GNU Radio

Wireless protocol analysis approach

Author:
Alex Verduin

Supervisors:
Maurits van der Schee
Karst Koymans

October 26, 2008

Final version

UNIVERSITEIT VAN AMSTERDAM

Master program System and Network Engineering

Abstract

This paper describes how GNU Radio software combined with the Universal Software Radio Peripheral (USRP) can be used to perform a wireless protocol analysis. GNU Radio and the USRP are an implementation of software defined radio (SDR). The design principle behind SDR is to bring the software as near to the radio antenna as possible, and to do most of the signal processing in software.

This research consists out of two phases. The first phase contains a literature study about the working and design of the USRP and GNU Radio. The second phase consist out of practical work which delivers an general approach, used to perform a wireless protocol analysis. This approach is structured in steps.

The first step of the approach begins with understanding the working of SDR and the GNU Radio software with the USRP. The second step involves setting up the test system. Analysing the protocol specifications and retrieve protocol specific parameters like the used frequency and modulation technique, is the next step. Derived from the protocol specifications, a flow graph can be created. The flow graph is a representation of the radio, connecting the signal process blocks together.

The following steps involve translating the flow graph to Python and C++ code and to capture sample data. The capture of sample data is very useful in a protocol analysis. This gives the opportunity to replay the sample and to fulfil analyses on different levels of the protocol and to correlate them. The last step describes how the gathered data can be analysed and could be interpreted.

As example protocol to demonstrate the general approach, this research involved analysing the Radio Data System (RDS) protocol. RDS is used to broadcast radio station information and for example, artist/song information next to the FM broadcast.

1 Acknowledgement

I would like to thank both supervisors for their assistance and information during my research. I also would like to thank *Dawei Shen* [48] for writing such an excellent explanation about creating signal processing blocks. His article gave me a very good insight in how signal blocks are constructed and work.

Contents

2 Introduction

Wireless protocols, like WiFi [35] or Bluetooth [4] are being used more and more. These protocols are mostly implemented into firmware [12] or hardware. This makes it difficult for system and network engineers to analyse the protocol on the lowest level. For this purpose it would be interesting to receive raw wireless traffic. This could be achieved by using a Software Defined Radio (SDR) [27]. With a software defined radio it is possible to modulate and demodulate the radio signal in software.

An implementation of SDR could be GNU Radio [42] in combination with Universal Software Radio Peripheral (USRP) [43]. The USRP hardware device is used to digitalize the received analog radio signal, so it can be imported into a computer. On the computer it is possible to build your own radio receiver or transmitter by the use of GNU Radio software. This creates the flexibility to build almost every kind of receiver or transmitter in software. There are of course limitations, for example performance.

In this paper is argued that GNU Radio in combination with the USRP is a useful tool in the work field of a System and Network Engineer. The System and Network Engineer could use the tool to analyse wireless protocols. The goal of this research project is to investigate what approach should be followed to use GNU Radio in combination with the USRP in a wireless protocol analysis. This approach is explained by an example of an analysis of a well known and documented protocol, namely the Radio Data System (RDS) [41]. RDS is a wireless protocol standard for sending small amounts of digital information using FM radio broadcasts. The RDS system transmit several types of information like track/artist info and station identification. The standard is maintained by the European Broadcasting Union [11].

2.1 Structure

This research is conducted into two phases. The first phase is a literature study about SDR (Software Defined Radio), the working of GNU Radio and the USRP. The result of the literature study is described in paragraph 2.4 combined with chapter 3. This study is essential to fully understand how to use the approach, defined in chapter 4.

The second phase describes the results of the practical work. In this phase an approach is proposed to perform a wireless protocol analysis. This is done by setting up a system with GNU radio and the USRP to perform a protocol analysis on a simple protocol, namely RDS.

The approach, described in chapter 4, consist out of a number of steps which will guide you to perform a protocol analysis. The steps defined in the approach are illustrated with examples of the RDS protocol, because the RDS protocol is analysed to create this approach. Even though RDS is a simple protocol this approach can also be used to perform more complex wireless protocol analysis.

The document finishes with the considerations, conclusions and future work. Installation instructions and the created source code can be found in the appendix.

2.2 Research question

The research question is formulated as follows:

> "How can a system and network engineer use the USRP and GNU Radio to fulfil a wireless protocol analysis?"

Derived from the research question, the following sub questions are defined:

- How does the USRP work?
- What are the limitations of the USRP and GNU Radio?
- How difficult is it to write your own code?
- What kind of approach should be followed in a protocol analysis?
- Which prior knowledge is required to use GNU Radio in a protocol analysis?

The above questions result in a clear view of the possibilities and limitations of GNU Radio and an approach to use GNU radio in a wireless protocol analysis. The research is split into two phases. The first is a literature study and the second phase consist out of practical work.

2.3 Scope

The scope of the project is to create an approach that can be followed to use Software Defined Radio in a wireless network analysis. This approach is illustrated with an analysis of an example protocol, namely RDS. The defined approach is generally defined. This means that the created approach also can be used for more complicated protocols.

2.4 Related Work

In [46] *David A. Scaperoth* describes the results of his research in which he uses GNU Radio as an Cognitive Radio. This research showed me that SDR can be used to configure a flexible radio platform with use of GNU Radio. The research also describes some limitations of the USRP, like the used USB connection and the calculation power needed on the host to process the signal.

Philip Balister and *Jeffrey H. Reed* describes in [38] the results of their research to use USRP in common software architectures for the Joint Tactical Radio System. This report helped me to understand the working of the USRP.

In [52] *Kalen Watermeyer* describes a design for a hardware platform for SDR that must be compatible with GNU Radio software. This demonstrates that GNU Radio software can be used with different kind of hardware peripheral and is not bound to the USRP.

Zang Li, *Wenyuan Xu*, *Rob Miller* and *Wade Trappe* from the state university of *New Jersey* describes in [53] how GNU Radio and the USRP can be used in experiments to secure wireless networks. This paper showed me that GNU Radio with the USRP is a good tool to perform prototyping of wireless protocols.

In [47] *Thomas Schmid*, *Oussama Sekkat* and *Mani B. Srivastava* describe their conducted research to the network performance impact of increased latency

in Software Defined Radios. They focused in their research on GNU Radio and the USRP. This research gives a clear view of the bad performance of SDR compared to dedicated radio, which is a limitation.

Lee K. Patton describes in [45] how GNU Radio and the USRP can be implemented to create a software defined radar. This research shows the limitations in performance and the synchronisation between the send and receive path.

In [40] *Prateek Mohan Dayal* describes the method of quadrature sampling used in the USRP. The understanding of quadrature sampling is not essential knowledge needed to use GNU Radio with the USRP, but it gave me a clearer view of how the sampling works with GNU Radio and the USRP.

Dominic Spill and *Andrea Bittau* of the *University College London* Describes in [49] how they used GNU Radio to perform eavesdropping on the Bluetooth protocol. This researched shows how GNU Radio with the USRP can be implemented to sniff wireless traffic. Their way of creating a set-up was inspiring for my research.

Ketan Mandke describes in [44] the early results of the research done at *Hydra: A Flexible MAC/PHY Multihop Testbed*. This research proves that GNU Radio with the USRP is flexible and well suited to create different kind of radios. This research showed me that GNU Radio can be used as a test bed for different protocols. This proved to me that it is possible to fulfil a wireless protocol analysis with GNU Radio and the USRP on different protocols.

Stefan Valentin, *Holger von Malm* and *Holger Karl* from the *University of Paderborn* describe in this [51] [50]two papers how to create a wireless network test bed and how to implement a physical and data-link control layer with GNU Radio software defined radio platform. The research described in [51] contains a performance test which shows that GNU Radio with the USRP is not suited for protocols that have data rates in the mega bits domain. The paper [50] gives a good explanation of how wireless protocols are constructed, and discusses some simple examples which helped to understand GNU Radio and the USRP.

3 Dataflow in USRP hardware and GNU Radio software

The USRP is the hardware element that combined with software tool kit GNU Radio is an implementation of SDR [27]. This section describes the flow of data though the USRP hardware and GNU Radio software. Before the specific details of the USRP [43] hardware and GNU Radio software are discussed, first some basics of SDR are described, because SDR is the design principle behind GNU Radio and the USRP

3.1 SDR basics

The design principle, shown in figure 1, is to bring the software code as near to the radio antenna as possible. This is achieved by using hardware that translates radio waves [23] to a data stream a computer can handle. The hardware should be transparent, from the view of the software.

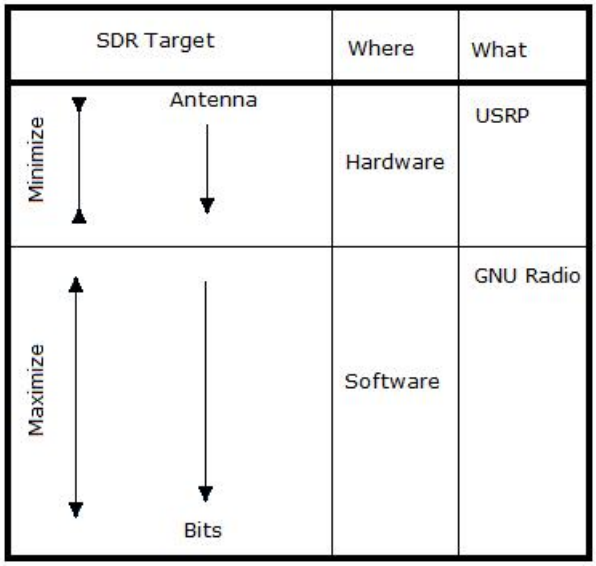

Figure 1: SDR design principle

A good definition of Software Defined Radio can be found on the *SDR Forum* [28]. The definition is stated as follows:

> "Radio in which some or all of the physical layer functions are Software Defined."

In above definition, the physical layer is by the *SDR Forum* defined as follows:

> "The layer within the wireless protocol in which processing of RF(Radio frequency), IF(intermediate frequency), or base-band signals includ-

ing channel coding occurs. It is the lowest layer of the ISO 7-layer model as adapted for wireless transmission and reception."

3.2 USRP

Wireless communication makes use of electromagnetic waves as a medium between nodes. Electromagnetic waves are characterized by *frequency* and *amplitude* [23]. This is shown in figure 2. The frequency is the number of waves per second. So a frequency of 101 MHz corresponds with 101.000.000 waves in one second. The amplitude is the maximum deviation of the wave from the centre. Wireless communication can use high frequencies to communicate. To sample and transfer those high frequencies the SDR implementation must use down converting which will be discussed in 3.2.6.

The USRP is used to create the connection between the RF-world (radio frequent [24]) and the computer (See figure 3). It takes the input of the antenna which receives radio waves and digitalizes those. The USRP must do as little as possible and only digitalize the signal and deliver it to the computer where all the calculations can be done in software.

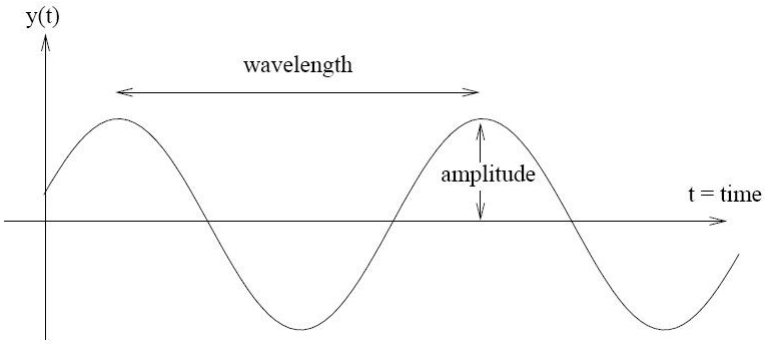

Figure 2: Radio wave. Source: [50]

To use the USRP in a wireless project it is necessary to know what components are used inside the USRP and what their functions are. This knowledge is necessary to understand to perform a wireless protocol analysis. GNU Radio is an open source project, the schematics of the USRP are also freely obtainable. The USRP is constructed out of the different components, which are described in detailed below:

- USB Controller
- ADC (Analog to Digital Converter)
- DAC (Digital to Analog Converter)
- PGA(Programmable Gain Amplifier)
- Daughterboards
- FPGA (Field Programmable Gate Array)

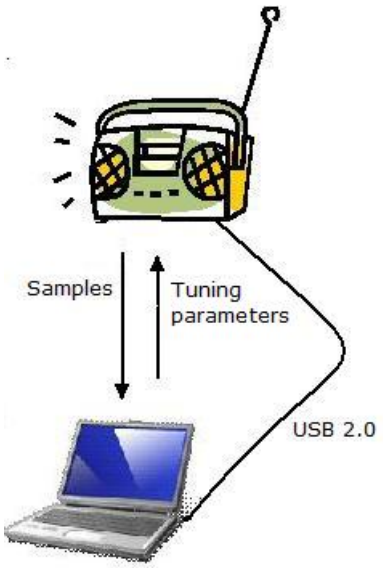

Figure 3: SDR hardware

3.2.1 USB 2.0 Controller

USB is used to connect the USRP to the computer as described in this quote from [34]. The FPGA will be described in paragraph 3.2.6.

> "The FPGA, in turn, connects to a USB2 interface chip, the Cypress FX2, and on to the computer. The USRP connects to the computer via a high speed USB2 interface only, and will not work with USB1.1."

As you can see the USRP only works with USB 2.0. In [33] can be found that this delivers a data throughput of maximal 32 MB/sec. The USB connection has a serious impact on the performance. This is researched in [47]. There is stated that the latency is on average 41.33 μsec. Also the data throughput can be a limitation like discussed in [18] were is tried to receive HDTV.

3.2.2 ADC (Analog to Digital Converter)

The analog to digital converter (ADC) has the function to digitize analog signals. This is used on the USRP to receive radio signals. The following quote from [1] shows the technical specifications:

> "There are 4 high-speed 12-bit AD converters. The sampling rate is 64M samples per second. In principle, it could digitize a band as wide as 32MHz"

The quote above also illustrates a limitation, because it is not possible to receive signals without any loss in a bigger bandwidth then 32 Mhz. The 32 Mhz is calculated by the use of the Nyquist theorem [19].

3.2.3 DAC (Digital to Analog Converter)

The digital to analog converter, converts a digital constructed signal to an analog signal. This is used in the USRP to transmit a analog radio signal. This quote from [6] shows the technical specifications:

> "At the transmitting path, there are also 4 high-speed 14-bit DA converters. The DAC clock frequency is 128 MS/s, so Nyquist frequency is 64MHz. However, we will probably want to stay below it to make filtering easier. A useful output frequency range is from DC (Direct Current)to about 44MHz."

The above quote also describes a limitation. For transmission it is not possible to send signals without loss above the 44MHz bandwidth. An explanation of Nyquist can be found in [19]

3.2.4 PGA(Programmable Gain Amplifier)

In the receive path, the programmable gain amplifier amplifies the received signal. This quote from [20] describes the function of the PGA:

> "There is a programmable gain amplifier (PGA) before the ADCs to amplify the input signal to utilize the entire input range of the ADCs, in case the signal is weak. The PGA is up to 20dB"

3.2.5 Daughterboards

Daughterboards make it possible to use the USRP in different frequency spectrum's. This is, because there are are other physically RF components needed to receive different frequency spectrum's. In [43] the follow description about daughterboards is given :

> "On the mother board there are four slots, where you can plug in up to 2 RX basic daughterboards and 2 TX basic daughterboards or 2 RFX boards. The daughterboards are used to hold the RF receiver interface or tuner and the RF transmitter. There are slots for 2 TX daughterboards, labeled TXA and TXB, and 2 corresponding RX daughterboards, RXA and RXB. Each daughterboard slot has access to 2 of the 4 high-speed AD / DA converters (DAC outputs for TX, ADC inputs for RX)."

The above quote shows that it is possible to connect multiple daughterboards on the USRP. This enables the USRP to send and receive simultaneously.

The next quote from [43] describes the technique used to identify a daughterboard. This means that the USRP will automatically be configured if a daughterboard is changed.

> "Every daughterboard has an I2C EEPROM (24LC024 or 24LC025) onboard which identifies the board to the system. This allows the host software to automatically set up the system properly based on the installed daughterboard."

The following enumeration shows the available daughterboards with their corresponding frequencies and transmission power. This information is retrieved from [3] [2].

Basic RX and Basic TX Receiving and transmitting from 1MHz to 250MHz.

LFRX and LFTX Receiving and transmitting up to 30 MHz with 100 mW transmitting power.

DBSRX Receiving in the range from 800MHz up to 2.4GHz with 100 mW transmitting power.

TVRX Complete receiver system in the 50-860 MHz range based on a TV tuner module. It receive a 6 MHz wide block of spectrum and all tuning and functions can be controlled from software.

RFX400 Receiving and transmitting in the range from 400 MHz up to 500 MHz with 100 mW transmitting power.

RFX900 Receiving and transmitting in the range from 800 MHz up to 1000 MHz with 200 mW transmitting power.

RFX1200 Receiving and transmitting in the range from 1150 MHz up to 1450 MHz with 200 mW transmitting power.

RFX1800 Receiving and transmitting in the range from 1.5 GHz up to 2.1 GHz with 100 mW transmitting power.

RFX2400 Receiving and transmitting in the range from 2.3 GHz up to 2.9 GHz with 10 mW transmitting power.

3.2.6 FPGA (Field Programmable Gate Array)

The FPGA (Field Programmable Gate Array) is the hart of the USRP. This quote from [14] explains the functions.

> "(..)all the ADCs and DACs are connected to the FPGA. This piece of FPGA plays a key role in the GNU Radio system. Basically what it does is to perform high bandwidth math, and to reduce the data rates to something you can squirt over USB2.0. The FPGA connects to a USB2 interface chip, the Cypress FX2. Everything (FPGA circuitry and USB Microcontroller) is programmable over the USB2 bus."

The performed high bandwidth math is called *Digital Down Converting*(DDC). The next quote from [15] explains how DCC works and gives an example.

> "First, it down converts the signal from the IF (Intermediate Frequency) band to the base band. Second, it decimates the signal so that the data rate can be adapted by the USB 2.0 and is reasonable for the computer's computing capability. (. . .)
>
> For example, suppose we want to design an FM receiver. The bandwidth of a FM station is generally 200kHz. So we can select the

decimation factor to be 250. Then the data rate across the USB is $64MHz/250 = 256kHz$, which is well suited for the 200kHz bandwidth without losing any spectral information. We can set the IF frequency of the DDC using usrp.set_rx_freq() method and set the decimation factor using usrp.set_decim_rate() method in Python. The decimation rate must be in [1, 256]."

The example in the previous quote, describes how DDC works and how it is used. It also describes that the correct value for the decimation is important to correctly sample the radio signal.

3.3 GNU Radio

The GNU Radio tool kit [42] provides all the functionality to create radios in software. To realise this, there are different kind of program languages involved, all with their special purpose. GNU Radio also includes a library of signal processing blocks like modulators, demodulators, filters etc. which are used to construct a radio. Essentially it needs the USRP to receive real radio waves or to transmit. You do not necessarily need a USRP. There is also the possibility to use a (pre-recorded) file as input.

GNU Radio's software is organized using a two-tier structure. All the performance-critical signal processing blocks are implemented in C++ [5], while the higher-level organizing, connecting and gluing the signal blocks together is done using Python [22]. There is also a graphical environment available to create a custom radio. This is called GNU Radio Companion (GRC).

3.3.1 GNU Radio Companion

The GNU Radio Companion [16] is a graphical user interface which allows GNU Radio components to be put together graphically. It is currently under development by Josh Blum. Figure 4 shows a screen shot of GRC. The screen shot shows a dial-tone example of which the working of the example is explained in figure 5.

figure 4 shows that a radio is constructed out of blocks. In the figure the blocks are two *signal sources* and one *audio sink*. Those blocks are called *"signal processing blocks"* and are part of GNU Radio. A flow graph represents a collection of signal processing blocks connected together.

GRC creates from the created flow graph an XML [37] file that is translated to Python code (discussed in paragraph 3.3.2). References to the signal processing blocks of the GNU Radio library are also included in GRC by the means of XML definition files. This creates the possibility to include custom made signal processing blocks by defining an XML file for the new blocks. The Python code of the dial-tone example shown in figure 4, can be found in appendix 9.2.

3.3.2 Python

The Python [22] script language is used to connect the signal processing blocks together. In Python the necessary signal sources, sinks and processing blocks are selected and configured with the correct parameters. The flow of data through the flow graph exists out of data in one of the following data-types:

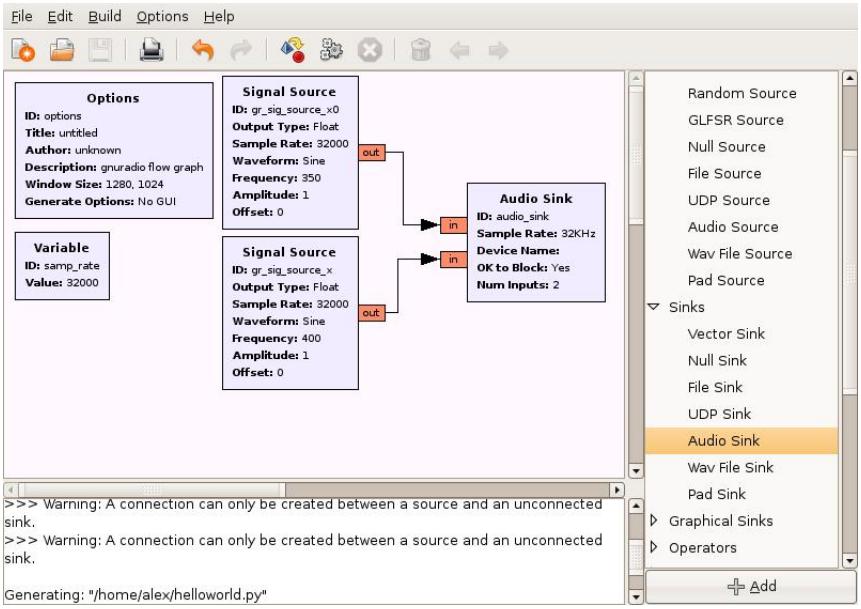

Figure 4: GNU Radio Companion screenshot

- Byte - 1 byte of data (8 bits)
- Short - 2 byte integer
- Int - 4 byte integer
- Float - 4 byte floating point
- Complex - 8 bytes

As mentioned previously, all sources, sinks and blocks are implemented as classes in C++. This results, no matter how complicated the radio is, in good readable Python code. The real heavy load is done in C++. Figure 5 shows an example of a Python code sample from [8]. As you can see it is not difficult to create a radio in Python.

```
Author : Eric Blossom

#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

def build_graph ():
sampling_freq = 48000
ampl = 0.1

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

return fg

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

"We start by creating a flow graph to hold the blocks and connections between them. The two sine waves are generated by the *gr.sig_source_f* calls. The *f* suffix indicates that the source produces floats. One sine wave is at 350 Hz, and the other is at 440 Hz. Together, they sound like the US dial tone.
audio.sink is a sink that writes its input to the sound card. It takes one or more streams of floats in the range -1 to +1 as its input. We connect the three blocks together using the *connect* method of the flow graph.
connect takes two parameters, the source endpoint and the destination endpoint, and creates a connection from the source to the destination. An endpoint has two components: a signal processing block and a port number. The port number specifies which input or output port of the specified block is to be connected. In the most general form, an endpoint is represented as a python tuple like this: (block, port_number). When port_number is zero, the block may be used alone. These two expressions are equivalent:
fg.connect ((src1, 0), (dst, 1))
fg.connect (src1, (dst, 1))

Once the graph is built, we start it. Calling *start* forks one or more threads to run the computation described by the graph and returns control immediately to the caller. In this case, we simply wait for any keystroke."

Figure 5: Code Sample. Source [8]

3.3.3 C++

Signal processing blocks process streams of data from their input port to their output port. The input and output ports of a signal process block are variable. So a block can have multiple outputs and multiple inputs. The signal processing blocks are written in C++. To use the C++ code in Python SWIG [31] is used.

SWIG is the wrapper [36] for the C++ modules and generate the corresponding Python code and library so that these classes and functions can be called from Python. As mentioned previously, most default signal blocks are already created in the GNU Radio project [42], or by third parties. So you only touch the C++ environment to create your own special signal processing blocks.

In appendix 9.6 you can see some example code of a signal processing block needed for decoding the RDS [41] signal. As you can see it requires some experienced C++ skills to create these blocks. If you do not have much C++ experience try to use signal blocks that are already part of the GNU Radio library, or try to find code from third parties.

3.4 Example projects

Software Defined Radios are already used for different kinds of projects. This section describes some of them:

FM Receiver Demodulates the FM-signal. Created by the *GNU Radio project*

Analog Television Receiver Demodulates the TV-signal. Created by the *GNU Radio project*

HDTV Reception Demodulates the HDTV-signal. Created by the *GNU Radio project*

GSM Demodulate the GSM signal. Created by *Henrik Omma* and *Heather Stern*

CDMA Demodulating CDMA signals. Created by *Zackary Anderson* and *Russell Ryan*

DVB-T Receive digital television. Created by *Vincenzo Pellegrini*

Bluetooth Signal processing software for bluetooth. Created by *Dominic Spill*

WiFi Signal processing software for WiFi. Created by *Vivek Raghunathan*

GPS Signal processing software for GPS. Created by *F.Firas*

4 Wireless protocol analysis approach

GNU Radio is too complicated to just take the equipment out of the box and conduct a protocol analysis with it. This chapter describes how to approach such a project. Every paragraph describes a step in the process. The following steps are defined:

- Understand the design of the USRP and GNU Radio
- Installing, configuring and code examples
- Study protocol specification and search for existing code
- Create flow graph
- Capture raw wireless samples
- Create the C++ and Python code
- Analyse the protocol

4.1 Understand the design of the USRP and GNU Radio

Before GNU Radio with the USRP can be used in a wireless protocol analysis, it is important to understand how GNU Radio and the USRP work. This is described in paragraph 3.2 and 3.3. Next, try to read the code example in paragraph 3.3.2, line by line and understand what is happening. Also understand what sampling [26] is and how digital to analog (and vice versa) converters work. This is described in paragraph 3.2.2 and paragraph 3.2.3.

Some basic knowledge of Python and C++ are essential. Python skills are necessary to program the *flow graph* (discussed in paragraph 3.3.1). C++ knowledge is necessary to understand how signal processing blocks are constructed. Also pay attention to the limitations of GNU Radio and USRP mentioned in chapter 3.2. Important limitations are the frequencies supported by the USRP. Check in paragraph 3.2.5 if there exist a suitable daughterboard for the frequency of interest. Latencies, bandwidth and timing issues between receiving, transmitting and the USRP, described in paragraph 3.2.1 could also be a problem.

4.2 Installing, configuring and code examples

GNU Radio can be installed on different kinds of operating systems, like MS Windows, Apple Mac OS X and Linux. GNU Radio is provided with good installation instructions. There are specific instructions for some operating systems like:

- Debian
- Fedora
- SuSE
- Ubuntu

- Mandriva
- Mac OS X
- NetBSD
- Windows

There are different versions of GNU Radio available. The latest stable release is currently version *3.1*. The most recent development version can be retrieved through *SVN* [29] from [30]. Before compiling and installing the source code, install all the necessary binary packages. After installing the software try some of the example scripts, to verify that the installation and hardware work properly. Pay some extra attention to the *dial tone* and *FM-radio* [39] example. Make sure the correct daughterboard is installed. From there adjust the sample code so that it is possible to write the signal to a file, and to read the recorded samples from file. With these exercises one can learn to work with different sources and sinks.

Problems encountered during this step was the compiling and installing of the *trunk* version of GNU Radio (Retrieved from *SVN*). The *trunk* version was needed because, the *binary* version of Ubuntu [32] contains an older version, which is discouraged to be used by the GNU radio project. In the *trunk* version a lot of bugs are solved [9]. In appendix 9.1 is described how GNU Radio is installed.

Most of the provided example code that is part of GNU Radio did not work. The code gave errors that libraries and specific modules could not be found. After analysing the problems, the cause was that some libraries were renamed and the implementation of methods where changed in the most recent version. This required rewriting of the code examples before they properly work.

4.3 Study protocol specification and search for existing code

The protocol specifications are essential to understand how a protocol works. Retrieve the protocol specifications and study them. Especially the parts discussing the physical layer [21] and the data link [7] layer. Interesting parameters are; used frequency spectrum, modulation and speed of the data transfers. These values provides the parameters for the demodulation and decoding blocks.

Next to studying the protocol specifications, search for existing code. As shown in paragraph 3.4 already a lot of code exist. Maybe somebody else has already created GNU Radio software blocks for the specific protocol. If code is available check the version it was built for.

For the RDS [41] analysis the specifications are maintained by the *European Broadcasting Union* [11] and are freely obtainable from their website. The specific specifications for the RDS protocol analysis are the following:

Frequency FM broadcast between 87.5 MHz and 108.0 MHz

Sub carrier frequency 57 kHz (suppressed) (This is the third harmonic of the pilot tone). See figure 6

Clock signal Retrieved from the pilot tone. $\frac{(3 \times 19kHz)}{48} = 1187,5$ bits per second as described in [41]

Modulation Customised form of two-phase phase-shift-keying

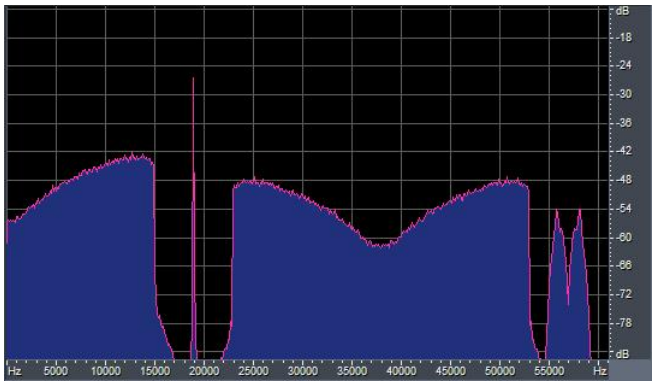

Figure 6: The figure (Source: [13]) shows a part of the FM spectrum, containing one broadcast station. On the left of the figure is shown the audio base-band signal. In a stereo broadcast this means the left + right channel. On 19 kHz is the pilot tone pictured. From around the 22 kHz the differences signal between the left and right (L - R) channel are pictured. On the 57 kHz the RDS signal is broadcasted with a suppressed carrier wave.

Because the RDS protocol is widely used, there is already some example code available [17]. This code was created by *Ronnie Gaensli*, and later adapted by *Ryan Shoff* followed by *Matteo Campanella*. The code consists out of four custom signal processing blocks and two Python library files. The four custom signal blocks that are written in C++ are more complicated to understand. Lack of comment in the code also contributes to this.

4.4 Create flow graph

Derived from protocol specifications it should be possible to create a flow graph (An example of a flow graph can be found in figure 7). The flow graph holds all the process signal blocks necessary to receive and decode the signal. Flow graphs start with a source. This could be for example the USRP or a file containing captured samples. The next signal processing block will be a filter (or multiple filters) to extract the radio spectrum [24] of interest from the incoming signal. After the radio signal is filtered it must be demodulated. Depending on the used modulation technology this delivers a binary stream. Next to this, will be a block that handles the error correction [10] and combining the binary data to its original format.

The last processing block is one of the kind *sink*. Examples of sinks are :

File The file sink gives the ability to write the signal to disk.

Sound card Directs the signal to the sound card.

Scope Useful to visualise the signal.

Screen Prints the samples to screen.

Figure 7 shows the flow graph of the RDS receiver. This graph is constructed using the RDS specifications [41] and the provided RDS sample code [17].

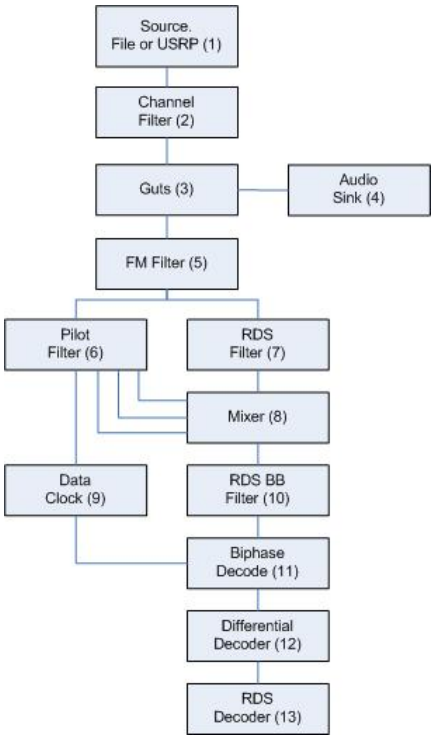

Figure 7: RDS Flow graph: **1. Source** The source could be the USRP (provided with tuning and decimation parameters) or a captured sample file. **2. Channel filter** Channel Filter cuts off all frequencies, 80 kHz above the tuned frequency. **3. Guts** Signal processing block for demodulating a broadcast FM signal. **4. Audio Sink** Connects the audio to the sound card. **5. FM filter** FM-Filter cuts off all frequencies, 70 kHz above the tuned frequency. **6. Pilot filter** Pilot filter only let the frequencies trough between 18 kHz and 20 kHz. **7. RDS Filter** RDS Filter only let the frequencies trough between 54 kHz and 60 kHz. **8. Mixer** Mixer adds the frequencies. It triples the 19 kHz pilot tone to reconstruct the suppressed 57 kHz carrier wave of the RDS signal. **9. Data Clock** Data clock takes the 19 kHz pilot tone and use it to reconstruct a clock signal. **10. RDS BB Filter** RDS Data filter, filters all the high frequencies above 1.5 kHz. **11. Biphase Decoder** Synchronises the signal with the clock signal and extracts the binary bits. **12. Differential Decoder** Determines by the difference between two receives bits the real data bit. **13. RDS Decoder** Implements the data link (and higher) layer(s). Does error correction etc. and translates the bits to characters.

4.5 Capture raw wireless samples

In this step code will be created to capture raw wireless signal. This step is important because this creates a signal source of a constant quality which

delivers a reliable reference source for the following (steps) analysis. For this step you need the frequency parameters of the protocol.

To capture RDS signal samples I created the capture code, which can be found in appendix 9.3. The code is based on examples provided by the GNU Radio software.

After gathering raw wireless samples, I created a script which takes the file created by the capture script as input. This convert script skips the first 20.000.000 samples and writes the following 500.000 samples to a new file. This is done to remove the tuning process at the beginning of the file, and to create a sample file with an exact number of samples. The convert script can be found in appendix 9.4.

4.6 Create the C++ and Python code

Start with creating the custom signal processing blocks. A good explanation of how this can be done is described in the article *How to Write a Signal Processing Block* [48]. This involves writing the C++ code and compiling and installing the code.

After constructing the custom blocks, put everything together. This is done in Python or in GRC (GNU Radio Companion). If the flow graph is more complex it is advisable to use Python over GRC, because the resulting Python code will be less clear with GRC, and there is no ability to add comments to the code. Also if you have created custom signal processing blocks you need to import them into GRC.

In the RDS protocol analysis the starting point was the sample code provided by [17]. First the brief install instructions provided with the code where performed. After fully understand the instructions it was possible to configure and compile the custom C++ RDS signal processing blocks. The document *How to Write a Signal Processing Block* [48] was a big help. The provided Python code from [17] makes use of the custom C++ RDS signal processing blocks. However this code did not function. The cause of this was (again) the difference in the used GNU Radio version. From there I re-wrote the Python code step by step. The result of the created code can be found in appendix 9.4.

4.7 Analyse the protocol

The final step is the analysis of the protocol. Described previously, an advantage of GNU Radio is the possibility to analyse the radio signal before it is translated into bits. This is in contradiction to dedicated RDS enabled radio receivers, where all converting is done in hardware and it is not easy to visualize the radio signal after the different processing stages. With GNU Radio it is relatively easy to visualise (plot) the signal after every signal processing block, as will be shown in this paragraph. Visualisation of the signal is done by connecting a file sink after every signal process block of the flow graph, and write the samples to file. From those files can be made graphical plots. The GNU Radio Software library provides Python code to create those plots. The code plots the relative amplitude against the recorded samples in time.

This approach of analysing can be used to perform root-cause [25] analysis. An example could be *Why do I have good RDS reception outside, but poor reception inside the building?* There are (at least) two ways of approaching

this. The first approach includes making plots of the proper working situation and compare them with the poor working situation. This could show after which signal processing block the signal is not sufficient any more. If it is not possible to create reference material in a good working environment use the protocol specification to reason what is going wrong after every process signal block.

If the plot functionality of GNU Radio is used, it is important that the correct number of samples is plot. This is because the frequency of a signal is related to time (see paragraph 3.2). So to plot a defined period of time, we need to know how many samples represents that period. The following calculation shows how to interpret samples versus time.

$$\frac{\text{Sample rate}}{\text{Decimation}} = 1 \text{ second of data} \tag{1}$$

The figures 8 to 20 show the protocol analysis of the RDS protocol. As mentioned in paragraph 4.3, the RDS signal is transmitted by use of FM broadcast. This means that the USRP must be tuned to a FM broadcast station that transmits RDS. In the beginning of the research I experienced, with the FM-Radio receiver sample script (part of GNU Radio) that the Dutch radio station *SkyRadio* had a good reception. Figure 8 shows that the station transmit RDS information. This radio station is used to perform the RDS protocol analysis.

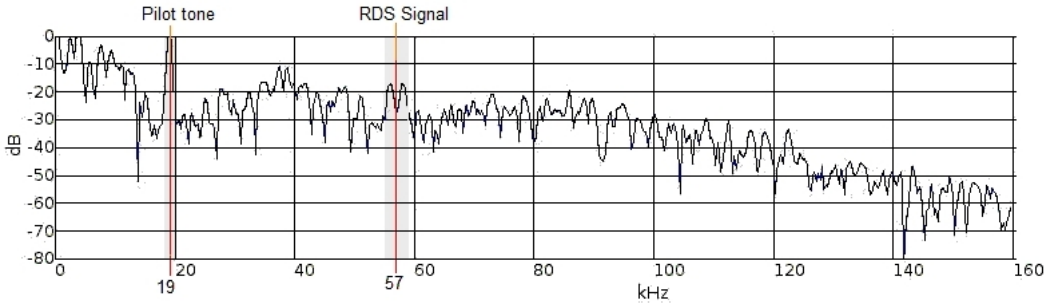

Figure 8: This figure shows a screen shot taken from GNU Radio example code *usrp_wfm_rcv.py*, tuned to the *SkyRadio* broadcast. In this figure we can see the 19 kHz pilot tone and the RDS broadcast with the suppressed carrier wave on the 57 kHz. This corresponds to figure 6, showing the FM spectrum of a broadcast.

Before starting the analysis, a sample file is created as described in paragraph 4.5. The created sample file is used as source for the RDS decode application, described in appendix 9.5. This means that the USRP is no longer needed. To perform the analysis, file sinks were placed after every signal process block. This resulted in multiple files from different stages of the RDS reception. From those files are made plots by using a plot script provided with GNU Radio. This creates the possibility to debug the protocol after every signal processing block. All figures in this analysis represent the same data. A little shift in data between the figures can be seen, which is probably caused by buffers in the processing blocks.

First the pilot tone will be analysed. As described in the protocol specifications, the pilot tone is an important part of the RDS protocol. The pilot tone is used to distract the clock signal and to reconstruct the carrier wave of the RDS signal. Figure 9 shows the pilot tone after the pilot filter.

The pilot filter is a generic GNU Radio signal processing block and functions as a bandpass filer. The bandpass filter is configured to reduces the amplitude of signals below 18 kHz and above the 20 kHz. In figure 8 is shown in the marked area around the 19 kHz which frequency spectrum will be let through by the filter.

To verify if the pilot tone is correct, we need to know if the signal represents a 19 kHz signal. This can be done by plotting the signal after the pilot filter and choosing a convenient block size to visualise the signal. In figure 9 is chosen to visualize 320 samples, because the sample rate in the RDS decode application is configured to 64.000.000 and the decimation is set to 200. This results in a representation of 320.000 samples to 1 second in time. This means that the plot must show 19 waves. This can been verified in figure 9, so we can assume that the pilot tone is valid.

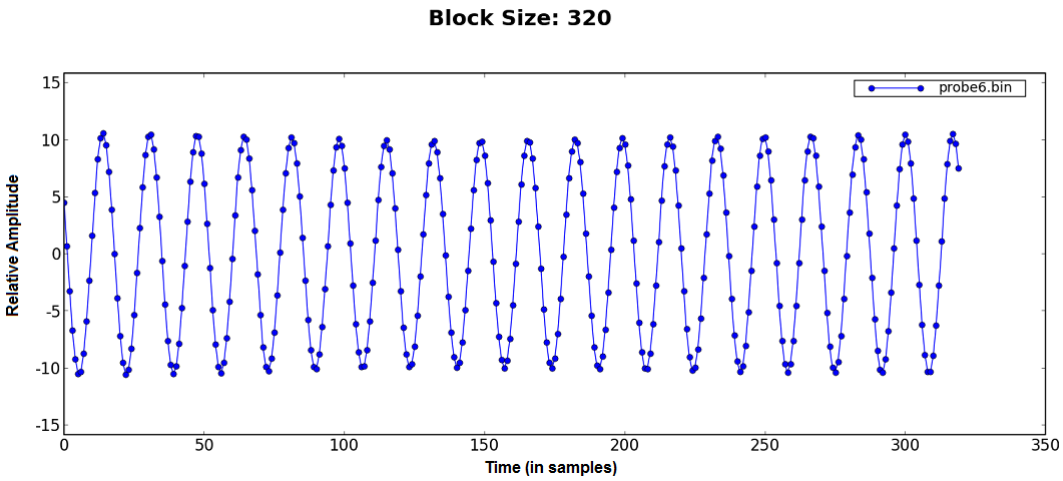

Figure 9: This plot shows the signal after the pilot filter. The plot shows exactly 320 samples which represents 1 thousands of a second, because the decimation rate is 320.000. The pilot tone is a 19 kHz signal which means that this plot must show 19 waves, which is correct.

Now we assuming that the pilot tone is correct, we can go to the next step. According to figure 7 we see that the next signal processing block is the data clock. The data clock is a custom signal processing block, which means that it is written for RDS decoding. The C++ code of this signal processing block can be found in appendix 9.6.

The function of the block is to generate a clock signal from the 19 kHz pilot tone. The C++ code does this by counting the number of zero crossings of the pilot tone. Every wave has two zero crossings. After every 16 zero crosses the clock signal changes from high to low, or from low to high. The next calculation shows how many zero crossings the pilot tone makes in one second (this is before the data clock).

$$19000 \text{ Hz} \times 2 = 38000 \text{ zero crossings} \tag{2}$$

If we divide the 38000 zero crossings by 16 we get the number of zero crossings the signal should have after the data clock signal block :

$$\frac{38000 \text{ zero crossings}}{16} = 2375 \text{ zero crossings} \tag{3}$$

The above calculation shows that the signal after the data clock should have 2375 zero crossings. To know how many waves this represents, the number of zero crossings is divide by two, because in this case every wave has two zero crossings. The result should be the bit rate of the RDS specifications [41] (1187,5 bit per second)

$$\frac{2375 \text{ zero crossings}}{2} = 1187{,}5 \tag{4}$$

Figure 10 shows the plot of the data clock signal. The block size is set to 2695 samples which shows much more samples then the plot of the pilot tone, shown in figure 9. This block size is chosen, so it should represent 10 periods of the data clock, which it does. Figure 11 shows a full bit in samples 20 - 290.

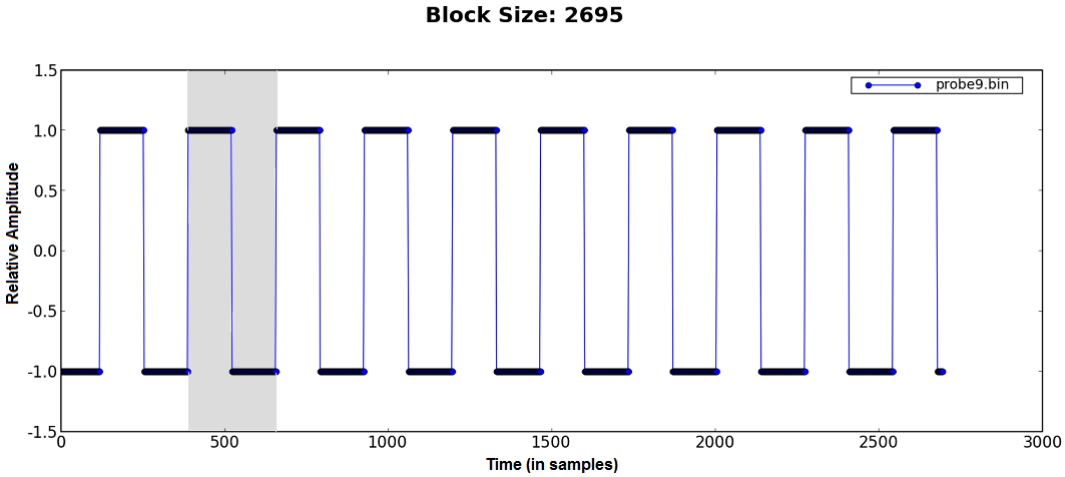

Figure 10: This plot shows the reconstructed data clock as described in the protocol specifications. The clock signal corresponds to 1187,5 bits per second. In the figure, one period (block signal) is marked. The plot shows 2695 samples which should represent 10 periods, which is correct.

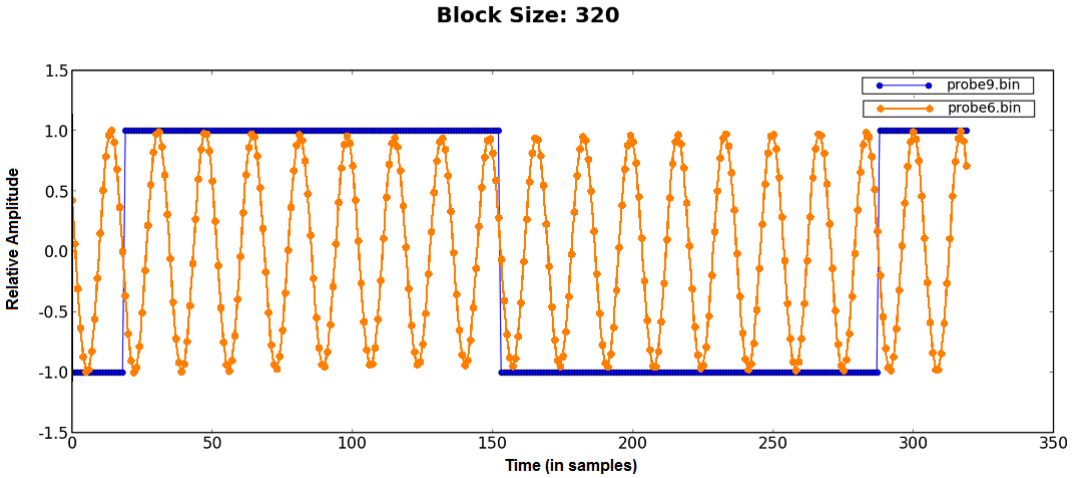

Figure 11: This plot shows the reconstructed data clock (blue) together with the pilot tone (red). Looking at the plot we can see that it shows a little bit more then one data bit. We can also see that 16 zero crosses of the pilot tone will change the data clock from high to low or from low to high.

Besides the data clock recovery, also the data signal has to be recovered from the FM signal. This is done in a number of steps. The first step is to filter the

RDS signal out of the FM signal. This is done by the RDS filter. The RDS filter is a bandpass filter, configured to reduces the amplitude of signals below 54 kHz and above the 60 kHz. In figure 8 is shown which frequency spectrum will be let through using a marked area around 57 kHz.

The protocol specifications [41] describes that the used modulation is a customised form of two-phase phase-shift-keying using biphase symbols. This means that two symbols eventually will correspond to one bit.

To verify if the RDS signal is correct the signal will be visualized. This is done with the same block size as described in the data clock recovery, because (again) we want to visualize 10 bits of data. Figure 12 shows the signal after the RDS Filter. This figure should show 20 recognisable symbols, which is true.

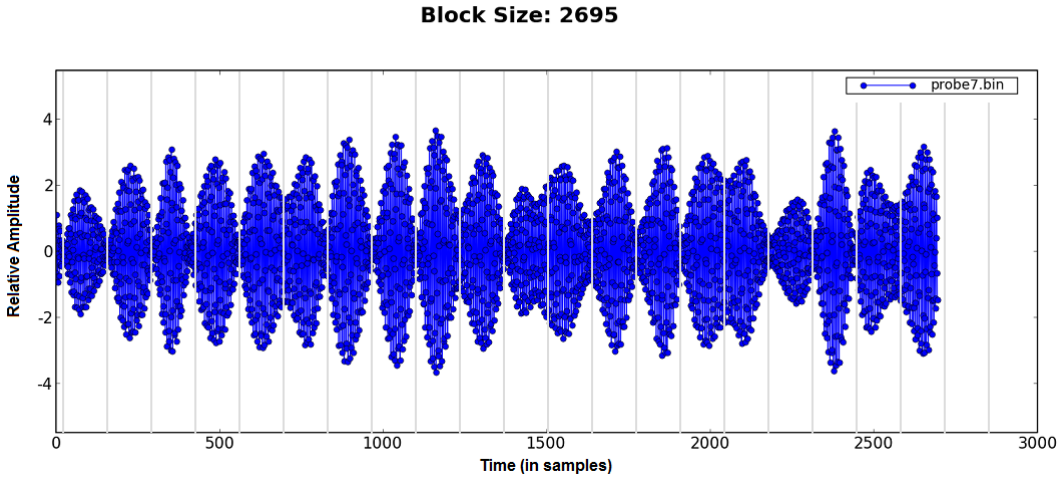

Figure 12: This plot shows the RDS signal after the RDS Filter. This plot, just like the plot of the data clock (see figure 10),represents 2695 samples, so it should represents 10 bits of data. As described in the RDS protocol specification, the used modulation makes use of biphase symbols, which means that there should be 20 symbols. We can also see that a low pas filter is needed, because the plot shows up black of the high frequent signals.

The RDS protocol specifications [41] describe that RDS makes use of a suppressed carrier wave. This carrier needs to be reconstructed and added to the RDS data signal which is visualised in figure 12. The carrier wave is reconstructed from the pilot tone. This is done by tripling the 19 kHz tone to 57 kHz and added to the RDS data signal. This is done in the mixer. Figure 13 shows the signal after the mixer.

The mixer is a generic GNU Radio signal processing block which takes multiple signals as input and *mixes* them all together. Figure 7 shows that the mixer receives three times the signal from the pilot filter. This reconstructs the carrier wave. Next to this, the RDS data signal is added.

To verify if the mixed signal is correct, we need to know if the signal represents 20 symbols or 10 biphase symbols, and is mixed with the reconstructed carrier wave. This is done by visualising the signal after the mixer, again with a block size of 2695 samples. Figure 13 shows the signal after the mixer. In the figure can been seen that the carrier wave has been added to the RDS signal. (Notice that the values on the y-axis have changed). There can also be seen that some samples are more amplified then others. Figure 13 still shows 20 symbols or 10 biphase symbols, so we can assume the signal is correct.

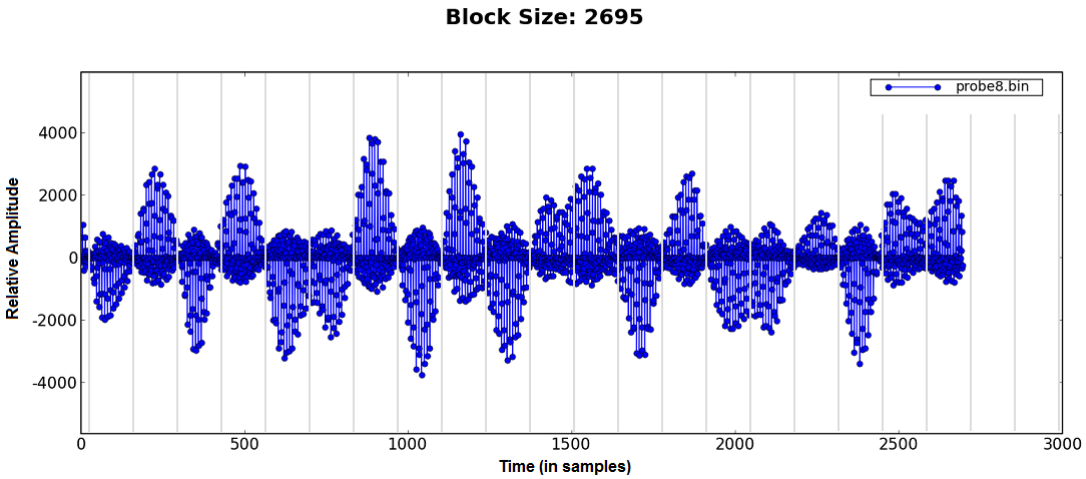

Figure 13: This plot shows the signal after the mixer. The mixer combines the filtered RDS signal with three times the pilot tone. This plot also shows 2695 samples so there are still 20 symbols or 10 biphase symbols recognisable. Also notice that the values on the x-axis are bigger in contradiction to the plot shown in figure 12. Looking at this plot, we can assume the signal is correct.

After the mixer, the signal still holds a high frequency signal. This is shown in figure 12 and figure 13. To create a clear signal as shown in figure 14 the RDS BB (Base Band) filter is implemented.

The RDS BB filter is a generic GNU Radio signal processing block and functions as a low pass filter. This filter reduces the amplitude of signals above the 1.5 kHz. This creates a more clear signal, as can been seen in figure 14

To verify if the filtered signal is correct, we need to show that the signal still represents 20 symbols or 10 biphase symbols and does not hold the high frequencies any more. This is done by visualising the signal after the RDS BB filter, with a block size of 2695 samples. This is shown in figure 14. In the plot can be seen that the signal does not hold the high frequencies any more and there are still 20 symbols or 10 biphase symbols recognisable. This means we can assume that the signal is correct.

Figure 15 shows the relation between the signal before the RDS BB filter and after the RDS BB filter.

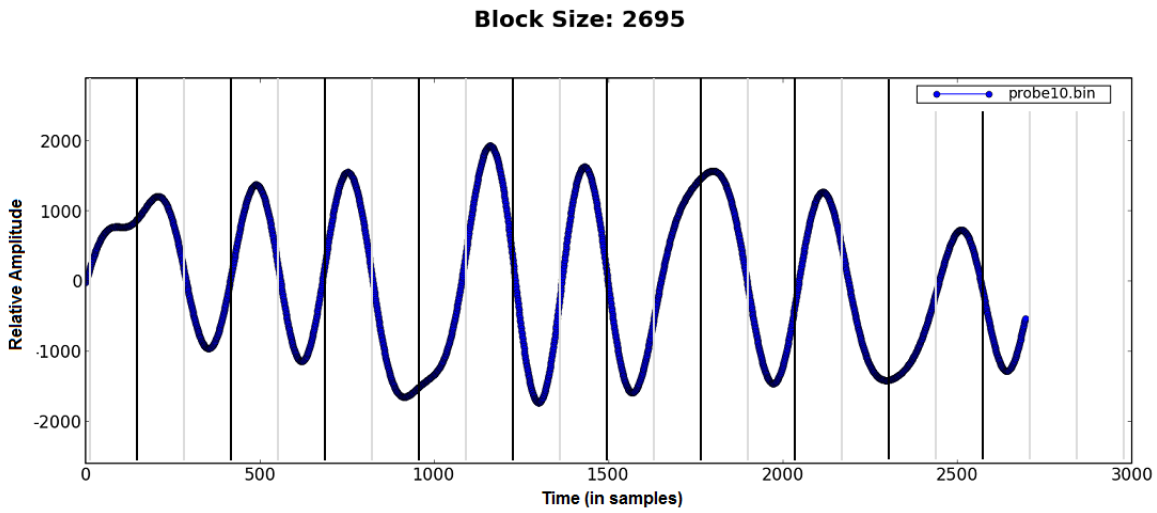

Figure 14: This plot shows the signal after the RDS BB Filter. The RDS BB Filter should reduces the amplitude of signals above the 1.5 kHz. The plot holds a sample block of 2695 samples so there should still 20 symbols or 10 biphase symbols recognisable. In the plot we can see that the high frequencies are filtered out of the signal, and that we still can recognise the symbols. This means we can assume that the signal is correct.

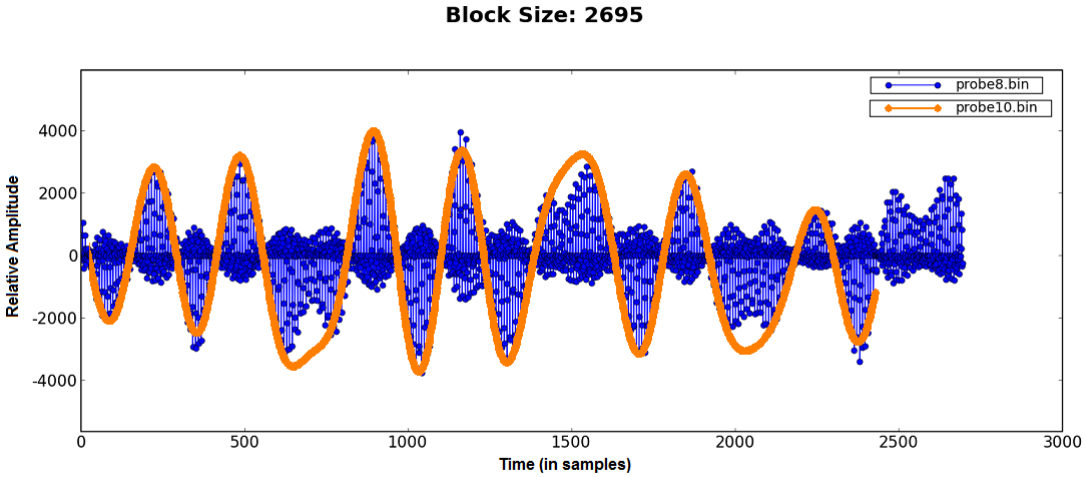

Figure 15: This plot shows the relation between the RDS signal before the RDS BB filter (blue) and the same signal after the (red) RDS BB filter. In this plot can be seen that the high frequency signals are filtered.

Now the RDS signal should be completely recovered and filtered from the FM signal. The next step is to translated the analog signal to a digital signal. How this should be done is described in the protocol specifications [41]. To do this, a biphase decoder is used.

The biphase decoder is a custom signal processing block. It has the function to retrieve the bits from the biphase symbols. It does this by aligning the data signal with the clock signal in such a way that the zero crossings of the data signal are in sync with the clock signal. When the signals are synced the RDS biphase symbols are translated to binary data. Figure 16 shows how the symbols are translated to a logical "1" or "0". Figure 17 shows how the symbols correspond to the reconstructed and filtered signal.

To verify if the data after the biphase decoder is correct, the signal after the biphase decoder can be visualised and matched to the manual recovered bits. The signal after the biphase decoder is shown in figure 18.

Plotting of the binary signal (figure 18) does not prove that the retrieved information is correct, because there is no knowledge about the transmitted bits. To determine of the bits are correct, the remaining signal processing blocks need to be implemented so the bits are converted to readable characters. Because the biphase decoder does synchronise both signals and outputs binary data we can assume that the stream is correct.

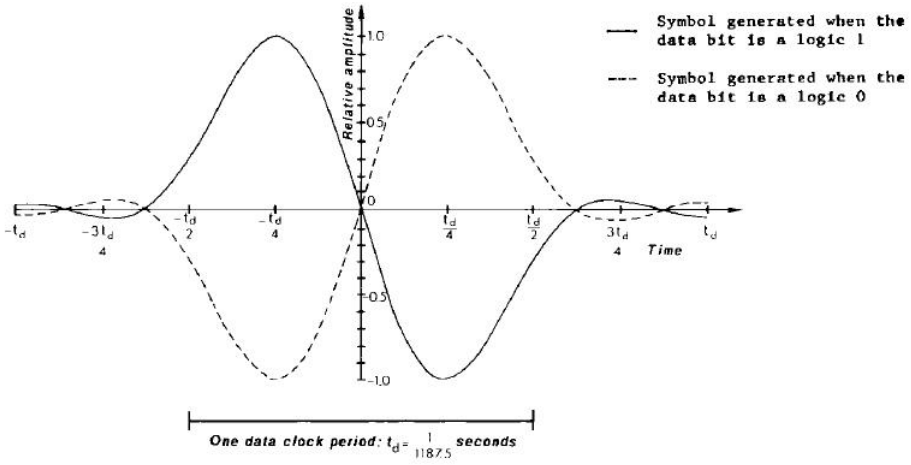

Figure 16: This figures shows how the symbols correspond to a logical " 1 " or " 0 ". Source : [41]

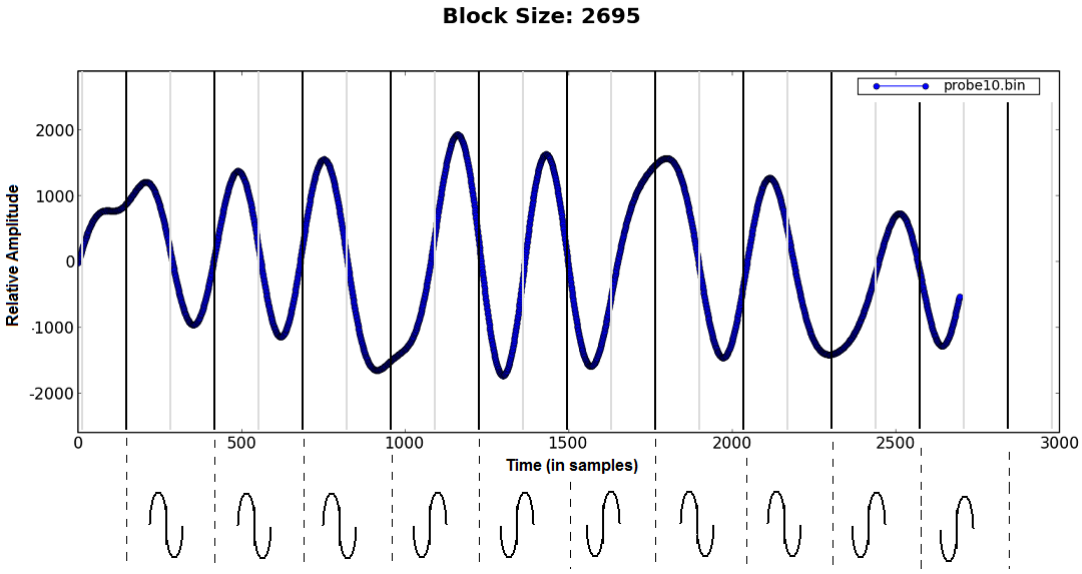

Figure 17: This plot shows the signal after the RDS BB Filter. The reconstructed symbols are shown under the plot. From those symbols the logical "1" or "0" can be distracted. The output of the biphase decoder should be : 1, 1, 1, 0, 0, 0, 1, 1, 0

Assuming that the biphase decoder outputs the correct bits the stream is still

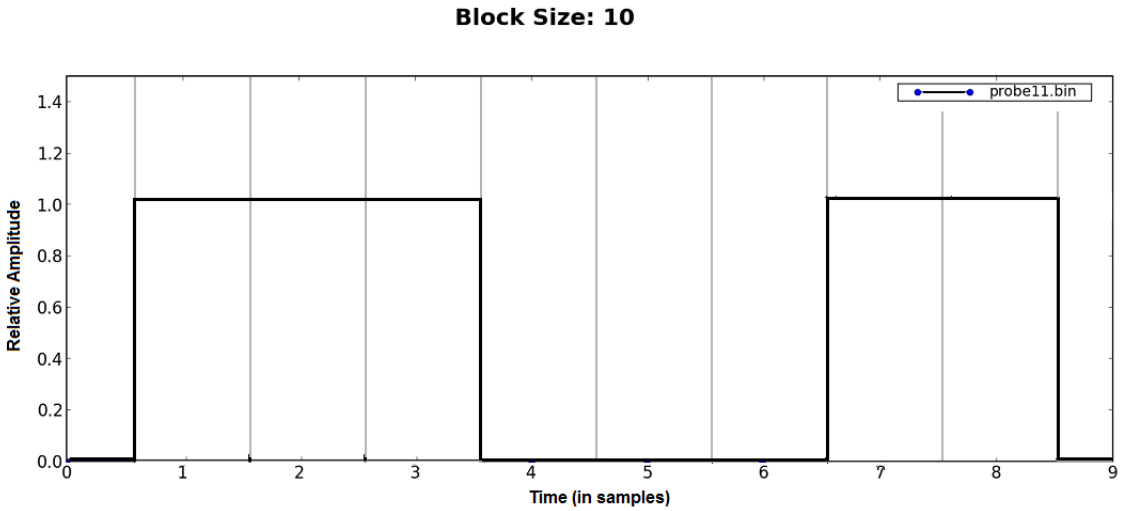

Figure 18: This figures shows the signal after the biphase decoder. The plot shows the translated bits. The biphase decoder should output digital data, which it clearly does.

differentially encoded, as described in the RDS protocol specifications [41]. This is done for robustness of the protocol. Differentially encoding means that the difference between two received bits is determines what the data bit represents.

The differential decoder is a custom signal processing block. It has the function to determine the differential between the previous and the current bit. The output of this signal processing block should hold the originally transmitted data. Figure 19 shows the digital signal after the differential decoder.

To verify the signal, the output of the biphase decoder needs to be translated by the use of the decoding rules described in the RDS protocol specifications [41]. The table (from [41]), shows those decoding rules.

Previous input	New input	New output
0	0	0
0	1	1
1	0	1
1	1	0

To check of the encoding did work correctly we can perform the encoding manually, with the values we take from figure 18. The result is shown in the table below.

Previous input	New input	Result
1	1	0
1	1	0
1	0	1
0	0	0
0	0	0
0	1	1
1	1	0
1	0	1
0	0	0

The above table shows that the output of the differential decoder must be a stream of bits of *0,0,1,0,0,1,0,1,0*. This corresponds to the signal visualised after the differential decoder, shown in figure 19. This means that we can assume that the differential encoding is done properly.

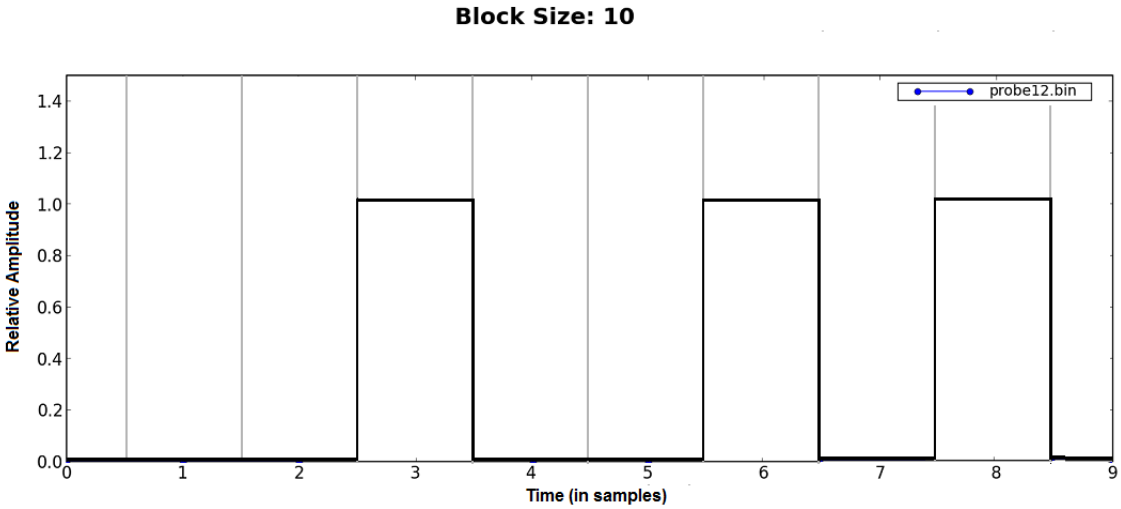

Figure 19: This figure shows the signal after the differential decoder. It should be a digital signal and correspond to the manual retrieved bits.

Assuming that all the previous signal processing blocks function correct, then the received and decoded bits can be converted into readable characters. Converting to characters is done by the RDS decoder signal processing block, as can be seen in figure 7.

The RDS decoder is a custom signal processing block, which implements the higher layer functionality of the RDS protocol. The RDS decoder block implements amongst other functionality, message discovery, error detection and also translation of the bits to readable characters. To verify that the data after the RDS Decoder is correct, we need to read the outputted characters and decide whether this text make sense or not.

If we look at the output of the RDS decode block, shown in figure 20, we see the station name of the FM broadcast station. This is the finale step of the RDS analysis and, we can assume that the received and captured RDS signal is correct.

```
usrp_decim = 200
chanfilt_decim = 1
audio_decimation = 10
adc_rate = 64000000
usrp_rate = 320000
demod_rate = 320000
audio_rate = 32000
usrp_rate = 320000
>>> gr_fir_ccf: using 3DNow!
>>> gr_fir_fff: using 3DNow!
>>> biphase decoder enter_looking
>>> biphase decoder enter_locked
Sync State Detected
==>  YR     <== -TP-  -Music-STEREO-Pop Music
==>  YRAD   <== -TP-  -Music-STEREO-Pop Music
==>  YRADIO <== -TP-  -Music-STEREO-Pop Music
==>SKYRADIO <== -TP-  -Music-STEREO-Pop Music
```

Figure 20: The RDS Decoder, decodes the signal and converts it to readable information.

5 Considerations

Because the research period spanned only two weeks, the scope of the research was limited to the analysis of a single wireless protocol. Since the analysed protocol is relative simple it can not be guaranteed that the created approach is the most optimal approach for all protocols. This research could have been done much faster and more in depth if there was more knowledge of Python and C++ available in the research. In the analysis of the RDS protocol there is assumed that the retrieved data is correct, because the application layer outputs readable characters. This does not prove that the received data is correct, because of implemented error correction (etc.) in the data link layer. Furthermore having to touch the *trunk* version instead of a stable release of GNU Radio was time consuming. As an System and Network Engineer it was hard to understand some specific radio characteristics because there was not much knowledge available in the research about radio technologies, like a physical scientist or a radio technician would have.

6 Conclusion

To fulfil a wireless protocol analyses the approach defined in chapter 4 can be followed, because it is possible to analyse a wireless protocol with the approach.

The USRP device works as described in section 3.2 and is suitable to use in a RDS protocol analysis. It may not be possible to analyse wireless protocols because of the limitations also mentioned in section 3.2.

Writing your own code is possible, but with the knowledge available at an average System and Network Engineer it is not easy to write your own signal processing code in C++, but lots of decoding blocks are available as mentioned in section 3.4. Gluing all the signal processing parts together in GRC is simple, but even writing the code in Python is not difficult.

To use GNU Radio in a wireless protocol analysis the System and Network Engineer must have basic knowledge about RF signals and how SDR are designed.

GNU Radio software in combination with the USRP hardware creates a good tool to perform wireless protocol analysis. This because it creates the possibility to analyse all the steps done in the physical layer of the protocol as shown in paragraph 4.7.

GNU Radio software includes already a lot of example scripts to perform a wireless protocol analysis. In this research also custom scripts (see appendix 9.3 and appendix 9.4), based on GNU Radio scripts are created. The RDS decode application, see appendix 9.5, is a rewrite of an already existing RDS decode application.

7 Future Work

This research was limited to analysing just one simple protocol. Applying the created general approach to multiple protocols would prove that the approach works for different protocols. The research was also limited to only receiving signals. Investigating what the added value of transmitting would be in protocol analysis would be a good extending of this research.

The defined approach is created for analysing protocols of which the specifications are available. The experience gathered in this research, with the specifications available, tell that without available specifications (reverse engineering) an analysis will be very difficult. To fulfil such an analysis there must be a lot of GNU Radio and radio experience present. Although this approach can not be used for such an analysis, the steps of capturing sample data an visualising signals can be helpful. Defining an general approach for such analysis can be a challenge.

In the protocol analysis of RDS, only the physical layer is addressed. Researching the upper layers can still be interesting. What could be investigated is: how many bits are reconstructed by the error correction, and how does this match to other radio broadcast stations?

8 Bibliography

References

[1] ADC explained. Website. http://gnuradio.org/trac/wiki/UsrpFAQ/Intro/ADC.

[2] Available miscellaneous daugtherboards. Website. http://www.ettus.com/downloads/miscdboards_v3b.pdf.

[3] Available transeiver daugtherboards. Website. http://www.ettus.com/downloads/transceiver_dbrds_v3b.pdf.

[4] Bluetooth. Website. https://www.bluetooth.org/apps/content/.

[5] C++. Website. http://www.cplusplus.com/doc/tutorial/.

[6] DAC explained. Website. http://gnuradio.org/trac/wiki/UsrpFAQ/Intro/DAC.

[7] Data link layer. Website. http://en.wikipedia.org/wiki/Data_link_layer.

[8] Dial-tone example explained. Website. http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html.

[9] Discuss-gnuradio mailing list. Website. http://www.mail-archive.com/discuss-gnuradio@gnu.org/msg04587.html.

[10] Error correction. Website. http://en.wikipedia.org/wiki/Error_correction.

[11] European Broadcasting Union. Website. http://www.ebu.ch/.

[12] Firmware. Website. http://en.wikipedia.org/wiki/Firmware.

[13] FM spectrum explained. Website. http://radioforum.nl/viewtopic.php?start=240&t=11566.

[14] FPGA explained. Website. http://www.nd.edu/~jnl/sdr/docs/tutorials/4.html.

[15] FPGA introduction. Website. http://www.gnuradio.org/trac/wiki/UsrpFAQ/Intro/FPGA.

[16] GNU Radio Companion. Website. http://www.joshknows.com/?key=grc.

[17] GNU Radio RDS signal blocks. Website. http://digilander.libero.it/iz2eeq/#rds.

[18] HDTV reception. Website. http://www.mail-archive.com/discuss-gnuradio@gnu.org/sg01926.html.

[19] Nyquist theorem. Website. http://www.cs.cf.ac.uk/Dave/Multimedia/node149.html.

[20] PGA explained. Website. http://www.nd.edu/~jnl/sdr/docs/tutorials/4.html.

[21] Physical layer. Website. http://en.wikipedia.org/wiki/Physical_layer.

[22] Python. Website. http://www.python.org/.

[23] Radio waves explained. Website. www.wisegeek.com/what-are-radio-waves.htm.

[24] RF-world explained. Website. http://en.wikipedia.org/wiki/Radio_frequency.

[25] Root cause analysis. Website. http://en.wikipedia.org/wiki/Root_cause.

[26] Sampling explained. Website. http://en.wikipedia.org/wiki/Sampling_rate.

[27] SDR explained. Website. http://en.wikipedia.org/wiki/Software_defined_radio.

[28] SDR forum. Website. http://www.sdrforum.org/.

[29] SVN. Website. http://subversion.tigris.org/.

[30] SVN location GNU Radio development version. Website. http://gnuradio.org/svn/gnuradio/trunk.

[31] SWIG explained. Website. http://www.swig.org/.

[32] Ubuntu. Website. http://www.ubuntu.com/.

[33] USB data speed. Website. http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html.

[34] USB explained. Website. http://www.olifantasia.com/gnuradio/usrp/files/usrp_guide.pdf.

[35] Wi-Fi. Website. http://en.wikipedia.org/wiki/Wifi.

[36] Wrapper explained. Website. http://en.wikipedia.org/wiki/Wrapper_pattern.

[37] XML explained. Website. http://www.w3.org/XML/.

[38] Philip Balister and Jeffrey H. Reed. Usrp hardware and software description. 2006. http://www.ece.vt.edu/swe/chamrad/crdocs/CRTM09_060727_USRP.pdf.

[39] Eric Blossom. Listening to FM radio in software, step by step. 2004. http://www.linuxjournal.com/article/7505.

[40] Prateek Mohan Dayal. Software based radio signal processing. 2004. http://www.geocities.com/pmd_iitgw/second.pdf.

[41] RDS Forum. The new RDS IEC 62106:1999 standard. Website. http://www.rds.org.uk/rds98/pdf/IEC%2062106-E_no%20print.pdf.

[42] Free Software Foundation. GNU Radio. Website. http://gnuradio.org/trac.

[43] Firas Abbas Hamza. The USRP under 1.5x magnifying lens! 2008. http://gnuradio.org/trac/attachment/wiki/UsrpFAQ/USRP_Documentation.pdf.

[44] Ketan Mandke. Early Results on Hydra: A Flexible MAC/PHY Multihop Testbed. 2007. http://users.ece.utexas.edu/~rheath/papers/2007/vtc1/paper.pdf.

[45] Lee K. Patton. A GNU Radio based software-defined radar. 2007. http://www.pattoncentral.org/download/patton_msthesis.pdf.

[46] David A. Scaperoth. Configurable SDR operation for cognitive radio applications using GNU Radio and the universal software radio peripheral. 2007. http://scholar.lib.vt.edu/theses/available/etd-05182007-235204/unrestricted/01thesis_whole5.pdf.

[47] Thomas Schmid and Mani B. Srivastava. An experimental study of network performance impact of increased latency in software defined radios. 2007. http://www.winlab.rutgers.edu/~wenyuan/papers/Seville.pdf.

[48] Dawei Shen. How to write a signal processing block. 2008. http://www.nd.edu/~jnl/sdr/docs/tutorials/10.pdf.

[49] Dominic Spill and Andrea Bittau. Bluesniff: Eve meets Alice and Bluetooth. 2007. http://www.usenix.org/event/woot07/tech/full_papers/spill/spill.pdf.

[50] Holger von Malm Stefan Valentin and Holger Karl. Implementing physical and data link control layer on the GNU software-defined radio platform. 2005. http://typo3.cs.uni-paderborn.de/fileadmin/Informatik/AG-Karl/Pubs/vmalm05-sa-gsr_aloha.pdf.

[51] Holger von Malm Stefan Valentin and Holger Karl. Evaluating the GNU software radio platform for wireless testbeds. 2006. http://www.cs.uni-paderborn.de/fileadmin/Informatik/AG-Karl/Pubs/TR-RI-06-273-gnuradio_testbed.pdf.

[52] Kalen Watermeyer. Design of a hardware platform for narrow-band software defined radio applications. 2006. http://rrsg.ee.uct.ac.za/theses/msc_theses/kwatermeyer_thesis.pdf.

[53] Rob Miller Zang Li, Wenyuan Xu and Wade Trappe. Securing wireless systems via lower layer enforcements. 2006. http://nesl.ee.ucla.edu/fw/thomas/wintech401-schmid.pdf.

9 Appendix

9.1 GNU Radio installation

In this project Ubuntu 8 is used, the following package need to be installed to compile and install GNU Radio:

```
sudo apt-get -y install g++ automake1.9 libtool python-dev fftw3-dev
   libcppunit-dev libboost-dev sdcc libusb-dev libasound2-dev libsdl1.2-dev
   python-wxgtk2.6 subversion guile-1.6 libqt3-mt-dev python-numpy-ext swig
   portaudio19-dev jack gkrellm wx-common libwxgtk2.6-dev alsa-base autoconf
   xorg-dev g77 gawk bison openssh-server emacs cvs usbview octave
```

The next code repairs the ability to test code and libraries before installing them

```
cp /etc/ld.so.conf /tmp/ld.so.conf
echo /usr/local/lib >> /tmp/ld.so.conf
sudo mv /tmp/ld.so.conf /etc/ld.so.conf
```

Download the source code, configure it and make it:

```
svn co http://gnuradio.org/svn/gnuradio/trunk gnuradio
cd gnuradio
./bootstrap
./configure
make
```

After the compiling has been completed, check to ensure that every component was correctly configured:

```
make check
```

If the check was correct, install the software:

```
sudo make install
```

After the installation one more step must be done to ensure that GNU Radio will operate correctly. Ubuntu uses udev to handle hotplug devices, and does not give by default non-root access to the USRP. That is why a group must be made to handle the USRP via USB, the steps taken were:

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

Now that the installation process has been completed successfully, it is a good idea to restart the computer before connecting the USRP to ensure that everything loads correctly.
After this is done tests can be done with USRP connected to the computer to ensure that everything is working fine.

```
ls -lR /dev/bus/usb | grep usrp
```

The steps given above were based on the instructions on: http://gnuradio.org/trac/wiki/UbuntuInstall.

9.2 GRC Dial Tone

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: untitled
# Author: unknown
# Description: gnuradio flow graph
# Generated: Wed Jun 25 11:54:47 2008
##################################################

from gnuradio import audio
from gnuradio import gr

tb = gr.top_block()

##################################################
# Callbacks
##################################################
def _set_samp_rate(_samp_rate):
global samp_rate
samp_rate = _samp_rate
gr_sig_source_x.set_sampling_freq(samp_rate)
gr_sig_source_x0.set_sampling_freq(samp_rate)

##################################################
# Variables
##################################################
samp_rate = 32000

##################################################
# Blocks
##################################################
audio_sink = audio.sink(32000, "", True)
gr_sig_source_x = gr.sig_source_f(samp_rate, gr.GR_SIN_WAVE, 400, 1, 0)
gr_sig_source_x0 = gr.sig_source_f(samp_rate, gr.GR_SIN_WAVE, 350, 1, 0)

##################################################
# Connections
##################################################
tb.connect((gr_sig_source_x0, 0), (audio_sink, 0))
tb.connect((gr_sig_source_x, 0), (audio_sink, 1))

tb.start()
raw_input('Press Enter to quit: ')
tb.stop()
```

9.3 RDS Capture

```
##################################################
#
# Author: Alex verduin,
# Credits : This script is based on example code provided with GNU Radio
# Description: Captures samples from the USRP and writs them to disk
# Generated: Wed Jun 25 11:54:47 2008
##################################################
#!/usr/bin/env python

# write to disk

#>>> bring in blocks from the main gnu radio package
from gnuradio import gr
from gnuradio import usrp
limit=1000000
u = usrp.source_c()                     # usrp is data source
freq=101.21
gain=50
adc_rate = u.adc_rate()                 # 64 MS/s
usrp_decim = 200
u.set_decim_rate(usrp_decim)
usrp_rate = adc_rate / usrp_decim           # 320 kS/s
chanfilt_decim = 1
demod_rate = usrp_rate / chanfilt_decim

subdev_spec = usrp.pick_rx_subdevice(u)
u.set_mux(usrp.determine_rx_mux_value(u, subdev_spec))
subdev = usrp.selected_subdev(u, subdev_spec)
subdev.set_gain(gain)

u.tune(0, subdev, freq)

#>>> create the flow graph
tb = gr.top_block()
skip=gr.skiphead(gr.sizeof_gr_complex, limit)
head=gr.head(gr.sizeof_gr_complex, limit)
```

```
sink=gr.file_sink (gr.sizeof_gr_complex, "input.bin")

tb.connect(u,skip)
tb.connect(skip,head)
tb.connect(head,sink)

tb.run()
```

9.4 RDS Sample converter

```
##################################################
#
# Author: Alex verduin,
# Credits : This script is based on example code provided with GNU Radio
# Description: Takes a configured domain of samples out of a bigger sample file
# Generated: Wed Jun 25 11:54:47 2008
##################################################
#!/usr/bin/env python

# write to disk

#>>> bring in blocks from the main gnu radio package
from gnuradio import gr
limit=500000 # Number of recorded samples
skipnumber = 20000000 # Skip the first number of samples

u = gr.file_source (gr.sizeof_gr_complex, "input.bin", 0) # Input is a file
tb = gr.top_block() # Create the top block
skip=gr.skiphead(gr.sizeof_gr_complex, skipnumber) # create a skip block
head=gr.head(gr.sizeof_gr_complex, limit) # Create a head block
sink=gr.file_sink (gr.sizeof_gr_complex, "output.bin") # The output is a file

tb.connect(u,skip) #Connect the source to the skip
tb.connect(skip,head) #Connect the skip to the head
tb.connect(head,sink) #connect the head to the sink

tb.run() #Run the graph
```

9.5 RDS Decode application

```
##################################################
#
# Author: Alex verduin,
# Credits : This script is a rewrite of code from http://digilander.libero.it/iz2eeq/#rds
# Description: Decodes RDS FM broadcast.
# Generated: Wed Jun 25 11:54:47 2008
##################################################
#!/usr/bin/env python

from gnuradio import gr, gru
from gnuradio import rds
from gnuradio import optfir
from gnuradio import blks2
from gnuradio import audio
from gnuradio import eng_notation
import sys
import math
import wx

level= int(sys.argv[1])
usrp_decim = 200 #usrp_decim = 200
chanfilt_decim = 1 #chanfilt_decim = 1
audio_decimation = 10  #audio_decimation = 10
adc_rate = 64000000 #adc_rate = 64000000
usrp_rate = adc_rate / usrp_decim  #usrp_rate = 320000
demod_rate = usrp_rate / chanfilt_decim #demod_rate = 320000
audio_rate = demod_rate / audio_decimation #audio_rate = 32000
usrp_rate = adc_rate / usrp_decim   #usrp_rate = 320000


print "Debug level = " +str(level)

#Print all the variables
print"usrp_decim = " +str(usrp_decim)
#print"chanfilt_decim = " +str(chanfilt_decim)
#print"audio_decimation = " +str(audio_decimation)
print"adc_rate = " +str(adc_rate)
#print"usrp_rate = " +str(usrp_rate)
#print"demod_rate = " +str(demod_rate)
#print"audio_rate = " +str(audio_rate)
#print"usrp_rate = " +str(usrp_rate)


tb = gr.top_block() # create the flow graph

file = gr.file_source (gr.sizeof_gr_complex, "RDSSampleSmall.bin", 0)# Recorded sample with RDS (sample) as input file. 0 =! non loop , 1 == loop behavior

#sink=gr.file_sink (gr.sizeof_gr_complex, "output"+str(level)+".bin")#Write the data to file. Datatype complex
#probe=gr.file_sink (gr.sizeof_float, "probe"+str(level)+".bin") #Write the data to file. Datatype float
audio_sink = audio.sink(audio_rate,'plughw:0,0', False) # Create an audio output with a audio rate of 32K


#Channel Filter cuts off al frequnties above 80 kHz
```

```
chan_filt_coeffs = optfir.low_pass (1,           # gain
                                    usrp_rate,   # sampling rate
                                       80e3,        # passband cutoff
                                       115e3,       # stopband cutoff
                                       0.1,         # passband ripple
                                       60)          # stopband attenuation
chan_filt = gr.fir_filter_ccf (chanfilt_decim, chan_filt_coeffs)


#FM-demodulator. Hierarchical block for demodulating a
#broadcast FM signal. The input is the down converted
#complex baseband signal.The output is the demodulated audio (float).
#demod_rate: input sample rate of complex baseband input.
#audio_decimation :much to decimate quad_rate to get to audio.
guts = blks2.wfm_rcv (demod_rate, audio_decimation)


#FM-Filter cuts off al frequnties above 70 kHz
coeffs = gr.firdes.low_pass (100.0,              # gain
                          demod_rate,       # sampling rate
                              70e3,# cutoff_freq
                              10e3,# transition_width
                              gr.firdes.WIN_HAMMING) #
fm_filter = gr.fir_filter_fff (1, coeffs)

#Piloot filter only let the frequenties trough between 18 kHz and 20 kHz
pilot_filter_coeffs = gr.firdes_band_pass(1,                      #
  demod_rate,         #sample rate
  18e3,   #center of low transition band
  20e3, #center of high transition band
  3e3, #transition_width
  gr.firdes.WIN_HAMMING) #
pilot_filter = gr.fir_filter_fff(1, pilot_filter_coeffs)

#RDS Filter only let the frequenties trough between 54 kHz and 60 kHz
rds_filter_coeffs = gr.firdes.band_pass (1,
                                         demod_rate,#sample rate
                                         54e3,#center of low transition band
                                         60e3,#center of high transition band
                                         3e3,#transition_width
                                         gr.firdes.WIN_HAMMING)#
rds_filter = gr.fir_filter_fff (1 , rds_filter_coeffs)

#RDS Data filter only let the frequenties lower then 1.5 kHz trough
rds_bb_filter_coeffs =  gr.firdes.low_pass (5,  # Gain
                                    demod_rate,# sampling rate
    1500,  # Cut off
    2000,  # transition_width
    gr.firdes.WIN_HAMMING) # window type
rds_bb_filter = gr.fir_filter_fff (1, rds_bb_filter_coeffs)

#Mixer adds the frequenties
mixer = gr.multiply_ff()

# Data clock reconstruction. It takes the 19kHz Pilot tone and use it to construct a clock signal
data_clock = rds.freq_divider(16)

# Biphase decoder
biphase_decoder = rds.biphase_decoder(audio_rate)

# Differential decoder
differential_decoder = rds.diff_decoder()

# RDS data decoder
msgq = gr.msg_queue()
rds_decoder = rds.data_decoder(msgq)


######
###Here all the blocks are connected
#####

tb.connect(file, chan_filt)
tb.connect(chan_filt, guts)
tb.connect(guts, audio_sink)
tb.connect(guts.fm_demod, fm_filter )
tb.connect(fm_filter, pilot_filter)
tb.connect(fm_filter, rds_filter)
tb.connect(pilot_filter, (mixer, 0))
tb.connect(pilot_filter, (mixer, 1))
tb.connect(pilot_filter, (mixer, 2))
tb.connect(rds_filter, (mixer, 3))
tb.connect(pilot_filter, data_clock)
tb.connect(mixer, rds_bb_filter)
tb.connect(rds_bb_filter, (biphase_decoder, 0))
tb.connect(data_clock, (biphase_decoder, 1))
tb.connect(biphase_decoder, differential_decoder)
#tb.connect(differential_decoder, rds_decoder)

#These connects can be used to create sample files.


if level == 2:
probe2=gr.file_sink (gr.sizeof_gr_complex, "probe"+str(2)+".bin") #Write the data to file. Datatype float
tb.connect(chan_filt, probe2)


if level == 3:
probe3=gr.file_sink (gr.sizeof_float, "probe"+str(3)+".bin") #Write the data to file. Datatype float
```

```
tb.connect(guts.fm_demod, probe3)

if level == 5:
probe5=gr.file_sink (gr.sizeof_float, "probe"+str(5)+".bin") #Write the data to file. Datatype float
tb.connect(fm_filter, probe5)

if level == 6:
probe6=gr.file_sink (gr.sizeof_float, "probe"+str(6)+".bin") #Write the data to file. Datatype float
tb.connect(pilot_filter, probe6)

if level == 7:
probe7=gr.file_sink (gr.sizeof_float, "probe"+str(7)+".bin") #Write the data to file. Datatype float
tb.connect(rds_filter, probe7)


if level == 8:
probe8=gr.file_sink (gr.sizeof_float, "probe"+str(8)+".bin") #Write the data to file. Datatype float
tb.connect(mixer, probe8)

if level == 9:
probe9=gr.file_sink (gr.sizeof_float, "probe"+str(9)+".bin") #Write the data to file. Datatype float
tb.connect(data_clock, probe9)

if level == 10:
probe10=gr.file_sink (gr.sizeof_float, "probe"+str(10)+".bin") #Write the data to file. Datatype complex
tb.connect(rds_bb_filter, probe10)

if level == 11:
probe11=gr.file_sink (gr.sizeof_float, "probe"+str(11)+".bin") #Write the data to file. Datatype complex
tb.connect(biphase_decoder, probe11)

if level == 12:
probe12=gr.file_sink (gr.sizeof_float, "probe"+str(12)+".bin") #Write the data to file. Datatype complex
tb.connect(differential_decoder, probe12)


tb.run() # Execute the graph
```

9.6 Example C++ code

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

#include <gr_rds_freq_divider.h>
#include <gr_io_signature.h>

gr_rds_freq_divider_sptr
gr_rds_make_freq_divider (int divider)
{
  return gr_rds_freq_divider_sptr (new gr_rds_freq_divider (divider));
}
static const int MIN_IN = 1; // mininum number of input streams
static const int MAX_IN = 1; // maximum number of input streams
static const int MIN_OUT = 1; // minimum number of output streams
static const int MAX_OUT = 1; // maximum number of output streams

gr_rds_freq_divider::gr_rds_freq_divider (int divider)
  : gr_sync_block ("gr_rds_freq_divider",
      gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
      gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
d_divider = 0;
DIVIDER = divider;
d_sign_last = d_sign_current = false;
d_out = 1;
}
```

```
/*
 * Our virtual destructor.
 */
gr_rds_freq_divider::~gr_rds_freq_divider ()
{
  // nothing else required in this example
}
int
gr_rds_freq_divider::work (int noutput_items,
       gr_vector_const_void_star &input_items,
       gr_vector_void_star &output_items)
{
  const float *in = (const float *) input_items[0];
  float *out = (float *) output_items[0];
  for (int i = 0; i < noutput_items; i++){
 d_sign_current = (in[i] > 0 ? true : false);
 if(d_sign_current != d_sign_last) {
 // A  zero cross
 if(++d_divider ==  DIVIDER) {
 d_out *= -1;
 d_divider = 0;
 }
 }
 out[i] = d_out;
 d_sign_last = d_sign_current;
  }
  // Tell runtime system how many output items we produced.
  return noutput_items;
```