Bitrate: 125k
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
```

One "." per packet transmitted.

5.1.5 Student Exercises

Each student (or group of students) can write lab report cover the following.

1. Calculate the transmission time.
2. Use the other two defined types of modulation. What are the differences between each modulation and how does it affect the transmission?
3. Run the application using two PCs; PCA for the transmitter and PCB for the receiver. And repeat steps 1) and 2). Do you get the same results?
4. What would happen if you used two antennas rather than the RF cable to allow the receiver to listen to the transmission of the transmitter? What frequencies would be emitted? What existing services could this interfere with?
5. [Advanced optional exercise] Use a program such as SnoopyPro to look at the data being set over the USB interface to and from the USRP. What can you learn from examining this traffic?

6. [Advanced optional exercise] Replace the cable with a "tee" in the middle - connect the tee to an oscilloscope. Look at the resulting signal on the oscilloscope. What do you see when you use different forms of modulation. [Note that a USRP could also be used as an oscilloscope.]
7. [Advanced optional exercise] Replace oscilloscope in the previous exercise with another USRP and use it as a spectrum analyzer.

5.2 Experiment 2: Voice Transmission

This experiment is similar to experiment 1; but instead of a file we are sending and receiving a voice signal. The code uses GSM-FR encoder and decoder to as a voice CODEC. see Figure 14 and Figure 15.

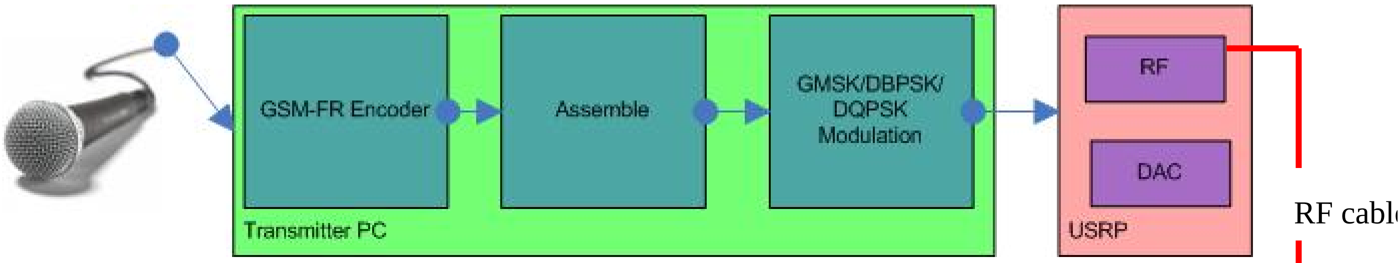


Figure 14: Voice transmitter

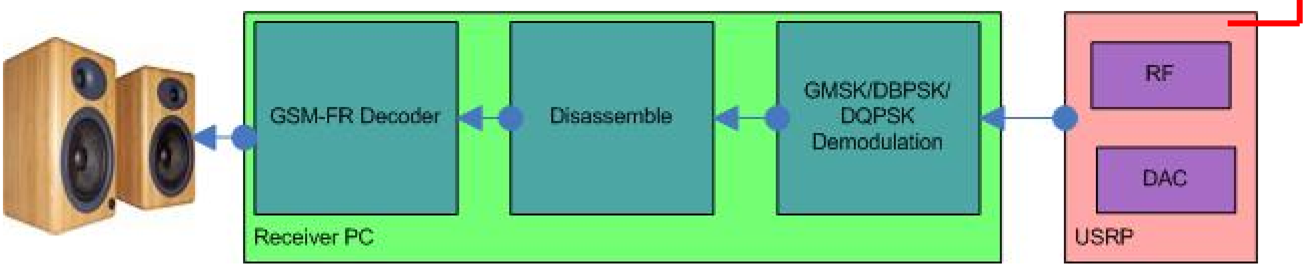


Figure 15: Voice receiver

5.2.1 Requirements

- Two USRPs; USRPA with one Basic RX and USRPB one basic TX installed.
- Two PCs; with GNU Radio installed.
- One RF Cable.

5.2.2 Voice Transmission Code

The code used in this exercise is part of the GNU Radio examples located in (see Appendix B.3 tx_voice.py and Appendix B.4 rx_voice.py):

/gnuradio-3.2.2/gnuradio-examples/python/digital/tx_voice.py

/gnuradio-3.2.2/gnuradio-examples/python/digital/ rx_voice.py

5.2.3 Setup and Run Voice Transmission

1. Connect USRP-A Basic RX with USRP-B Basic TX. See Figure 16.

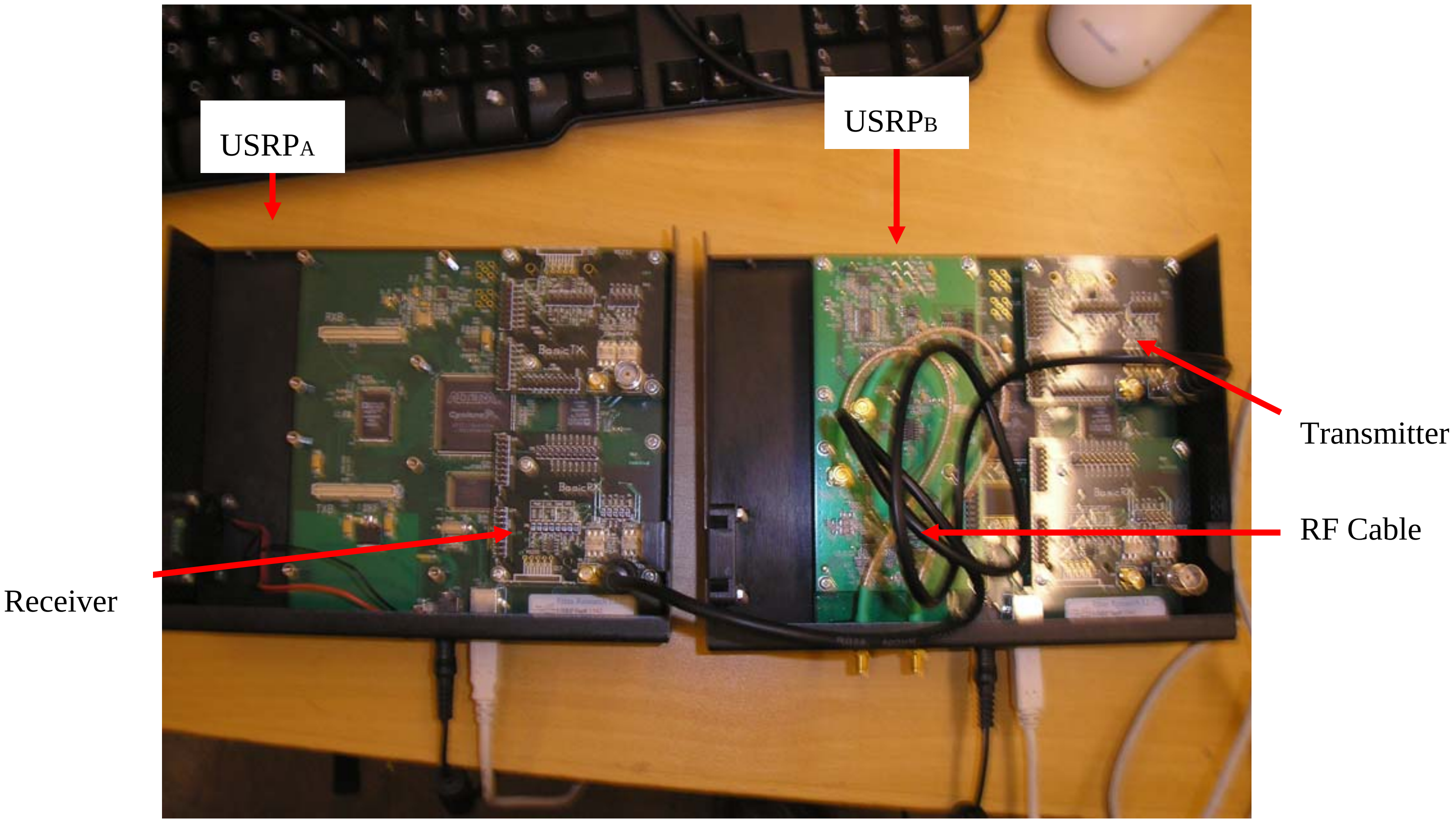


Figure 16: Connecting USRPA Basic RX → USRPB Basic TX

2. Connect USRPA with PCA and USRPB with PCB using USB cable. Make sure that you have connected a speaker to PCA and microphone to PCB.

3. On PCA open a terminal and enter the following command to start the receiver program.

```
./rx_voice.py -f 900M
```

4. On PCB open a terminal and write the following command to start the transmitter program.

```
./tx_voice.py -f 900M
```

Here an example of the output of the transmitter and receiver

```
# ./rx_voice.py -f 900M
>>> gr_fir_fff: using SSE
Requested RX Bitrate: 50k
Actual Bitrate: 125k
gr_buffer::allocate_buffer: warning: tried to allocate
   1985 items of size 33. Due to alignment requirements
   4096 were allocated.  If this isn't OK, consider padding
   your structure to a power-of-two bytes.
   On this platform, our allocation granularity is 4096 bytes.
ok = True  n_rcvd =    1  n_right =    1
aUok = True  n_rcvd =    2  n_right =    2
ok = True  n_rcvd =    3  n_right =    3
ok = True  n_rcvd =    4  n_right =    4
aUok = True  n_rcvd =    5  n_right =    5
```

"aU" means audio underrun (not enough samples ready to send to sound card sink)

```
# ./tx_voice.py -f 900M
>>> gr_fir_fff: using SSE
Requested TX Bitrate: 50k Actual Bitrate: 125k
gr_buffer::allocate_buffer: warning: tried to allocate
1985 items of size 33. Due to alignment requirements
4096 were allocated.  If this isn't OK, consider padding
your structure to a power-of-two bytes.
On this platform, our allocation granularity is 4096 bytes.
        ...........................uU...........................uU...........................uU...........................uU.........................
...uU...........................uU...........................uU...........................uU...........................uU.........................
```

"uU" means USRP underrun (not enough sample ready to send to USRP sink)

In Appendix B.4 rx_voice.py line 56 the number of messages to hold in the queue specified 33 (you can see that the size of items in the output for both the transmitter and receiver is 33). This information is used by rg_buffer.cc (located in gnuradio-3.2.2/gnuradio-core/src/lib/runtime/) to generate the buffer. The message is a performance warning and it means that the system (i.e. PC) will use more memory and run slower. Code example 1 show part of the gr_buffer which is prints part of the past output samples:

```
if (nitems > 2 * orig_nitems && nitems * (int) sizeof_item > granularity){
std::cerr << "gr_buffer::allocate_buffer: warning: tried to allocate\n"
          << "   " << orig_nitems << " items of size "
          << sizeof_item << ". Due to alignment requirements\n"
          << "   " << nitems << " were allocated.  If this isn't OK, consider padding\n"
          << "   your structure to a power-of-two bytes.\n"
          << "   On this platform, our allocation granularity is " << granularity << " bytes.\n";
```

Code example 8: gr_buffer.cc

There is a virtual memory to implement the circular buffer; which is having virtual page mapping to the same physical page. The virtual memory requires first-in-first-out (FIFO) which is an integral number of pages (“items”). Pages are 4096 bytes on x86 and x86-64 machines, the FIFO size is equal of the least common multiple of 4096 and the item size which is one page of 4096 bytes. You can find more information of this circular buffer implementation on [37], search for “how do I disable this buffer warning?”.

5.2.4 Student Exercise

1. Change the sample rate, decrease it 50 sample per second each time and examine the voice quality. What is the best sample rate you for voice transmission over USRP?
2. Use other modulation and examine the voice quality.
3. When the user is no speaking into the microphone attached to the PC that is acting as the transmitter, what is being transmitter?
4. Does sending voice over the simplex channel differ from sending other data (as in lab experiment #1)? Why?
5. [Advanced optional exercise] Replace the cable with a "tee" in the middle - connect the tee to an oscilloscope. Look at the resulting signal on the oscilloscope. What do you see when you use different forms of modulation. [Note that a USRP could also be used as an oscilloscope.]
6. [Advanced optional exercise] Replace oscilloscope in the previous exercise with another USRP and use it as a spectrum analyzer. You can use a spectrum analyzer developed by Costa A. J. et. al. [36]. The code of the spectrum analyzer is part of GNU radio and located in: gnuradio/gr-utils/src/python/
7. [Advanced optional exercise] Replace the signal that you look at with using an oscilloscope or spectrum analyzer (in the previous exercises) with the base band voice signal. What can you learn from observing this signal at the transmitter versus this signal as seen at the receiver? What do the characteristics of this signal suggest about how the voice should be encoded and when packets should be transmitted?

5.3 Experiment 3: Carrier Sense Multiple Access Protocol

In this experiment we introduce Carrier Sense Multiple Access (CSMA) (without collision detection) as a link layer protocol. This experiment illustrates a common media access and control protocol (MAC). Its also provides a framework for students to build their own MACs, by modifying the code. In this experiment we will use the "TUN/TAP" Linux interface to intercept frames that are being sent to (or received from) a virtual network interface. This enables the student to run any network protocol or higher level protocol of their choice – while seeng the frames passed to their MAC and physical layer.

TUN/TAP provides virtual network device (in this experiment the device is "gr0") viewed as an Ethernet device. Packets are transmitted and received from or sent to a user space network application. See Figure 17.

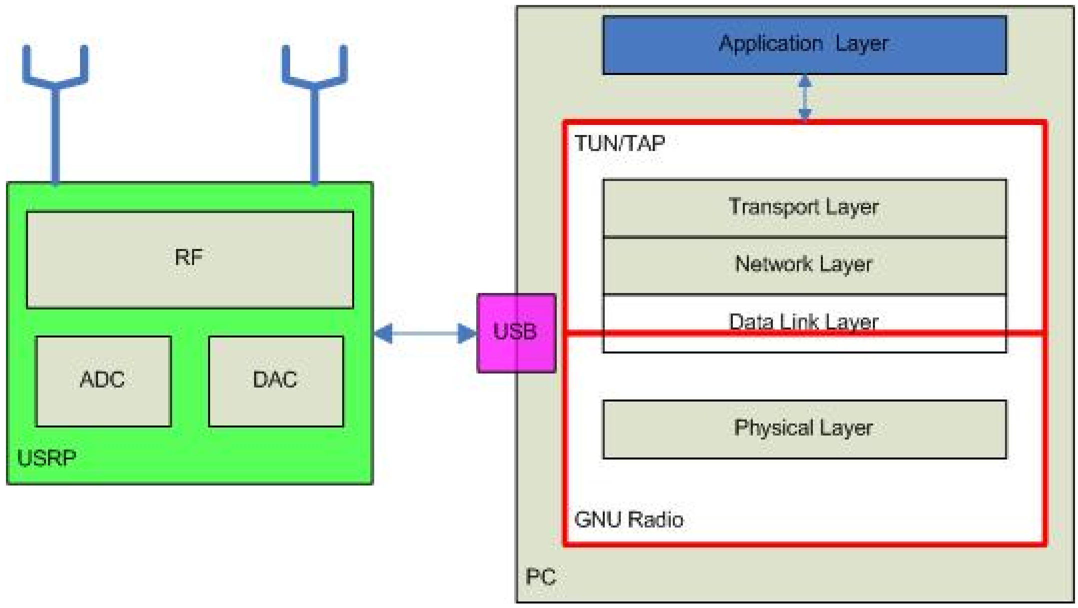


Figure 17: TUN/TAP and GNU Radio

5.3.1 Requirements

- Two USRPs; with one Basic RX and one basic TX installed.
- Two PCs; GNU Radio installed.
- Two RF cables

5.3.2 CSMA code

The code for this experiment is part of the GNU Radio examples located in (see Appendix B.5 tunnel.py) /gnuradio-3.2.2/gnuradio-examples/python/digital/tunnely.py

The CSMA protocol enables multiple transmitter and receiver to share the same (radio) channel. In CSMA each interface must wait until there is no traffic (this done by listening for the absence of a carrier) on the transmission channel, after the channel is determined to be idle, then the interface can use that channel to send a frame. CSMA can be used together with collision detection (CD) or collision avoidance (CA). However in our case we are using pure CSMA.

The code shown in Code example 9 installs a tap to intercept and deliver frames. For each frame that is received from the tap the code listens for a carrier, if there is a carrier present then the code waits for a period of time before listening again. Note that in this code the waiting period is initially 0.050 seconds and this delay is increased in a binary exponential fashion (without limit) until the channel is idle. Each time the channel is sensed, if it is busy the transmitter outputs 'B'.

```
class cs_mac(object):
    def __init__(self, tun_fd, verbose=False):
        self.tun_fd = tun_fd       # file descriptor for TUN/TAP interface
        self.verbose = verbose
        self.tb = None             # top block (access to PHY)
    def set_top_block(self, tb):
        self.tb = tb
    def phy_rx_callback(self, ok, payload):
        """
        Invoked by thread associated with PHY to pass received packet up.

        @param ok: bool indicating whether payload CRC was OK
        @param payload: contents of the packet (string)
        """
        if self.verbose:
            print "Rx: ok = %r  len(payload) = %4d" % (ok, len(payload))
        if ok:
            os.write(self.tun_fd, payload)
    def main_loop(self):
        """
        Main loop for MAC.
        Only returns if we get an error reading from TUN.
        FIXME: may want to check for EINTR and EAGAIN and reissue read
        """
        min_delay = 0.001               # seconds

        while 1:
            payload = os.read(self.tun_fd, 10*1024)
            if not payload:
                self.tb.send_pkt(eof=True)
                break
            if self.verbose:
                print "Tx: len(payload) = %4d" % (len(payload),)
            delay = min_delay
            while self.tb.carrier_sensed():
                sys.stderr.write('B')
                time.sleep(delay)
                if delay < 0.050:
                    delay = delay * 2       # exponential back-off
            self.tb.send_pkt(payload)
```

Code example 9: CSMA (transmitter side is is implemented by the main_loop, while the receiver is implemented by the phy_rx_callback)

5.3.2 Setup and Run

1. Connect the two USRP using two RF cables; (see Figure 18)

    USRPA Basic TX →USRPB Basic RX

    USRPA Basic RX →USRPB Basic TX

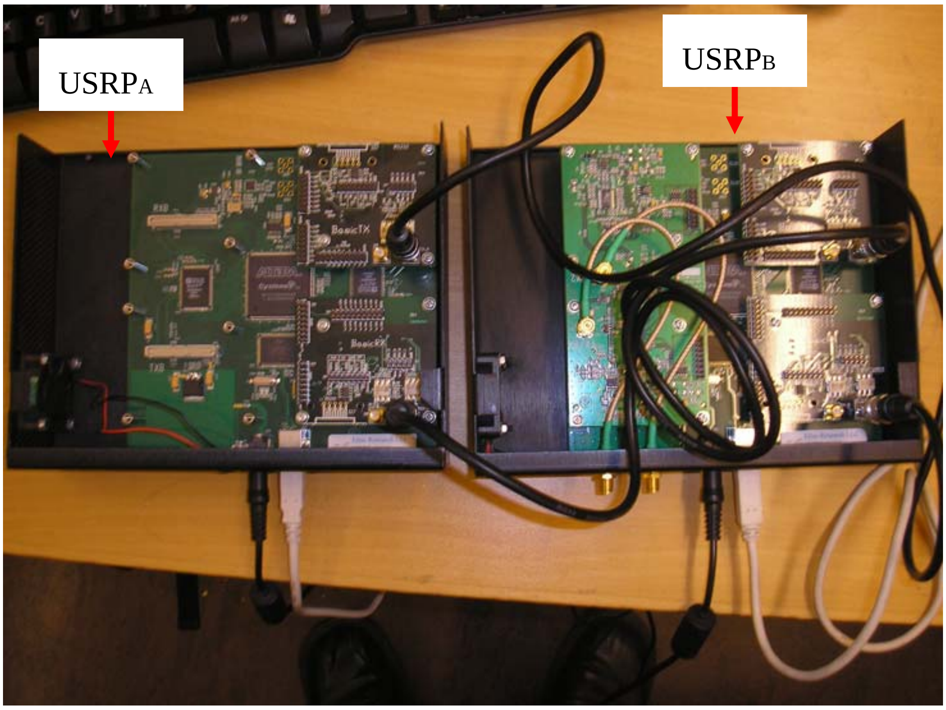


Figure 18: Connecting two USRP

2. On PCA open two terminals

    On the first terminal enter the following line which tells the program to use a frequency of 423 MHz with a 500 K bit per second and output some information for each packet in the terminal’s window.

    ```
    $./tunnel.py --freq 423.0M --bitrate 500k -v
    ```

    In the second terminal write the following line to configure interface gr0

    ```
    $ ifconfig gr0 192.168.200.1
    ```

3. On PCB open two terminals

    On the first terminal enter the following line which tells the program to use a frequency of 423 MHz with a 500 K bit per second and output some information for each packet in the terminal’s window.

    ```
    $./tunnel.py --freq 423.0M --bitrate 500k -v
    ```

    In the second terminal write the following line to configure interface gr0

    ```
    $ ifconfig gr0 192.168.200.2
    ```

The following is the output from the first window of the PCA.

```
# ./tunnel.py --freq 423.0M --bitrate 500k -v
>>> gr_fir_fff: using SSE
bits per symbol = 1
Gaussian filter bt = 0.35
Tx amplitude     0.25
modulation:      gmsk_mod
bitrate:         500kb/s
samples/symbol:    2
USRP Sink: A: Basic Tx
Requested TX Bitrate: 500k Actual Bitrate: 500k
bits per symbol = 1
M&M clock recovery omega = 2.000000
M&M clock recovery gain mu = 0.175000
M&M clock recovery mu = 0.500000
M&M clock recovery omega rel. limit = 0.005000
frequency error = 0.000000

Receive Path:
modulation:      gmsk_demod
bitrate:         500kb/s
samples/symbol:    2
USRP Source: A: Basic Rx
Requested RX Bitrate: 500k
Actual Bitrate: 500k
modulation:     gmsk
freq:           423M
bitrate:        500kb/sec
samples/symbol:   2
Carrier sense threshold: 30 dB

Allocated virtual ethernet interface: gr0
You must now use ifconfig to set its IP address. E.g.,

  $ sudo ifconfig gr0 192.168.200.1
```

Be sure to use a different address in the same subnet for each machine.

```
Tx: len(payload) =   90
Tx: len(payload) =   54
Tx: len(payload) =  153
Tx: len(payload) =   82
Tx: len(payload) =  235
Rx: ok = False  len(payload) =  235
Tx: len(payload) =   78
Tx: len(payload) =  235
```

5.3.3 Student Exercises

1. Open a terminal window on one machine and perform ping command. Look at the delay sending a packet from one machine to the other and back. Compare and analysis your results with Ethernet and IEEE 802.11b network.
2. Capture the traffic using Ethereal and analysis what you got.
3. Modify the code so that you can detect if there is any collision.

5.4 Experiment 4: Bluetooth (or IEEE 802.15.4) sniffer

Bluetooth is low rate low power wireless personal area network solution. Bluetooth devices operate at 2.4 GHz band. The 2.4 GHz band that is used is 83.5 MHz wide (from 2.400 to 2.435 GHz). This band is divided into 79 channels with a channel spacing of 1 MHz. Bluetooth uses spectrum that may be used by other wireless systems (i.e. IEEE 802.11 wireless local area networks, locators (such as used in anti-theft systems in vehicles), cordless telephones, etc.) and may cause interference to other wireless systems as well as receive interference from those other systems. Each Bluetooth device makes 1600 hops per second to implement a fast frequency hopping spread spectrum scheme (at 1/1600 hops per seconds this means that each transmission occurs in a 0.625 millisecond long time slot).

Each Bluetooth device is either a master or a slave. The master Bluetooth device is the device that initiates data exchange and the master Bluetooth device is the device that responds to the master. Both the master and slave devices must use the same sequence of frequency hops to communicate, the master device orders the clock of the piconets, where slaves keeps track of their clocks' offset form the master. In this experiment we will build an application to sniff Bluetooth packets.

It is hard to sniff Bluetooth because of its wide frequency band and fast random hopping (calculated by the master device). We need eleven USRPs to sniff the 83.5 MHz wide band (USRP can work with 8 MHz wide band centred in a frequency), or we can use four USRP2. see 6.2,

Table 7 compares between USRP and USRP2.

5.4.1 Bluetooth Implementation

In this experiment we will use gr-bluetooth. This code was developed by Dominic Spill and Michael Ossmann [35] and they made the code freely available [26]. In this experiment we will use in which the Bluetooth baseband layer for GNU Radio to implement the Bluetooth baseband processing. In this experiment students will see an example of a SDR. This SDR will be used to listen to packets exchanged between a cellular phone and a Bluetooth headset. Note that Bluetooth uses its own audio coding (using the SBC CODEC), but to listen to the audio requires installing this CODEC.

Bluetooth MAC address is Bluetooth Device Address (BD_ADDR) which is 48 bits comprised of three parts (see Figure 19). Local Area network Profile (LAP) is 24 bits section of the BD_ADDR, Address Portion UAP is 8 bits, and NAP is 16 bits. The NAP and UAP together expresses the company ID which is unique for each Bluetooth device.

NAP (16 bits)	UAP (8bits)	LAP (24 bits)

Figure 19: Bluetooth BD_ADDR

5.4.1.1 Equipment

- One USRP; with one RFX2400 installed (with reverse polarity SMA connector).
- One PC; GNU Radio installed.
- One 2.4 GHz antenna, this antenna we are using has the following specification

```
Part                 number:               30223
Type:                                       whip
Frequency:                 2.4               GHz
Gain:                    5                   dBi
Radiation            Angle:          H360°/V23°
Range:                    200                  m
Dimensions              (mm):            197x19
Contact:                                 Rev-SMA
Cable:                                        --
Trivia: Multiangle
```

5.4.2 Installing the system

The software consists of a signal processing block and a front-end command line tool. The code can be downloaded from the internet site http://sourceforge.net/projects/gr-bluetooth/. using a web browser, browse to this side and choose "file", then download gr-bluetooth-0.3.tar.gz, extend Samples and download gr-bluetooth-samples.tar.gz. Next follow the instructions below:

1. Open a terminal window and connect to the directory where you downloaded your files, then enter the following command to unpack and install the code:

$ tar -xzf gr-bluetooth-0.3.tar.gz

$ cd gr-bluetooth-0.3

$ ./configure

$ make

$ sudo make install

$ cd ..

2. Copy the file gr-bluetooth-samples.tar.gz to the directory gr-bluetooth/src/python, extract it and rename the output directory to sample. This can be done using the following commands (assuming that you have downloaded the files into the directory /tmp)

$ cp /tmp/gr-bluetooth-samples.tar.gz gr-bluetooth/src/python gr-bluetooth-samples.tar.gz

$ cd gr-bluetooth/src/python

$ tar –xzf gr-bluetooth-samples.tar.gz

$ mv gr-bluetooth-samples.tar.gz samples

3. Connect the USRP to your PC. See Figure 20.

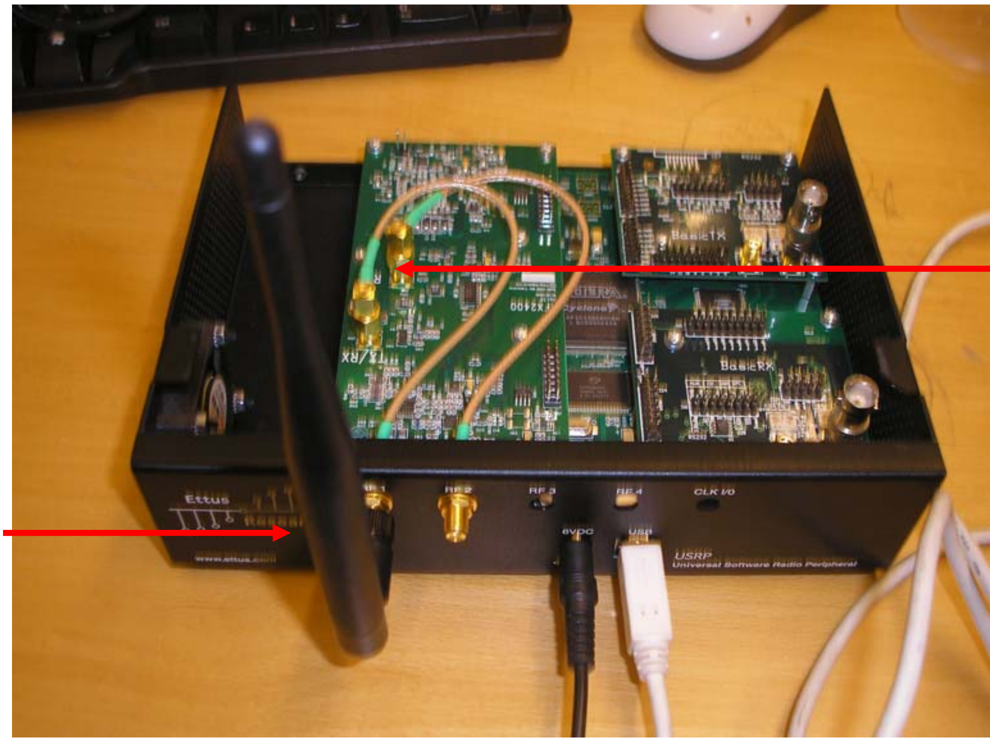


Figure 20: USRP 2.4 GHz Antenna (designed for use with WLAN devices)

5.4.3 Student Exercise

The student can carry a report to the instructor includes the solution of this exercises. This exercise uses captured files which are prepared in section 5.4.2 and no need for the USRP to solve the questions. captured files are:

- headset1.cfile: This sample file captured during a call between cell phone and Bluetooth headset at 2.4765 GHz centred frequency with 8 MHz bandwidth using usrp_rx_cfile.py.
- headset3.cfile: This sample file captured during a call between cell phone and Bluetooth headset at 2.476 GHz centred frequency with 2 MHz bandwidth using usrp_rx_cfile.py.
- keyboard1.cfile: This sample file captured during a keyboard typing rapidly plus idle cell phone and headset at 2.4765 GHz centred frequency with 8 MHz bandwidth usrp_rx_cfile.py.

The usrp_rx_cfile.py is part of GNU radio which used to read samples from the USRP and write to file formatted as binary outputs single precision complex float values or complex short values (interleaved 16 bit signed short integers).

This exercise is centred on btrx.py application located on gr-bluetooth/src/python directory.

Table 6 shows btrx.py options.

Table 6: btrx.py options.

code	Option
-h	Show this help message and exit
-N	Number of samples to collect
-R	Select USRP Rx side A or B
-S	All-piconet sniffer
-a	Using a particular aliasing receiver implementation
-c	Comma separated list of ddc frequencies
-e	Use specified Ethernet interface for USRP2
-d	Set fgpa decimation rate to DECIM
-f	Set USRP frequency to FREQ
-g	Set USRP gain in dB
-i	Use named input file instead of USRP
-l	LAP of the master device
-m	Use USRP2 at specified MAC address
-n	Channel number for hop reversal (0-78)
-p	Reverse hopping sequence to determine master clock
-r	Sample rate of input
-s	Input interleaved shorts instead of complex floats
-t	Power squelch threshold in dB
-w	Direct output to a tune interface
-2	use USRP2 (or file originating from USRP2) instead of USRP

1. Find packets and display Local Area network Profile (LAP) in headset3.cfile sample file.
2. Discover the Upper Address Portion (UAP) by CRC in keyboard1.cfile sample file.
3. Discover UAP/CLK1-6 by time interval in headset1.cfile sample file.
4. Decode all piconets on all available channels in keyboard1.cfile sample file.

5.5 Experiment 5: IEEE 802.11 Implementation

In this experiment we will use the BBN 802.11 implementation by the Adaptive Dynamic Radio Open-source Intelligent Team and funded by DARPA's ACERT program. This project used GNU Radio and implemented an 802.11 receiver and transmitter [28].

5.5.1 Requirements

- One USRP; with one RFX2400 installed (with reverse polarity SMA connector).
- One PC; GNU Radio version **3.1.1** installed.
- One 2.4 GHz antenna, this antenna we are using has the following specification

```
Part number: 30223
Type: whip
Frequency: 2.4 GHz
Gain: 5 dBi
Radiation Angle: H360°/V23°
Range: 200 m
Dimensions (mm): 197x19
Contact: Rev-SMA
Cable: --
Trivia: Multiangle
```

5.5.2 Installing BBN 802.11

This section describes how to build BBN 802.11. You will experience a problem installing BBN 802.11 as described in [29] and this release's build guide [25]. The problem is that BBN 802.11 is not longer available with this SVN version. However, you can get the correct code from the BBN80211 - The Comprehensive GNU Radio Archive Network [30]. You will see two versions (i.e. douggeiger for USRP-1 and usrp2_version), get the proper version according to your USRP device (we are using a USRP rather than the newer USRP2). To get the BBN 802.11 code do the following:

svn co https://128.2.212.19/cgran/projects/bbn_80211/branches/ douggeiger/

Before you install their code make sure that you have GNU Radio version **3.1.1** installed on your Linux platform; if you have version 3.2.2 you will receive the following error -- thus it is very important that you install the earlier version of the GNU Radio software. Now you can install the BBN code. If you have not installed the earlier version of the GNU radio code you will experience an error as shown below:

root@ala-laptop:/sdr/bbn/gr-bbn/src/examples# ./bbn_80211b_rx.py -f 2.437G -v -b

Traceback (most recent call last):

File "./bbn_80211b_rx.py", line 126, in <module>

main ()

File "./bbn_80211b_rx.py", line 121, in main

app = app_flow_graph()

File "./bbn_80211b_rx.py", line 109, in __init__

self.u = usrp_rx(options.decim, options.verbose, options.gain,options.freq)

```
 File "./bbn_80211b_rx.py", line 57, in __init__
   gr.hier_block2.__init__(self, "usrp_rx", gr.io_signature(0, 0, 0), gr.io_signature(1, 2, gr.sizeof_gr_complex))
 File "/usr/local/lib/python2.5/site-packages/gnuradio/gr/hier_block2.py",
line 42, in __init__
   self._hb = hier_block2_swig(name, input_signature, output_signature)
 File "/usr/local/lib/python2.5/site-packages/gnuradio/gr/gnuradio_swig_py_runtime.py",
line 995, in hier_block2_swig
   return _gnuradio_swig_py_runtime.hier_block2_swig(*args, **kwargs)
RuntimeError: Hierarchical blocks do not yet support arbitrary or
variable numbers of inputs or outputs (usrp_rx)
```

The problem is that this BBN code was not converted to use the hier_block2 API which is needed for GNU Radio version 3.2.0 and later.

If GNU Radio version **3.2.x** I already installed on your machine you have to delete all *gnuradio* directories and *usrp** files from /usr/local/, then install GNU Radio **3.1.1**. Finally go to douggeiger (you can change the douggeiger name and for directory organization point of view; we recommend to put BBN 802.11 in the dorectory gnuradio-3.1.1.) and do execute the commands: ./bootstrap && ./configure && make && sudo make install.

5.5.3 Setup and Implementation

In this exercise you need one USRP with RFX2400 daughter board installed and 2.4 GHz antenna; as described in 5.5.2. See Figure 21.

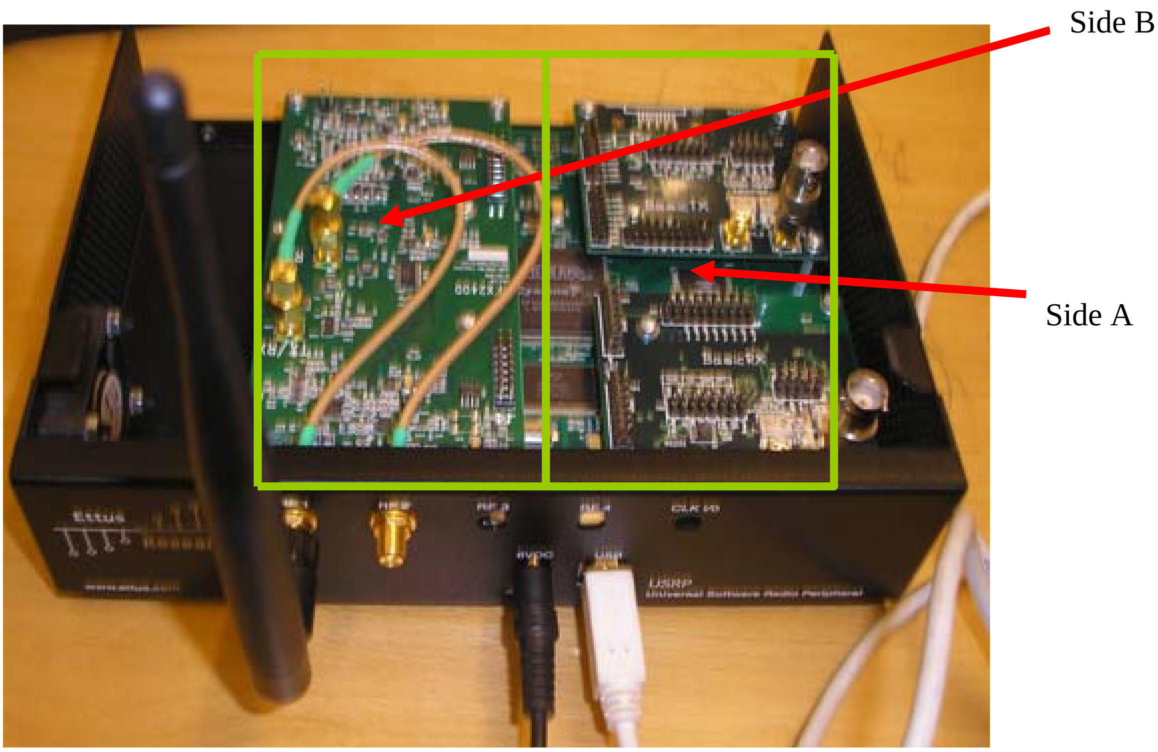


Figure 21: RFX2400, with an Antenna. Note the two sides of the USRP (A and B)

1. Open a terminal window and connect to the directory gr-bbn/src/examples/, then run the receiver by entring:

./ bbn_80211b_rx.py -R B -f 2.437G -v –b

This will tell the program to use the receiver on “B” (-R B) side, the frequency 2.437 GHZ (-f 2.437G), verbose (-v), and Barker Spreading (-b).

The output will be similar to that shown below.

```
ala@hlllab2:~/gnuradio/gnuradio-3.1.1/adroitgrdevel/gr-bbn/src/examples> bbn_80211b_rx.py -R  B -f 2.437G -v -b

Bits Per Encoded Sample = 8

adc frequency =  64000000

decimation frequency =  16

input_rate =  4000000

gain =  45.0

desired freq =  2437000000.0

baseband frequency 2432000000.0

dxc frequency -5000000.0

Samples per data bit =  8

>>> gr_fir_ccf: using SSE

gr_vmcircbuf_createfilemapping: createfilemapping is not available

PKT: len=84, rssi=-43, src=00:1a:70:3e:4F:29, time=15856, rate=1 Mbps

PKT: len=84, rssi=-40, src=00:1a:70:3e:4F:29, time=18280, rate=1 Mbps

PKT: len=84, rssi=-40, src=00:1a:70:3e:4F:29, time=19664, rate=1 Mbps

PKT: len=84, rssi=-43, src=00:1a:70:3e:4F:29, time=21000, rate=1 Mbps

PKT: len=84, rssi=-41, src=00:1a:70:3e:4F:29, time=34456, rate=1 Mbps
```

2- Open a new terminal and run the transmitter (you will see your frame captured by the receiver) do this:

./ bbn_80211b_tx.py -T B -f 2.437G -b

This will tell the program to use the transmitter on “B” (-T B) side, the frequency 2.437 GHZ (-f 2.437G), and Barker Spreading (-b)

The output will be similar to that shown below.

```
ala@hlllab2:~/gnuradio/gnuradio-3.1.1/adroitgrdevel/gr-bbn/src/examples> bbn_80211b_tx.py -T  B -f 2.437G -b

Using TX d'board B: Flex 2400 Tx MIMO B

>>> gr_fir_ccf: using SSE

spb:      8

interp:   32
```

The output on the receiver terminal will be similar to that shown below.

```
ala@hlllab2:~/gnuradio/gnuradio-3.1.1/adroitgrdevel/gr-bbn/src/examples> bbn_80211b_rx.py -R  B -f 2.437G -v -b
```

Bits Per Encoded Sample = 8

adc frequency = 64000000

decimation frequency = 16

input_rate = 4000000

gain = 45.0

desired freq = 2437000000.0

baseband frequency 2432000000.0

dxc frequency -5000000.0

Samples per data bit = 8

>>> gr_fir_ccf: using SSE

gr_vmcircbuf_createfilemapping: createfilemapping is not available

uO

“uO” means USRP overrun (USRP samples dropped because they weren't read in time.

5.5.4 Student Exercises

1. Give examples of how to receive packets from bbn_80211b_rx.py (without dropping them).
2. Look at all sniffed packets and check the rate of each packet. Do you think this system is a full IEEE 802.11b sniffer? Why?

6. Evaluation and Analysis

In this section we will evaluate each of the laboratory experiment from a pedagogical point of view. We should start by noting that these experiments target senior undergraduate student and instructors. The undergraduate student must have studied the following subjects attempting these laboratory experiment s:

1. The student need to have studied at least one high level programming language, preferably object oriented programming language. This will enable the student to understand and the GNU Radio code.
2. Communication systems and computer networks.
3. Signals and systems
4. Digital signal processing.

6.1 GNU Radio: Analysis

The GNU Radio provides a extensiv library of signal processing blocks and a glue to tie thises blocks. The radio can be build by creating a flow graph. The signal processing blocks are implemented in C++ programing language, while programers construc the graph and run them in Python.

There is no enghough documentation of how GNU Radio is implemented, during runig application we found that there are some messages printed from diffrernt *classes* (for example see 5.2.3) and tracing and understanding these message takes some time. The GNU Radio developers did not found acceptable way to provide unifed documentation for the system [38]. However, there is some documentatins for GNU Radio C++ blocks, and you can get help from other developers in [39].

The Gnu Radio has many releases developed. In release version **3.2.x** the higher block of the system is updated. This will affect applications developed under old release from running in new releses.

6.2 USRP: Analysis

The USRP is a device we used in this thesis to develop undergraduate’s experiment . This device has various daughterboards which operate on different radio frequency bands (from DC to 2.9 GHz); you have to plug-in a sutable daughterboard for you application.

When we are running our applications we experience that closing application using Cotrol+z will not flush the application running process; is you are going to run any application after the one application you will receive a error. We used to unplug the USRP DC power off, and then pulg it in again. Another solution introduced to us is to see all running process, and then “kill” Python process. You can do that by:

ps (to see all running process. find the number of Python application process and enter)

kill -9 <Python process number>

USRP2 was developed and goes to the market on May 25, 2009. There are some benefits of using USRP2 than USRP,

Table 7 describes these benefits:

Table 7: USRP and USRP2 [15]

USRP	**USRP2**
8 MHz instantaneous of RF bandwidth	25 MHz instantaneous of RF bandwidth
The radio can be accessible from one computer	the radio to be accessible from more than one computer
USB interface	Gigabit Ethernet interfaces
Lowest cost	Highest cost
Slower FPGA	Faster FPGA
ADCs (12-bits 64 MS/s)	ADCs (14-bits 100 MS/s)
DACs (140bits 128 MS/s)	DACs (16-bits 400 MS/s)

6.1 Laboratory exercises: Analysis

The laboratory exercises were designed based upon the idea of step-by-step learning. The undergraduate student initialy follow the steps presented in each experiment to solve a problem and understands subject terms. These experiments start with simple communication systems first, a little bit complex systems, and finaly real world systems. In each experiment, the student must solve specific problems and submit a written report to the instructor. The instructor can choose which experiment are sutable for the students.

Experiment 1: simplex data communication, in a simple application to data transmission. In this initial experiment , the student can exmine in detalis how the three lower layers of the OSI model are implemented and different methods of modulation can be used – while supporting the same higher level protocol. The student can develop a ptotocol in any packet format, and can use the code represented for this exprimet to develop a feedback from the receiver.

Experiment 2 shows the student encoded voice data can be transmitted over a digital channel. In the exercises the student is ask to think about what would happen if the cable were replaced with a pair of antennas and the RF signal were to be transmitted and transmitt on the air.

Experiment 3 exposed the student to a spesific MAC layer protocol. The student see how he or she might can bulid his or her own MAC protocol A central element of this experiment is that the MAC protocol simply implements a protocol. (An optional exercise for this experiment would be to ask the student to write the protocol specification that is actually implemented by the code.). It is difficult for the student to implemetnt CSMA/with collition detection, because of the anttena power limitation. However, the student can implement CSMA/with collition avoidance.

Experiment 4 takes students deep to Bluetooth protocol. This experiment illustrates some very sophisticated aspects of protocol analysis and has some important observations for student's about the lack of security through obscurity (specifically that fast frequency hopping and not putting the complete MAC address in Bluetooth frames does not prevent someone from listing to these packets nor does it hide the devices), and let bluetooth works.

Experiment 5 tells the student how to implement 802.11 protocol. This experiment is sutable only for last year undergraduate students. Moreover we can use it only with IEEE 802.11 and **not** IEEE 802.11b because of the limitation of USB2 transmission.

7. Conclusions and suggested future work

We developed laboratory experiment for undergraduate students to help them understands media and access control protocols protocol. The experiments are designed in a way that easy to understand experiments first, and the complicated experiments. Instructors might use these experiments and add more exercises to develop their own lessons plan and course material.

In this thesis we present software defined radio application built on USRP and GNU Radio. Thus, our first goal was achieved. However, we did **not** develop our own application using USRP and GNU Radio, which is goal two. If we look at the laboratory experiments we can see that it includes different kind of applications, in which we spend our time. But if this thesis was designed to build specific application using USRP and GNU Radio, then we can spend our time on single application. Moreover, the development time for applications using USRP and GNU Radio is varied form application to other. For example, Bluetooth (or IEEE 802.15.4) sniffer developed by two developers and they spends three months to make it running.

In conclusion, we can say that it is not easy job to implement applications using USRP and GNU Radio because of the weak documentation of the GNU Radio. And if we started this thesis again we would develop a documentations tool for GNU Radio to help developers to implement their own applications.

The computer science department of Grove City College, has developed some exercises based on SDR for undergraduate projects [13]. These exercises enable students to receive real-time waveforms; specifically to receive AM, FM, and SSB signals. They are reported to be developing a plug-in for commercial radio broadcasts in which an AM radio will have the current FM station quality and the quality of broadcast FM stations will be CD quality.

Compared to our solution, the Grove City College research targets broadcast radio, while we focus on wireless local area networks and personal area networks.

7.1 Future work

1) Create experiments based on GNU Radio Companion (a graphical tool for creating signal graph to generate flow graph source code) [31].
2) Create experiments base simulink [32].
3) Create experiment to listen to a GSM cell phone [33].
4) Create experiment for ZigBee.
5) Create fully a receiver experiment for 802.11b

References

[1] Susan Karlin,”Tools & Toys: Hardware for your Software Radio”, IEEE Spectrum, 34(10) , Oct. 2006, pp51-54.

[2] Paul Burns, ”Software Defined Radio for 3G”. London. Artech House, 2003.

[3] Bruce A. Fette, et al, ”Cognitive Radio Technology” Newnes, 2006, 656 pages, ISBN-10: 0750679522, ISBN-13: 978-0750679527.

[4] T.W. Parks and J.J. McClellan, “Chebyshev Approximation for Nonrecursive Digital Filters with Linear Phase,” IEEE Transactions on Circuit Theory, Vol. 19, 1972, pp. 189–194.

[5] L.R. Rabiner, J.H. McClellan, and T.W. Parks, “FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximations,” Proceedings of the IEEE, Vol. 63, 1975, pp. 595–610.

[6] Walter Tuttlebee, et al, ”Software Defined Radio: Enabling Technology”, USA, John Wiley & Sons Ltd, 2002.

[7] John Bard and Vincent J. Kovarik Jr., ”Software Defined Radio: the software communication Architecture”, John Wiley & Sons Ltd, 2007.

[8] GNU Radio, web page” www.gnuradio.org “, visited 2009-02-10

[9] F. Ge, Q. Chen, Y. Wang, T. W. Rondeau, B. Le, and C. W. Bostian, "Cognitive Radio: From Spectrum Sharing to Adaptive Learning and Reconfiguration," in IEEE Aerospace Conference. Big Sky Montana, MT, March 2008.

[10] Python Programming Language, web page “ www.python.org ”, visited 2009-02-10.

[11] Peter Norton, Alex Samuel, David Aitel, Eric Foster-Johnson, Leonard Richardson, Jason Diamond, Aleatha Parker, and Michael Roberts, Beginning Python, Canada, Wiley Publishing, Inc, 2005.

[12] SDR Forum, Web site “ www.sdrforum.org ”, visited 2009-02-10.

[13] William Birmingham and Leah Acker, ”Software-defined radio for undergraduate projects”, ACM, session:Embded systems and architecture, Volume 39, Issue 1, March 2007, pp. 293 – 297, ISSN:0097-8418.

[14] Open Source SCA Implementation - Embedded, web page “http://ossie.wireless.vt.edu/index.html”, visited 2009-02-11.

[15] Ettus Research LLC, web page ” www.ettus.com”, visited 2009-11-11.

[16] General Dynamics C4 Systems, Web site ” www.gdc4s.com” visited 2009-07-01.

[17] Analog Design, AD9862 12-/14-Bit Mixed Signal Front-End (MxFE®) Processor for Broadband Communications, Data Sheet, Revision 0, Dec. 2002, internet site "http://www.analog.com/static/imported-files/data_sheets/AD9860_9862.pdf"

[18] IEEE, Standard Hardware Description Language Based on the Verilog® Hardware Description Language –Description, IEEE Std 1364-1995, IEEE, Oct. 1996, ISBN: 1-55937-727-5, E-ISBN: 0-7381-3065-6.

[19] IEEE SystemVerlog Working Group, IEEE 1800, Web Site ” http://www.eda.org/sv-ieee1800/”.

[20] Deepak Kumar Tala, Verilog Tutorial, web page, Jan. 10, 2009 http://www.asic-world.com/verilog/veritut.html.

[21] Eric Blossom, "Exploring GNU Radio: Tools fo Exploring the RF spectrum", Linux Journal, Issue 122, June 2004.

[22] Eric Blossom, How to Write a Signal Processing Block, Web page, Jul 21,2006, http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html.

[23] Greg Colvin and Beman Dawes, Smart Pointers, Web Page, March 11, 2009, http://www.boost.org/libs/smart_ptr/smart_ptr.htm.

[24] Joseph Mitola III,web page, Nov 21, 2008 ”http://web.it.kth.se/~maguire/jmitola/”.

[25] Build Guide- GNU Radio, web page,2009-11-05 ” http://gnuradio.org/trac/wiki/BuildGuide”.

[26] Gr-Bluetooth, web page, Aug 18, 2009,” http://sourceforge.net/projects/gr-bluetooth/”

[27] “Wireless an Mobile Network Architecture”, *G.Q.Maguire Jr.* <maguire@kth.se>, http://www.it.kth.se/courses/IK2555/Coursepage-Spring-2010.html

[28] Troxel Gregory D, Blossom Eric, et al “Adaptive Dynamic Radio Open-source Intelligent Team (ADROIT): Cognitively-controlled Collaboration among SDR Nodes”, Networking Technologies for Software Defined Radio Networks, 2006. SDR '06.1st IEEE Workshop, Sep 2006, pp 8-17, ISBN: 1-4244-0733-8.

[29] Other Code – GNU Radio, web page, “http://gnuradio.org/trac/wiki/OtherCode”, visited Nov 7, 2009.

[30] BBN80211 - The Comprehensive GNU Radio Archive Network, web page, “https://128.2.212.19/wiki/BBN80211”, visited Nov 7, 2009.

[31] GNU Radio Companion, web page, “http://gnuradio.org/trac/wiki/GNURadioCompanion” , visited Nov 7,2009.

[32] Simulink-USRP: Universal Software Radio Peripheral (USRP) Blockset, web page, visited Nov 7, 2009.

[33] The NetBSD Packages Collection, web page, “http://ftp.sunet.se/pub/NetBSD/packages/pkgsrc/ham/gnuradio-gsm/README.html”, visited Nov 7,2009.

[34] Alister Burr, “Modulation and coding: for wireless communications”, Prentice Hall/Pearson Education, 2001, ISBN: 0201398575.

[35] Michael Ossmann and Dominic Spill, "Building an All-Challe Bluetooth Monitor", ShmooCon 2009, 6 February 2009.

[36] Costa A. j. et. al., “Spectrum analyzer with USRP, GNU Radio and MATLAB”, 7th Conference on Telecommunication, Portugal, May 2009.

[37] Discuss-Gnuradio Archives, web page, “http://lists.gnu.org/archive/html/discuss-gnuradio/”.

[38] GNU Radio 3.2svn C++ API Documentation, web page, “http://gnuradio.org/doc/doxygen/index.html”, May 22, 2009.

Appendix A: gr_block.h

```
// gr_block.h
/* -*- 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.
 */

#ifndef INCLUDED_GR_BLOCK_H
#define INCLUDED_GR_BLOCK_H

#include <gr_runtime.h>
#include <string>

class gr_block {

 public:

  virtual ~gr_block ();

  std::string name () const { return d_name; }
  gr_io_signature_sptr input_signature () const  { return d_input_signature; }
  gr_io_signature_sptr output_signature () const { return d_output_signature; }
  long unique_id () const { return d_unique_id; }

  unsigned history () const { return d_history; }
  void  set_history (unsigned history) { d_history = history; }

  bool fixed_rate() const { return d_fixed_rate; }

  // ----------------------------------------------------------------
  //            override these to define your behavior
  // ----------------------------------------------------------------
```

```
virtual void forecast (int noutput_items,
                       gr_vector_int &ninput_items_required);

virtual int general_work (int noutput_items,
                          gr_vector_int &ninput_items,
                          gr_vector_const_void_star &input_items,
                          gr_vector_void_star &output_items) = 0;

virtual bool check_topology (int ninputs, int noutputs);

virtual bool start();

virtual bool stop();

// ----------------------------------------------------------------

void set_output_multiple (int multiple);
int  output_multiple () const { return d_output_multiple; }

void consume (int which_input, int how_many_items);

void consume_each (int how_many_items);

void  set_relative_rate (double relative_rate);

double relative_rate () const { return d_relative_rate; }

/*
 * The following two methods provide special case info to the
 * scheduler in the event that a block has a fixed input to output
 * ratio.  gr_sync_block, gr_sync_decimator and gr_sync_interpolator
 * override these.  If you're fixed rate, subclass one of those.
 */
virtual int fixed_rate_ninput_to_noutput(int ninput);

virtual int fixed_rate_noutput_to_ninput(int noutput);

// ----------------------------------------------------------------------------

private:

std::string           d_name;
gr_io_signature_sptr  d_input_signature;
gr_io_signature_sptr  d_output_signature;
int                   d_output_multiple;
double                d_relative_rate;        // approx output_rate / input_rate
gr_block_detail_sptr  d_detail;               // implementation details
long                  d_unique_id;            // convenient for debugging
```

```
  unsigned             d_history;
  bool                 d_fixed_rate;


protected:

  gr_block (const std::string &name,
            gr_io_signature_sptr input_signature,
            gr_io_signature_sptr output_signature);

  void set_input_signature (gr_io_signature_sptr iosig){
    d_input_signature = iosig;
  }

  void set_output_signature (gr_io_signature_sptr iosig){
    d_output_signature = iosig;
  }

  void set_fixed_rate(bool fixed_rate){ d_fixed_rate = fixed_rate; }

  // These are really only for internal use, but leaving them public avoids
  // having to work up an ever-varying list of friends

public:
  gr_block_detail_sptr detail () const { return d_detail; }
  void set_detail (gr_block_detail_sptr detail) { d_detail = detail; }
};

long gr_block_ncurrently_allocated ();

#endif /* INCLUDED_GR_BLOCK_H */
```

Appendix B: Laboratory Experiments

Appendix B.1 benchmark_tx.py

```
#!/usr/bin/env python
#
# Copyright 2005,2006,2007,2009 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 3, 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., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random, time, struct, sys

# from current dir
import usrp_transmit_path

#import os
#print os.getpid()
```

```python
#raw_input('Attach and press enter')

class my_top_block(gr.top_block):
    def __init__(self, modulator, options):
        gr.top_block.__init__(self)

        self.txpath = usrp_transmit_path.usrp_transmit_path(modulator, options)

        self.connect(self.txpath)

# /////////////////////////////////////////////////////////////////////////////
#                                    main
# /////////////////////////////////////////////////////////////////////////////

def main():

    def send_pkt(payload='', eof=False):
        return tb.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        print "ok = %r, payload = '%s'" % (ok, payload)

    mods = modulation_utils.type_1_mods()

    parser = OptionParser(option_class=eng_option, conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice", choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(mods.keys()),))

    parser.add_option("-s", "--size", type="eng_float", default=1500,
                      help="set packet size [default=%default]")
    parser.add_option("-M", "--megabytes", type="eng_float", default=1.0,
                      help="set megabytes to transmit [default=%default]")
    parser.add_option("","--discontinuous", action="store_true", default=False,
                      help="enable discontinous transmission (bursts of 5 packets)")
    parser.add_option("","--from-file", default=None,
```

```
                  help="use file for packet contents")

usrp_transmit_path.add_options(parser, expert_grp)

for mod in mods.values():
    mod.add_options(expert_grp)

(options, args) = parser.parse_args ()

if len(args) != 0:
    parser.print_help()
    sys.exit(1)

if options.tx_freq is None:
    sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
    parser.print_help(sys.stderr)
    sys.exit(1)

if options.from_file is not None:
    source_file = open(options.from_file, 'r')

# build the graph
tb = my_top_block(mods[options.modulation], options)

r = gr.enable_realtime_scheduling()
if r != gr.RT_OK:
    print "Warning: failed to enable realtime scheduling"

tb.start()                       # start flow graph

# generate and send packets
nbytes = int(1e6 * options.megabytes)
n = 0
pktno = 0
pkt_size = int(options.size)

while n < nbytes:
    if options.from_file is None:
        data = (pkt_size - 2) * chr(pktno & 0xff)
```

```
        else:
            data = source_file.read(pkt_size - 2)
            if data == '':
                break;

        payload = struct.pack('!H', pktno & 0xffff) + data
        send_pkt(payload)
        n += len(payload)
        sys.stderr.write('.')
        if options.discontinuous and pktno % 5 == 4:
            time.sleep(1)
        pktno += 1

    send_pkt(eof=True)

    tb.wait()                       # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Appendix B.2 benchmark_rx.py

```
#!/usr/bin/env python
#
# Copyright 2005,2006,2007,2009 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 3, 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., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import struct
import sys

# from current dir
import usrp_receive_path

#import os
#print os.getpid()
```

```
#raw_input('Attach and press enter: ')

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

        # Set up receive path
        self.rxpath = usrp_receive_path.usrp_receive_path(demodulator, rx_callback, options)

        self.connect(self.rxpath)

# /////////////////////////////////////////////////////////////////////////////
#                                   main
# /////////////////////////////////////////////////////////////////////////////

global n_rcvd, n_right

def main():
    global n_rcvd, n_right

    n_rcvd = 0
    n_right = 0

    def rx_callback(ok, payload):
        global n_rcvd, n_right
        (pktno,) = struct.unpack('!H', payload[0:2])
        n_rcvd += 1
        if ok:
            n_right += 1

        print "ok = %5s  pktno = %4d  n_rcvd = %4d  n_right = %4d" % (
            ok, pktno, n_rcvd, n_right)


    demods = modulation_utils.type_1_demods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option, conflict_handler="resolve")
```

Line

```
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice", choices=demods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(demods.keys()),))

    usrp_receive_path.add_options(parser, expert_grp)

    for mod in demods.values():
        mod.add_options(expert_grp)

    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    # build the graph
    tb = my_top_block(demods[options.modulation], rx_callback, options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    tb.start()        # start flow graph
    tb.wait()         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
```

```
        pass
```

Appendix B.3 tx_voice.py

```
#!/usr/bin/env python
#
# Copyright 2005,2006,2007,2009 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 3, 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., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import audio
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

from gnuradio.vocoder import gsm_full_rate

import random
import time
import struct
import sys

# from current dir
```

```python
import usrp_transmit_path

#import os
#print os.getpid()
#raw_input('Attach and press enter')


class audio_rx(gr.hier_block2):
    def __init__(self, audio_input_dev):
        gr.hier_block2.__init__(self, "audio_rx",
                                gr.io_signature(0, 0, 0), # Input signature
                                gr.io_signature(0, 0, 0)) # Output signature
        sample_rate = 8000
        src = audio.source(sample_rate, audio_input_dev)
        src_scale = gr.multiply_const_ff(32767)
        f2s = gr.float_to_short()
        voice_coder = gsm_full_rate.encode_sp()
        self.packets_from_encoder = gr.msg_queue()
        packet_sink = gr.message_sink(33, self.packets_from_encoder, False)
        self.connect(src, src_scale, f2s, voice_coder, packet_sink)

    def get_encoded_voice_packet(self):
        return self.packets_from_encoder.delete_head()


class my_top_block(gr.top_block):

    def __init__(self, modulator_class, options):
        gr.top_block.__init__(self)
        self.txpath = usrp_transmit_path.usrp_transmit_path(modulator_class, options)
        self.audio_rx = audio_rx(options.audio_input)
        self.connect(self.txpath)
        self.connect(self.audio_rx)


# /////////////////////////////////////////////////////////////////////////////
#                                   main
# /////////////////////////////////////////////////////////////////////////////
```

```
def main():

    def send_pkt(payload='', eof=False):
        return tb.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        print "ok = %r, payload = '%s'" % (ok, payload)

    mods = modulation_utils.type_1_mods()

    parser = OptionParser(option_class=eng_option, conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice", choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(mods.keys()),))
    parser.add_option("-M", "--megabytes", type="eng_float", default=0,
                      help="set megabytes to transmit [default=inf]")
    parser.add_option("-I", "--audio-input", type="string", default="",
                      help="pcm input device name.  E.g., hw:0,0 or /dev/dsp")
    usrp_transmit_path.add_options(parser, expert_grp)

    for mod in mods.values():
        mod.add_options(expert_grp)

    parser.set_defaults(bitrate=50e3)  # override default bitrate default
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        sys.exit(1)

    if options.tx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)
```

```
    # build the graph
    tb = my_top_block(mods[options.modulation], options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: failed to enable realtime scheduling"


    tb.start()                        # start flow graph

    # generate and send packets
    nbytes = int(1e6 * options.megabytes)
    n = 0
    pktno = 0

    while nbytes == 0 or n < nbytes:
        packet = tb.audio_rx.get_encoded_voice_packet()
        s = packet.to_string()
        send_pkt(s)
        n += len(s)
        sys.stderr.write('.')
        pktno += 1

    send_pkt(eof=True)
    tb.wait()                         # wait for it to finish


if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pa
```

Appendix B.4 rx_voice.py

```
#!/usr/bin/env python
#
# Copyright 2005,2006,2009 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 3, 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., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
from gnuradio import audio
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

```

```
from gnuradio.vocoder import gsm_full_rate

import random
import struct
import sys

# from current dir
import usrp_receive_path

#import os
#print os.getpid()
#raw_input('Attach and press enter')


class audio_tx(gr.hier_block2):
    def __init__(self, audio_output_dev):
        gr.hier_block2.__init__(self, "audio_tx",
                                gr.io_signature(0, 0, 0), # Input signature
                                gr.io_signature(0, 0, 0)) # Output signature

        self.packet_src = gr.message_source(33)
        voice_decoder = gsm_full_rate.decode_ps()
        s2f = gr.short_to_float ()
        sink_scale = gr.multiply_const_ff(1.0/32767.)
        audio_sink = audio.sink(8000, audio_output_dev)
        self.connect(self.packet_src, voice_decoder, s2f, sink_scale, audio_sink)

    def msgq(self):
        return self.packet_src.msgq()

```

```python
class my_top_block(gr.top_block):
    def __init__(self, demod_class, rx_callback, options):
        gr.top_block.__init__(self)
        self.rxpath = usrp_receive_path.usrp_receive_path(demod_class, rx_callback, options)
        self.audio_tx = audio_tx(options.audio_output)
        self.connect(self.rxpath)
        self.connect(self.audio_tx)

# /////////////////////////////////////////////////////////////////////////////
#                                   main
# /////////////////////////////////////////////////////////////////////////////

global n_rcvd, n_right

def main():
    global n_rcvd, n_right

    n_rcvd = 0
    n_right = 0

    def rx_callback(ok, payload):
        global n_rcvd, n_right
        n_rcvd += 1
        if ok:
            n_right += 1

        tb.audio_tx.msgq().insert_tail(gr.message_from_string(payload))

        print "ok = %r  n_rcvd = %4d  n_right = %4d" % (
            ok, n_rcvd, n_right)

```

```python
    demods = modulation_utils.type_1_demods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option, conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")

                parser.add_option("-m", "--modulation", type="choice", choices=demods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
                            % (', '.join(demods.keys()),))
    parser.add_option("-O", "--audio-output", type="string", default="",
                      help="pcm output device name.  E.g., hw:0,0 or /dev/dsp")
    usrp_receive_path.add_options(parser, expert_grp)

    for mod in demods.values():
        mod.add_options(expert_grp)

    parser.set_defaults(bitrate=50e3)  # override default bitrate default
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    # build the graph
```

```
    tb = my_top_block(demods[options.modulation], rx_callback, options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    tb.run()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Appendix B.5 tunnel.py

```
#!/usr/bin/env python
#
# Copyright 2005,2006,2009 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 3, 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., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#


# ////////////////////////////////////////////////////////////////////////////
#
#    This code sets up up a virtual ethernet interface (typically gr0),
#    and relays packets between the interface and the GNU Radio PHY+MAC
#
#    What this means in plain language, is that if you've got a couple
#    of USRPs on different machines, and if you run this code on those
#    machines, you can talk between them using normal TCP/IP networking.
#
# ////////////////////////////////////////////////////////////////////////////


from gnuradio import gr, gru, modulation_utils
from gnuradio import usrp
```

Line

```python
from gnuradio import eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser

import random
import time
import struct
import sys
import os

# from current dir
import usrp_transmit_path
import usrp_receive_path

#print os.getpid()
#raw_input('Attach and press enter')


# ////////////////////////////////////////////////////////////////////////////
#
#   Use the Universal TUN/TAP device driver to move packets to/from kernel
#
#   See /usr/src/linux/Documentation/networking/tuntap.txt
#
# ////////////////////////////////////////////////////////////////////////////

# Linux specific...
# TUNSETIFF ifr flags from <linux/tun_if.h>

IFF_TUN         = 0x0001   # tunnel IP packets
IFF_TAP         = 0x0002   # tunnel ethernet frames
IFF_NO_PI       = 0x1000   # don't pass extra packet info
IFF_ONE_QUEUE   = 0x2000   # beats me ;)

def open_tun_interface(tun_device_filename):
    from fcntl import ioctl

    mode = IFF_TAP | IFF_NO_PI
```

```python
    TUNSETIFF = 0x400454ca

    tun = os.open(tun_device_filename, os.O_RDWR)
    ifs = ioctl(tun, TUNSETIFF, struct.pack("16sH", "gr%d", mode))
    ifname = ifs[:16].strip("\x00")
    return (tun, ifname)


# /////////////////////////////////////////////////////////////////////////////
#                             the flow graph
# /////////////////////////////////////////////////////////////////////////////

class my_top_block(gr.top_block):

    def __init__(self, mod_class, demod_class,
                 rx_callback, options):

        gr.top_block.__init__(self)
        self.txpath = usrp_transmit_path.usrp_transmit_path(mod_class, options)
        self.rxpath = usrp_receive_path.usrp_receive_path(demod_class, rx_callback, options)
        self.connect(self.txpath)
        self.connect(self.rxpath)

    def send_pkt(self, payload='', eof=False):
        return self.txpath.send_pkt(payload, eof)

    def carrier_sensed(self):
        """
        Return True if the receive path thinks there's carrier
        """
        return self.rxpath.carrier_sensed()


# /////////////////////////////////////////////////////////////////////////////
#                           Carrier Sense MAC
# /////////////////////////////////////////////////////////////////////////////

class cs_mac(object):
```

```
"""
Prototype carrier sense MAC

Reads packets from the TUN/TAP interface, and sends them to the PHY.
Receives packets from the PHY via phy_rx_callback, and sends them
into the TUN/TAP interface.

Of course, we're not restricted to getting packets via TUN/TAP, this
is just an example.
"""
def __init__(self, tun_fd, verbose=False):
    self.tun_fd = tun_fd       # file descriptor for TUN/TAP interface
    self.verbose = verbose
    self.tb = None             # top block (access to PHY)

def set_top_block(self, tb):
    self.tb = tb

def phy_rx_callback(self, ok, payload):
    """
    Invoked by thread associated with PHY to pass received packet up.

    @param ok: bool indicating whether payload CRC was OK
    @param payload: contents of the packet (string)
    """
    if self.verbose:
        print "Rx: ok = %r  len(payload) = %4d" % (ok, len(payload))
    if ok:
        os.write(self.tun_fd, payload)

def main_loop(self):
    """
    Main loop for MAC.
    Only returns if we get an error reading from TUN.

    FIXME: may want to check for EINTR and EAGAIN and reissue read
    """
    min_delay = 0.001               # seconds
```

```
        while 1:
            payload = os.read(self.tun_fd, 10*1024)
            if not payload:
                self.tb.send_pkt(eof=True)
                break

            if self.verbose:
                print "Tx: len(payload) = %4d" % (len(payload),)

            delay = min_delay
            while self.tb.carrier_sensed():
                sys.stderr.write('B')
                time.sleep(delay)
                if delay < 0.050:
                    delay = delay * 2       # exponential back-off

            self.tb.send_pkt(payload)


# /////////////////////////////////////////////////////////////////////////////
#                                   main
# /////////////////////////////////////////////////////////////////////////////

def main():

    mods = modulation_utils.type_1_mods()
    demods = modulation_utils.type_1_demods()

    parser = OptionParser (option_class=eng_option, conflict_handler="resolve")
    expert_grp = parser.add_option_group("Expert")
    expert_grp.add_option("", "--rx-freq", type="eng_float", default=None,
                          help="set Rx frequency to FREQ [default=%default]", metavar="FREQ")
    expert_grp.add_option("", "--tx-freq", type="eng_float", default=None,
                          help="set transmit frequency to FREQ [default=%default]", metavar="FREQ")
    parser.add_option("-m", "--modulation", type="choice", choices=mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%default]"
```

```
                         % (', '.join(mods.keys()),))

    parser.add_option("-v","--verbose", action="store_true", default=False)
    expert_grp.add_option("-c", "--carrier-threshold", type="eng_float", default=30,
                          help="set carrier detect threshold (dB) [default=%default]")
    expert_grp.add_option("","--tun-device-filename", default="/dev/net/tun",
                          help="path to tun device file [default=%default]")

    usrp_transmit_path.add_options(parser, expert_grp)
    usrp_receive_path.add_options(parser, expert_grp)

    for mod in mods.values():
        mod.add_options(expert_grp)

    for demod in demods.values():
        demod.add_options(expert_grp)

    (options, args) = parser.parse_args ()
    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    # open the TUN/TAP interface
    (tun_fd, tun_ifname) = open_tun_interface(options.tun_device_filename)

    # Attempt to enable realtime scheduling
    r = gr.enable_realtime_scheduling()
    if r == gr.RT_OK:
        realtime = True
    else:
        realtime = False
        print "Note: failed to enable realtime scheduling"



    # If the user hasn't set the fusb_* parameters on the command line,
    # pick some values that will reduce latency.

    if options.fusb_block_size == 0 and options.fusb_nblocks == 0:
```

```
        if realtime:                        # be more aggressive
            options.fusb_block_size = gr.prefs().get_long('fusb', 'rt_block_size', 1024)
            options.fusb_nblocks    = gr.prefs().get_long('fusb', 'rt_nblocks', 16)
        else:
            options.fusb_block_size = gr.prefs().get_long('fusb', 'block_size', 4096)
            options.fusb_nblocks    = gr.prefs().get_long('fusb', 'nblocks', 16)

    #print "fusb_block_size =", options.fusb_block_size
    #print "fusb_nblocks    =", options.fusb_nblocks

    # instantiate the MAC
    mac = cs_mac(tun_fd, verbose=True)


    # build the graph (PHY)
    tb = my_top_block(mods[options.modulation],
                      demods[options.modulation],
                      mac.phy_rx_callback,
                      options)

    mac.set_top_block(tb)    # give the MAC a handle for the PHY

    if tb.txpath.bitrate() != tb.rxpath.bitrate():
        print "WARNING: Transmit bitrate = %sb/sec, Receive bitrate = %sb/sec" % (
            eng_notation.num_to_str(tb.txpath.bitrate()),
            eng_notation.num_to_str(tb.rxpath.bitrate()))

    print "modulation:     %s"   % (options.modulation,)
    print "freq:           %s"      % (eng_notation.num_to_str(options.tx_freq))
    print "bitrate:        %sb/sec" % (eng_notation.num_to_str(tb.txpath.bitrate()),)
    print "samples/symbol: %3d" % (tb.txpath.samples_per_symbol(),)
    #print "interp:         %3d" % (tb.txpath.interp(),)
    #print "decim:          %3d" % (tb.rxpath.decim(),)

    tb.rxpath.set_carrier_threshold(options.carrier_threshold)
    print "Carrier sense threshold:", options.carrier_threshold, "dB"

    print
```

```
    print "Allocated virtual ethernet interface: %s" % (tun_ifname,)
    print "You must now use ifconfig to set its IP address. E.g.,"
    print
    print "  $ sudo ifconfig %s 192.168.200.1" % (tun_ifname,)
    print
    print "Be sure to use a different address in the same subnet for each machine."
    print


    tb.start()    # Start executing the flow graph (runs in separate threads)

    mac.main_loop()    # don't expect this to return...

    tb.stop()     # but if it does, tell flow graph to stop.
    tb.wait()     # wait for it to finish


if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

TRITA-ICT-EX-2009:206

www.kth.se