

Bachelor Thesis

**Software-Defined Radio with GNU Radio  
and USRP/2 Hardware Frontend:  
Setup and FM/GSM Applications**

Matthias Föhnle

Advisor: Prof. Dr.-Ing. Frowin Derr

**Hochschule Ulm**  
**University of Applied Sciences**  
Institute of Communication Technology  
Ulm, Germany

Winter term 2009/2010



Hereby I declare that I have written the enclosed bachelor thesis entirely on my own and have not used other than the referenced sources. Any concepts or quotations applicable to these sources are clearly attributed to them.

This bachelor thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Ulm, 29.01.2010

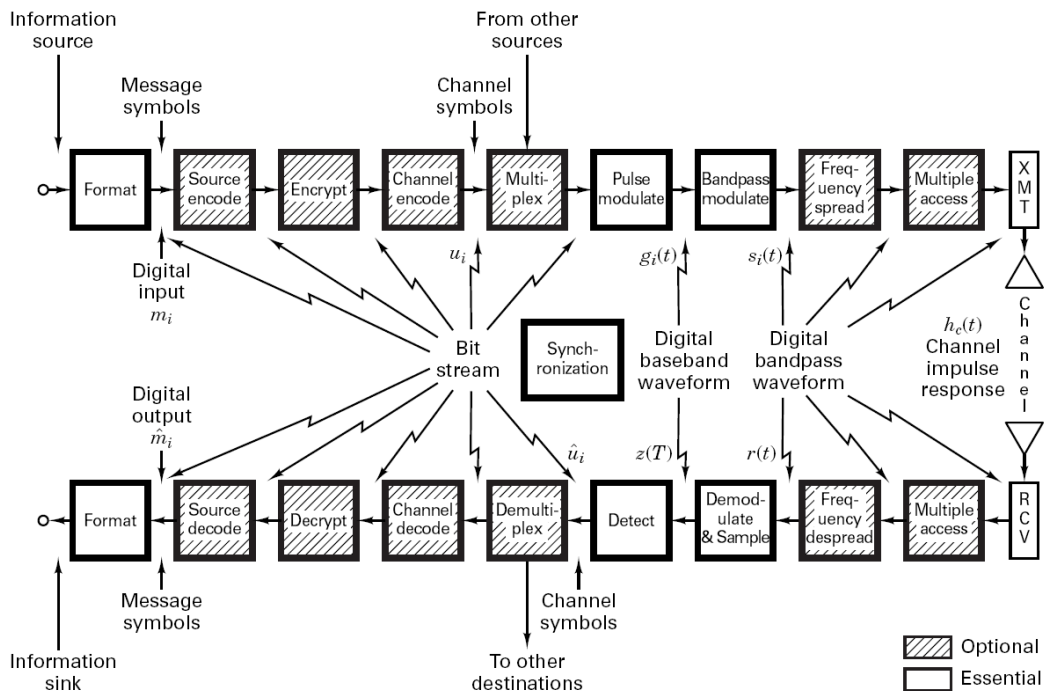
---

Matthias Föhnle



**Abstract**

Recent progress in digital communication system design rests mainly upon software algorithms instead of dedicated hardware. In a typical communication system, as in Figure 1, not only source, line and channel coding, encryption and multiplexing are digitally implemented. Based on FPGA's and DSP's, pulse shaping and modulation is also done by digital signal processing. For instance OFDM, the preferred modulation scheme for the future LTE mobile system, cannot be thought of without digital implementation.



**Figure 1: Block diagram of a typical digital communication system [1]**

All efforts to push digital implementation towards the antenna (wireless) or the cable (wire based) can be summarized in SDR, Software Defined Radio. Together with high-speed A/D- and D/A-converters and a more or less universal RF frontend, various transmission formats (e.g. Bluetooth, WLAN, DECT, ZigBee) may be provided on the same hardware. The only thing to change is the software, which implements the protocol stack and the physical layer functions.



In roughly three chapters, the bachelor thesis covers Software Defined Radio based on the Open Source project GNU Radio and the Universal Software Radio Peripheral (USRP and USRP2). It starts with installation and configuration aspects of GNU Radio and its graphical interface GNU Radio Companion (GRC), offers some background knowledge of how GNU Radio works in conjunction with USRP and addresses an Audio-specific clock synchronization issue. In the following chapter some GNU Radio projects like a FM RDS receiver, a capture utility for the GNU Radio-USRP connection and a simple transmit samples generator are presented.

The last part treats an open GSM system based on OpenBTS and a hardware setup. Installation and operation details of OpenBTS and Asterisk as SIP-server, S-parameter measurements of the RF setup, some solved porting issues (USRP2 in place of USRP) and a preliminary implementation of an automatic SIP registration utility are given. Installation guides and scripts can be found in the annex.

To sum up, all documented projects have been run successfully. GNU Radio together with USRP/2 hardware proved to be a valuable lab platform for implementing complex radio system prototypes in a short time.

Ulm/Germany, February 2010

Matthias Föhnle (matthias@faehnle.org)

Frowin Derr (derr@hs-ulm.de)



## Table of contents

<b>1. Introduction</b> .....	<b>7</b>
1.1. Task and target of this thesis.....	7
1.2. Abbreviations used in conjunction with SDR (GNU Radio, GRC, USRP/2).....	7
<b>2. SDR platform and basics</b> .....	<b>8</b>
2.1. SDR block diagram.....	8
2.2. Universal Software Radio Peripheral (USRP) version 1 and 2 .....	10
2.3. USRP/2 daughterboards .....	12
2.4. SDR environment and system setup .....	13
2.5. GNU Radio .....	14
2.6. GNU Radio Companion (GRC) .....	15
2.7. “Getting started”-problems .....	25
<b>3. Adapted GNU Radio Projects</b> .....	<b>30</b>
3.1. RDS FM receiver .....	30
3.2. Capture samples from USRP2 in C++.....	32
3.3. Wireshark trace .....	37
3.4. Sending Samples.....	39
<b>4. OpenBTS</b> .....	<b>46</b>
4.1. Block diagram – traditional GSM in short .....	46
4.2. Laboratory setup .....	49
4.3. S-parameters of circulator and directional coupler .....	51
4.4. Components of OpenBTS .....	54
4.5. Asterisk.....	56
4.6. Automated SIP user registration .....	58
4.7. SWLOOPBACK.....	60
4.8. Porting issues USRP/2 .....	61
4.9. OpenBTS I & Q data .....	63
<b>5. Conclusion</b> .....	<b>67</b>
<b>6. Annex</b> .....	<b>A</b>
6.1. Annex A: Installation guide for Fedora 9 with GNU Radio 3.2.2 and USRP2.....	A
6.2. Annex B: Automated installation script for GNU Radio 3.2.2, USRP2, ALSA and RDS on Fedora 9.....	H
6.3. Annex C: Install.sh .....	K
6.4. Annex D: RDS receiver for USRP2.....	N
6.5. Annex E: 3-port s-parameter measurement of laboratory setup .....	S
6.6. Annex F: Sending samples source code with byte orders and csv generation .....	T



## List of abbreviations

<b>3GPP</b>	3rd Generation Partnership Project
<b>ADC</b>	Analog-to-digital converter
<b>ALSA</b>	Advanced Linux Sound Architecture
<b>API</b>	Application Programming Interface
<b>ARFCN</b>	Absolute Radio Frequency Channel Number
<b>AuC</b>	Authentication Center
<b>BMC</b>	Biphase mark code
<b>BS</b>	Base Station
<b>BSC</b>	Base Station Controller
<b>BSS</b>	Base Station Subsystem (BTS + BSC)
<b>BTS</b>	Base Transceiver Station
<b>CF</b>	Center frequency
<b>CGRAN</b>	Comprehensive GNU Radio Archive Network
<b>CLI</b>	Command line interface
<b>CSV</b>	Comma-separated values
<b>DAC</b>	Digital-to-analog converter
<b>DC</b>	Direct current
<b>DDC</b>	Digital down converter
<b>Downlink</b>	Signal sent from BS → MS
<b>DUC</b>	Digital up converter
<b>DUT</b>	Device under test
<b>FM</b>	Frequency Modulation
<b>FM</b>	Frequency Modulation
<b>FPGA</b>	Field-programmable gate array
<b>GbitE</b>	Gigabit Ethernet
<b>(G)MSC</b>	(Gateway) Mobile Switching Center
<b>GNU</b>	GNU's Not Unix
<b>GRC</b>	GNU Radio Companion
<b>GSM</b>	Global System for Mobile Communications
<b>GUI</b>	Graphical User Interface
<b>HLR</b>	Home Location Register
<b>I/Q</b>	Inphase & Quadrature
<b>IF</b>	Intermediate Frequency
<b>IMSI</b>	International Mobile Subscriber Identity
<b>IP</b>	Internet Protocol
<b>LO</b>	Local Oscillator
<b>LTE</b>	Long Term Evolution
<b>MS</b>	Mobile station
<b>NCO</b>	Numerically-controlled oscillator
<b>PBX</b>	Private branch exchange
<b>PCB</b>	Printed Circuit Board
<b>PPM</b>	Parts per million
<b>PLL</b>	Phase-locked loop



<b>PSTN</b>	Public switched telephone network
<b>RDS</b>	Radio Data System
<b>RF</b>	Radio Frequency
<b>Rx</b>	Receive path
<b>S/PDIF</b>	Sony/Philips Digital Interconnect Format
<b>SDR</b>	Software-Defined Radio
<b>SIM</b>	Subscriber Identity Module
<b>SIP</b>	Session initiation protocol
<b>SMA</b>	SubMiniature version A (coaxial RF connector)
<b>STDOUT</b>	Standard output (where a application writes ist output data to)
<b>SWIG</b>	Simplified Wrapper and Interface Generator
<b>TCP</b>	Transmission Control Protocol
<b>TDMA</b>	Time Division Multiple Access
<b>TMSI</b>	Temporary Mobile Subscriber Identity
<b>TTL</b>	Transistor-transistor logic
<b>Tx</b>	Transmit path
<b>UDP</b>	User Datagram Protocol
<b>UE</b>	User Equipment
<b>Um</b>	GSM air interface
<b>Uplink</b>	Signal sent from MS → BS
<b>USB</b>	Universal Serial Bus
<b>USRP</b>	Universal Software Radio Peripheral
<b>VCC</b>	Common-collector voltage (IC power supply pin)
<b>VLR</b>	Visitor Location Register
<b>VoIP</b>	Voice over IP
<b>XML</b>	Extensible Markup Language

## Used paths

**\$GNURADIOROOT** = /gnuradio-3.2.2  
**\$PYTHONPATH** = /usr/local/lib/python2.5/site-packages  
**\$OPENBTSROOT** = /openbts-2.5Lacassine



## 1. Introduction

### 1.1. Task and target of this thesis

This work provides an insight into the wide area of Software-Defined Radio. Starting with installation of a Linux system including an automated installer for GNU Radio framework on Fedora 9, an introduction to SDR techniques and the used hardware frontends are given. Furthermore, a software GSM BTS is set up using the OpenBTS project. Covering basics in GSM and S-parameter measurements, a laboratory setup for GSM communication is built. For a more detailed insight into the structure of GNU Radio and OpenBTS, code extensions in C++ as well as some GRC project adaptations are additionally provided.

### 1.2. Abbreviations used in conjunction with SDR (GNU Radio, GRC, USRP/2)

Since this document deals with many abbreviations, a one-sentence-introduction of some fundamental terms might be useful at the beginning. Later on, all of them are explained in detail. However, some fundamentals in communications engineering and computer science will be necessary for comprehension of this work.

Abbreviation	In chapter	Short description
<b>SDR</b>	2.1	Software-Defined Radio describes the technique of using a universal hardware frontend to receive and transmit RF signals with waveforms defined in software applications
<b>USRP/2</b>	2.2, 2.3	USRP and USRP2 Universal Software Radio Peripheral are the two used hardware frontends, which build the interface between an host PC and the RF domain by mixing the transmitted and received signals to an software definable IF
<b>GNU Radio</b>	2.5	GNU Radio is a an open source software development framework, that builds the interface to the RF hardware USRP/2
<b>GRC</b>	2.6	GNU Radio Companion is an application distributed with GNU Radio, in which software radios can be built within a flow graph using numerous predefined blocks for signal sources, sinks and modulations
<b>OpenBTS</b>	4	Open source project which implements a GSM base station using USRP





## 2. SDR platform and basics

Starting with a significant citation from GNU Radio founder Eric Blossom, Software-Defined Radio can be defined as:

„Software radio is the technique of getting code as close to the antenna as possible. It turns radio hardware problems into software problems.“ [2]

In principle, a universal applicable hardware serves as interface between the baseband and the RF. The waveform of a transmitted signal is fully generated through software, as well as a received signal is fully processed and demodulated within software algorithms. In SDR, the processing power required for signal processing is sourced out to a universal host (PC).

An important benefit for industrial usage is the possibility to change complete processing stacks or modulation schemes just with a software update. This also saves costs and time for new hardware developments.

### 2.1. SDR block diagram

A universal SDR structure with the specific software (GNU Radio) and hardware (USRP/2) is given in Figure 2.

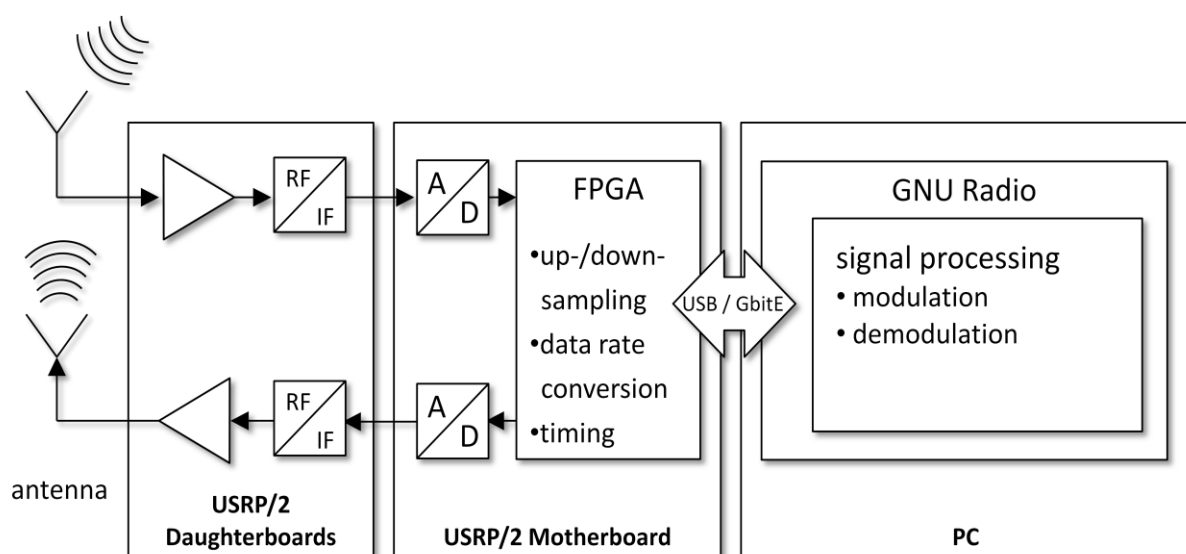


Figure 2: Software-Defined Radio block diagram



In Figure 2, the Software-Defined Radio (SDR) structure is divided into three blocks. The left one builds the RF frontend of the hardware which serves as interface to the analog RF domain. In the second block, the intelligence of the hardware part is implemented, forming the interface between the digital and the analog world. In the third block, the whole signal processing is done - fully designed in software.

Getting more detailed, the interface to the analog world is given as mentioned on left side of Figure 2. An analog RF signal can be received or transmitted over antennas, or can also be directly connected via SMA connectors to the SMA ports of RF frontend called daughterboards. The upper path (arrow towards the daughterboard) marks the receive path (Rx), the lower path describes the transmit path (Tx). Both paths can operate autonomously. The possible operation frequency range is very modular (from DC to 5.9 GHz), depending on the available daughterboards for USRP/2. Daughterboards form the RF frontend of USRP/2 and are connected to the USRP/2 motherboard. On USRP/2 motherboard, the analog signals are converted to digital samples and mixed down to baseband within the FPGA. Also a decimation of the sample rate is performed, see chapter 2.2.

Regarding Figure 2, data sampled by the FPGA are sent to the host by USB or Gigabit Ethernet respectively what is used – USRP or URSP2. Connected to the host computer (right block in Figure 2), the GNU Radio framework controls the further signal processing capabilities. GNU Radio is an open source framework, providing various pre-assembled signal processing blocks for waveform creation and analysis in software radio development. Further information about signal processing and the GNU Radio environment are elaborated in chapter 2.5.

In the GNU Radio environment, Python and C++ are used as main programming languages as well as a signal flow application called GNU Radio Companion (GRC).

Very detailed documentation about USRP and daughterboards with information gathered from mailing lists can be found at:

[http://www.gnuradio.org/redmine/attachments/129/USRP\\_Documentation.pdf](http://www.gnuradio.org/redmine/attachments/129/USRP_Documentation.pdf)



## 2.2. Universal Software Radio Peripheral (USRP) version 1 and 2



Figure 3: USRP



Figure 4: USRP2

As explained in chapter 2.1, the USRP serves as interface between digital (host) and analog (RF) domain. In May 2009, the well-proven Universal Software Radio Peripheral (USRP) product became extended by an enhanced product named USRP2. USRP2 uses a different FPGA, faster ADCs and DACs with a higher dynamic range and a Gbit-Ethernet connection. All USRP daughterboards can be used furthermore.

	USRP	USRP2
<b>Manufacturer</b>	Ettus Research	
<b>ADCs</b>	64 MS/s 12-bit	100 MS/s 14-bit
<b>DACs</b>	128 MS/s 14-bit	400 MS/s 16-bit
<b>Mixer</b>	programmable decimation- and interpolation factors	
<b>Max. BW</b>	16 MHz	50 MHz
<b>PC connection</b>	USB 2.0 (32 MB/s half duplex)	Gigabit Ethernet (1000 MBit/s)
<b>RF range</b>	DC – 5.9 GHz, defined trough RF daughterboards	

Table 1: USRP versus USRP2

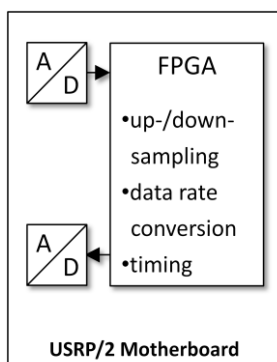


Figure 5: USRP/2 SDR Block

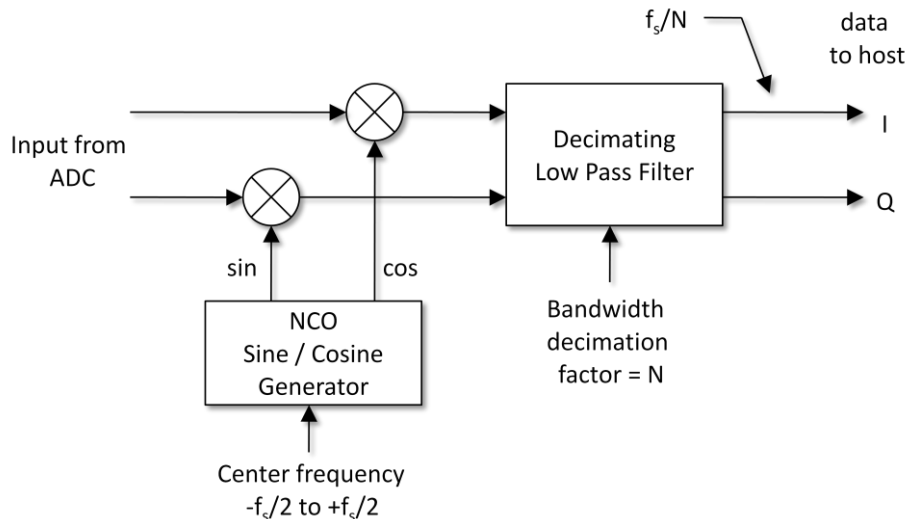
In USRP2 motherboard, an analog to digital converter (ADC) samples the received signal and converts it to digital values depending on the ADCs dynamic range of 14 bit.

How many times per second the analog signal can be measured is defined by the sampling rate of used ADCs - yielding in 100e6 results per second at a sampling rate of 100 mega samples per second (MS/s) for USRP2.

The digital sample values are transferred to the FPGA and



processed with digital down converters (DDC) to meet exactly the requested output frequency and sample rate. Below, the schematic of a DDC is shown.



**Figure 6: Digital Down Converter Block Diagram (according to [2])**

The digitized samples from ADC are mixed down to the desired IF by being multiplied with a sine respectively cosine function resulting in the I and Q path. The frequency is generated with a numerically-controlled oscillator (NCO) which synthesizes a discrete-time, discrete-amplitude waveform within the FPGA. Via the used NCO, very rapid frequency hopping is feasible.

Afterwards a decimation of the sampling rate is performed by an arbitrary decimation factor  $N$ . The sampling rate ( $f_s$ ) divided by  $N$  results in the output sample rate, sent to host. In transmit path, the same procedure is done vice versa using digital up converters (DUC) and digital analog converters (DAC).

The FPGA also supports time dependent applications which e.g. use TDMA. A free running internal counter allows incoming samples to be sent in strictly definable timestamps.

Towards host side, USRP2 uses the Gbit-Ethernet connection, allowing a significant higher throughput than USRP with USB 2.0 achieves. The USB connection sustains 32 MB/s in half duplex, so transmission and reception of samples isn't possible synchronously. Using 4 byte complex samples (16-bit I and 16-bit Q) and respecting the Nyquist criterion leads to a usable (complex) spectral bandwidth of 8 MHz. For Gbit-Ethernet in USRP2, the theoretically



data rate of 125 MB/s allows for a theoretical (complex) RF bandwidth of about 31,25 MHz. A value of 25 MHz usable bandwidth, as given in [3], may serve as a realistic limit.

### 2.3. USRP/2 daughterboards

On most daughterboards, the signal is already filtered, amplified and tuned to a baseband frequency dependent on the boards IF bandwidth and local oscillator frequency. There are also so called Basic Rx/Tx boards with no frequency conversion or filtering. They only provide a direct RF connection to the motherboard.

Identifier	Frequency range	Area of application
<b>Transceiver</b>		
RFX900	750 to 1050 MHz	GSM (Low Band)
RFX1200	1150 to 1450 MHz	GPS
RFX1800	1,5 to 2,1 GHz	DECT, GSM (High Band)
RFX2400	2,3 to 2,9 GHz	WLAN, Bluetooth
XCVR 2450	2,4 - 2,5 and 4,9 - 5,9 GHz	WLAN
<b>Transmitter, Receiver</b>		
Basic TX, Basic RX	1 to 250 MHz	Misc baseband operations
TVRX Receiver	50 to 860 MHz	VHF, DAB

Table 2: Frequency range of several USRP/2 daughterboards

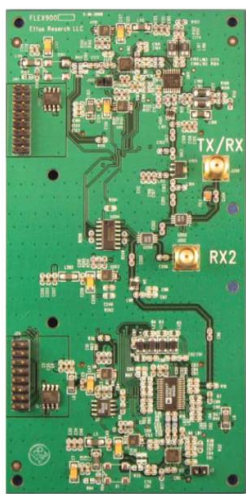


Figure 7: RFX900



Figure 8: TVRX



## 2.4. SDR environment and system setup

During the task, the laboratory setup was built using the following equipment.

<b>Host characteristics</b>	
Processor	Intel Pentium 2.8 GHz (Dual Core), 512 MB RAM
OS	Fedora 9
GNU Radio	3.2.2
OpenBTS	2.5Lacassine
<b>SDR equipment</b>	
SDR RF hardware	USRP, USRP2
USRP Daughterboards	TVRX, 2xRFX900, 2xRFX1800, Basic Rx/Tx, XCVR2450
<b>Signal generator and measurement equipment</b>	
Network analyzers	Rohde & Schwarz ZVA 8 Vector Network Analyzer (300 kHz – 8 GHz)
	Rohde & Schwarz ZVM Vector Network Analyzer (10 MHz - 20 GHz)
Oscilloscope	Tektronix TDS460A (400MHz, 100MS/s)
Signal generators	Marconi Instruments signal generator 2022C (10kHz - 1GHz)
	TTi TG330 Function generator
Spectrum analyzer	Rohde & Schwarz FSL (9kHz - 6GHz)
<b>Mobile equipment</b>	
DC Power Supply	Agilent E3631A
GSM mobile	Nokia 5330 XpressMusic
<b>RF equipment</b>	
Attenuators	Huber & Suhner 20 dB attenuators
	Rohde & Schwarz RF step attenuator 139dB
Circulator	Renaissance 3A5BC
Directional coupler	Hewlett Packard 776D (940 – 1900 MHz)
Screened case	MTS MSB-0011-USB (418x268x116)



To allow an easy host system setup, an installation guide for GNU Radio 3.2.2 on Fedora 9 is provided in Annex A of this document. After having installed Fedora 9, an install script may be run, which unpacks the provided files and afterwards installs automatically what's needed for an easy entry to a SDR platform. Further configuration statements can be inserted in *install.sh*.

In Annex C the full script is attached. Annex B provides additional comments for all statements to give a better understanding what's done. Note: It is not necessary to understand bash script programming like it is done here for use of GNU Radio. The comments will give just a short excursion to bash-scripting. Being anyway interested in bash-programming, the "Advanced Bash-Scripting Guide" on <http://tldp.org/LDP/abs/html> or similar sources may be useful.

## 2.5. GNU Radio

The GNU Radio project was founded by Eric Blossom with the intention of creating an open source framework for software radios. Providing the application programming interface (API) for USRP/2, GNU Radio represents the centre of SDR development with USRP/2 frontend. In GNU Radio, various pre-assembled signal processing blocks for waveform creation and analysis are implemented.

For 3<sup>rd</sup> party open source projects based on GNU Radio, George Nychis founded the Comprehensive GNU Radio Architecture Network (CGRAN). The RDS receiver of chapter 3.1 can be found there.

GNU Radio runs under several operating systems like Linux, Mac OS X, NetBSD. Also a Cygwin porting for Windows exists, but due to the limited hardware control, the full functionality is not guaranteed. Python and C++ are used as main programming languages in GNU Radio as well as the below introduced GNU Radio Companion (GRC).

Since the GNU Radio framework is the central point of data streams sent towards and received from USRP/2, its structure will be illustrated step by step during the whole thesis.



## 2.6. GNU Radio Companion (GRC)

Since it is easier to handle information flow graphically, GNU Radio offers with its application GNU Radio Companion the possibility to form a flow chart with graphical block elements. This application provides numerous predefined blocks, organized in different groups like signal sources, signal sinks as well as modulation and demodulation functions.

As signal source for instance, USRP/2, audio card, wav files, signal generators or UDP/TCP ports may be used. Being installed with GNU Radio, GRC can be run from Linux by simply typing “*grc*” in an xterm shell.

GRC is best illustrated by the typical FM radio receiver example.

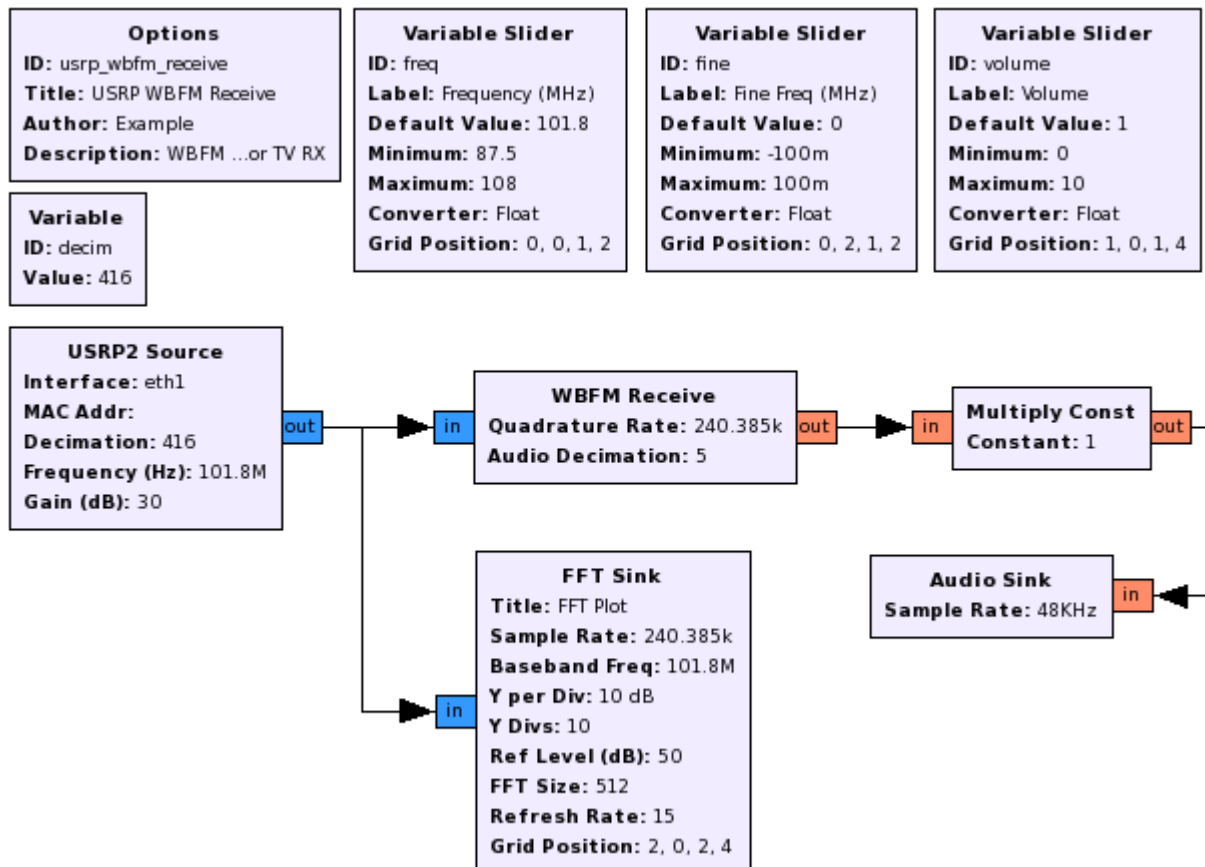


Figure 9: GRC FM receiver example usrp\_wbfm\_receive.grc adapted for USRP2





Figure 9 shows an adapted GRC example for a FM receiver with audio sink, based on `$GNURADIOROOT/grc/examples/usrp_wbfm_receive.grc`.

Since this example originally uses USRP as data source, the source block has to be replaced by “USRP2 Source” block from category USRP (drag and drop). Blocks are connected to each other by clicking first to an out connector, followed by a click on the in connector. Double clicking opens the property window of a block.

Basic understanding in working with graphical signal flow graphs may be extended by having a look at

<http://www.gnuradio.org/trac/wiki/GNURadioCompanion>.

### 2.6.1. GRC USRP2 Source block

First of all, the USRP2 Source block needs to be configured according to project dependent requirements. Since the provided `usrp_wbfm_receive.grc` example as shown in Figure 9 ought to be used, the complex output type from USRP2 Source block needs to connect to the WBFM Receive block.

Below, a set of provided USRP2 Source block parameters according Figure 10 is described. All red marked parameters are mandatory and/or not identified as valid until they are colored black!

- Output Type

This parameter provides a selection of output data type according to the selected block. Possible choices here are Short or Complex.

The most common format depending on the common USRP2 configuration is 16-bit I & 16-bit Q with I in top of a 32-bit word. Detailed information about USRP data structure is provided in chapter 3.4.

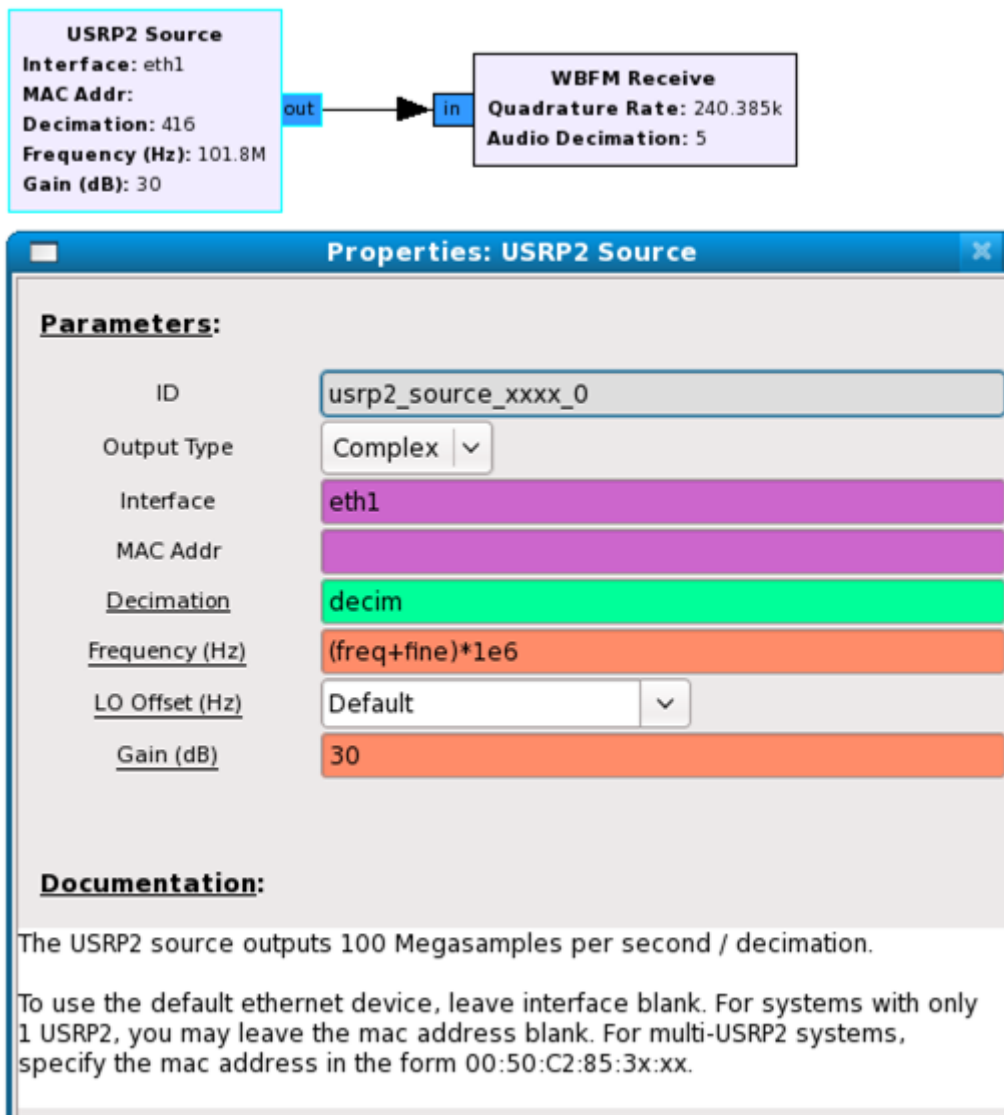


Figure 10: USRP2 block properties

- Interface  
Linux notation of the interface where USRP2 is connected to.
- MAC Addr  
MAC Address of the USRP2 Ethernet interface. This parameter is only mandatory if several USRP2 devices are used over one PC Ethernet interface card.
- Decimation  
This parameter describes the internal decimation factor of the USRP2. The ADC sample rate of USRP2 is at 100 MS/s. The ADC chips' rate of 100e6 divided by this decimation number results in the output sample rate sent to the PC. The minimum



value for decimation is 4, maximum is 512. Further information about decimation rate calculations is provided in chapter 3.2.3.

- Frequency (Hz)  
Sets the USRP2 receiver frequency, e.g. 101.8 MHz for FM reception.
- LO Offset (Hz)  
Set receive LO offset frequency of the used daughterboard. According to [4] this value can be ignored (for further study).
- Gain (dB)  
This parameter sets the receiver gain. To receive a FM radio signal, a gain of at least 30 dB has to be set.

The properties of a GRC block are told “parameters”. These can be set statically by writing a fixed number in there or with variable values, which may be used for some dependent calculations (like sample rates) or which can be changed while the project is running. A variable, as known in every programming language, can be defined by the appropriate GRC block called “Variable”.

One possibility to change the variables’ value while the project is running is to use sliders. In GRC slider block, the default value, a minimum and a maximum value are set. The default value defines the value where the slider resides while the project is started, the minimum and maximum values define the range of possible values.

Parameters can be expressed by variables, see Figure 9. For instance the variable “decim” is used as decimation factor. GRC recognizes, if a variable is set for a parameter and has an appropriate data type. If not, GRC marks this value red and the GRC project can’t be run.

The benefit by gathering a variable here is that one variables’ value has to be changed and other variables’ contents are calculated autonomously. In the GRC WBFM Receive example, the parameter “Quadrature Rate” from WBFM Receive block is calculated by  $100e6$  divided by “decim” variable – so we get a value of 240.385kS/s.



There are also 3 variable sliders, with which customizations of volume and frequency values can be done in the GUI during the running GRC project.

Further information of USRP2 in conjunction with GNU radio can be found at <http://gnuradio.org/trac/wiki/USRP2GenFAQ> and <http://gnuradio.org/trac/wiki/USRP2UserFAQ>.

### 2.6.2. Explanation of blocks

At first, the “WBFM Receive” block from category Modulators is considered exemplarily. As shown in Figure 11, this block has a blue input and orange output connector. In GRC, the connector color specifies the data type.

Blue defines a connector with complex data type,

- orange the data type float,
- yellow stands for data type short,
- and magenta for characters.

Connectors of different colors need to be connected via a type conversion block found in “Type Conversions”-category.

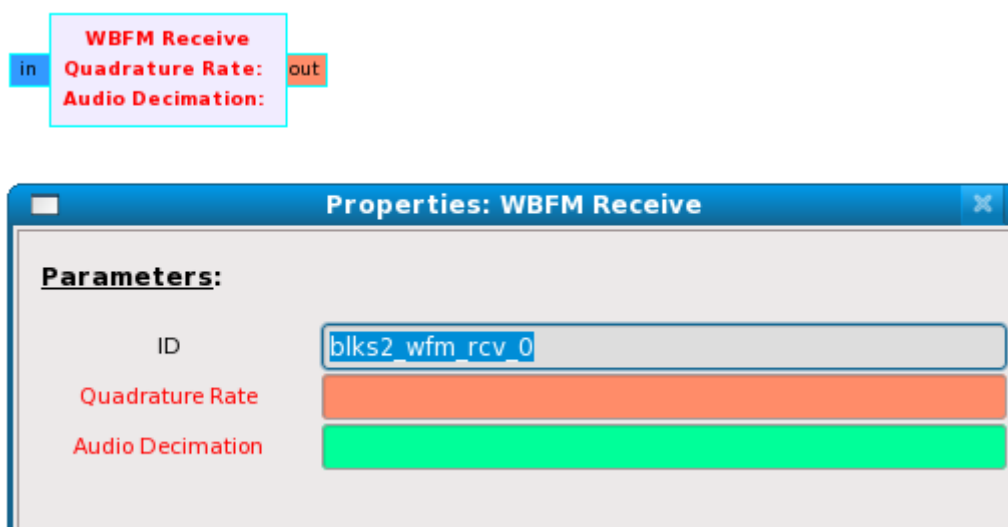


Figure 11: GRC block “WBFM Receive” with opened properties window



After a double click to the “WBFM Receive” block, the ID and two more parameters are shown. ID identifies the name to the according xml-definition for this block. This xml block name (here `blks2_wfm_rcv`) is followed by an underscore and an auto incremented number making it possible to use several blocks of the same type in one GRC project.

<code>blks2_</code>	<code>wfm_rcv</code>	<code>_0</code>
python package	name of python function	block id in current GRC project

If there are several blocks with identical names and different connector types, an additional underscore with the provided type conversion is shown in the ID (e.g. FM Demod block). The abbreviation `_cf` for instance stands for complex input and float output stream of a block.

In Figure 9, some more blocks were used which aren't

- FFT sink

The FFT sink block creates a python window in which the Fourier spectrum of the connected signal is shown.

- Multiply Const

This block serves as a variable audio volume control. During runtime, the current value of the volume slider variable is multiplied with the audio stream, resulting in an adjustable audio amplitude.

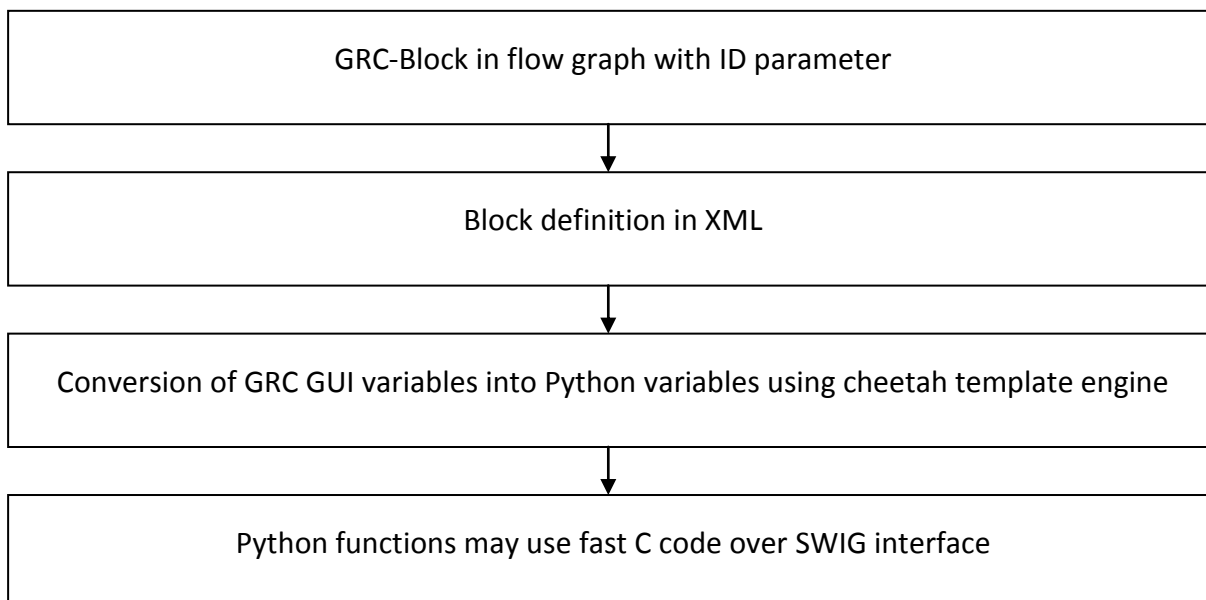
- Audio Sink

The audio sink block builds the software interface to the PC audio card. By connecting a GRC stream to this block, the signal will become audible through audio speakers connected to the sound card.



### 2.6.3. Programming layers of GRC

In the following graph, the processing layers below a GRC block are described.



### 2.6.4. XML block definition

Definitions of GRC standard xml blocks can be found in `"/usr/local/share/gnuradio/grc/blocks/"`, depending on the installation path prefixes. As mentioned above, the block ID refers to the file xml file, here `"blks2_wfm_rcv.xml"`.

### 2.6.5. Abstract of xml source code

FILE: `/usr/local/share/gnuradio/grc/blocks/blks2_wfm_rcv.xml`

```

<?xml version="1.0"?>
<!--
#####
##WBFM Receive
#####
-->
<block>
  <name>WBFM Receive</name>
  <key>blks2_wfm_rcv</key>
  <import>from gnuradio import blks2</import>
  <make>blks2.wfm_rcv(
    quad_rate=$quad_rate,
    audio_decimation=$audio_decimation,
  )</make>
  <param>
    <name>Quadrature Rate</name>
    <key>quad_rate</key>
  </param>
</block>
  
```



```

        <type>real</type>
    </param>
    <param>
        <name>Audio Decimation</name>
        <key>audio_decimation</key>
        <type>int</type>
    </param>
    <sink>
        <name>in</name>
        <type>complex</type>
    </sink>
    <source>
        <name>out</name>
        <type>float</type>
    </source>
</block>

```

By considering this code, the xml structure depending on tags becomes clear.

- **<import>**

```
<import>from gnuradio import blks2</import>
```

Package import syntax according to Python conformation. This tag imports the blks2 module from `$PYTHONPATH/gnuradio` folder (see tag descriptions below).

- **<make>**

```
<make>blks2.wfm_rcv(
    quad_rate=$quad_rate,
    audio_decimation=$audio_decimation,
)</make>
```

The make-tags are handled as cheetah-templates (see [www.cheetahtemplate.org](http://www.cheetahtemplate.org) for further information). The templates replace the GRC variables starting with \$ from GRC GUI before transferring them to the called function (here blks2.wfm\_rcv from gnuradio.blks2 package).

- **<param>**

```
<param>
    <name>Audio Decimation</name>
    <key>audio_decimation</key>
    <type>int</type>
</param>
```

The param-tags define usable variables for this block in GRC. With the key-tag, the GRC-variable name is given (audio\_decimation), that is replaced by the template-machine with the value stored in \$audio\_decimation. The declaration is done in the make-tag by audio\_decimation=\$audio\_decimation.



- `<sink>`  
Defines input data type in type-tag
- `<source>`  
Defines output data type in type-tag

### 2.6.6. Python code

To get the information what is really done after putting the GRC variables through the cheetah template, it's needed to follow the import-tag. Since a function is called in the make-tag (`blks2.wfm_rcv`), this function is included from the imported package `gnuradio.blks2`. Hence we have to search for the function in `$PYTHONPATH/gnuradio/blks2/`.

Included in this folder there are just three `__init__.py*` files, that define this folder as Python module. The file `__init__.py` consists of a for-loop that according to the explanation parses the `blks2impl`-directory to import all contained modules. For fundamentals about Python packages and modules have a look at

<http://docs.python.org/tutorial/modules.html#packages>.

Section of FILE: `$PYTHONPATH/gnuradio/blks2/__init__.py`

```
# Semi-hideous kludge to import everything in the blks2impl directory
# into the gnuradio.blks2 namespace. This keeps us from having to remember
# to manually update this file.
```

“Redirected“ by the `blks2/__init__.py`, the function is found in `../blks2impl` directory: here

`$PYTHONPATH/gnuradio/blks2impl/wfm_rcv.py`.

Further information about how the FM demodulation is done is provided under:

<http://radioware.nd.edu/documentation/basic-gnuradio/exploring-the-fm-receiver>





For users, which aren't experienced in GNU Radio and/or Python, having a look at the following resources will be helpful.

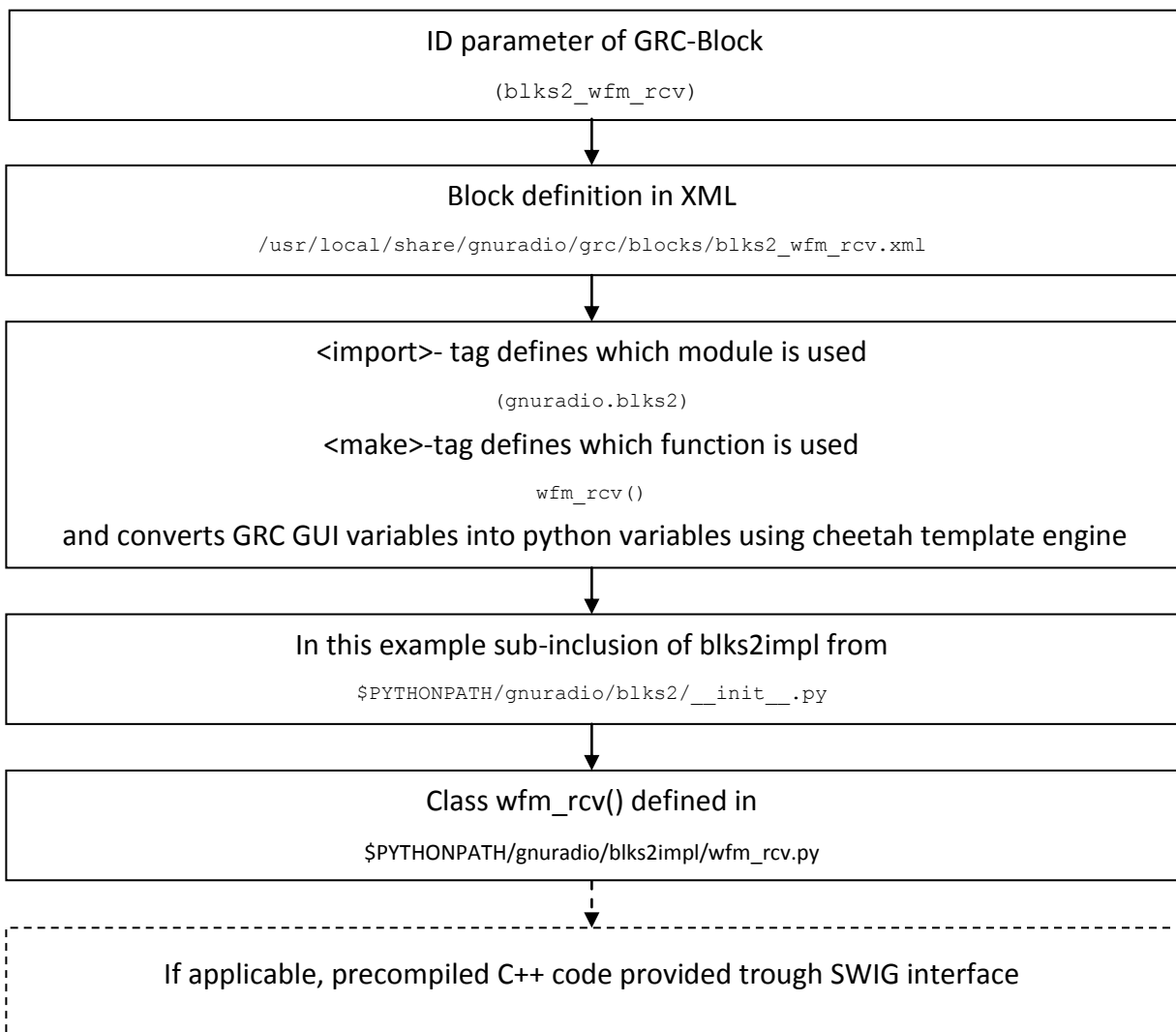
- <http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>
- <http://gnuradio.org/trac/wiki/Tutorials/WritePythonApplications>

These resources give a good basic knowledge about Python in addition with GNU Radio and also about FM basics.

### 2.6.7. SWIG interface

Likewise there are more complex blocks such as the USRP/2 Source and Sink blocks. In these blocks a SWIG interface is used for inclusion of compiled C++ code, which runs essentially faster than the Python interpreter. More information about SWIG is provided under:

<http://www.swig.org/exec.html>





## 2.7. "Getting started"-problems

A first introduction to GNU Radio and USRP2 often leads to

```
$GNURADIOROOT/gnuradio-examples/python/audio/dial_tone.py
```

to have a look at the provided code.

Later on, often mentioned examples like the GRC project `usrp2_fft.py` in

```
$GNURADIOROOT/grc/examples/usrp/usrp2_fft.grc
```

or

```
$GNURADIOROOT/gnuradio-examples/python/usrp2/usrp2_wfm_rcv.py
```

might be interesting to demodulate received FM signals.

If a FM modulated signal is received with USRP2, the demodulated signal (like FM radio) may be sent to an audio sink like a sound card. Implementing this chain will definitively result in resampling problems, since an audio card has its own clock - running independent from USRP2. Even if a sample rate supported by the used audio card is found, whose multiple of meets exactly the ADC rate  $100\text{e}6$ , the two clocks will run apart of each other due to clock instabilities in ppm range.

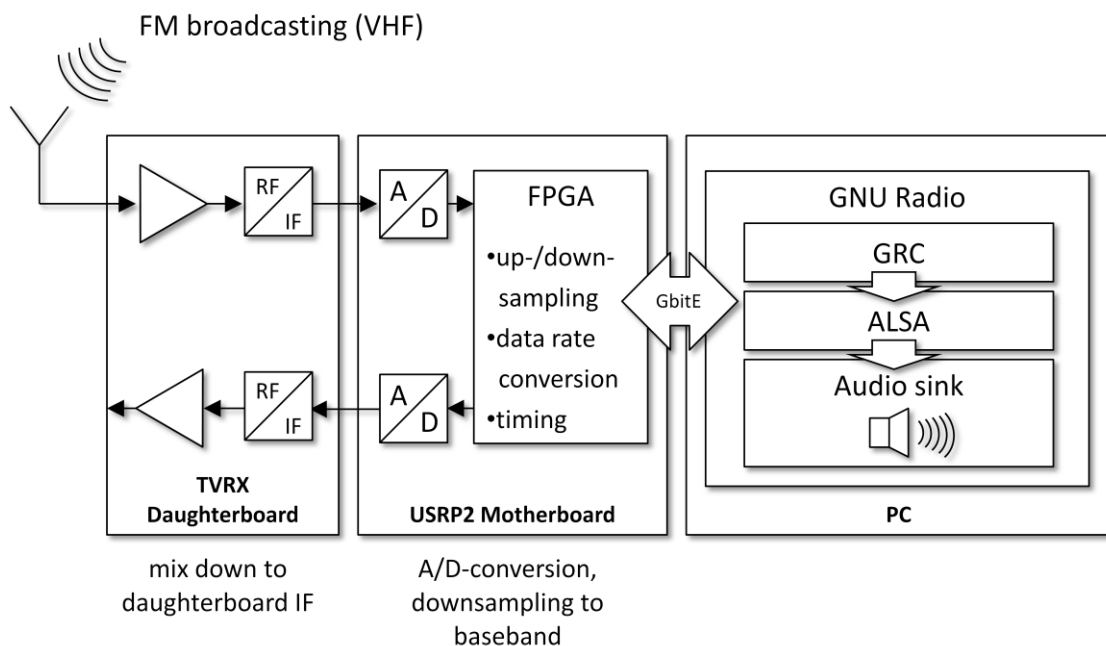


Figure 12: Signal path from FM VHF to audio sink



### 2.7.1. Audio sink problems

In Figure 9, the USRP2 decimation rate is set to 416 and the audio decimation to 5. This results in a sample rate of 48.077kS/s.

$$\frac{100e6 \text{ S/s}}{416 * 5} \approx 48\,077 \text{ S/s}$$

Sending a data stream with this sampling rate to an audio sink operating at 48000 S/s, 77 Samples each second will be buffered – resulting in a non-audible deeper audio frequency. By running about half an hour, a buffer with a size of 138 600 samples (equivalent to 541 kbyte) is filled, since not all provided data can be handled by the audio sink at 48k.

$$77 \text{ S/s} * 60s * 30 = 138600 \text{ S}/30 \text{ min}$$

Unplugging the Ethernet cable between PC and USRP2 after this time (while the audio capture is still running) will result in a continuous play of the sound which was streamed into USRP2. Since running on a sample rate of 48k, the sound card needs nearly 3 seconds to clear the filled buffer and keeps on playing in this time as the USRP2 would still be connected.

$$\frac{138.6e3 \text{ S}}{48e3 \text{ S/s}} \approx 2.89 \text{ s}$$

Depending on the used audio driver and hardware, someday a buffer overflow or underflow occurs – printing out one of the following error codes:

- “u” = USRP
- “a” = audio (sound card)
- “O” = overrun (PC not keeping up with received data from usrp or audio card)
- “U” = underrun (PC not providing data quickly enough)
- “aUaU” = audio underrun (not enough samples ready to send to sound USRP sink)
- “S” = indicates a sequence number error in Ethernet packets marking an overrun from USRP to PC like “O” [3][5][6]



The most common audio architecture used in Linux is “Advanced Linux Sound Architecture” (ALSA). It includes a software resampler device called plughw, which provides a sample rate adaptation in a given range, see

<http://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html> and  
[http://www.alsa-project.org/alsa-doc/alsa-lib/pcm\\_plugins.html](http://www.alsa-project.org/alsa-doc/alsa-lib/pcm_plugins.html)

Using plughw:0,0 as audio output device seems to adjust USRP2’s sample rate better to the audio sink. Nevertheless after about 3 hours a reproducible overflow occurred.

### 2.7.2. ALSA supported audio interfaces in GNU Radio

In GNU Radio, the used audio module can be specified in the system wide config file `/usr/local/etc/gnuradio/conf.d/gnuradio-core.conf`. In section `[audio]` the `audio_module = auto` is set by default, that makes the system selecting one. Some more possibilities are listed in this file such as `audio_alsa` and `audio_oss`.

As shown in `$PYTHONPATH/gnuradio/audio.py`, all possible modules are defined in an array. If “auto” is defined, the first fitting module (typically `audio_alsa`) will be used. Using ALSA, we need to have a look at `/usr/local/etc/gnuradio/conf.d/gr-audio-alsa.conf` for our system-wide GNU Radio ALSA configuration. In this config file, among other things the default input and output devices can be set and the verbose mode can be enabled. If the verbose mode is enabled, detailed information about the audio cards’ supported formats, sampling rates, number of channels and sample resolution is printed out each time a GNU Radio application is run that connects to the ALSA driver.

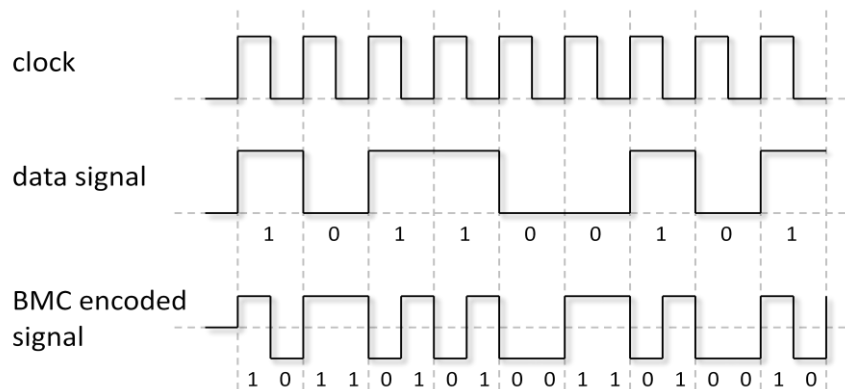
### 2.7.3. Audio synchronization approach

In digital audio processing, the clocking issues are crucial. Since digital audio recording and playback devices use their own oscillators (differing a little from each other), synchronization needs to be done. One approach is called Wordclock, used in several well known audio formats like S/PDIF and AES/EBU. It is a master clock signal to synchronize slave devices – independently of their own clocks.



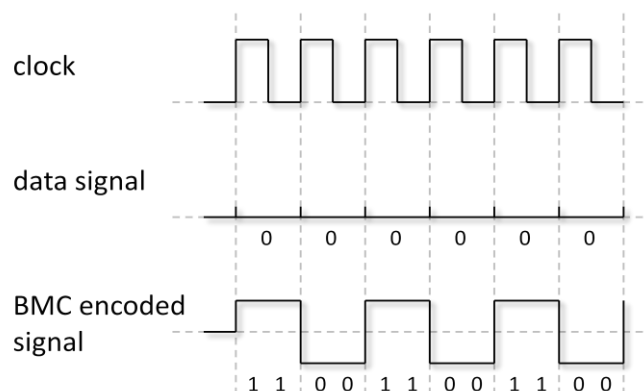
In S/PDIF, a 44,1 kHz Wordclock signal (simple rectangular signal) may be used as master clock. The appropriate coding format in S/PDIF is biphase-mark-code (BMC). By encoding a binary data stream with BMC, the encoded signal changes polarity for every data bit, according to every new clock period. Additionally, a polarity change can occur in the middle of the data bit, depending on the current data signal. A data bit logical “1” results in change of polarity in the middle of bit time, a “0” holds the polarity of the encoded signal constant.

[7][8][9]



**Figure 13: Biphase mark code**

The frequency of the clock is twice the bit rate, if all data bits are set to 0, called “digital black” (see Figure 14). This fact is used in purchasable Wordclock generators for clock synchronizations.



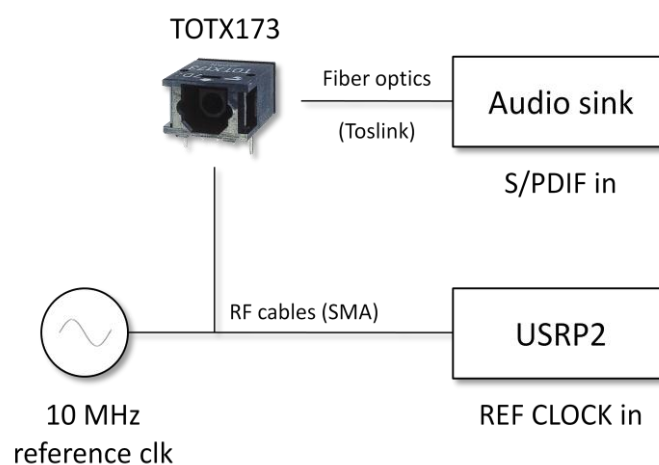
**Figure 14: Biphase mark code with all data bits set to zero**

This synchronization feature of S/PDIF might also be used to synchronize an audio card by sending the USRP/2 clock signals there, similarly like word clock generators do. No data



signal is included in the stream (only “0”-bits), just a clock reference is built during changed polarity.

One possibility for audio synchronization might be to lock USRP2 initially to an external reference signal (10 MHz), sent to the REF CLOCK SMA connector at the front side of USRP2. An optical Toslink sender chip like Toshiba TOTX173 might be used to convert TTL conform signals from the given reference clock to an optical fiber, which is connected to S/PDIF input port of the audio sink.



**Figure 15: Block diagram for possible clock synchronization**

In this assembly, the RF input needs to be adjusted to TTL levels as requested as input signal range of TOTX173. Also the TOTX173 needs to be supplied with VCC and be soldered onto PCB. Further information provided in datasheet:

<http://www.toshiba.com/taec/components/Datasheet/TOTX173.pdf>

Another possibility might be to extract the USRP2 onboard clock. Enabling the test clock pins on the motherboard by putting the line “clocks\_enable\_test\_clk(true,1)” in USRP2 firmware results in an output test clock on the two middle pins of the 4 pin connector J503. Pins 1 and 4 are ground connectors. For further information have a look at

<http://www.mail-archive.com/discuss-gnuradio@gnu.org/msg19306.html>

Since the audio sync problem was not crucial for the projects of this work, further investigation has been moved to another work topic. This chapter summarized the considerations done so far without any implementation aspects.



### 3. Adapted GNU Radio Projects

Since USRP2 release is quite new, most available projects and documentations are provided for USRP only. The following chapter will describe how USRP2 data flow is handled and how USRP projects may be adapted to be used with USRP2.

#### 3.1. RDS FM receiver

Radio Data System (RDS) is a protocol to transmit additional information over FM radio broadcast. There is a project for GNU Radio provided by <http://www.cgran.org> contributed by Dimitrios Symeonidis. By installing the source code, a new block called gr-rds will be built which demodulates a received RDS signal embedded in a FM radio signal. The source code is available on

<https://www.cgran.org/wiki/RDS>

and also contained in the install.tar archive mentioned in Annex B. If being interested in, explicit information about the RDS standard may be found in

[http://www.rds.org.uk/rds98/rds\\_standards.htm](http://www.rds.org.uk/rds98/rds_standards.htm)

For adapting the already functional RDS project, no direct knowledge of RDS protocol is necessary. Only a exchange of the USRP/2 interface has to be done.

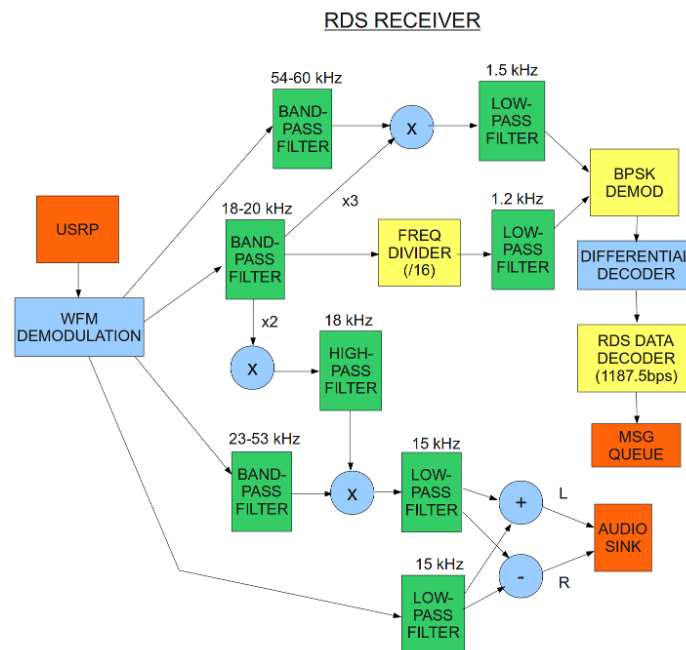


Figure 16: Flow graph of RDS Receiver block gr-rds [10]



Since this project supports only USRP so far, some extensions need to be done in the file `$GNURADIOROOT/gr-rds/src/python/usrp_rds_rx.py` (if installed as described in Annex B). Copying this file to a new file, e.g. `usrp2_rds_rx.py`, preserves the original file from being overwritten. The TVRX daughterboard is used as FM receive frontend.

Line 3 change `usrp` to `usrp2`:

```
from gnuradio import gr, usrp2, optfir, blks2, rds, audio
```

comment out line 8

```
#from usrpm import usrp_dbid
```

To connect to USRP2 the following options can be used.

```
# connect to USRP
usrp_decim = 416                # old: usrp_decim = 250
self.u = usrp2.source_32fc("eth1") # old: self.u = usrp.source_c(0, usrp_decim)
print "USRP MAC: ", self.u.mac_addr() # old: print "USRP Serial: ", self.u.serial_number()
adc_rate = self.u.adc_rate()      # 100 MS/s
usrp_rate = adc_rate / usrp_decim # ~240 kS/s
audio_decim = 5
audio_rate = usrp_rate / audio_decim # ~48k kS/s
self.u.set_decim(usrp_decim)
```

In the following, checking of available daughterboards is by-passed. The USRP2 configuration used here is based on one fixed subdevice (TVRX receiver).

```
# if options.rx_subdev_spec is None:
#     options.rx_subdev_spec = usrp.pick_subdev(self.u,
#         (usrp_dbid.TV_RX, usrp_dbid.TV_RX_REV_2, usrp_dbid.BASIC_RX))

# self.u.set_mux(usrp.determine_rx_mux_value(self.u, options.rx_subdev_spec))
# self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
# print "Using d'board", self.subdev.side_and_name()

# gain, volume, frequency
self.gain = options.gain
# if options.gain is None:
#     g = self.subdev.gain_range()
#     self.gain = g[1]
```





Further non-existent functions are also commented out from GUI

```
# hbox.Add((5,0), 1)
# self.myform['gain'] = form.quantized_slider_field(parent=self.panel, size=hbox,
# label="Gain", weight=3, range=self.subdev.gain_range(), callback=self.set_gain)
# hbox.Add((5,0), 0)

# r = usrp.tune(self.u, 0, self.subdev, target_freq)
# r = self.u.set_center_freq(target_freq)

# self.myform['gain'].set_value(gain)
# self.subdev.set_gain(gain)
# self.u.set_gain(gain)
```

### 3.2. Capture samples from USRP2 in C++

For debugging reasons and to track the communication between USRP2 and GNU Radio, the source code of a capture software is included in

`$GNURADIOROOT/usrp2/host/apps/rx_streaming_samples.cc`

Executing the compiled version `rx_streaming_samples` allows a byte-by-byte capturing of the samples that the PC receives via the Gbit-Ethernet interface. Running

`$GNURADIOROOT/usrp2/host/apps/rx_streaming_samples -h`

from command line interface offers following options:

Usage: `rx_streaming_samples [options]`

Options:

```
-h show this message and exit
-e ETH_INTERFACE specify ethernet interface [default=eth0]
-m MAC_ADDR mac address of USRP2 HH:HH [default=first one found]
-f FREQUENCY specify receive center frequency in Hz [default=0.0]
-d DECIM specify receive decimation rate [default=5]
-g GAIN specify receive daughterboard gain [default=0]
-N NSAMPLES specify number of samples to receive [default=infinite]
-o OUTPUT_FILENAME specify file to receive samples [default=none]
-s write complex<short> [default=complex<float>]
-v verbose output
```

This code uses the `rx_sample_handler` to directly handle received raw sample data from USRP2. In GNU Radio C++ API, further information about the used data types can be found. The received raw sample data is packed into big-endian 32-bit unsigned ints for transport when standard USRP2 configuration is used. The packets contain 16-bit I data followed by 16-bit Q data organized in a 32-bit word.



Wire format (big-endian)					
I1	Q1	I2	Q2	I3	Q3
0x01FC	0x0021	0x01F9	0x0043	0x01F3	0x0067

Host format (little-endian)					
I1	Q1	I2	Q2	I3	Q3
0xFC01	0x2100	0xF901	0x4300	0xF301	0x6700

### 3.2.1. Convert `rx_streaming_samples` outfile to csv

The source code of `rx_streaming_samples.cc` makes clear, how received samples from USRP2 are processed. The copy functions are found in

`$GNURADIOROOT/usrp2/host/lib/copiers.cc`.

In `copy_u2_16sc_to_host_16sc` for instance, a byte-swap function is run through, if the application `rx_streaming_samples` is executed with parameter `“-s”`. The samples are written into an output file using parameter `“-o”`.

The function `copy_u2_16sc_to_host_16sc` performs a conversion from USRP2 “wire format” to “host format” (big-endian to little-endian format). With the given code it becomes clear which data types are used and how the byte-by-byte rotation is done.

The output file is filled with complex <float> - organized in 2-byte I followed by 2-byte Q data little-endian format. A reformatting of the outfile allows handling the ADC values in a common manner.

For data visualization (e.g. in a spreadsheet program) a csv-file with integer values is created. I & Q data are separated with comma as delimiter. Further details concerning the csv-format can be found at

[http://en.wikipedia.org/wiki/Comma-separated\\_values](http://en.wikipedia.org/wiki/Comma-separated_values)

The following code meets this requirement, if the data are captured with parameter `“-s”` from `rx_streaming_samples` application and stored in a file called `outfile.txt`. After the code having saved to a file called `convert_outfile_to_csv.c`, it can be compiled by running

```
g++ convert_outfile_to_csv.c -o convert_outfile_to_csv
```



By running the compiled version, all converted data are printed to STDOUT – so redirecting the data stream to a file by

```
convert_outfile_to_csv > outfile.csv
```

might be useful.

### 3.2.2. C++ source code for csv conversion

```
#include <iostream>
#include <fstream>
#include <complex>

using namespace std;

int main() {

    char * buffer;
    char buffer_c;
    int length = 16
    int iq_blk_len = 4; // 4 byte
    buffer = new char [iq_blk_len];

    ifstream infile;

    infile.open("outfile.txt", ios::binary);
    std::complex<int16_t> cplex;
    int16_t var_r;
    int16_t var_q;

    cout<<"I:,Q:\n";

    int i = 0;
    while(infile.good()) {
        i+=1;
        infile.read(buffer,iq_blk_len);

        // LITTLE ENDIAN AFTER CONVERSION 2 HOST!
        var_r =(((buffer[1]&0xFF)<<8) + (buffer[0]&0xFF));
        var_q =(((buffer[3]&0xFF)<<8) + (buffer[2]&0xFF));

        cplex =std::complex<int16_t>(var_r,var_q);
        cout<<cplex.real()<<" "<<cplex.imag()<<endl;

    }

    infile.close();
    delete[] buffer;

    cout<<endl;

    return 0;
}
```

Chapter 3.4 provides more detailed information about byte reordering and source code compilation. First, some variables are defined. Important data types are char (size 8 bit) and int16\_t (size 16 bit).



The recorded `outfile.txt` (see above chapter) is opened and read by 4 byte blocks. Every 4 byte block of the file consists of 16 bit I and 16 bit Q data. Before writing `var_r` and `var_q` to STDOUT, a byte swap is executed since the samples are stored in little-endian format. After parsing the whole `outfile.txt` and printing all samples (separated by a comma) to STDOUT, the file is closed and the allocated character pointer `buffer` is freed.

### 3.2.3. Data illustration in a diagram

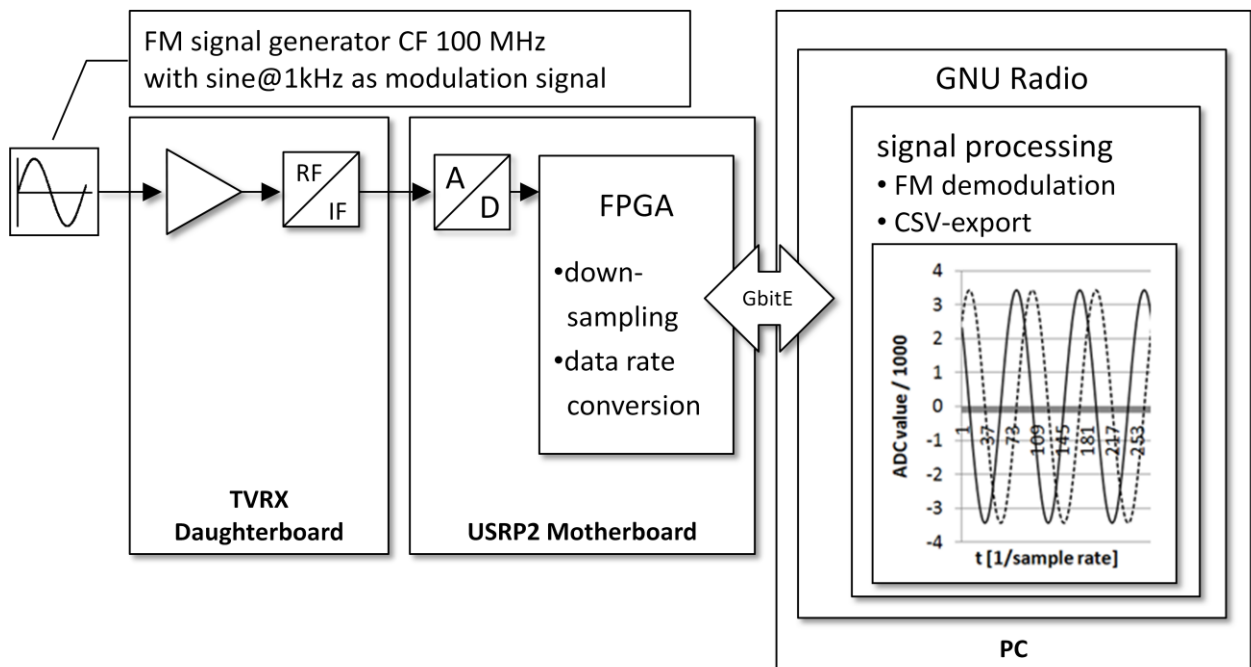


Figure 17: SDR block for FM signal processing

Having connected a signal generator with a carrier frequency of 100.001 MHz and using TVRX board, the following command produces a valid file named `outfile.txt`.

```
./rx_streaming_samples -e eth1 -f 1e8 -d 416 -o outfile.txt -N 50k -s -g 30
```



The meaning of the command line parameters is:

- select eth1 as interface where USRP2 is connected to
- set the USRP2s' center frequency to 100 MHz
- set a decimation factor of 416 – resulting in 240.384 kS/s
- specify that the recorded data should be written to a file called outfile.txt
- use complex<short> as data type written to file
- set receive daughterboard gain to 30

Because the USRP2s' center frequency is set to 100 MHz, the 1k offset from the signal generators' frequency is written to our outfile. After opening the converted data in a spreadsheet program, the comma separated values need to be divided into columns if not done automatically. Afterwards a column oriented chart is generated as shown in Figure 18.

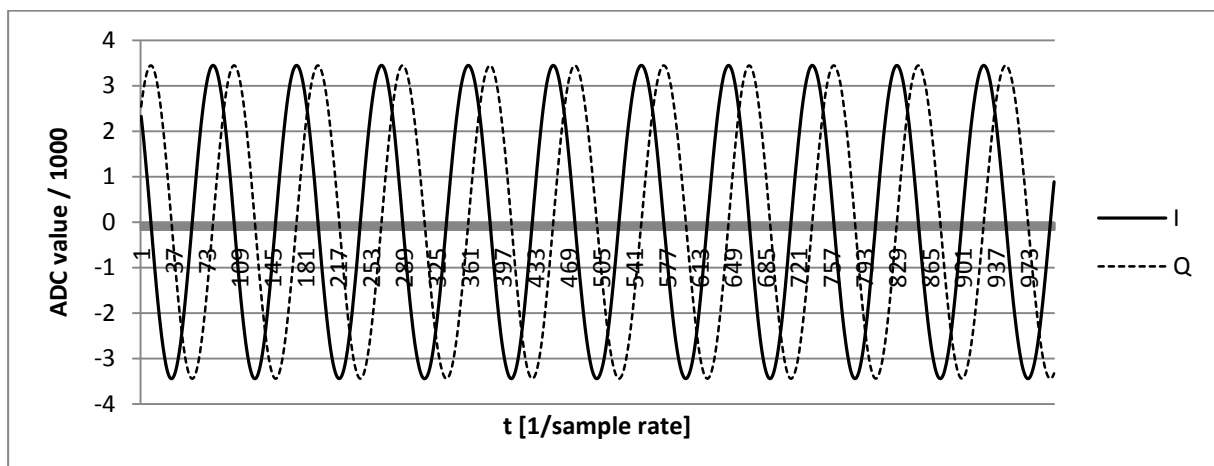


Figure 18: Illustration of csv data after byte swapping

Considering how many samples per period were taken and dividing our sample rate through that value, the former input frequency can be determined. Potentially a little shifted frequency may occur originated from relative clock stabilities between source and USRP2.

$$\frac{1e8 \text{ Hz} / \text{decimation}}{\text{samples per period}} = f [\text{Hz}]$$



In Figure 18 for example, approximately 95 samples per period were taken. Regarding the used decimation factor, this results in a calculated frequency of 2.53 kHz, instead of the expected 1 kHz signal.

$$\frac{1e8 \text{ Hz} / 416}{95} = 2.53 \text{ kHz}$$

This deviation is due to clock instabilities. An offset of 1.53 kHz related to a center frequency of 100 MHz corresponds to a clock stability of about 15 ppm.

$$\frac{1e8 \text{ Hz}}{(2530 - 1000) \text{ Hz}} = 15.3e^{-6} = 15 \text{ ppm}$$

This agrees well with USRP2 internal clock stability of about 20 ppm. [3]

### 3.3. Wireshark trace

Wireshark, formerly known as Ethereal, is a well known open source application for sniffing Ethernet traffic. The source code is provided under:

<http://www.wireshark.org>

Sniffing the network traffic allows to examine the data streams, which are interchanged between USRP2 and the host. On the next page, a section of a Wireshark trace is shown. The stream direction of this Ethernet packet is from USRP2 to PC, also visible in the source and destination fields of the header (IntelCor describes the host Gbit-Ethernet card).

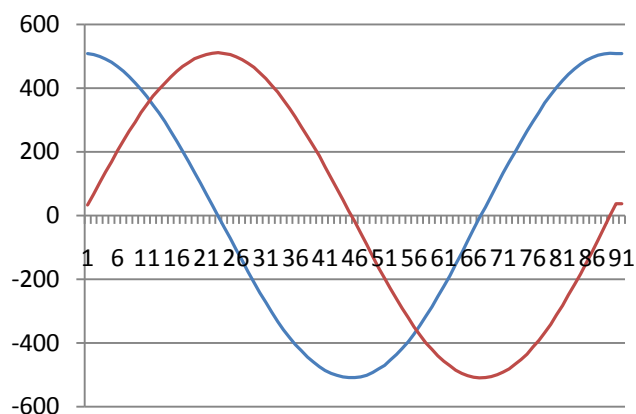


Figure 19: CSV visualization of traced samples



```
No.      Time      Source      Destination      Protocol Info
   84 0.166222 IeeeRegi_85:30:ed IntelCor_35:3e:c2 0xbeef Ethernet II
```

Frame 84 (1512 bytes on wire, 1512 bytes captured)

Data (1498 bytes)

```
...
0480 01 fc 00 21 01 f9 00 43 01 f3 00 67 01 eb 00 88 ...!...C...g...
0490 01 e1 00 a8 01 d4 00 cb 01 c5 00 eb 01 b4 01 09 .....
04a0 01 a0 01 25 01 8b 01 43 01 74 01 5d 01 5b 01 76 ...%...C.t.].[.v
04b0 01 42 01 8c 01 26 01 a0 01 08 01 b3 00 e9 01 c5 .B...&.....
...
05b0 01 43 fe 78 01 61 fe 90 01 7a fe aa 01 92 fe c8 .C.x.a...z.....
05c0 01 a8 fe e5 01 bb ff 06 01 cc ff 26 01 db ff 48 .....&...H
05d0 01 e7 ff 6b 01 f0 ff 8f 01 f7 ff b3 01 fb ff da ...k.....
05e0 01 fd 00 00 01 fc 00 25 .....%
```

This data packet is fragmented and converted to decimal values as shown below:

```
I- I -I- Q -I- I -I- Q -I- I -I- Q -I- I -I- Q -I
|01 fc|00 21|01 f9|00 43|01 f3|00 67|01 eb|00 88|
| 508| 33| 505| 67| 499| 103| 491| 136|
```

This exactly results in the following decimal I and Q values

508,33	233,453	-273,430	-509,-20	-223,-458	294,-414
505,67	202,468	-303,409	-507,-55	-189,-472	323,-392
499,103	168,480	-333,387	-502,-91	-153,-486	353,-368
491,136	135,492	-359,362	-493,-129	-118,-496	378,-342
481,168	101,499	-383,336	-482,-164	-81,-503	402,-312
468,203	67,505	-406,309	-471,-197	-43,-508	424,-283
453,235	32,509	-425,278	-454,-230	-8,-510	443,-250
436,265	-2,511	-445,248	-437,-262	26,-509	460,-218
416,293	-38,508	-460,219	-418,-292	61,-506	475,-184
395,323	-72,505	-475,187	-396,-321	98,-500	487,-149
372,349	-107,498	-487,152	-372,-350	133,-492	496,-113
347,374	-141,489	-496,117	-344,-376	167,-482	503,-77
322,396	-176,478	-502,84	-316,-400	200,-467	507,-38
294,416	-211,465	-507,49	-287,-421	232,-453	509,0
264,435	-243,448	-509,14	-255,-441	265,-436	508,37

and exactly to one period of the appropriate chart built with these CSV values as it can be seen in Figure 19.



### 3.4. Sending Samples

Before a data stream is sent with USRP/2, the appropriate I & Q values as well as the timing context of the samples have to be calculated. In this example, a complex sine wave will be calculated. By stepping through the unit circle in equivalent degree values (e.g. 22.5°) and plotting this amplitude in equidistant time distances ( $\Delta t$ ), a sine respectively cosine function results.

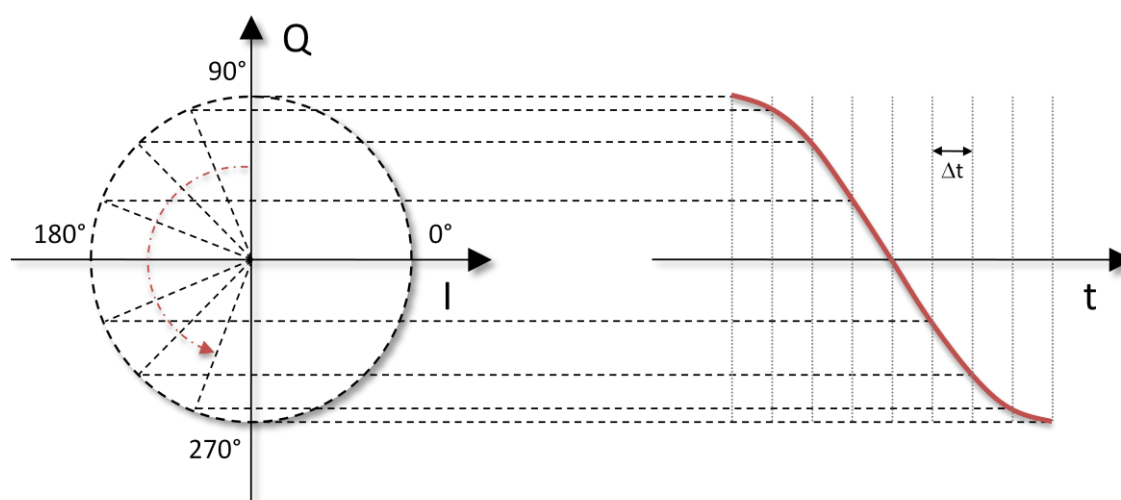


Figure 20: Principle of I & Q values for a sine function between 90° and 270°

In the following code, one sine wave period is represented with exactly 200 complex values.

```
short sample[400];
// calculate sine wave with amplitude of 2^12 in 200 steps
for (int i = 0; i < 200; i++) {
    // first 2 bytes (size of short is 2 byte) are cos-function (I)
    // and second sin-function (Q). (i<<1) multiplies increment variable i by two
    sample[i<<1] = 4096*cos(2*M_PI*(i % 200)/200);
    sample[(i<<1) + 1] = 4096*sin(2*M_PI*(i % 200)/200);
}
```

These discrete values are not related to time scale so far. To calculate the desired output frequency the DAC clock frequency as well as any interpolation coefficients of the USRP2 has to be taken into account.





USRP2 is run with 400 MHz DACs and a static interpolation factor of 4, implemented in the USRP2 firmware. [3] [11]

With an additional interpolation factor of 500 and with the given sine period of 200 samples the output frequency can be calculated as follows.

$$\text{Output sample rate: } \frac{\text{DAC rate}}{\text{interpolation factor}} = \frac{100 \text{ MS/s}}{500} = 200 \text{ kS/s}$$

$$\text{Output frequency: } \frac{200 \text{ kS/s}}{200 \text{ S/period}} = 1 \text{ kHz}$$

Sending 200e3 samples per second the resulting frequency is about 1 kHz.

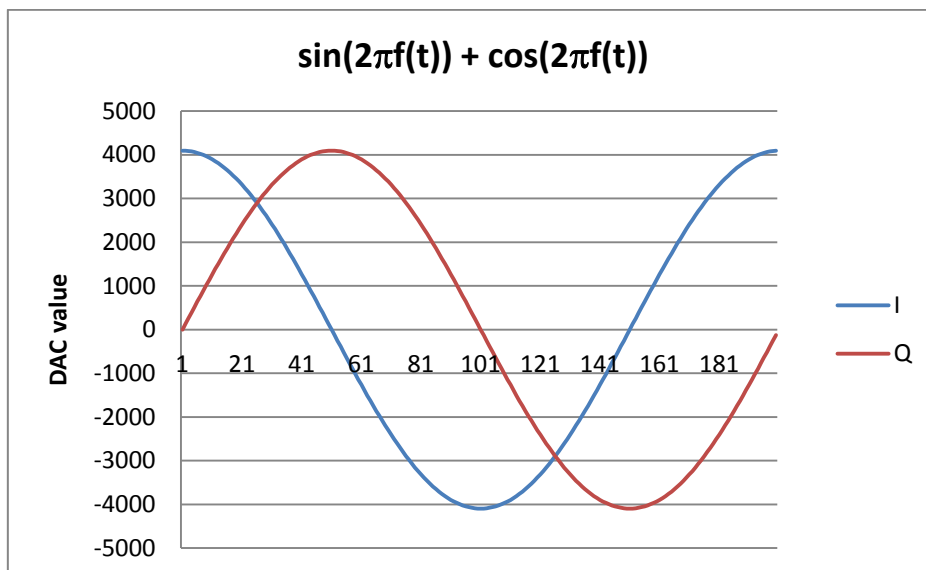


Figure 21: Calculated sine and cosine functions



### 3.4.1. Problems with byte orders

While calculating the I and Q values of the sine function, the hex values are printed out and written to a binary file. Confusingly displayed values and values written to the file are not identical. This is related to PC architecture. Here, a common Intel architecture with an internal little-endian memory ordering was used. Since the function `fwrite()` steps through the sample-array byte by byte, this results in a platform dependent byte order in the output file (named `outfile`), which differs from the displayed values on screen.

#### Source code to demonstrate byte order problems

```
#include <iostream>
#include <math.h>
#include <iomanip>

using namespace std;

int main(int argc, char *argv[])
{
    short sample[20];

    // set output format to hex
    cout<<hex;

    // calculate sine wave with amplitude of 2^12 in 200 steps, but output only 10 I&Q samples
    for (int i = 0; i < 10; i++) {
        // first 2 bytes (size of short is 2 byte) are cos-function (I)
        // and second sin-function (Q). (i<<1) multiplies increment variable i by two
        sample[i<<1] = 4096*cos(2*M_PI*(i % 200)/200);
        sample[(i<<1) + 1] = 4096*sin(2*M_PI*(i % 200)/200);

        // print samples to output, formatted in 4 digits with leading zeros
        cout<<setw(4)<<setfill('0')<< (sample[i<<1] ) << " ";
        cout<<setw(4)<<setfill('0')<< (sample[(i<<1) + 1])<<endl;
    }

    // open and if not exists create file binary file sine_out_2.bin for writing
    FILE * pFile;
    pFile = fopen("sine_out_2.bin","wb");

    // write sample array byte by byte from memory to file
    fwrite(sample,1,sizeof(sample),pFile);

    // close file pointer
    fclose(pFile);

    // if necessary, insert a system("pause") for windows or the like here
    return 0;
}
```



Using the g++ compiler, the source code can be compiled by execution of

```
g++ source_filename.c -o outfile
```

Normally, the executable flag of the created outfile will be set automatically. If not, this can be done in Linux with a:

```
chmod a+x outfile
```

The application can be run by executing:

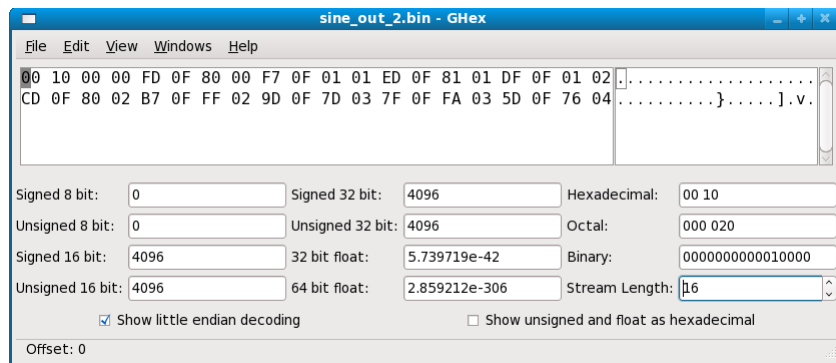
```
./outfile
```

During execution the following 10 hex values will be displayed on STDOUT:

```

STDOUT:
1000 0000
0ffd 0080
0ff7 0101
0fed 0181
0fdf 0201
0fcd 0280
0fb7 02ff
0f9d 037d
0f7f 03fa
0f5d 0476

```



**Figure 22: Byte order of sine\_out\_2.bin (little-endian)**

Examining the generated output file sine\_out\_2.out with a hex editor, the different byte order can be noticed, see Figure 22. The data on screen are displayed in big-endian (as usual), but written to the file in little-endian format (as it is common in Intel architectures).



To avoid this problem, the `fprintf()` function may be used, which operates hardware platform independently:

```
#include <iostream>
#include <math.h>
#include <iomanip>

using namespace std;

int main(int argc, char *argv[])
{
    short sample[20];

    // open and if not exists create file binary file sine_out_3.bin for writing
    FILE * pFile;
    pFile = fopen("sine_out_3.bin","wb");

    // set output format to hex
    cout<<hex;

    // calculate sine wave with amplitude of 2^12 in 200 steps, but output only 10 I&Q samples
    for (int i = 0; i < 10; i++) {
        // first 2 bytes (size of short is 2 byte) are cos-function (I)
        // and second sin-function (Q). (i<<1) multiplies increment variable i by two
        sample[i<<1] = 4096*cos(2*M_PI*(i % 200)/200);
        sample[(i<<1) + 1] = 4096*sin(2*M_PI*(i % 200)/200);

        // write every byte (character = 8 bit) with platform independent function fprintf to file
        fprintf(pFile,"%c",sample[i<<1] & 0xff);
        fprintf(pFile,"%c",sample[i<<1]>>8 & 0xff);
        fprintf(pFile,"%c",sample[(i<<1) + 1] & 0xff);
        fprintf(pFile,"%c",sample[(i<<1) + 1]>>8 & 0xff);

        // print samples to output, formatted in 2 digits with leading zeros
        cout <<setw(2)<<setfill('0') <<(sample[i<<1] & 0xff);
        cout <<setw(2)<<setfill('0') << (sample[i<<1]>>8 & 0xff)<<" ";
        cout <<setw(2)<<setfill('0') << (sample[(i<<1) + 1] & 0xff);
        cout <<setw(2)<<setfill('0') << (sample[(i<<1) + 1]>>8 & 0xff)<<endl;
    }

    // close file pointer
    fclose(pFile);

    // if necessary, insert a system("pause") for windows or the like here
    return 0;
}
```



Now the standard output on screen and the file content are identical:

```

STDOUT:
0010 0000
fd0f 8000
f70f 0101
ed0f 8101
df0f 0102
cd0f 8002
b70f ff02
9d0f 7d03
7f0f fa03
5d0f 7604
    
```

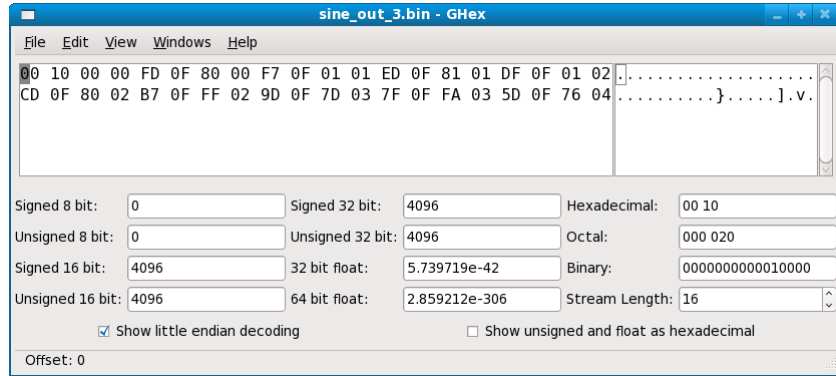


Figure 23: Platform independent byte order of sine\_out\_2.bin (little-endian)

The standard output function of GNU Radio to USRP2 (tx\_samples) turns the input little-endian format into the big-endian format that is transferred via Gigabit Ethernet. So the file format “little-endian” as described above has to be used to write sine wave samples correctly to the USRP2, see the following sample program call.

```
GNURADIO_ROOT/usrp2/host/apps/tx_samples -e eth1 -f 0 -I sine_out_2.bin -i 500 -r
```

Annex F provides a source code that displays the following values and also creates a csv file with the calculated decimal I & Q values. Byte ordering functions can be traced step by step.

I	I (dec)	Q (dec)	I (hex)	Q (hex)	I (hex)      Q (hex)				I (hex)      Q (hex)				
					Big Endian (MSB left)				Little Endian (MSB right)				
0	4096	0	1000	0000	10	00	00	00	00	10	00	00	00
1	4093	128	0ffd	0080	0f	fd	00	80	fd	0f	80	00	
2	4087	257	0ff7	0101	0f	f7	01	01	f7	0f	01	01	
...			...		...				...				
197	4077	-385	0fed	fe7f	0f	ed	fe	7f	ed	0f	7f	fe	
198	4087	-257	0ff7	feff	0f	f7	fe	ff	f7	0f	ff	fe	
199	4093	-128	0ffd	ff80	0f	fd	ff	80	fd	0f	80	ff	



The little-endian conversion can be done as shown in the code below, or with the network function `htonl` from `gruel/inet.h`. This network function isn't held in the C++ standard, but provided with `gcc`. Its argument is a 32-bit value.

These byte swaps may be done with the following code lines:

```
full_sample[i] = ((sample[i<<1] >> 8) & 0xff | (sample[i<<1] << 8))<< 16 |  
                ((sample[(i<<1) + 1] >> 8) & 0xff | (sample[(i<<1) + 1] << 8) & 0xff00);
```

- First I-byte is shifted 8 bits right followed by a bitwise AND operation, that defines a bit mask for the last byte.
- Second I-Byte is shifted 8 bit to the left and also operated with the appropriate bit mask. The bit masks are necessary because some compilers may cause trouble if a shift operation of a negative int-value results in writing 0xf-bytes in the left-hand bytes.
- On first and second I-byte there is a bitwise OR operation performed, which results in the required 16 bit word.
- This 2-byte word is shifted left by 16 bits, since the I-bytes define the upper 16 digits of our 32 bit data structure.
- The same byte swap operation is done respectively with the Q-bytes. Finally both 16 bit words are combined with a bitwise OR operation, resulting in a full 32-bit little-endian I&Q sample.



## 4. OpenBTS

The OpenBTS Project implements a GSM access point in open source software. Standard GSM-compatible mobiles can access via the GSM air interface (Um) to USRP and so register to OpenBTS software running on the host. With this project, the functionality of a GSM Base Transceiver Station (BTS) from cellular operators can be emulated. The project was founded by David A. Burgess and Harvind S. Samra with the intention to offer a low cost GSM BTS for instance as being used in developing countries. [12]

Project source code and further information is provided under

<http://www.gnuradio.org/redmine/wiki/1/OpenBTS>

The projects basic requirements are

- USRP with 2xRFX900 or 2xRFX1800 transceiver daughterboards
- Asterisk Server
- GNU Radio API

### 4.1. Block diagram – traditional GSM in short

A traditional GSM core network consists of the following core elements:

- Base Transceiver Station (BTS) and Base Station Controller (BSC)
- Mobile Switching Center (MSC) / Visitor Location Register (VLR)
- Home Location Register (HLR) / Authentication Center (AuC)

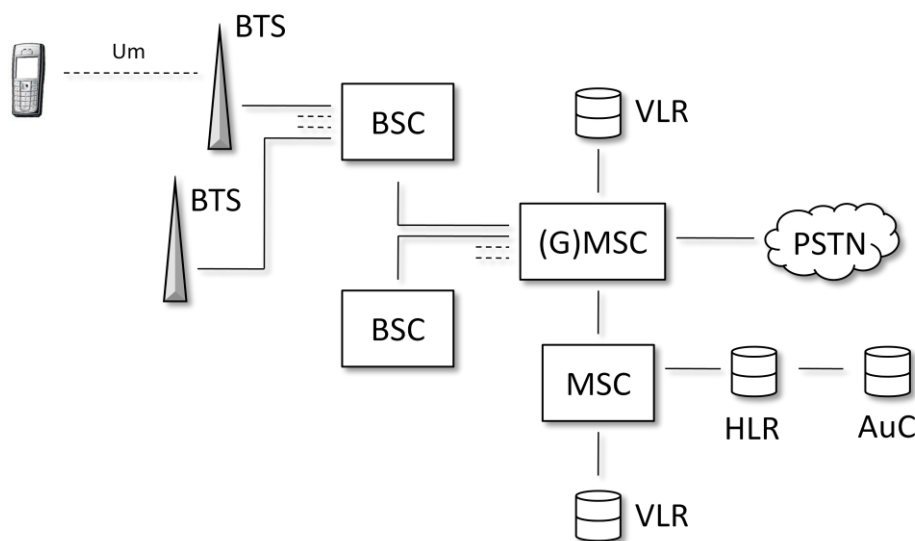


Figure 24: summarized GSM block diagram



The BTS describes the air interface (Um) to the mobiles. Several BTS are connected to a BSC, responsible for control functions like handovers while the mobile is moving between radio cells. The Gateway MSC (GMSC) serves as interface between Public Switched Telephone Network (PSTN) and the radio cells the mobile is connected to. MSCs control the connected BSCs and house information of all mobile subscribers connected to the respective MSC area. VLR and HLR are the databases, where information about the mobile subscriber is held in. In HLR, all necessary subscriber data are permanently stored and transferred to the VLR of a guest MSC as a temporary copy, each time the mobile logs into a foreign MSC area. The AuC is responsible for storage of the secret key  $K_i$ , the authentication process using SRES and the generation of  $K_c$  for encrypting the voice data. The other interfaces, e.g. PSTN interface in Figure 24, are not needed in the following.

Further information about GSM standard and network architecture is provided under

<http://www.etsi.org/Website/Technologies/gsm.aspx>

or under numerous GSM network illustrations like

<http://ccnga.uwaterloo.ca/~jscouria/GSM/gsmreport.html>

### **Authentication scheme in short**

If a mobile wants to log into a radio cell, the Base Station Subsystem (BSS) sends an “Authentication Request” with a randomly chosen number RAND that serves as a challenge. The mobile calculates a value called SRES based on RAND with the  $K_i$  secret key stored in its SIM card. Submitting the SRES to BSS is called “Authentication Response”. The BSS verifies the SRES and grants access to the network if it’s correct. More information can be found in [13].

Since OpenBTS doesn’t have access to an operator’s AuC (in which the subscriber’s secret key information is stored), it just doesn’t care about the mobiles calculated SRES value and accepts it. For the same reason, the Um interfaces is operated in the no-cipher mode which means that voice data is not encrypted

After reception of the authentication request, OpenBTS tries to register the mobile subscriber as SIP-user in Asterisk PBX. So OpenBTS works as an Um-to-SIP gateway.





Further information as well as an installation guide can be found at

[https://81.56.142.154/Cour/These/OpenBTS/OpenBTS\\_Guide\\_En\\_v0.1.pdf](https://81.56.142.154/Cour/These/OpenBTS/OpenBTS_Guide_En_v0.1.pdf)

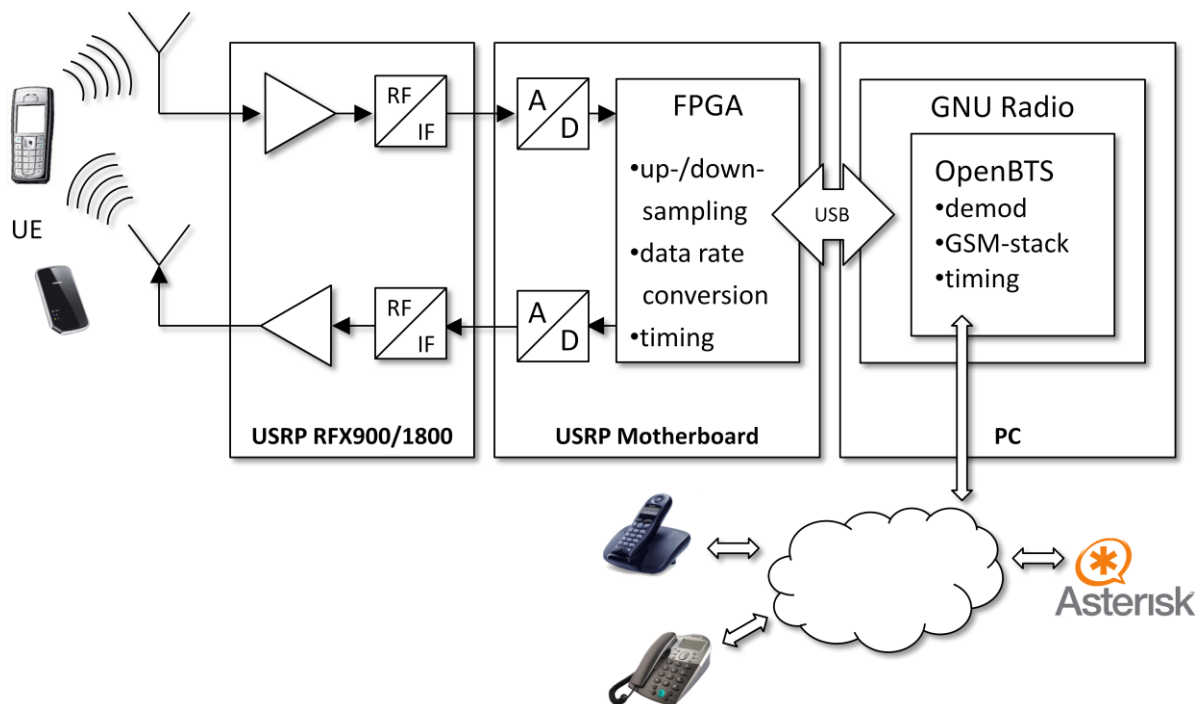


Figure 25: Functional principle of SDR with OpenBTS



## 4.2. Laboratory setup

To avoid disturbances in local cellular networks, the operator's licensed frequencies can only be used in a shielded environment. A respective laboratory setup is shown below.

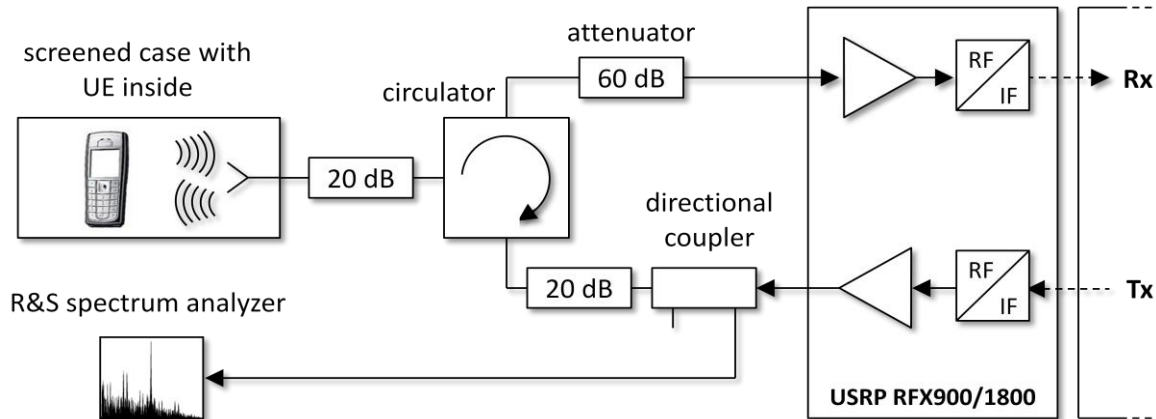


Figure 26: Block diagram of OpenBTS laboratory setup

Starting from Tx SMA connector of the RFX board which is plugged into slot A of USRP motherboard, a directional coupler is attached. Using this coupler, the Tx signal is decoupled to the RF input of a spectrum analyzer with a decoupling attenuation of about 20 dB. The spectrum analyzer traces the Tx signal and visualizes signal strength, used frequency and operation status of the OpenBTS application.

The second decoupling port of the directional coupler isn't used and can be left open. This can be done because a signal which might be reflected there (starting from Tx chain of UE) is attenuated by a 20 dB attenuator, by circulator decoupling, another 20 dB attenuator, 20 dB directional coupler decoupling and vice versa, resulting in an attenuation of approx. 140 dB. Also mismatching of connectors can be neglected.

Behind the coupler, a 20 dB attenuator absorbs a part of the transmitted signal from USRP. The following center of the assembly is built by a circulator, used for decoupling of the receive and the transmit path. A signal arriving on an arbitrary port is passed to the next port in direction of arrow only with little insertion loss, to the next port against the arrow though

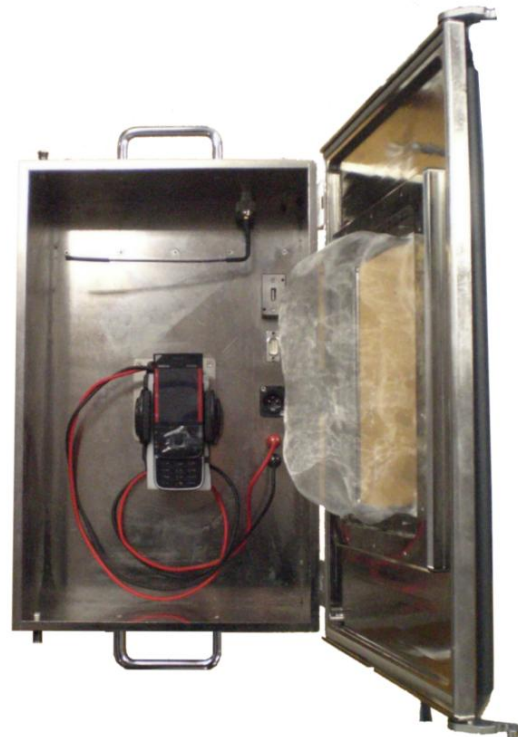


with high attenuation. So the Tx signal from USRP is passed to the UE but decoupled from USRP Rx chain and a signal sent by the mobile is circulated to USRP Rx board but decoupled from USRP Tx chain.

Having passed the circulator, the transmit signal from USRP is attenuated once more and radiated by an antenna mounted in the screened case.



**Figure 27: RF screened case**



**Figure 28: Opened case with UE and antenna inside**

The same antenna serves as receive antenna in USRP Rx chain, which is equivalent to UE Tx path. A signal sent by the mobile is attenuated about 20 dB, passed by the circulator in the direction of arrow, again attenuated about 60 dB and finally reaches the Rx SMA connector of RFX board plugged into USRP slot B.

### 4.3. S-parameters of circulator and directional coupler

With scattering parameters, also abbreviated S-parameters, information about the response characteristics of the two devices can be gathered.

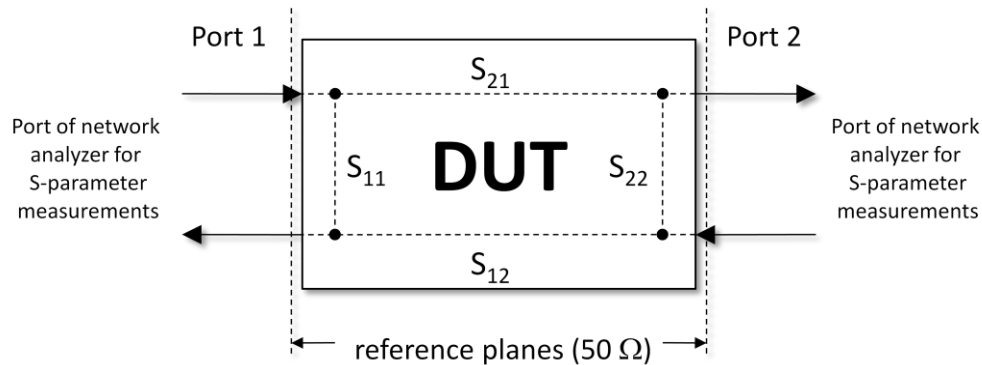


Figure 29: S-parameters of a 2 port device (DUT = device under test)

In short, on a 2-port device, the  $S_{21}$  parameter defines the ratio between output wave at port 2, to input wave at the opposite port. Plotting this behavior over a frequency range gives detailed information about the amplitude and phase relations from the input to the output port. S-parameter measurements are done with a network analyzer. The parameter  $S_{12}$  measures the wave ratio between port 1 (output) and port 2 (input) of the device under test (DUT).  $S_{11}$  and  $S_{22}$  are called the reflection parameters. Their measurement tells mainly how matched the input port 1 respectively 2 is by comparing how much power is reflected while a signal is sent to port 1 respectively 2. It should be noted that the impedances (usually  $50\ \Omega$ ) of both reference planes (see Figure 29) need to be matched.

The circulator is in fact a 3-port device, so with one 3-port measurement all 9 possible S-parameters are measured at once. To get a general survey about the OpenBTS laboratory setup, a measurement of the whole assembly around the circulator is shown below.

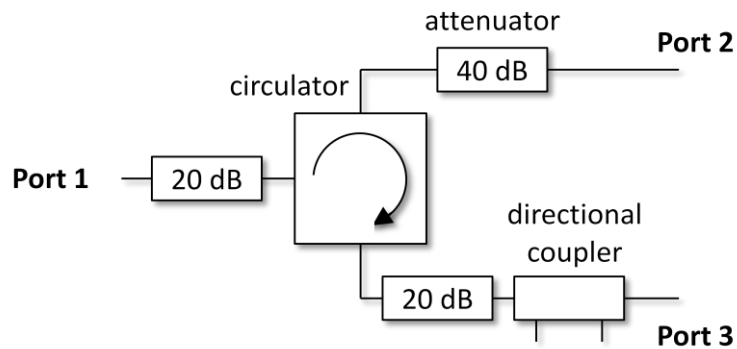


Figure 30: S-parameter measurement setup

Measuring this whole setup with a 3 port measurement on a network analyzer (Rohde & Schwarz ZVA 8), results in 9 charts as mentioned above. The interesting s-parameters are  $S_{21}$ ,  $S_{31}$ ,  $S_{13}$ ,  $S_{23}$  since they show the insertion loss and decoupling isolations. Figure 31 illustrates the decoupling attenuation of the circulator from port 1 to port 3. A full 3-port s-parameter measurement of the laboratory setup is provided in Annex E.

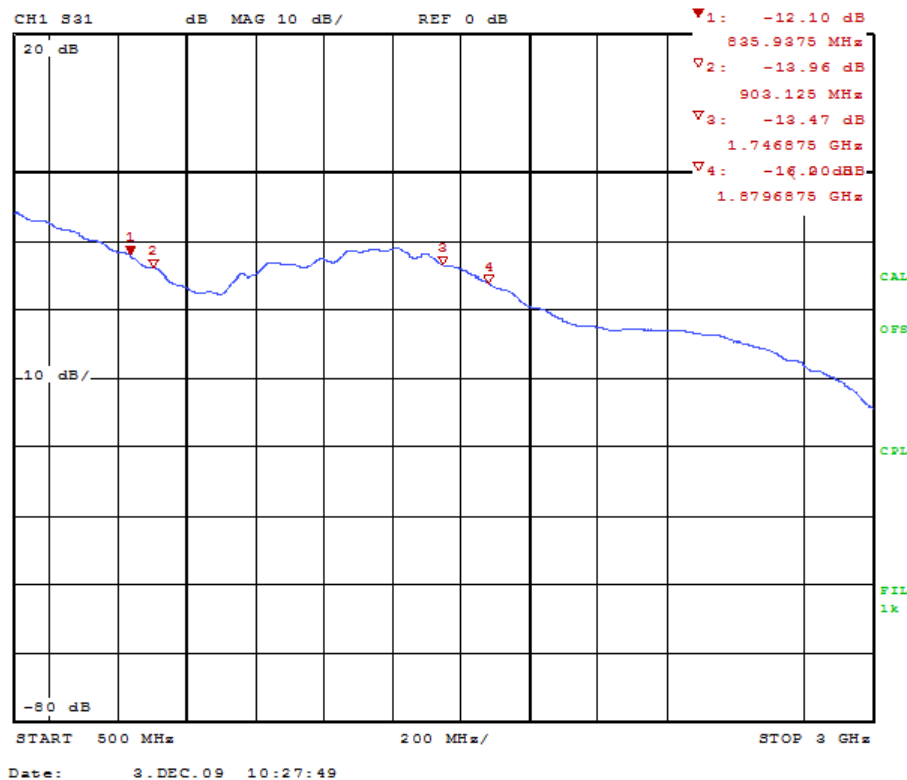


Figure 31: Decoupling of the circulator from port 1 to port 3 ( $S_{31}$ )



### Measured frequencies:

In GSM standard 3GPP TS 45.005 V9.1.0 (2009-11), thirteen possible frequency bands were defined. The four common bands in use are GSM-850/900/1800/1900, so afterwards the edge frequencies as well as the appropriate mid-frequency are listed. The mid-frequencies are calculated values and don't need to have an associated absolute radio frequency channel number (ARFCN).

	uplink			downlink		
<b>GSM 850</b>	824	836.5	849	869	881.5	894
<b>GSM 900</b>	890	902.5	915	935	947.5	960
<b>GSM 1800</b>	1710	1747.5	1785	1805	1842.5	1880
<b>GSM 1900</b>	1850	1880	1910	1930	1960	1990

**Table 3: Common used GSM bands with low, high and mid-frequencies in MHz**

To have an overview about the usability of the used circulator, the measurement frequencies are chosen in the mid-frequency ranges of all bands regarding uplink and downlink paths. Uplink and downlink indicate the RF direction as seen from the mobile. Uplink describes the direction to the base station, downlink means receiving a signal from the base station. So uplink is the Tx path and downlink the Rx path as seen from mobile. Regarding this, different frequencies have to be considered between the circulator ports (see Figure 30). From port 1 to port 2 the insertion loss in uplink frequencies has to be observed as well as the decoupling attenuation to port 3 in same frequency ranges. Port 3 to port 1 has to be considered for downlink frequencies respectively the decoupling attenuation from port 3 to port 2.

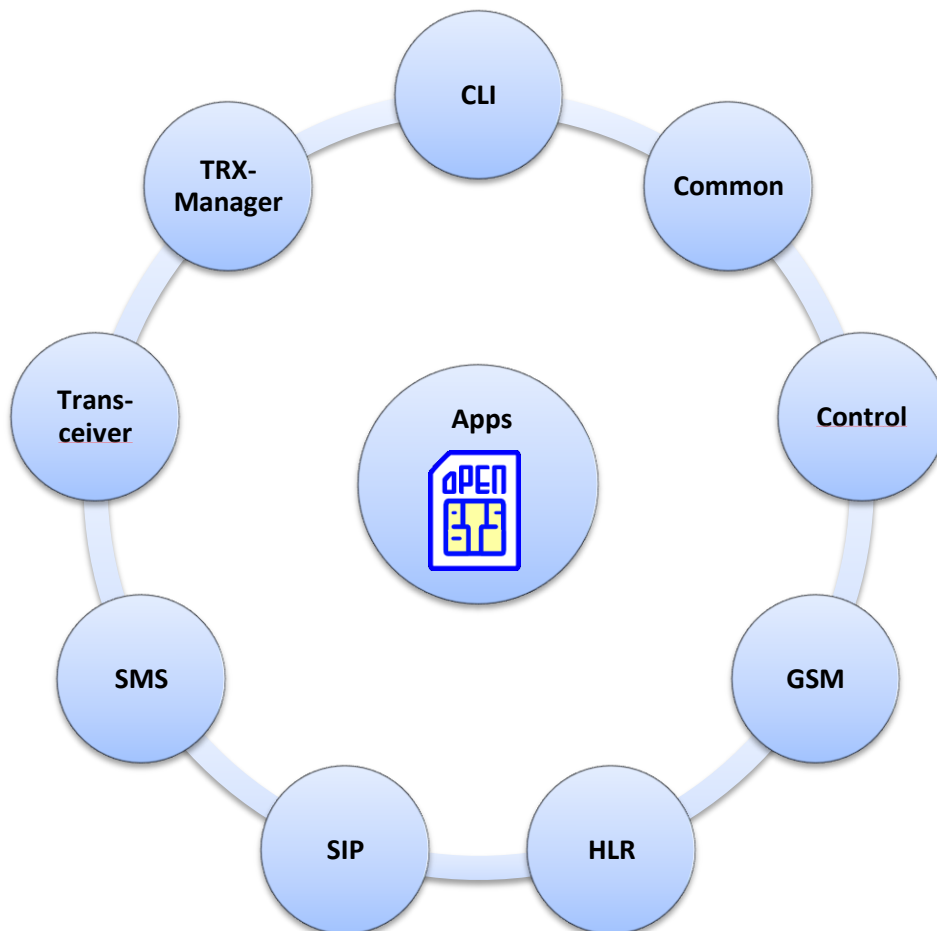
from \ to	port 1	port 2	port 3
port 1	-	836.5, 902.5 [-69 dB] 1747.5, 1880 [-61.5 dB]	836.5, 902.5 [-54.6 dB] 1747.5, 1880 [-57.3 dB]
port 2	-	-	-
port 3	881.5, 947.5 [-49.6 dB] 1842.5, 1960 [-42.5 dB]	881.5, 947.5 [-74.4 dB] 1842.5, 1960 [-78.2 dB]	-

**Table 4: Frequencies in MHz measured for decoupling and insertion loss of circulator**



#### 4.4. Components of OpenBTS

Effecting a clear topology, the source code of the OpenBTS project is structured as shown below.



Apps	In the apps folder, the application executable OpenBTS as well as its sources and the global configuration file OpenBTS.config are stored. Furthermore an application called sendSimple shows how SMS over SIP is implemented.
CLI	In this directory, relevant source code of the command line interface for OpenBTS can be found.
CommonLibs	Defines some often used classes and functions like the vector class used to organize data streams and socket wrappers.



Control	In this directory, control layer functions are held. They are responsible for call control (mobile originated/terminated calls), radio resource (paging, access grant), mobility management (location updates) and SMS control functions.
GSM	Implementation of GSM stack
HLR	Interface to Asterisk so far, performing the IMSI authentication. A MySQL connection is planned.
SIP	The SIP interface is held in this directory. Reading and writing of messages like registrations and invitations connecting to Asterisk are formed here.
SMS	Handling of SMS messages
Transceiver	Defines the radio interface, which accesses to USRP. All USRP specific configurations like setup of registers are done here. All Tx and Rx data pass these classes. It operates basically as lowest layer in GSM stack, performing modulation, demodulation and detection of GSM bursts.
Transceiver52M	Replacement for Transceiver when using an external highly stable 52 MHz reference clock for USRP. The 52 MHz clock fits well into the clock hierarchy of GSM and does not require further resampling.
TRXManager	TRX Manager builds the interface to the transceiver using UDP sockets. It is responsible for master clock indications, power control, tuning and timeslot control.

For some directories, such as for the transceiver, more detailed information is provided in a contained README file.





## 4.5. Asterisk

Asterisk is a software implementation of a telephone PBX, allowing attached phones to communicate with each other and connect them to telephone services like the PSTN and VoIP services.

OpenBTS uses Asterisk to register each connected mobile over its SIM-Card as SIP user. The SIP username is build using the 4 characters “IMSI” directly followed by the 15-digit IMSI number provided in the operators SIM card. Before OpenBTS version 2.5 Lacassine, only the IMSI number without “IMSI”-prefix was registered as username.

So if a “new” mobile tries to register to OpenBTS, its IMSI number has to be added to the Asterisk config files, because otherwise no call can be set up. The IMSI number can be found in the OpenBTS log file (test.out by default) which is declared in the OpenBTS.config file.

```
GSMLogicalChannel.cpp:76:send: L3 SAP0 sending MM Identity Request type=IMSI
GSML3Message.cpp:162:parseL3: L3 recv MM Identity Response mobile id=IMSI=001010123456789
SIPEngine.cpp:148:Register: SIPEngine::Register mState=NULL 0 callID 685931626
SIPInterface.cpp:107:addCall: creating SIP message FIFO callID 685931626
SIPInterface.cpp:167:write: write REGISTER sip:127.0.0.1 SIP/2.0
SIPInterface.cpp:192:drive: read SIP/2.0 404 Not found
SIPInterface.cpp:114:removeCall: removing SIP message FIFO callID 685931626
MobilityManagement.cpp:190:LocationUpdatingController: registration ALLOWED: IMSI=001010123456789
```

In the above shown transaction, the mobile is asked for its IMSI and responds therewith. Afterwards OpenBTS tries to register the mobile at Asterisk server running on localhost (127.0.0.1) via SIP REGISTER but gets back a “404 Not found”.

The registration to OpenBTS is “ALLOWED”, but the SIP registration failed – the user isn’t known in Asterisk. If the SIP registration has been accomplished successfully, a “SUCCESS” appears in the log file:

```
SIPInterface.cpp:167:write: write REGISTER sip:127.0.0.1 SIP/2.0
SIPInterface.cpp:192:drive: read SIP/2.0 200 OK
MobilityManagement.cpp:189:LocationUpdatingController: registration SUCCESS: IMSI=001010123456789
```



### 4.5.1. Asterisk configuration

The required config files are sip.conf and extensions.conf from the Asterisk server installation (normally found at /etc/asterisk, depending on the operating system an installation).

Creating a new SIP user is done by editing the sip.conf as follows:

```
[IMSI001010123456789]
canreinvite=no
type=friend
allow=gsm
context=sip-external
host=dynamic
```

#### In extensions.conf add

```
[macro-dialGSM]
exten => s,1,Dial(SIP/${ARG1})
exten => s,2,Goto(s-${DIALSTATUS},1)
exten => s-CANCEL,1,Hangup
exten => s-NOANSWER,1,Hangup
exten => s-BUSY,1,Busy(30)
exten => s-CONGESTION,1,Congestion(30)
exten => s-CHANUNAVAIL,1,playback(ss-noservice)
exten => s-CANCEL,1,Hangup
```

This code adds an Asterisk macro called dialGSM, recommended by David Burgess. After macro definition, the local phone number for the SIP-user can be defined in the appropriate context “sip-external”. [14]

```
[sip-external]
exten => 2103,1,Macro(dialGSM,IMSI001010123456789)
exten => 2104,1,Macro(dialGSM,IMSI001234567891010)
```

The local number 2103 is assigned to SIP-user IMSI001010123456789 e.q. and the dialGSM macro is executed, starting the call with:

```
exten => s,1,Dial(SIP/IMSI001010123456789)
```

More information about macros in Asterisk can be found e.q. on

[http://www.asteriskguru.com/tutorials/extensions\\_conf.html](http://www.asteriskguru.com/tutorials/extensions_conf.html)



## 4.6. Automated SIP user registration

Browsing every time the OpenBTS log file while an unknown IMSI-number tries to register is quite time-consuming. There are several options to solve this issue. The most reliable way is certainly to extend the registration procedure directly where it happens. In the following, a possible solution is explained:

### 4.6.1. Code modifications

`$OPENBTSROOT/Control/MobilityManagement.cpp`

```

...
191     if (success)
192         LOG(INFO) << "registration SUCCESS: " << mobID;
193     else {
194         LOG(INFO) << "registration ALLOWED: " << mobID;

        // print an information in the log that a new IMSI was detected
195         LOG(INFO) << "NEW IMSI DETECTED, WRITING TO SIP.CONF: " << mobID.digits();

        // create a file pointer and jump at the end of file /etc/asterisk/sip.conf
196         FILE * pFile;
197         pFile = fopen("/etc/asterisk/sip.conf","a");

198         // write the new SIP user in the sip.conf with the recommended options
199         fprintf(pFile,"\n[IMSI%s]\ncanreinvite=no\ntype=friend\nallow=gsm\ncontext=sip-
external\nhost=dynamic\n",mobID.digits());
        // close file pointer to sip.conf and open extensions.conf
200         fclose(pFile);
201         pFile = fopen("/etc/asterisk/extensions.conf","a");

        // dice a internal phone number between 2500 und 2999
202         short rnd = (rand() % 500 + 2500);
        // write number and SIP user to extensions.conf and close file pointer

203         fprintf(pFile,"\nexten => %i,1,Macro(dialGSM,IMSI%s)\n",rnd,mobID.digits());
204         fclose(pFile);
        // write a log info while reloading asterisk configuration
205         LOG(INFO) << "RELOADING ASTERISK CONFIG... " << system("/etc/init.d/asterisk
reload");
206
207     }
...

```



---

The code above extends the `Control::LocationUpdatingController` in `MobilityManagement.cpp`. For several reasons this is the right place to catch the IMSI.

On the SIP message “404 Not found” as seen in the log file section mentioned above can’t be relied on, because this SIP message also may occur if a called number doesn’t exist in Asterisk. Also the first Location Update Request from a mobile can’t be used, because the mobile may send its TMSI if it was logged into another radio cell before. So it’s better to monitor the variable “success” in the `LocationUpdatingController`.

Registration with the given code works only if the Asterisk config files are prepared as shown in chapter 4.5.1. In particular, the context statement with its associated parameters are static entries at the end of the file. With `mobID.digits()` the current IMSI can be read out.

To run the modified code, OpenBTS has to be rebuilt by execution of “make” in the `$OPENBTSROOT`.

#### 4.6.2. Code modifications with welcome SMS

After the successful registration, the assigned subscriber number is appended to `extensions.conf`. Therefore a SMS might be sent out to the subscriber as written below.

This source code is adapted from `sendSMS()`-function in `$OPENBTSROOT/CLI/CLI.cpp` and modified for an automatic transaction.

A connection to Asterisk is established and a SIP message sent with message body

*"Welcome to IKTmobile - your number is "*

followed by the assigned phone number. The delays were added because the registration process needs a short while. Instead of using `sleep()` function here, this code might be swapped out to build a external stand-alone tool that can be executed without delaying the OpenBTS code (like `$OPENBTSROOT/apps/sendSimple`).




---

```

/*-----*/
UDPsocket sock(0,"127.0.0.1",gConfig.getNum("SIP.Port"));
unsigned port = sock.port();
unsigned callID = random();
sleep(5);
// Just fake out a SIP message.
const char form[] = "MESSAGE sip:IMSI%s@127.0.0.1 SIP/2.0\nVia: SIP/2.0/UDP
127.0.0.1;branch=z9hG4bK776sgdkse\nMax-Forwards: 2\nFrom: %s
<sip:%s@127.0.0.1>:%d;tag=49583\nTo: sip:IMSI%s@127.0.0.1\nCall-ID:
%d@127.0.0.1:5063\nCSeq: 1 MESSAGE\nContent-Type: text/plain\nContent-Length:
%d\n\n%s\n";

char txtBuf[150];
snprintf(txtBuf,sizeof(txtBuf),"%s%i","Welcome to IKTmobile - your number is
",rnd);
char outbuf[2048];

sprintf(outbuf,form,mobID.digits(),"1000","1000",port,mobID.digits(),callID,strlen(txtBuf),txtBuf);
sock.write(outbuf);
sleep(2);
sock.write(outbuf);
sock.close();
/*-----*/

```

#### 4.7. SWLOOPBACK

OpenBTS offers an implementation of a software loopback operation concerning USRP. By enabling this compilation flags, no USRP needs to be available since all of the Transceiver interface data is written to and read from a software buffer.

The flag is enabled by adding the additional parameters to the configure command.

```
./configure CXXFLAGS=-DSWLOOPBACK CPPFLAGS=-DSWLOOPBACK
```



### 4.8. Porting issues USRP/2

So far, OpenBTS uses USRP only as SDR platform. USRP has some significant differences that affect the handling with USRP2

- PC connection is USB instead of Ethernet
- Different sample rates
- Different FPGA accessibility
- Number of usable transceiver boards

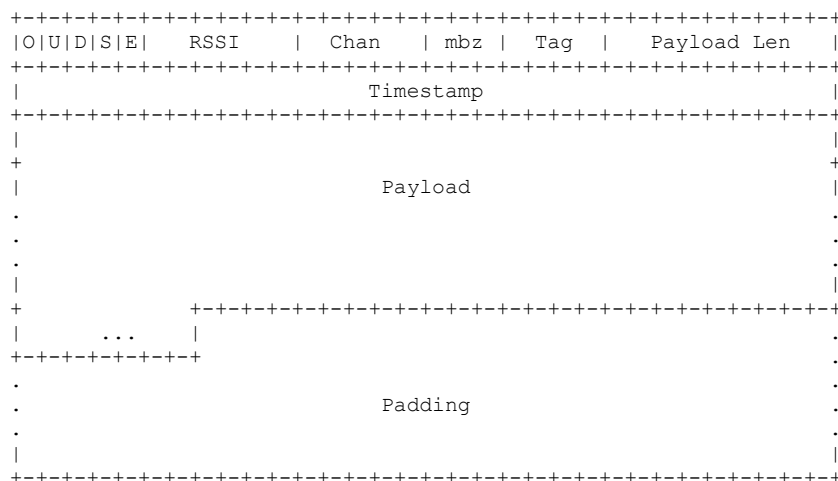
In addition to this, not all necessary functions are implemented in GNU Radio for USRP2 so far. So in-band signaling is planned to be implemented in 2010, but inoperable right now.

#### 4.8.1. In-band signaling

In-band signaling means a technique to send control information within the normal flow of user data, e.g. the Dual-tone multi-frequency signaling in analog telephone lines. For instance, as OpenBTS needs in-band signaling for timing issues - GSM uses TDMA which needs a very strict time handling – this control information has to be embedded in the USB data flow. George P. Nychis developed in-band signaling for USRP in GNU Radio.[15]

The respective data structure is shown in Figure 32.

The layout is 32-bits wide. All data is transmitted in little-endian format across the USB.



**Figure 32: Data structure of an in-band signaling block**  
 from \$GNURADIOROOT/usrp/doc/inband-signaling-usb

...  
 S Start of Burst Flag: Set in an OUT packet if the data is the first segment of what is logically a continuous burst of data.



Must be zero in IN packets.

- E End of Burst Flag: Set in an OUT packet if the data is the last segment of what is logically a continuous burst of data. Must be zero in IN packets. Underruns are not reported when the FPGA runs out of samples between bursts.

...

Timestamp: 32-bit timestamp.

On IN packets, the timestamp indicates the time at which the first sample of the packet was produced by the A/D converter(s) for that channel. On OUT packets, the timestamp specifies the time at which the first sample in the packet should go out the D/A converter(s) for that channel. If a packet reaches the head of the transmit queue, and the current time is later than the timestamp, an error is assumed to have occurred and the packet is discarded. As a special case, the timestamp 0xffffffff is interpreted as "Now".

The time base is a free running 32-bit counter that is incremented by the A/D sample-clock.

#### 4.8.2. Lack of USRP2 transceiver slots

To reduce crosstalk between the RX and TX path of the transceiver boards, simply two transceivers are used – one for RX chain and one for TX. Daughterboard slot A serves as transmitter, slot B as receiver. Because of USRP2 offers only one transceiver slot, crosstalk with standard RFX daughterboard will raise a challenge.

Some information about transceiver handling:

<http://gnuradio.org/redmine/wiki/gnuradio/OpenBTSDesktopTestingKit>

#### 4.8.3. Different sample rates

USRP runs at 64 MHz internal clock, USRP2 on 100 MHz. Because GSM clocks are derived from a clock running at 13 MHz, resampling is already done in OpenBTS. By use of an external 52 MHz clock (multiple of 13) this difference between USRP and USRP2 won't be an issue.



#### 4.8.4. Clocking issues

Because in Europe the GSM 1900 band isn't in use by operators, this might be a nice "test frequency" in a lab setup over the air interface. But in this frequency range, problems with the stability of USRPs' on board oscillator occur.

A very informative article about this issue written by David Burgess can be found at

<http://gnuradio.org/redmine/wiki/gnuradio/OpenBTSClocks>

Also the solution in re-clocking USRP to 52 MHz with an external oscillator can be found at

<http://gnuradio.org/redmine/wiki/gnuradio/OpenBTSClockModifications>

#### 4.9. OpenBTS I & Q data

Details of data handling between OpenBTS and USRP are found in *USRPDevice.cpp* in the *Transceiver* directory. As usual, GNU Radio is used as interface to USRP, by sending data in 32-bit words. The data word is separated in 16-bit I and 16-bit Q data.

The data stream provides also information about start and end of burst as well as the timestamp they have to be sent. The data are fragmented into 504-byte blocks with 8 bytes appended to achieve the common 512-byte USB packet size. The appended 8 bytes consist of the in-band signaling flags, the payload length and the timestamp (see Figure 32). If the payload length is smaller than 504 bytes, also shorter data packets may be sent provided that the payload length is also adjusted.

Figure 33 shows some bursts written to USRP while no mobile is attached. The period of approx. 231 samples stems from the guard interval of a typical GSM timeslot.

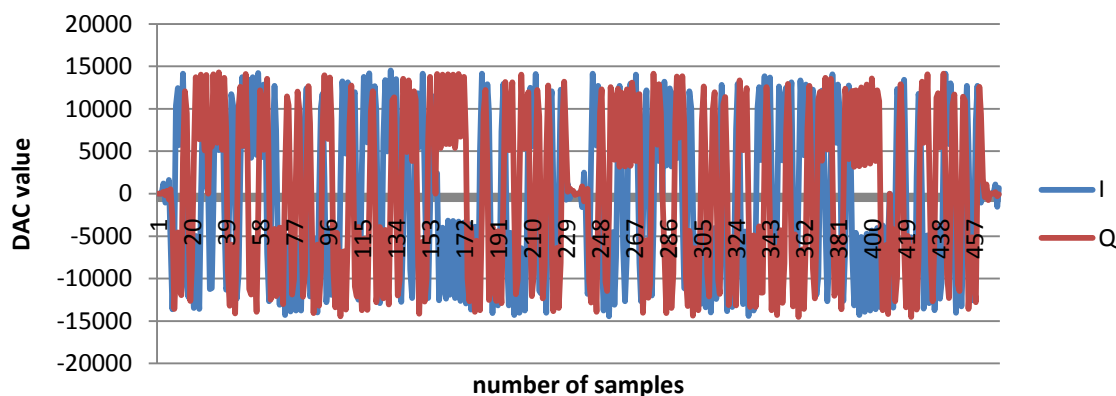


Figure 33: OpenBTS data bursts in Tx path





To understand how data rates in OpenBTS are calculated, a short excursion to GSM standard and OpenBTS source code has to be done. GSM bases on a 13 MHz master clock and provides a symbol rate of 270.833 kbit/s (clock decimation of 48) calculated as follows:

$$\frac{\textit{TDMA multiframe period}}{\textit{TDMA frames in one multiframe} * \textit{slots per frame}} = \textit{Duration of one TDMA time slot}$$

$$\frac{120 \textit{ ms}}{26 * 8} = 577 \textit{ }\mu\textit{s}$$

$$\frac{\textit{bits per TDMA timeslot}}{\textit{duration of one TDMA timeslot}} = \textit{GSM data rate} \quad \frac{156.25 \textit{ bit}}{577 \textit{ }\mu\textit{s}} = 270.833 \textit{ kbit/s}$$

Since the USRP DACs can only interpolate with integer factors to provide their sample rate of 128 MS/s (see Table 1), it's impossible to result in 13 MHz or any integer divider of it. This is why OpenBTS uses a resampler algorithm with factors 65 and 96 (defined in `radioInterface.h`) to result in an appropriate USRP sample rate of 400 kS/s, which can be achieved by a USRP decimation factor of 320.

$$\frac{270.833 \textit{ kS/s}}{65/96} = 400 \textit{ kS/s} * 320 = 128 \textit{ MS/s}$$

The resampling is done in `radioInterface.cpp`, calculation of interpolation factor 320 is done in `USRPDevice.cpp`, defined as `decimRate*2`. The desired USRP sample rate of 400kS/s is assigned in `runTransceiver.cpp`.



Having understood this context, the OpenBTS data rate with a burst periodicity of approx. 231 samples (see Figure 33) can be calculated as follows:

$$231 * \frac{65}{96} = 577 \mu s \rightarrow 1 \text{ GSM TDMA time slot}$$

$$\frac{128e6 \text{ S/s}}{320 * \frac{65}{96}} = 270.833 \frac{\text{kBit}}{\text{s}} \rightarrow \text{GSM symbol rate}$$

This resampling issue can be solved, if the USRP is relocked to 52 MHz, since this is an integer multiple of 13 MHz GSM master clock, see Transceiver52M in chapter 4.4. While doing this, some changes in the USRP firmware as well as in OpenBTS.config have to be done.

Further information provided under:

<http://www.gnuradio.org/redmine/wiki/gnuradio/OpenBTSClockModifications>

The data bursts as shown in Figure 33 are generated by writing the USRP I/Q samples into a csv-file (done by the modified `USRPDevice.cpp` below) and visualized through a spreadsheet program.

After having changed the file `USRPDevice.cpp` it's necessary to rerun `make` in the Transceiver directory.

`OPENBTSROOT/Transceiver/USRPDevice.cpp`

```
int USRPDevice::writeSamples(short *buf, int len, bool *underrun,
                             unsigned long long timestamp,
                             bool isControl)
{
#ifdef SWLOOPBACK
    if (!m_uTx) return 0;
    ...
    unsigned isEnd = (numPkts < 2);
    uint32_t *outPkt = new uint32_t[128*numPkts];
    int pktNum = 0;

// USRP-HACK
FILE * pFile;
pFile = fopen ("outfile.csv","a");
// /USRP-HACK
```




---

```

while (numWritten < len) {
    // pkt is pointer to start of a USB packet
    ...
    timestamp += payloadLen/2/sizeof(short);
    isStart = 0;

    pktNum++;

// USRP-HACK
for (int i=2;i<=(payloadLen/4)+2;i++) {
    fprintf (pFile, "%i,%i\n", (int16_t) (pkt[i] & 0xffff), (int16_t) (pkt[i] >> 16));
}
// /USRP-HACK
    pkt[0] = host_to_usrp_u32(pkt[0]);
    pkt[1] = host_to_usrp_u32(pkt[1]);

}
m_uTx->write((const void*) outPkt, sizeof(uint32_t)*128*numPkts, NULL);
delete[] outPkt;

// USRP-HACK
fclose (pFile);
// /USRP-HACK

    samplesWritten += len/2/sizeof(short);
    return len/2/sizeof(short);
#else
    ...

FILE * pFile;
pFile = fopen("anyfile.csv", "a");

for (int i=2; i<=(payloadLen/4)+2;i++) {
    fprintf(pFile, "%i,%i\n", (int16_t) (pkt[i] & 0xffff), (int16_t) (pkt[i] >>
16));
}

fclose(pFile);

```



## 5. Conclusion

The given text deals with the wide application- and research-oriented field of Software-Defined Radio (SDR). With the open source project GNU Radio and the hardware platform USRP/2, very complex wireless transmission systems can be explored, even with a relatively small budget. Principally all of the needed modules and information can be found in the internet. It is the credit of this text to bring these widespread information together.

This work integrates many aspects of SDR projects based on GNU Radio and USRP/2. Starting with installation guides and scripts for Linux, GNU Radio and GRC, first receiver implementations like FM and FM RDS are run successfully. Specific aspects, like synchronization issues at the RF frontend and the sound card, details of the data transfer at the USB and Gigabit Ethernet interface and first portation issues of the OpenBTS project to USRP2 are addressed. Highlight of this thesis is the successful operation of a complete OpenBTS GSM system. For a better understanding, all achieved goals and aspects are presented in Figure 34.

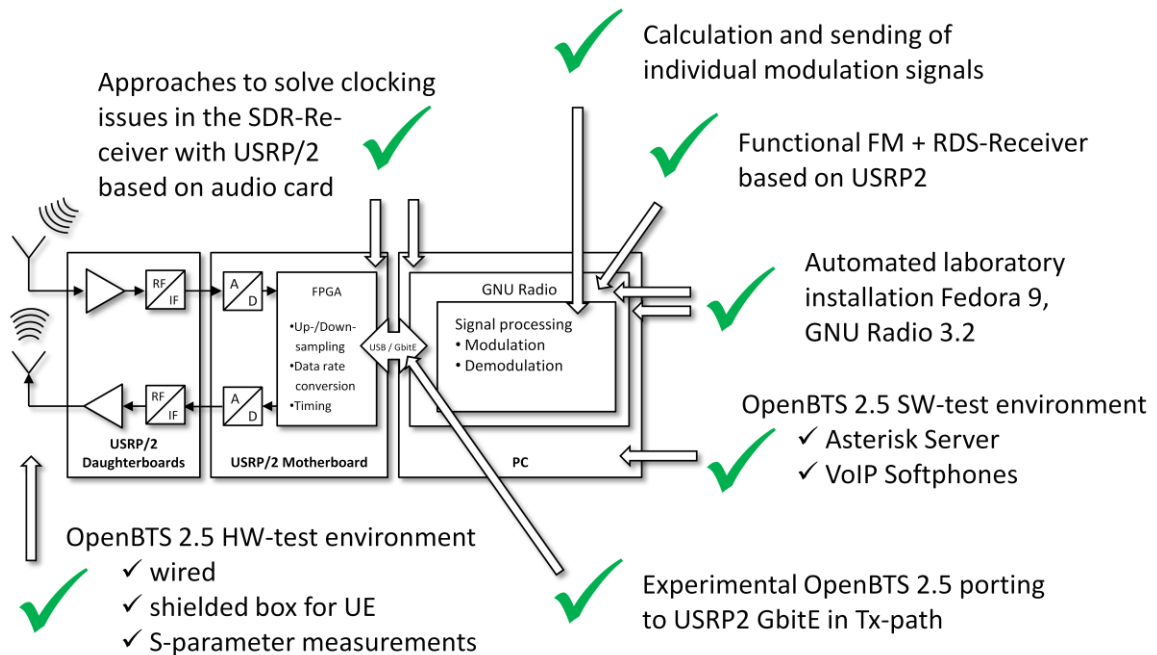


Figure 34: Results of the thesis



Every finished project puts new issues to work on in the future. They are illustrated in Figure 35.

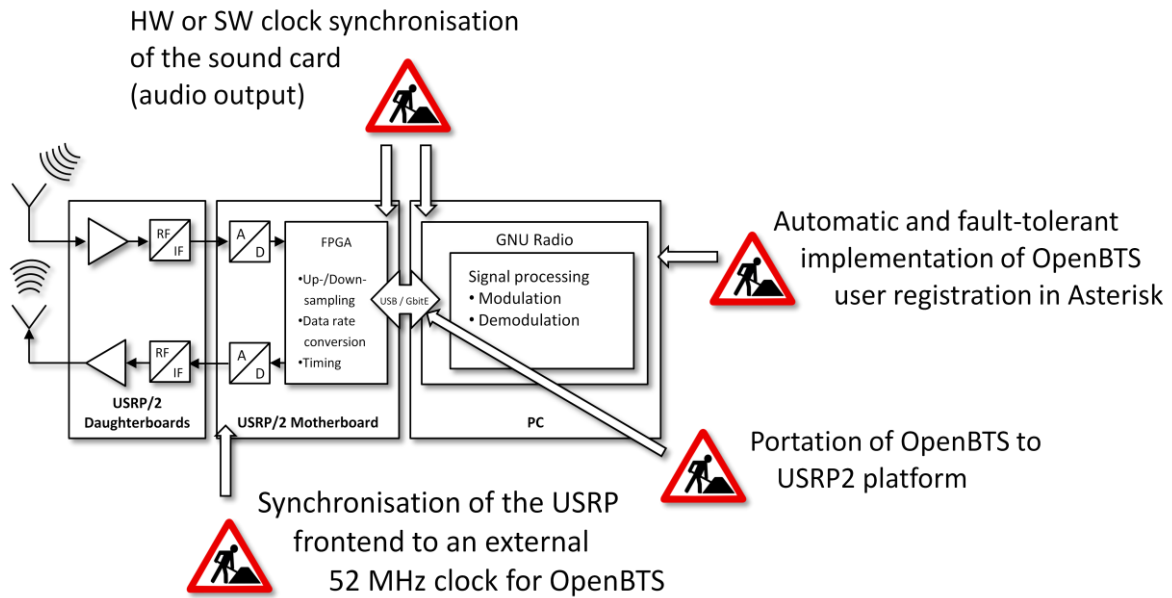


Figure 35: Lookout of the thesis

The author is grateful for all code and documentation found in the internet and created by many anonymous contributors. He is particularly indebted to George Nychis, David Burgess and Jonathan Corgan for very valuable mail contributions.



## List of literature

1. **Sklar, Bernard.** *Digital Communications: Fundamentals and Applications*. s.l. : Prentice Hall, 2001.
2. **Blossom, Eric.** *Exploring GNU Radio*. [Online] [Cited: 12 10, 2009.] <http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>.
3. **Ettus, Matt (et al.).** *USRP2GenFAQ*. [Online] [Cited: 01 20, 2010.] <http://www.gnuradio.org/redmine/wiki/gnuradio/USRP2GenFAQ>.
4. **Abbas, Firas (et al.).** *GNU Radio - UsrcFAQDBoards*. [Online] [Cited: 01 11, 2010.] <http://www.gnuradio.org/redmine/wiki/gnuradio/UsrcFAQDBoards>.
5. **Ettus, Matt (et al.).** *USRP2UserFAQ*. [Online] [Cited: 01 20, 2010.] <http://www.gnuradio.org/redmine/wiki/gnuradio/USRP2UserFAQ>.
6. —. *UsrcFAQGen*. [Online] [Cited: 01 20, 2010.] <http://www.gnuradio.org/redmine/wiki/gnuradio/UsrcFAQGen>.
7. *Biphase mark code*. [Online] [Cited: 01 10, 2010.] [http://en.wikipedia.org/wiki/Biphase\\_mark\\_code](http://en.wikipedia.org/wiki/Biphase_mark_code).
8. **Engdahl, Tomi.** *S/PDIF Interface*. [Online] [Cited: 01 20, 2010.] <http://www.epanorama.net/documents/audio/spdif.html>.
9. **RME (german).** *Word clock module*. [Online] [Cited: 01 10, 2010.] [http://www.rme-audio.de/download/9632wcm\\_d.pdf](http://www.rme-audio.de/download/9632wcm_d.pdf).
10. **Symeonidis, Dimitrios.** *RDS*. [Online] [Cited: 01 11, 2010.] [https://www.cgran.org/attachment/wiki/RDS/rds\\_rx.png](https://www.cgran.org/attachment/wiki/RDS/rds_rx.png).
11. **Blossom, Eric.** *Basic TX first Nyquist zone frequencies*. [Online] [Cited: 01 10, 2010.] <http://www.mail-archive.com/discuss-gnuradio@gnu.org/msg18077.html>.
12. **Kestrel Signal Processing, Inc.** *The OpenBTS Project*. [Online] [Cited: 01 10, 2010.] <http://www.openbts.sourceforge.net>.
13. **GSM Security.** *What are Ki, Kc, RAND and SRES*. [Online] [Cited: 01 10, 2010.] <http://www.gsm-security.net/faq/gsm-ki-kc-rand-sres.shtml>.
14. **Burgess, David.** *OpenBTSSettingUpAsterisk*. [Online] [Cited: 01 10, 2010.] <http://gnuradio.org/redmine/wiki/gnuradio/OpenBTSSettingUpAsterisk>.



15. **Nychis, George.** *Enabling MAC Protocol Implementations on Software-Defined Radios.* [Online] [http://www.andrew.cmu.edu/user/gnychis/nychis\\_nsdi09.pdf](http://www.andrew.cmu.edu/user/gnychis/nychis_nsdi09.pdf).
16. **EBU and RDS Forum.** *RDS Standards.* [Online] [Cited: 01 10, 2010.] [http://www.rds.org.uk/rds98/rds\\_standards.htm](http://www.rds.org.uk/rds98/rds_standards.htm).
17. **3GPP.** [Online] [Cited: 01 10, 2010.] [http://www.3gpp.org/ftp/Specs/archive/45\\_series/45.005/45005-910.zip](http://www.3gpp.org/ftp/Specs/archive/45_series/45.005/45005-910.zip).
18. *Configuring IP Phones for use with Asterisk.* [Online] [Cited: 01 10, 2010.] [http://www.asteriskguru.com/tutorials/asterisk\\_voip\\_ipphone.html](http://www.asteriskguru.com/tutorials/asterisk_voip_ipphone.html).

## List of figures

Figure 1: Block diagram of a typical digital communication system [1] .....	2
Figure 2: Software-Defined Radio block diagram .....	8
Figure 3: USRP .....	10
Figure 4: USRP2 .....	10
Figure 5: USRP/2 SDR Block.....	10
Figure 6: Digital Down Converter Block Diagram (according to [2]).....	11
Figure 7: RXF900.....	12
Figure 8: TVRX .....	12
Figure 9: GRC FM receiver example usrp_wb_fm_receive.grc adapted for USRP2 .....	15
Figure 10: USRP2 block properties.....	17
Figure 11: GRC block “WBFM Receive” with opened properties window.....	19
Figure 12: Signal path from FM VHF to audio sink.....	25
Figure 13: Biphas mark code .....	28
Figure 14: Biphas mark code with all data bits set to zero .....	28
Figure 15: Block diagram for possible clock synchronization .....	29
Figure 16: Flow graph of RDS Receiver block gr-rds [10] .....	30
Figure 17: SDR block for FM signal processing.....	35
Figure 18: Illustration of csv data after byte swapping.....	36
Figure 19: CSV visualization of traced samples.....	37




---

Figure 20: Principle of I & Q values for a sine function between 90° and 270° .....	39
Figure 21: Calculated sine and cosine functions .....	40
Figure 22: Byte order of sine_out_2.bin (little-endian) .....	42
Figure 23: Platform independent byte order of sine_out_2.bin (little-endian) .....	44
Figure 24: summarized GSM block diagram .....	46
Figure 25: Functional principle of SDR with OpenBTS .....	48
Figure 26: Block diagram of OpenBTS laboratory setup .....	49
Figure 27: RF screened case .....	50
Figure 28: Opened case with UE and antenna inside .....	50
Figure 29: S-parameters of a 2 port device (DUT = device under test) .....	51
Figure 30: S-parameter measurement setup .....	52
Figure 31: Decoupling of the circulator from port 1 to port 3 ( $S_{31}$ ) .....	52
Figure 32: Data structure of an in-band signaling block from \$GNURADIOROOT/usrp/doc/inband-signaling-usb .....	61
Figure 33: OpenBTS data bursts in Tx path .....	63
Figure 34: Results of the thesis .....	67
Figure 35: Lookout of the thesis.....	68

---





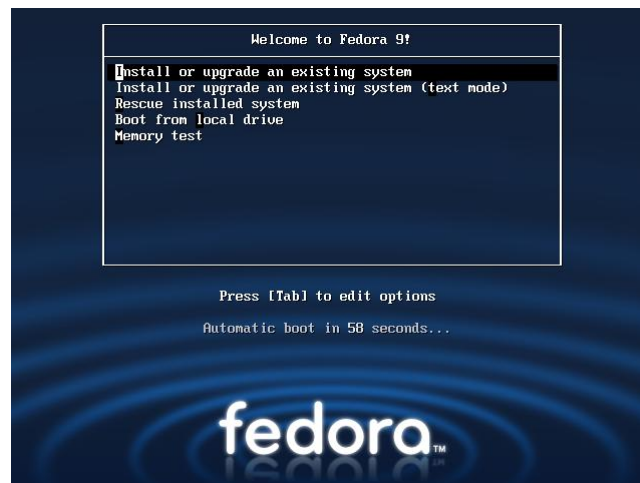
## 6. Annex

### 6.1. Annex A: Installation guide for Fedora 9 with GNU Radio 3.2.2 and USRP2

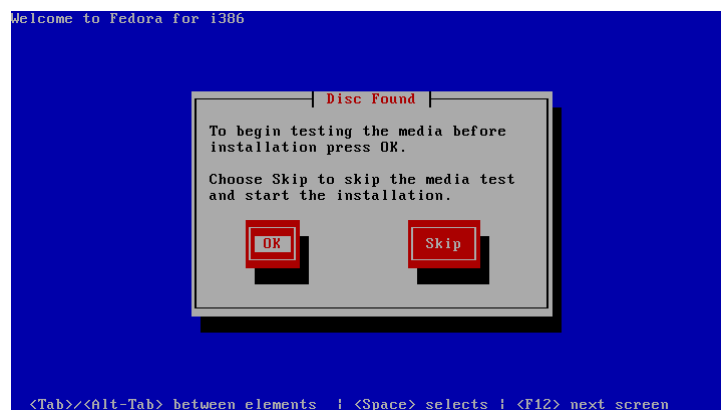
Requirements: Fedora 9 DVD-Image

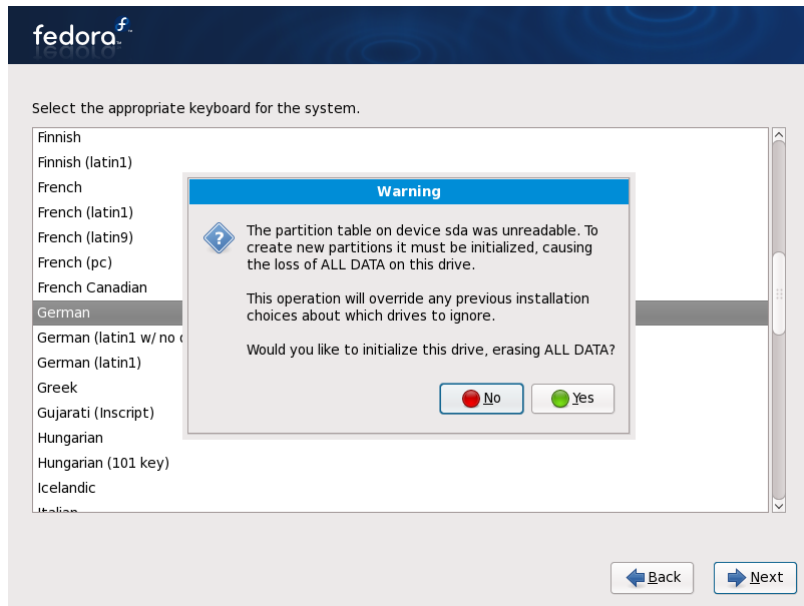
<http://download.fedoraproject.org/pub/fedora/linux/releases/9/Fedora/i386/iso/Fedora-9-i386-DVD.iso>

Below, a short installation how-to for Fedora 9, GNU Radio 3.2.2 and USRP2 is given. Fedora 9 was chosen as Linux distribution for our GNU Radio/USRP2 environment, so the following installation steps are based thereon.



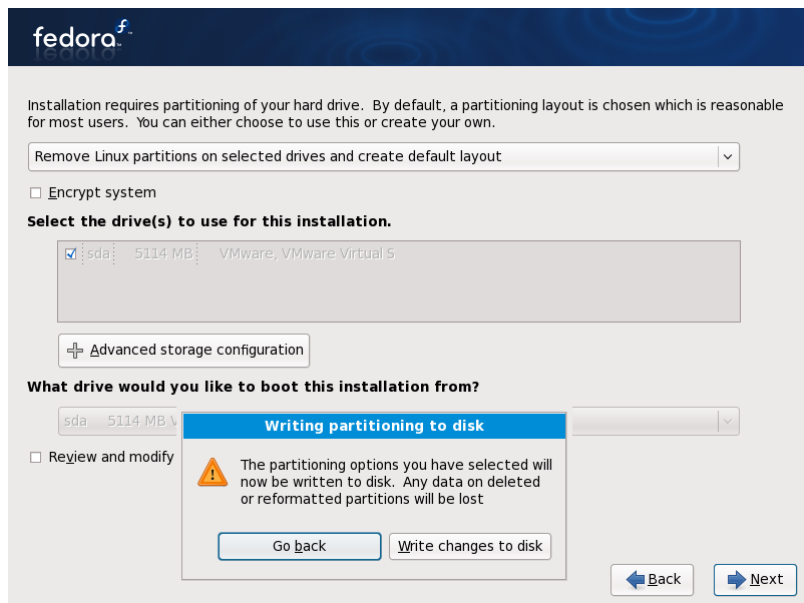
First you have boot from Fedora 9 DVD and choose “Install or upgrade existing system“. The following question for testing your media may be skipped...





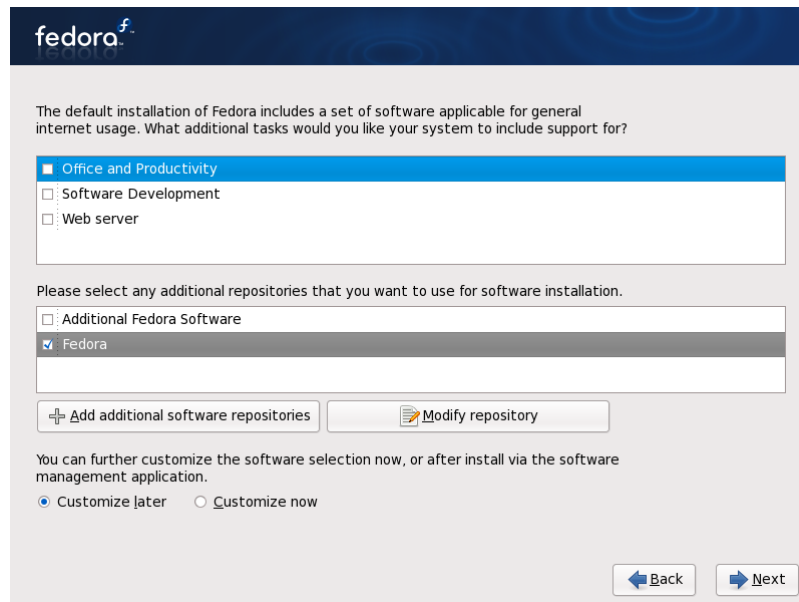
After selecting system language and keyboard layout you are asked for confirmation to overwrite all data, if you're going to install Fedora on a clean hard drive.

Afterwards the time zone and root password has to be set. In the following dialog we leave the partitioning of the hard drive to the fedora installer and confirm the choice with "Write changes to disk". You may choose an advanced partitioning of your hard drive if you know what you're doing.

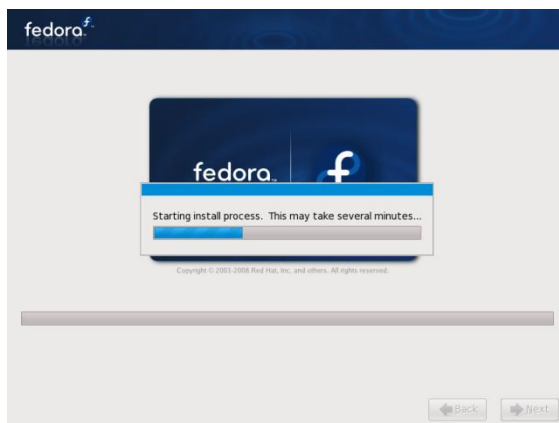




Having written the partition table, you're asked for additional software to be installed. We uncheck "Office and Productivity" since we're going to install all we need later on.



Now you have to wait until Fedora has been installed and finally reboot your pc...

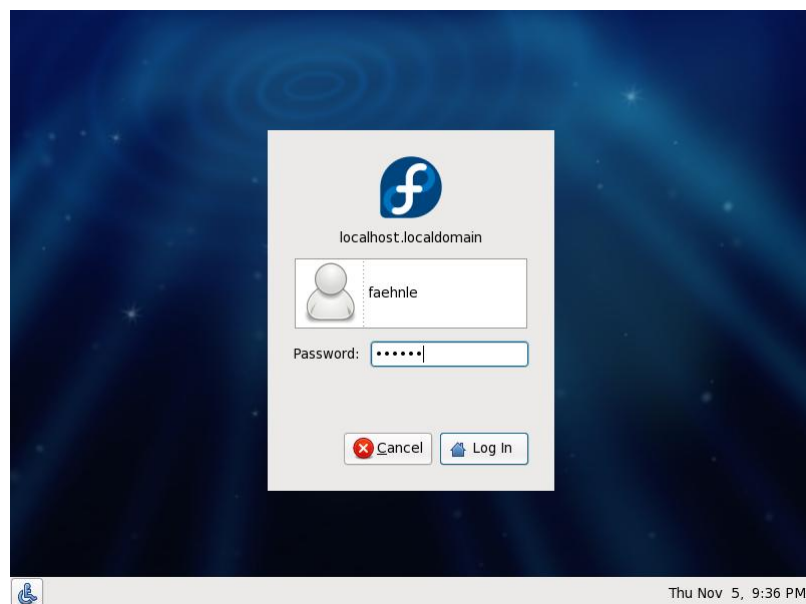




After rebooting, some last steps have to be done. You have to agree to the license information, create a standard user profile for non root privileged work, set your local date and time and finally decide if you want to send your hardware information to fedora project.

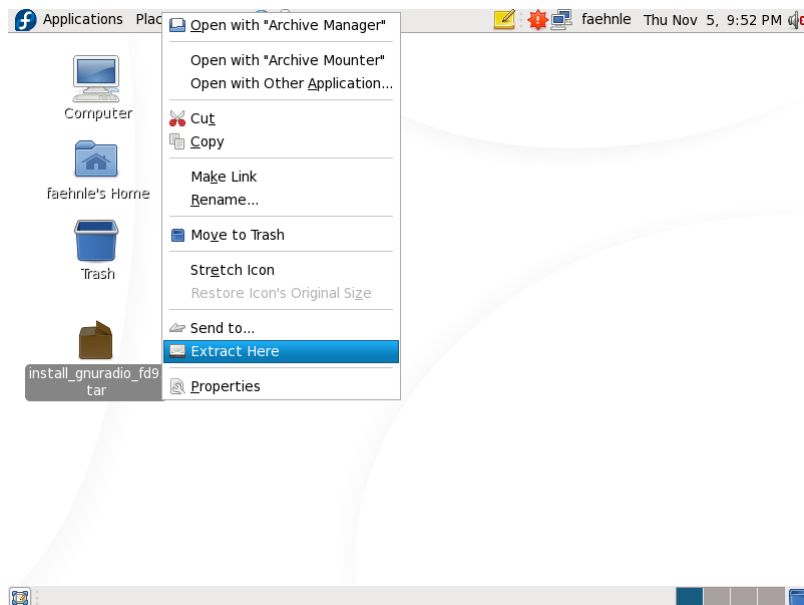


Passed another reboot, your system starts the first time with your login screen - so log in with your created user.

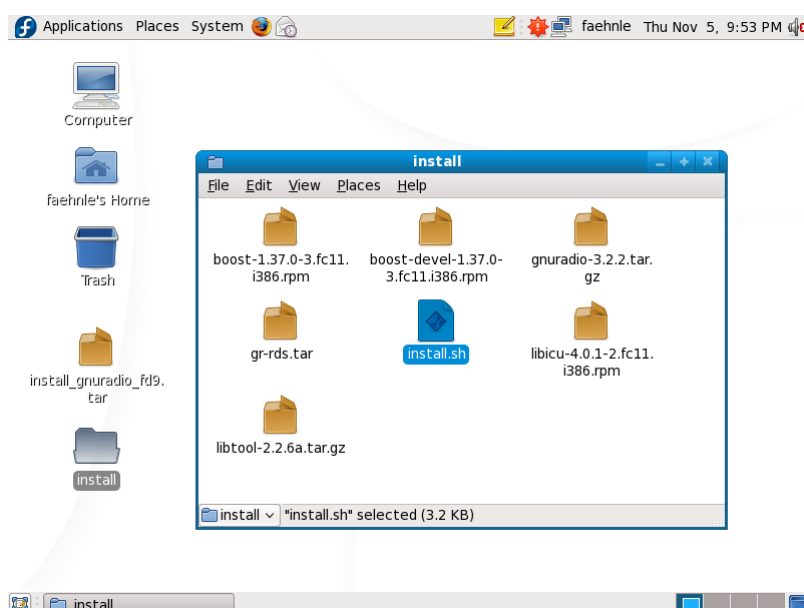




Now you have a real clean Fedora 9 installation. In the next step we go on with the file `install_gnuradio_fd9.tar` extracted on desktop.

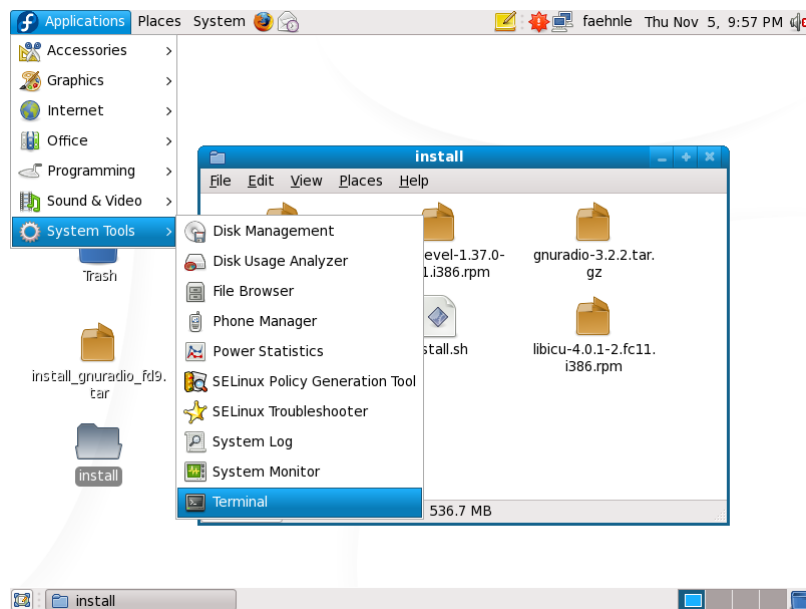


This archive includes all dependencies necessary to configure and make GNU Radio 3.2.2 with USRP2 (`grc`, `python`) support on THAT Fedora installation. Don't use the provided install script `install.sh` for another Linux distribution without checking dependencies!

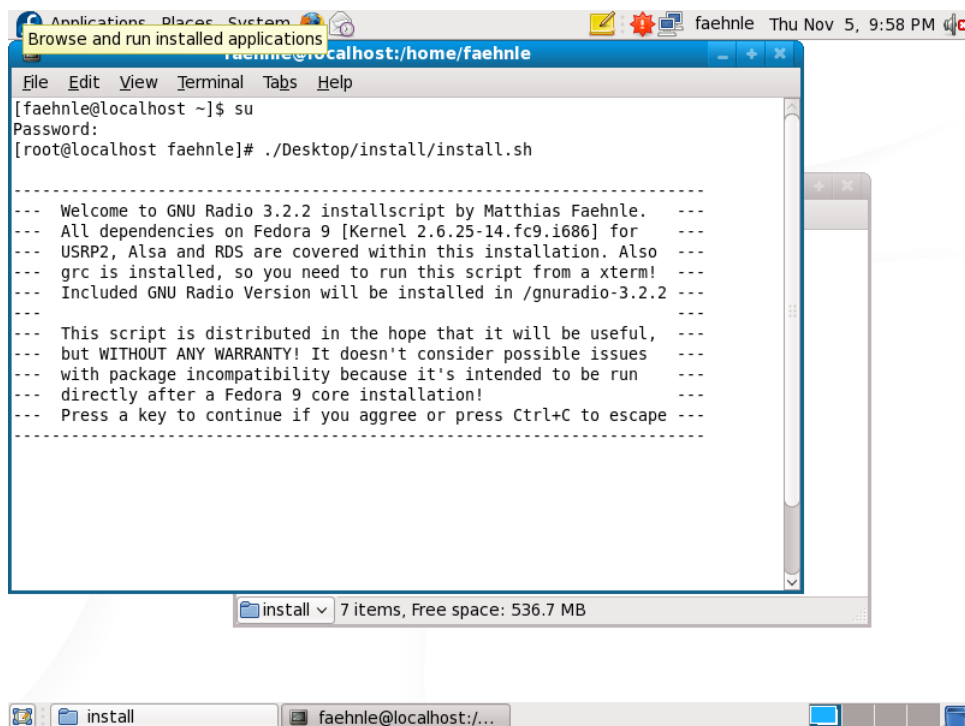




Since the provided `install.sh` script needs root privileges for installing and making packets, you need to run in out of an xterm as root. Open the Terminal as shown below.



In the terminal you have to gain root privileges by typing “`su`” followed by your created root password during Fedora installation. Having gathered root privileges, the `install.sh` needs to be run by typing “`./Desktop/install/install.sh`” if you have extracted the tar archive to your users desktop.

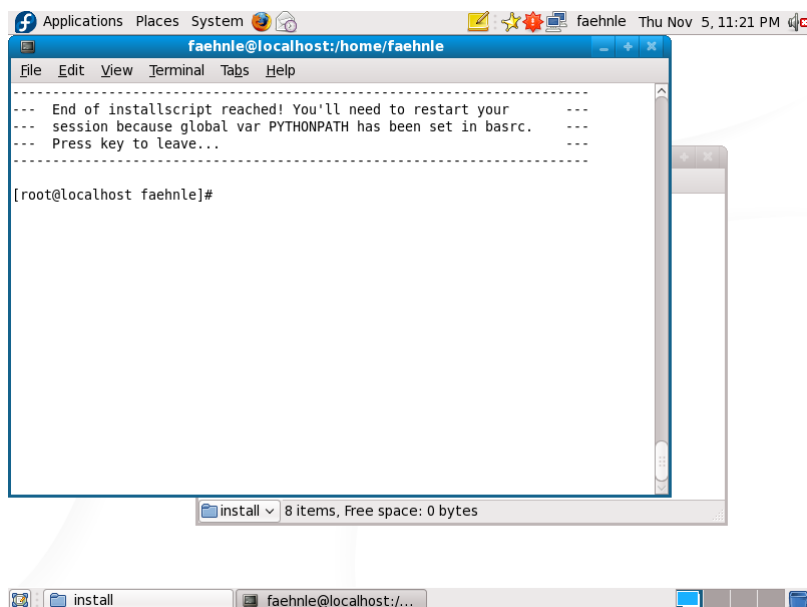


```
faehnle@localhost:~/home/faehnle
File Edit View Terminal Tabs Help
[faehnle@localhost ~]$ su
Password:
[root@localhost faehnle]# ./Desktop/install/install.sh

-----
--- Welcome to GNU Radio 3.2.2 installscrip by Matthias Faehnle. ---
--- All dependencies on Fedora 9 [Kernel 2.6.25-14.fc9.i686] for ---
--- USRP2, Alsa and RDS are covered within this installation. Also ---
--- grc is installed, so you need to run this script from a xterm! ---
--- Included GNU Radio Version will be installed in /gnuradio-3.2.2 ---
-----
--- This script is distributed in the hope that it will be useful, ---
--- but WITHOUT ANY WARRANTY! It doesn't consider possible issues ---
--- with package incompatibility because it's intended to be run ---
--- directly after a Fedora 9 core installation! ---
--- Press a key to continue if you aggree or press Ctrl+C to escape ---
-----

install 7 items, Free space: 536.7 MB
```

Having understood and agreed to the shown banner, you can start the installation by pressing a key. If the installation has finished, you should restart your system because of some environment variables were set during the installation, which are not correctly set until a new logon.



```
faehnle@localhost:~/home/faehnle
File Edit View Terminal Tabs Help
-----
--- End of installscrip reached! You'll need to restart your ---
--- session because global var PYTHONPATH has been set in basrc. ---
--- Press key to leave... ---
-----

[root@localhost faehnle]#

install 8 items, Free space: 0 bytes
```



## 6.2. Annex B: Automated installation script for GNU Radio 3.2.2, USRP2, ALSA and RDS on Fedora 9

Starting by a so called “shebang” or “sha-bang” at the head of a script, you can tell your system a set of commands, which will be fed to the command interpreter indicated. Here this is just a shell-script need to run in `/bin/sh`.

```
#!/bin/sh
```

After this, some information about the script and author are provided that you just see, if your editing the file. There’s also a note that this script is provided under GPL License.

```
# Copyright 2009 Matthias Faehnle, University of Applied Sciences, Ulm, Germany [ff]
```

At first the screen is cleared now and a welcome message is printed out in the shell that contains information about the requirements and what is done by running the script.

```
clear
echo "
```

```
-----
--- Welcome to GNU Radio 3.2.2 install script by Matthias Faehnle. ---
--- All dependencies on Fedora 9 [Kernel 2.6.25-14.fc9.i686] for --- [ff]
```

Also there is information, that you can abort this script by pressing Ctrl+C (like you can do it with every script that doesn’t contain a signal handler) or go on by pressing any other key, what is done by the read-command.

```
--- Press a key to continue if you agree or press Ctrl+C to escape ---
-----"
```

```
read -n 1
```

Now we change the current directory to the relative path, where the script is run from. After this operation we can set the variable `SCRIPTDIR` to the absolute path from `/` and access later to it from wherever the script is run from and in which directory we currently are.

```
cd $(dirname $0)
SCRIPTDIR=$(pwd)
```

Because we need root privileges to for installation of packets, the current user name is read out and checked if it’s equal to “root”. If not, the script aborts with the shown error.





```
WHO=$(whoami)
```

```
if [ "$WHO" != "root" ]; then
    echo "You are logged in as $WHO, please log in as root - script stopped"
    exit
fi
```

It is also checked if the expected Fedora 9 Kernel is used to prevent dependency problems. If the estimated Kernel isn't found, the following notice will be print to screen.

```
if [ ! -f /boot/vmlinuz-2.6.25-14.fc.i686 ]; then
    echo "Your distribution doesn't seem to be Fedora 9 [Kernel 2.6.25-14.fc9.i686] -
    continue by pressing a key or press Ctrl+C to escape!"
fi
```

Now the installation starts by calling the Fedora packet manager yum with the required packets. Parameter "y" causes yum not to ask for each packet to be installed. These packages need to be installed to meet requirements for GNU Radio 3.3.2, Alsa sound, and the gr-rds block that later will be installed.

```
### requirements gnuradio-3.2.2 [gnuradio.org] + alsa + gr-rds [www.cgran.org]
yum -y install gcc gcc-c++ fftw-devel cppunit-devel wxPython-devel alsa-lib-devel python-devel
swig pygsl gsl-devel SDL-devel python-cheetah python-lxml pygtk2-devel PyOpenGL libxml2-devel
```

Since we need some newer packages than they are provided by Fedora 9, some packets are built in the install.tar und now installed to the system.

```
### install boost > 1.35 from rpms
rpm -i $SCRIPTDIR/libicu-4.0.1-2.fc11.i386.rpm
rpm -i $SCRIPTDIR/boost-1.37.0-3.fc11.i386.rpm
rpm -i $SCRIPTDIR/boost-devel-1.37.0-3.fc11.i386.rpm
```

Now we can start with the execution and installation of GNU Radio. It is installed by this script in /gnuradio-3.2.2. After unpacking with tar, the directory is changed to /gnuradio-3.2.2 and a "configure" is performed. The given parameters tell GNU Radio to enable the correspondent functions und to make the necessary libraries. After the configuration a "make" and "make install" need to be executed to start the makefiles doing their compilation work.



```

### unpack gnuradio to /gnuradio-3.2.2 and install
cd /
GNUSRC=$SCRIPTDIR/gnuradio-3.2.2.tar.gz
tar -xf $GNUSRC
GNUDIR=/gnuradio-3.2.2
cd $GNUDIR

./configure --enable-gnuradio-core --enable-usrp2 --enable-gr-audio-alsa --enable-grc --
enable-gr-wxgui
make
make install

```

Having installed GNU Radio, we start preparing the dependencies for the RDS block provided from <http://www.cgran.org>. Because this block was made with a newer libtool version, we have to compile libtool-2.X in a folder – here unpacked to the SCRIPTDIR.

```

### prepare libtool > 1.5 for RDS
cd $SCRIPTDIR
tar -xf libtool-2.2.6a.tar.gz
cd libtool-2.2.6

./configure
make
make install

```

After the libtool is renewed to an appropriate version, the gr-rds block is extracted to the \$GNURADIOROOT/gr-rds.

```

### unpack and install gr-rds block
cd $GNUDIR
RDSSRC=$SCRIPTDIR/gr-rds.tar
tar -xf $RDSSRC
RDSDIR=$GNUDIR/gr-rds
cd $RDSDIR

```

Because the gr-rds installer searches for libxml2-devel library header files in /usr/include/libxml/, we have to check if the later on needed file “parser.h” exists in there. Since by default in Fedora 9 the file is found in in /usr/include/libxml2/libxml, a symbolic link is created if the file doesn’t exist in the expected location.

```

if [ ! -f /usr/include/libxml/parser.h ]; then
    ln -s /usr/include/libxml2/libxml /usr/include/libxml
fi

```



Also the symbolic link ltmain.sh is tested and replaced because a different path is used in the default configuration.

```
if [ ! -f ltmain.sh ]; then
    yes|rm ltmain.sh && ln -s /usr/local/share/libtool/config/ltmain.sh
fi
```

Having checked all dependencies for gr-rds, the block is installed as known.

```
./configure
make
make install
```

For several reasons (e.g. GRC), we need to set the \$PYTHONPATH environment variable in the shell to the python package path. This is done automatically for every user on time you log in, if the export command is written to /etc/bashrc. The following command prints the export command just at the very end of bashrc.

```
### write envvar PYTHONPATH to bashrc
echo "export PYTHONPATH=/usr/lib/python2.5/site-packages" >> /etc/bashrc
```

At last, the script waits for a input key and prints out information that the installation has finished and you have to restart your session for the env var \$PYTHONPATH to be set.

```
echo "
-----
--- End of installsript reached! You'll need to restart your ---
--- session because global var PYTHONPATH has been set in basrc. ---
--- Press key to leave... ---
-----"
read -n 1
```

### 6.3. Annex C: Install.sh

```
#!/bin/sh
#
# Copyright 2009 Matthias Faehnle, University of Applied Sciences, Ulm, Germany
#
# Install script for GNU Radio 3.2.2, USRP2 and RDS on Fedora 9 [Kernel 2.6.25-14.fc9.i686]
#
# This script 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.

clear
echo "
-----
--- Welcome to GNU Radio 3.2.2 install script by Matthias Faehnle. ---
--- All dependencies on Fedora 9 [Kernel 2.6.25-14.fc9.i686] for ---
--- USRP2, Alsa and RDS are covered within this installation. Also ---
--- grc is installed, so you need to run this script from a xterm! ---
--- Included GNU Radio Version will be installed in /gnuradio-3.2.2 ---
---
--- This script is distributed in the hope that it will be useful, ---
--- but WITHOUT ANY WARRANTY! It doesn't consider possible issues ---
--- with package incompatibility because it's intended to be run ---
--- directly after a Fedora 9 core installation! ---
--- Press a key to continue if you agree or press Ctrl+C to escape ---
-----"

read -n 1

cd $(dirname $0)
SCRIPTDIR=$(pwd)

WHO=$(whoami)

if [ "$WHO" != "root" ]; then
    echo "You are logged in as $WHO, please log in as root - script stopped"
    exit
fi

if [ ! -f /boot/vmlinuz-2.6.25-14.fc.i686 ]; then
    echo "Your distribution doesn't seem to be Fedora 9 [Kernel 2.6.25-14.fc9.i686] -
continue by pressing a key or press Ctrl+C to escape!"
fi

### requirements gnuradio-3.2.2 [gnuradio.org] + alsa + gr-rds [www.cgran.org]
yum -y install gcc gcc-c++ fftw-devel cppunit-devel wxPython-devel alsa-lib-devel python-devel
swig pygsl gsl-devel SDL-devel python-cheetah python-lxml pygtk2-devel PyOpenGL libxml2-devel

### install boost > 1.35 from rpms
rpm -i $SCRIPTDIR/libicu-4.0.1-2.fc11.i386.rpm
rpm -i $SCRIPTDIR/boost-1.37.0-3.fc11.i386.rpm
rpm -i $SCRIPTDIR/boost-devel-1.37.0-3.fc11.i386.rpm

```



```

### unpack gnuradio to /gnuradio-3.2.2 and install
cd /
GNUSRC=$SCRIPTDIR/gnuradio-3.2.2.tar.gz
tar -xf $GNUSRC
GNUDIR=/gnuradio-3.2.2
cd $GNUDIR

./configure --enable-gnuradio-core --enable-usrp2 --enable-gr-audio-alsa --enable-grc --
enable-gr-wxgui
make
make install

### prepare libtool > 1.5 for RDS
cd $SCRIPTDIR
tar -xf libtool-2.2.6a.tar.gz
cd libtool-2.2.6

./configure
make
make install

### unpack and install gr-rds block
cd $GNUDIR
RDSSRC=$SCRIPTDIR/gr-rds.tar
tar -xf $RDSSRC
RDSDIR=$GNUDIR/gr-rds
cd $RDSDIR
if [ ! -f /usr/include/libxml/parser.h ]; then
    ln -s /usr/include/libxml2/libxml /usr/include/libxml
fi
if [ ! -f ltmain.sh ]; then
    yes|rm ltmain.sh && ln -s /usr/local/share/libtool/config/ltmain.sh
fi

./configure
make
make install

### write envvar PYTHONPATH to bashrc
echo "export PYTHONPATH=/usr/lib/python2.5/site-packages" >> /etc/bashrc

echo "
-----
--- End of installscript reached! You'll need to restart your ---
--- session because global var PYTHONPATH has been set in basrc. ---
--- Press key to leave... ---
-----"
read -n 1

```



## 6.4. Annex D: RDS receiver for USRP2

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

from gnuradio import gr, usrp2, optfir, blks2, rds, audio
from gnuradio.eng_option import eng_option
from gnuradio.wxgui import slider, form, stdgui2, fftsink2, scopesink2, constsink_gl
from optparse import OptionParser
from rdspanel import rdsPanel
#from usrpm import usrp_dbid
import sys, math, wx, time

class rds_rx_graph (stdgui2.std_top_block):
    def __init__(self, frame, panel, vbox, argv):
        stdgui2.std_top_block.__init__(self, frame, panel, vbox, argv)

        parser=OptionParser(option_class=eng_option)
        parser.add_option("-R", "--rx-subdev-spec", type="subdev", default=None,
            help="select USRP Rx side A or B (default=A)")
        parser.add_option("-f", "--freq", type="eng_float", default=101.8e6,
            help="set frequency to FREQ", metavar="FREQ")
        parser.add_option("-g", "--gain", type="eng_float", default=None,
            help="set gain in dB")
        parser.add_option("-s", "--squelch", type="eng_float", default=0,
            help="set squelch level (default is 0)")
        parser.add_option("-V", "--volume", type="eng_float", default=None,
            help="set volume (default is midpoint)")
        parser.add_option("-O", "--audio-output", type="string", default="plughw:0,0",
            help="pcm device name (default is plughw:0,0)")

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

        self.frame = frame
        self.panel = panel

        # connect to USRP
        #
        usrp_decim = 250
        usrp_decim = 416
        self.u = usrp.source_c(0, usrp_decim)
        self.u = usrp2.source_32fc("eth1")
        #
        print "USRP Serial: ", self.u.serial_number()
        print "USRP MAC: ", self.u.mac_addr()
        #
        adc_rate = self.u.adc_rate()           # 64 MS/s
        adc_rate = self.u.adc_rate()           # 100 MS/s
        #
        usrp_rate = adc_rate / usrp_decim      # 256 kS/s
        usrp_rate = adc_rate / usrp_decim      # ~240 kS/s
        #
        audio_decim = 8
        audio_decim = 5
        #
        audio_rate = usrp_rate / audio_decim   # 32 kS/s
        audio_rate = usrp_rate / audio_decim   # ~48k kS/s
        self.u.set_decim(usrp_decim)

        #
        if options.rx_subdev_spec is None:
        #
            options.rx_subdev_spec = usrp.pick_subdev(self.u,
                (usrp_dbid.TV_RX, usrp_dbid.TV_RX_REV_2, usrp_dbid.BASIC_RX))
        #
        self.u.set_mux(usrp.determine_rx_mux_value(self.u, options.rx_subdev_spec))
```



```

#         self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
#         print "Using d'board", self.subdev.side_and_name()

#         # gain, volume, frequency
#         self.gain = options.gain
#         if options.gain is None:
#             g = self.subdev.gain_range()
#             self.gain = g[1]
#
#         self.gain = -30

#         self.vol = options.volume
#         if self.vol is None:
#             g = self.volume_range()
#             self.vol = float(g[0]+g[1])/2

#         self.freq = options.freq
#         if abs(self.freq) < 1e6:
#             self.freq *= 1e6

#         # channel filter, wfm_rcv_pll
#         chan_filt_coefs = optfir.low_pass (1,
#                                           usrp_rate,
#                                           80e3,
#                                           115e3,
#                                           0.1,
#                                           60)
#         self.chan_filt = gr.fir_filter_ccf (1, chan_filt_coefs)
#         self.guts = blks2.wfm_rcv_pll (usrp_rate, audio_decim)
#         self.connect(self.u, self.chan_filt, self.guts)

#         # volume control, audio sink
#         self.volume_control_l = gr.multiply_const_ff(self.vol)
#         self.volume_control_r = gr.multiply_const_ff(self.vol)
#         self.audio_sink = audio.sink(int(audio_rate), options.audio_output, False)
#         self.connect ((self.guts, 0), self.volume_control_l, (self.audio_sink, 0))
#         self.connect ((self.guts, 1), self.volume_control_r, (self.audio_sink, 1))

#         # pilot channel filter (band-pass, 18.5-19.5kHz)
#         pilot_filter_coefs = gr.firdes.band_pass(1,
#                                                  usrp_rate,
#                                                  18.5e3,
#                                                  19.5e3,
#                                                  1e3,
#                                                  gr.firdes.WIN_HAMMING)
#         self.pilot_filter = gr.fir_filter_fff(1, pilot_filter_coefs)
#         self.connect(self.guts.fm_demod, self.pilot_filter)

#         # RDS channel filter (band-pass, 54-60kHz)
#         rds_filter_coefs = gr.firdes.band_pass (1,
#                                                usrp_rate,
#                                                54e3,
#                                                60e3,
#                                                3e3,
#                                                gr.firdes.WIN_HAMMING)
#         self.rds_filter = gr.fir_filter_fff (1, rds_filter_coefs)
#         self.connect(self.guts.fm_demod, self.rds_filter)

#         # create 57kHz subcarrier from 19kHz pilot, downconvert RDS channel
#         self.mixer = gr.multiply_ff()
#         self.connect(self.pilot_filter, (self.mixer, 0))
#         self.connect(self.pilot_filter, (self.mixer, 1))
#         self.connect(self.pilot_filter, (self.mixer, 2))
#         self.connect(self.rds_filter, (self.mixer, 3))

#         # low-pass the baseband RDS signal at 1.5kHz
#         rds_bb_filter_coefs = gr.firdes.low_pass (1,
#                                                  usrp_rate,
#                                                  1500,
#                                                  2e3,
#                                                  gr.firdes.WIN_HAMMING)
#         self.rds_bb_filter = gr.fir_filter_fff (1, rds_bb_filter_coefs)

```



```

self.connect(self.mixer, self.rds_bb_filter)

# 1187.5bps = 19kHz/16
self.rds_clock = rds.freq_divider(16)
#self.rds_clock = gr.fractional_interpolator_ff(0, 1/16.)
clock_taps = gr.firdes.low_pass (1, # gain
                                usrp_rate, # sampling rate
                                1.2e3, # passband cutoff
                                1.5e3, # transition width
                                gr.firdes.WIN_HANN)
self.clock_filter = gr.fir_filter_fff (1, clock_taps)
self.connect(self.pilot_filter, self.rds_clock, self.clock_filter)

# bpsk_demod, diff_decoder, rds_decoder
self.bpsk_demod = rds.bpsk_demod(usrp_rate)
self.differential_decoder = gr.diff_decoder_bb(2)
self.msgq = gr.msg_queue()
self.rds_decoder = rds.data_decoder(self.msgq)
self.connect(self.rds_bb_filter, (self.bpsk_demod, 0))
self.connect(self.clock_filter, (self.bpsk_demod, 1))
self.connect(self.bpsk_demod, self.differential_decoder)
self.connect(self.differential_decoder, self.rds_decoder)

#
# this is used for tuning to stations automatically
#
self.probe = gr.probe_avg_mag_sqr_d_f(5, 0.1)
self.connect (self.pilot_filter, self.probe)

self._build_gui(vbox, usrp_rate, audio_rate)

# set initial values
self.set_gain(self.gain)
self.set_vol(self.vol)
if not(self.set_freq(self.freq)):
    self._set_status_msg("Failed to set initial frequency")

##### GUI #####

def _set_status_msg(self, msg, which=0):
    self.frame.GetStatusBar().SetStatusText(msg, which)

def _build_gui(self, vbox, usrp_rate, audio_rate):

    def _form_set_freq(kv):
        return self.set_freq(kv['freq'])

    if 0:
        self.src_fft = fftsink2.fft_sink_c (self.panel, title="Data from USRP",
                                           fft_size=512, sample_rate=usrp_rate)
        self.connect (self.u, self.src_fft)
        vbox.Add (self.src_fft.win, 4, wx.EXPAND)

    if 0:
        post_fm_demod_fft = fftsink2.fft_sink_f (self.panel, title="Post FM
Demod",
                                           fft_size=512, sample_rate=usrp_rate, y_per_div=10, ref_level=0)
        self.connect (self.guts.fm_demod, post_fm_demod_fft)
        vbox.Add (post_fm_demod_fft.win, 4, wx.EXPAND)

    if 0:
        rds_fft = fftsink2.fft_sink_f (self.panel, title="RDS baseband",
                                       fft_size=512, sample_rate=usrp_rate, y_per_div=20, ref_level=20)
        self.connect (self.rds_clock, rds_fft)
        vbox.Add (rds_fft.win, 4, wx.EXPAND)

    if 0:
        rds_scope = scopesink2.scope_sink_f(self.panel, title="RDS timedomain",
                                           sample_rate=usrp_rate,num_inputs=2)
        self.connect (self.rds_bb_filter, (rds_scope,1))
        self.connect (self.rds_clock, (rds_scope,0))
        vbox.Add(rds_scope.win, 4, wx.EXPAND)

```





```

#         if 1:
#             const_sink = constsink_gl.const_sink_c(
#                 self.panel,
#                 title="Constellation Plot",
#                 sample_rate=usrp_rate,
##                 frame_rate=5,
##                 const_size=2048,
##                 M=4,
##                 theta=0,
##                 alpha=0.005,
##                 fmax=0.06,
##                 mu=0.5,
##                 gain_mu=0.005,
##                 symbol_rate=usrp_rate/4.,
##                 omega_limit=0.005,
#             )
#             self.connect (self.rds_bb_filter, gr.float_to_complex(), const_sink)
#             vbox.Add(const_sink.win, 4, wx.EXPAND)

self.rdspanel = rdsPanel(self.msgq, self.panel)
vbox.Add(self.rdspanel, 4, wx.EXPAND)

# control area form at bottom
self.myform = form.form()

# 1st line
hbox = wx.BoxSizer(wx.HORIZONTAL)
self.myform.btn_down = wx.Button(self.panel, -1, "<<")
self.myform.btn_down.Bind(wx.EVT_BUTTON, self.Seek_Down)
hbox.Add(self.myform.btn_down, 0)
self.myform['freq'] = form.float_field(
    parent=self.panel, sizer=hbox, label="Freq", weight=1,
    callback=self.myform.check_input_and_call(_form_set_freq,
self._set_status_msg))
hbox.Add((5,0), 0)
self.myform.btn_up = wx.Button(self.panel, -1, ">>")
self.myform.btn_up.Bind(wx.EVT_BUTTON, self.Seek_Up)
hbox.Add(self.myform.btn_up, 0)
self.myform['freq_slider'] = form.quantized_slider_field(
    parent=self.panel, sizer=hbox, weight=3,
    range=(87.5e6, 108e6, 0.1e6), callback=self.set_freq)
hbox.Add((5,0), 0)
vbox.Add(hbox, 0, wx.EXPAND)

# 2nd line
hbox = wx.BoxSizer(wx.HORIZONTAL)
hbox.Add((5,0), 0)
self.myform['volume'] = form.quantized_slider_field(parent=self.panel,
sizer=hbox,
            label="Volume",
            weight=3,
            range=self.volume_range(),
callback=self.set_vol)
#         hbox.Add((5,0), 1)
#         self.myform['gain'] = form.quantized_slider_field(parent=self.panel,
sizer=hbox,
            label="Gain",
            weight=3,
            range=self.subdev.gain_range(),
callback=self.set_gain)
#         hbox.Add((5,0), 0)
#         vbox.Add(hbox, 0, wx.EXPAND)

##### EVENTS #####

def set_vol (self, vol):
    g = self.volume_range()
    self.vol = max(g[0], min(g[1], vol))
    self.volume_control_l.set_k(10**(self.vol/10))
    self.volume_control_r.set_k(10**(self.vol/10))
    self.myform['volume'].set_value(self.vol)
    self.update_status_bar ()

```




---

```

def set_freq(self, target_freq):
#     r = usrp.tune(self.u, 0, self.subdev, target_freq)
#     r = self.u.set_center_freq(target_freq)
#     if r:
#         self.freq = target_freq
#         self.myform['freq'].set_value(target_freq)
#         self.myform['freq_slider'].set_value(target_freq)
#         self.rdspanel.frequency.SetLabel('%3.2f' % (target_freq/1e6))
#         self.update_status_bar()
#         self.bpsk_demod.reset()
#         self.rds_decoder.reset()
#         self.rdspanel.clear_data()
#         self._set_status_msg("OK", 0)
#         return True
#     else:
#         self._set_status_msg("Failed", 0)
#         return False

def set_gain(self, gain):
#     self.myform['gain'].set_value(gain)
#     self.subdev.set_gain(gain)
#     self.u.set_gain(gain)

def update_status_bar (self):
#     msg = "Volume:%r, Gain:%r, Freq:%3.1f MHz" % (self.vol, self.gain,
self.freq/1e6)
#     self._set_status_msg(msg, 1)

def volume_range(self):
#     return (-20.0, 0.0, 0.5) # hardcoded values

def Seek_Up(self, event):
#     new_freq = self.freq + 1e5
#     if new_freq > 108e6:
#         new_freq=88e6
#     self.set_freq(new_freq)
#     print self.probe.level()

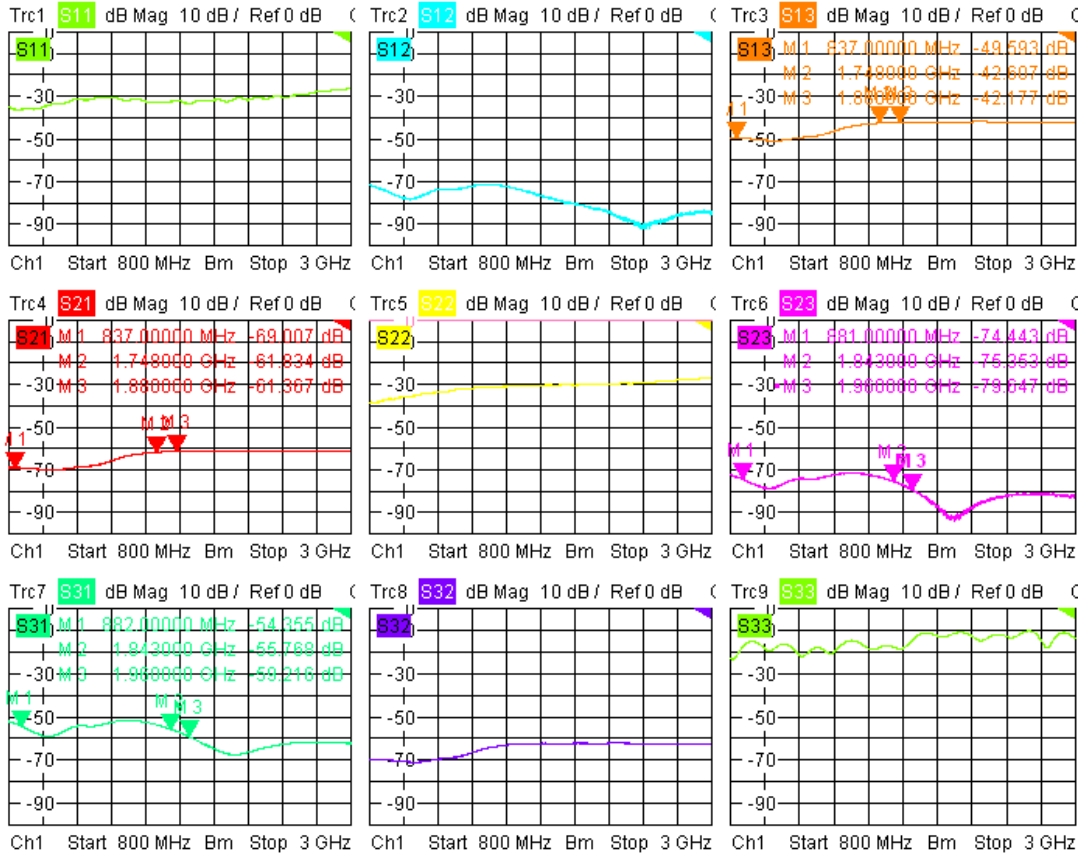
def Seek_Down(self, event):
#     new_freq = self.freq - 1e5
#     if new_freq < 88e6:
#         new_freq=108e6
#     self.set_freq(new_freq)
#     print self.probe.level()

if __name__ == '__main__':
#     app = stdgui2.stdapp (rds_rx_graph, "USRP RDS RX")
#     app.MainLoop ()

```



### 6.5. Annex E: 3-port s-parameter measurement of laboratory setup





## 6.6. Annex F: Sending samples source code with byte orders and csv generation

```

#include <iostream>
#include <math.h>

using namespace std;

int main(int argc, char *argv[])
{
    short sample[400];
    uint32_t full_sample[200];

    // calculate sine wave with amplitude of 2^12 in 200 steps
    for (int i = 0; i < 200; i++) {
        // first 2 bytes (size of short is 2 byte) are cos-function (I)
        // and second sin-function (Q). (i<<1) multiplies increment variable i by two
        sample[i<<1] = 4096*cos(2*M_PI*(i % 200)/200);
        sample[(i<<1) + 1] = 4096*sin(2*M_PI*(i % 200)/200);
    }

    // open and if not exists create file sine_out.csv for writing
    FILE * pFile;
    pFile = fopen("sine_out.csv","w+");

    // step through sample array to perform byte operations
    for(int i =0; i<200; i++) {
        // write current I and Q sample[i] in decimal and hex notation to stdout
        cout<<dec<<"\tI(dec): "<<sample[i<<1]<<"\tQ(dec): "<<sample[(i<<1) + 1];
        cout<<hex<<"\tI(hex): "<<sample[i<<1]<<"\tQ(hex): "<<sample[(i<<1) + 1];

        // write current decimal I and Q sample[i] divided by a comma to csv file
        fprintf(pFile,"%i,%i\n",sample[i<<1], sample[(i<<1) + 1]);

        // build 32 bit array full_sample in little endian format (16 bit I + 16 bit Q)
        // this is how data are sent to USRP2 in tx_16sc function (GNU Radio)
        full_sample[i] = ((sample[i<<1] >> 8) & 0xff | (sample[i<<1] << 8))<< 16 |
            ((sample[(i<<1) + 1] >> 8) & 0xff|(sample[(i<<1) + 1] << 8) & 0xff00);

        // write current full_sample[i] in hex notation to stdout
        cout<<hex<<"\tfull_sample(hex): "<<full_sample[i<<1]<<endl;
    }

    // close file pointer
    fclose(pFile);

    // if necessary, insert a system("pause") for windows or the like here
    return 0;
}

```