Universität Paderborn — Fakultät EIM-I
Fachgebiet Rechnernetze

# Implementing physical and data link control layer on the GNU software-defined radio platform

**Studienarbeit**
am Fachgebiet Rechnernetze
Prof. Dr. H. Karl

Institut für Informatik
Fakultät für Elektrotechnik,
Informatik und Mathematik
Universität Paderborn

vorgelegt von
**Holger von Malm**
am
02.12.2005

Betreuer: Prof. Dr. H. Karl
Stefan Valentin M.A.

Holger von Malm
Matrikelnummer: 6066676
Marblicksweg 63
59555 Lippstadt

# Abstract

In a traditional wireless communication system many functions are implemented in hardware. These functions are normally unchangeable after they are implemented. A Software-defined Radio (SDR) is a radio communication system which uses software for the modulation and demodulation of radio signals. The flexibility achieved by this approach can ideally be used in industry and research for rapid prototyping of radio systems, because software development is cheaper and less time-consuming than hardware development. In this student research project an SDR is analyzed, that is based on the SDR development platform GNU Radio and the Universal Software Radio Peripheral (USRP). We evaluate, whether performance of this specific SDR is sufficient for prototyping of wireless communication systems. Therefore, we implement basic functions for wireless communication, consisting of physical functions (e.g. modulation, bit synchronization), functions to provide reliable data transfer and a simple MAC protocol. The performance of this system is measured in terms of latency. The results imply that the GNU Radio/USRP platform can be used for systems which communicate with a low data rate. As GNU Radio has a clear modular concept, it is suitable for rapid prototyping of wireless PHY and MAC functions.

# Zusammenfassung

In herkömmlichen drahtlosen Kommunikationssystemen werden viele Funktionen in Hardware implementiert. I.d.R. sind diese Funktionen nach der Implementierung nicht mehr änderbar. Ein Software-defined Radio (SDR) ist ein Kommunikationssystem, welches mit Hilfe von Software Funksignale erzeugen und demodulieren kann. Die Flexibilität, die durch diesen Ansatz erreicht wird, kann in der Industrie und Forschung ideal für die Prototypenfertigung von Funksystemen genutzt werden, da Softwareentwicklung weitaus weniger Geld und Zeit kostet als die Entwicklung von Hardware. In dieser Studienarbeit wird ein SDR untersucht, welches auf der SDR Entwicklungsplattform GNU Radio und dem Universal Software Radio Peripheral (USRP) basiert. Hierbei wird analysiert, ob die Leistung dieses SDR ausreicht um Prototypen für drahtlose Kommunikationssysteme herzustellen. Zu diesem Zweck wurden Basisfunktionen der drahtlosen Kommunikation auf der GNU Radio/USRP-Plattform implementiert. Diese setzten sich zusammen aus physischen Funktionen (z.B. Modulationsverfahren, Bitsynchronisation), Funktionen welche eine zuverlässige Datenübertragung garantieren, sowie einem einfachen MAC-Protokoll. Zur Leistungsbestimmung dieses Systems wurden Latenzzeitmessungen durchgeführt. Hieraus resultierte, dass die GNU Radio/USRP-Plattform für Kommunikationssysteme mit einer geringen Datenrate geeignet ist. Da GNU Radio ein übersichtliches modulares Konzept verfolgt, eignet es sich für die schnelle Prototypenentwicklung von physischen Funktionen und MAC-Protokollen.

# Contents


**1 Introduction** **1**

**2 Wireless communication using software-defined radios** **3**

2.1 Data communication . . . 3
2.2 Specifics of wireless communication . . . 5
2.3 Basic functions of wireless communication . . . 7
2.3.1 Modulation . . . 9
2.3.2 Synchronization . . . 10
2.3.3 Forward error correction: Convolution codes . . . 10
2.3.4 Error control by Cyclic Redundancy Check . . . 12
2.3.5 Framing . . . 12
2.3.6 Media Access Control (MAC) protocols . . . 13
2.4 Software-defined radios . . . 15
2.5 The GNU Radio SDR platform . . . 17
2.6 The Universal Software Radio Peripheral hardware front-end . . . 20

**3 Implementation of PHY/MAC functions on the GNU Radio platform** **23**

3.1 Send function . . . 23
3.1.1 Source vector . . . 24
3.1.2 Framer . . . 24
3.1.3 CRC encoder . . . 25
3.1.4 FEC encoder . . . 25
3.1.5 Preamble . . . 25
3.1.6 Non Return to Zero (NRZ) . . . 25
3.1.7 Interpolation . . . 26
3.1.8 Gaussian filter . . . 27
3.1.9 Frequency modulation . . . 28
3.1.10 Gain . . . 28
3.1.11 USRP TX . . . 29
3.2 Receiver function . . . 29
3.2.1 USRP RX . . . 29
3.2.2 Filter . . . 29
3.2.3 Demodulation . . . 29
3.2.4 Integrate filter . . . 30

3.2.5 Correlator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.2.6 FEC decoder (Viterbi decoder) . . . . . . . . . . . . . . . . . 31
3.2.7 CRC decoder . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.2.8 Extract frame . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.3 MAC protocols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**4 Performance study 33**
4.1 Experimental setup and parameterization . . . . . . . . . . . . . . . . 33
4.2 Measurement results for the round trip time . . . . . . . . . . . . . 35
4.3 Measurement results for the calculation time per signal block . . . . 36
4.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**5 Conclusion 41**

**A Tutorial of measurement 43**
A.1 How to measure a signal block . . . . . . . . . . . . . . . . . . . . . 43
A.2 How to measure round trip time . . . . . . . . . . . . . . . . . . . . 46

**B Acronyms 49**

**C Bibliography 52**

# List of Figures


2.1 Block diagram of a communication system . . . . . . . . . . . . . . . 3
2.2 Layers and protocols of the ISO/OSI model . . . . . . . . . . . . . 4
2.3 Wavelength and amplitude . . . . . . . . . . . . . . . . . . . . . . . . 5
2.4 Frequency range of wireless electromagnetic channels . . . . . . . . 6
2.5 Wireless Communication . . . . . . . . . . . . . . . . . . . . . . . . 7
2.6 Block diagram of a wireless communication system: sender . . . . . 7
2.7 Block diagram of a wireless communication system: receiver . . . . 8
2.8 Modulation schemes . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.9 Principle of a convolution encoder . . . . . . . . . . . . . . . . . . . 10
2.10 Implementation of a convolution encoder using shift registers . . . . 11
2.11 Byte stuffing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.12 The pure ALOHA protocol with multiple access . . . . . . . . . . . 14
2.13 Vulnerable periode of frames . . . . . . . . . . . . . . . . . . . . . . 14
2.14 Throughput of the pure ALOHA protocol . . . . . . . . . . . . . . . 15
2.15 Classic communication systems . . . . . . . . . . . . . . . . . . . . . 16
2.16 A software-defined radio . . . . . . . . . . . . . . . . . . . . . . . . 16
2.17 The basic architecture of an SDR . . . . . . . . . . . . . . . . . . . 17
2.18 GNU Radio signal block . . . . . . . . . . . . . . . . . . . . . . . . 18
2.19 The USRP main board with two TX and two RX daughter boards [5] 20

3.1 Signal graph of the sender . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 Architecture of the frame . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Architecture of the frame additional with CRC part . . . . . . . . . 25
3.4 Output signals of the NRZ block . . . . . . . . . . . . . . . . . . . . 26
3.5 The operation of a Finite Impulse Response (FIR) filter [8] . . . . . 26
3.6 Output signals of the Interpolation block . . . . . . . . . . . . . . . 27
3.7 Output signals of the Gaussian filter block . . . . . . . . . . . . . . 27
3.8 Output signals of the frequency modulation block . . . . . . . . . . 28
3.9 Output signals of the gain block . . . . . . . . . . . . . . . . . . . . 28
3.10 Signal graph of the receiver . . . . . . . . . . . . . . . . . . . . . . 29
3.11 Output signals of the demodulation block . . . . . . . . . . . . . . . 30
3.12 Output signals of the integrate filter block . . . . . . . . . . . . . . 30
3.13 Protocol of a sender and receiver terminal . . . . . . . . . . . . . . 32

4.1 Experimental setup of the communication system . . . . . . . . . . 34

4.2 Round trip time . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Histogram of the average round trip time (RTT) per bit . . . . . . 36
4.4 Measurements of the sender signal blocks. . . . . . . . . . . . . . . . 37
4.5 Measurements of the receiver signal blocks. . . . . . . . . . . . . . . . 39

# Chapter 1

# Introduction

In a traditional wireless communication system many functions are implemented in hardware, e.g. the modulation. They are normally unchangeable and define the functionality of the system. Many communication systems use different modulation schemes and protocols, so a communication is often only possible between homogeneously systems. To communicate with different wireless terminals a radio system is needed that can switch between modulation schemes and protocols. Software-defined Radio (SDR) is a technology that allows this. An SDR is a radio communication system which uses software for the modulation and demodulation of radio signals. If a special radio modulation is needed only the software has to be changed. Thus, SDRs are flexible and much more efficient than traditional wireless communication systems. This technology can ideally be used in industry and research for rapid prototyping, because software development is cheaper and less time-consuming than hardware development. SDRs can also be used in education to learn about SDR techniques. Students can develop own communication systems or can test own protocols and modulation schemes using an SDR platform.
In future the SDR technology can be used to build radios which can be programmed to cover all wireless standards (e.g. GSM, 3G, WLAN).

In this student research project we analyze an SDR, that is based one GNU Radio and the Universal Software Radio Peripheral (USRP). We evaluate, whether performance of this specific SDR is sufficient for prototyping of wireless communication systems. Therefore, we implement basic functions for wireless communication, consisting of physical functions (e.g. modulation), functions to provide reliable data transfer and a simple MAC protocol. The performance of this system is measured in terms of latency (e.g. round trip time).

This work is structured into five chapters. In Chapter 2 we introduce wireless communication systems. We discuss the characteristics of wireless communication, differences to wired communication and describe basic functions of a wireless communication system. At the end of this chapter SDR is introduced as a special approach in wireless communication systems and one specific implementation, the GNU Radio/USRP platform is closer described. Chapter 3 is focused on the imple-

mentation of our communication system. The experimental setup and the results of the measurements are described in Chapter 4. In Chapter 5 we summarize our work and mention some aspects of future work.

# Chapter 2

# Wireless communication using software-defined radios

This chapter gives a basic understanding of communication systems. We consider the structure, the physical background and the problems that arise when information is sent over a wireless channel. To get a closer look how a communication system works and what services and functions are needed, the ISO/OSI model is introduced. After discussing the basic of communication systems, we point out the characteristics of wireless communication. We get to know how wireless transmission works and what problems a wireless channel brings. To handle these problems special protocols are utilized. Therefore the pure ALOHA protocol will be introduced. At the end of this chapter SDR is discussed as a special platform for wireless communication. GNU Radio SDR framework and the USRP is one possible platform to implement SDRs and is considered in this chapter.

## 2.1 Data communication

Technical communication systems are developed to transmit messages from a sender to a receiver. The basic structure of a communication system can be represented by the block digram shown in Figure 2.1

Figure 2.1: Block diagram of a communication system

The communication chain has three basic parts, the transmitter, the channel and the receiver. The transmitter converts the incoming information from a source to an electrical signal that is suitable for transmission. These signals have to pass a physical channel (e.g. a cable) or a transmission medium (e.g. air) before they reach a receiver. At the receiver side the electrical signal has to be re-transformed to the original information.
Beside the task of sending and receiving information over a channel, there are many

other tasks a communication system has to do. One import task is to guarantee the correctness of the received information. No existing channel is completely error-free. In order to receive a correct information, redundancy is often sent. On the receiver side the redundancy is needed to correct errors which have occurred while transmission.
Because of the complexity a communication system with all its function has, we use a reference model. The Open System Interconnection (OSI) reference model divides a network system into different layers with different tasks. Each layer performs services requested by the lower or upper layer. Figure 2.2 illustrates the ISO/OSI model. The first layer in the model is the physical layer (PHY). This layer is the most basic network layer, providing the means of transmitting raw bits. Data link control (DLC) is the next higher layer. The main work of this layer is to provide reliable data transfer across the physical link. For a better handling of big quantities of information, the data gets packed into smaller units called frames. Another part of the DLC is the Media Access Control (MAC). This component controls the transmission of frames. These two layers provide basic functions for data transfer which are used by the upper layers and communication protocols. The upper layers provide functions for frame addressing, routing, transporting and communication with applications.

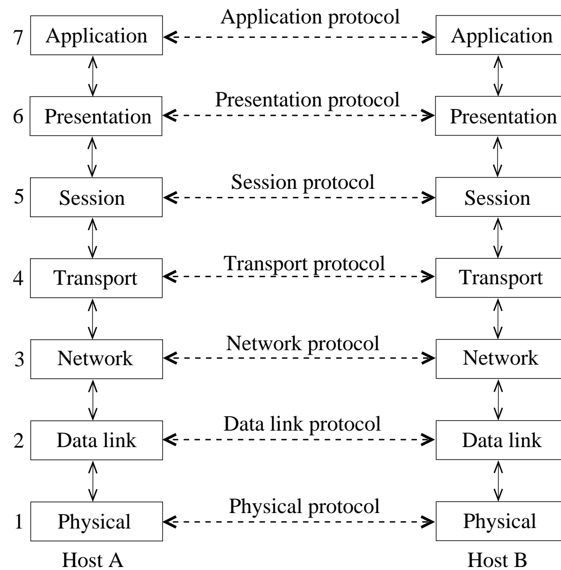



Figure 2.2: Layers and protocols of the ISO/OSI model

## 2.2 Specifics of wireless communication

Wireless communication employs electromagnetic waves for data transmission. A wave is characterized by the *frequency* $f$ and the *wavelength* $\lambda$. The frequency is the number of cycles of the repetitive waveform per second and is measured in Hertz. The length of a wave is the distance between repeating units of a wave pattern. *Amplitude* is the maximum deflection of a waves oscillation. The state of an oscillation in a certain moment and in a certain position is called *phase*. By changing the frequency, phase or amplitude of a wave information can be transmitted. This method is called modulation. In Figure 2.3 amplitude and wavelength are illustrated on a sinusoidal wave.

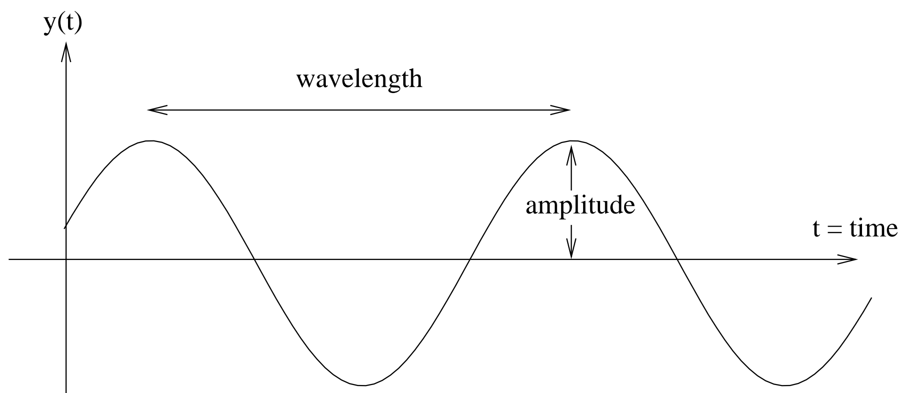



Figure 2.3: Wavelength and amplitude

In radio communication systems antennas radiate the electromagnetic waves. The design of antennas depends amongst others on the frequency of the wave. For an efficient radiation of the electromagnetic energy the antenna must be longer than $\frac{1}{10}$ of the wavelength [16]. Electromagnetic waves with a large wavelength have low frequencies and high frequencies have small wavelength. Figure 2.4 gives a overview of the frequency band. E.g. the IEEE 802.11b standard for wireless communication works in the 2.4 GHz frequency band. GSM mobile phones operates in two frequency bands, one at 900 MHz, and one at 1800 MHz.

Radio transmission is very different from wired transmission. Though both use electromagnetic waves, in a wired medium these waves are directed and are well protected against interferences. A wireless communication system radiates electromagnetic waves undirected. Also the wireless channel has no option to protect against interferences. With increasing distance a radio signal loses its power and so the range of the signal is limited. This problem is called *path-loss*. When electromagnetic waves in High Frequency (HF) range propagate a problem occurs called *multi-path propagation*. In this frequency range signals can be reflected by solid obstacles like walls or trees. Because of the reflecting signals the receiver may receive multiple copies of the signal.

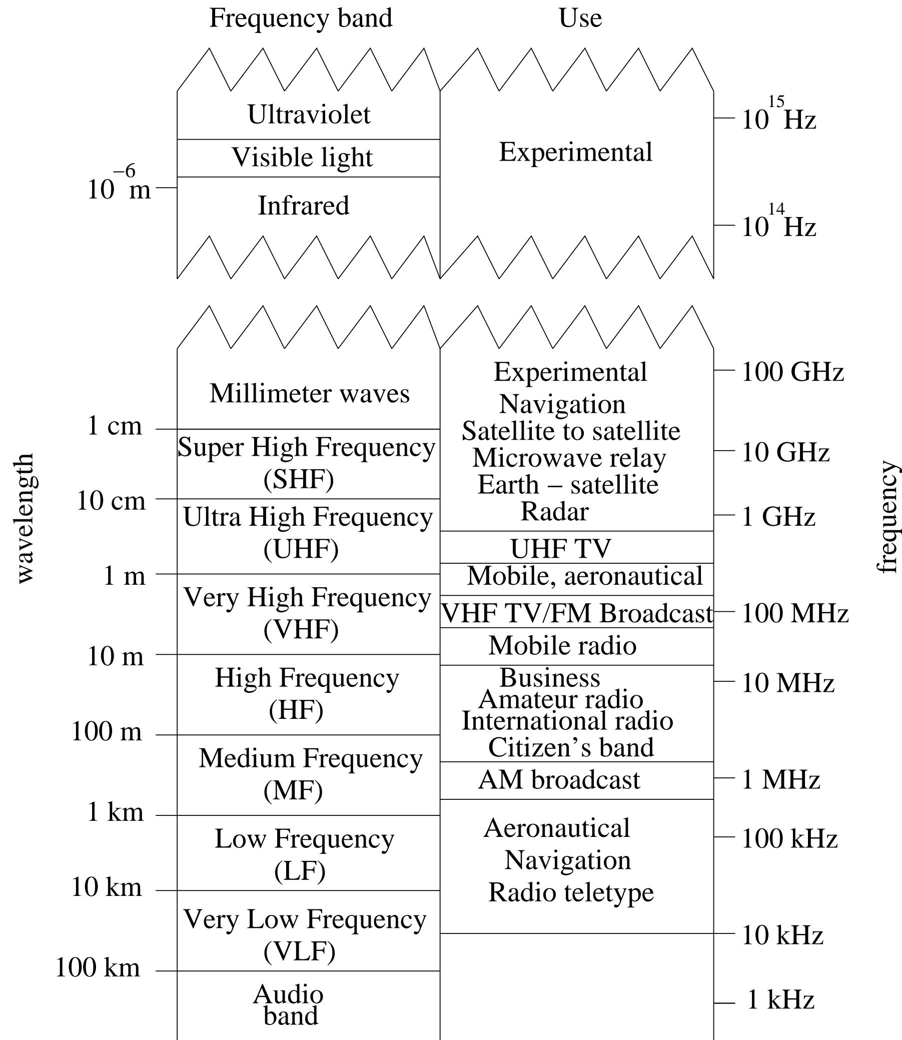



Figure 2.4: Frequency range of wireless electromagnetic channels

If the reflecting signals overlap a problem occurs named *multi-path fading*. Another problem makes wireless communication harder to handle as communication over a wired channel. Because of the undirected wireless channel, wireless communication is always a broadcast communication. When more than two radio terminals exist which are listening and sending on the same frequency with the same transmission power, many new problems occur. So a transmission can be interfered by other terminals. Interferences happens when the signal is overlaid by another signal of the same frequency (e.g. other radio sources). Thereby the original information of the signal may get lost. In the following we consider some scenarios in which multiple terminals exist(Figure 2.5).

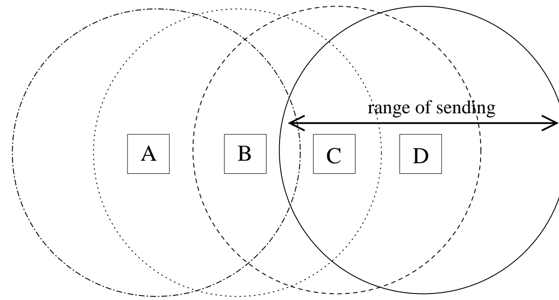



Figure 2.5: Wireless Communication

In Figure 2.5 four terminals A, B, C and D are shown. The transmission power from terminal A ranges only to terminal B. So A and B interfere each other when they send at the same time. Terminal C can interfere with B and D, but not with terminal A. When A and C simultaneously send to terminal B a problem occurs that is known as the *hidden terminal*. Terminal C has not the ability to prof if terminal A sends a message to B because A is out of range (i.e. for C terminal A is hidden). So C considers that the medium is free and sends a message to B. At terminal B signals from A and C interfere so both messages get destroyed.
Another problem occurs when B sends to A and C wants to send a message to D. When C proves the medium, it detects a transmission and will not send until the transmission is over. Effectively terminal C can transmit to D, because it does not interfere the transmission between B and A. This situation is called the *exposed terminal problem*.

## 2.3 Basic functions of wireless communication

In 2.2 we have discussed what problems a wireless channel produce. So a radio communication system has to be designed by considering the characteristics of this channel. Thus, special functions are needed to assure correctness of the data that is sent to the receiver. There are just some functions we are interested in which are shown in the following block diagrams.

Before the data from a source is sent via the wireless channel it passes through different blocks. These blocks can be assigned into the PHY and DLC/MAC layer as shown in Figure 2.6 and Figure 2.7. The DLC provides reliable data transfer

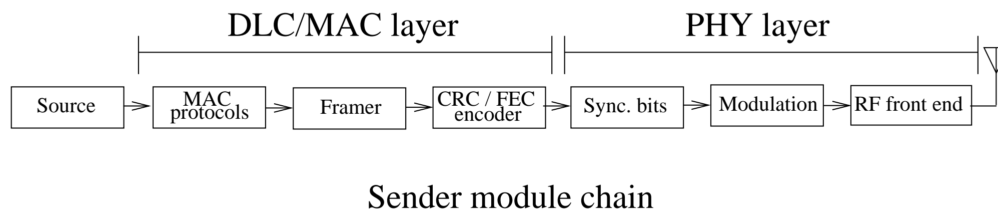



Figure 2.6: Block diagram of a wireless communication system: sender

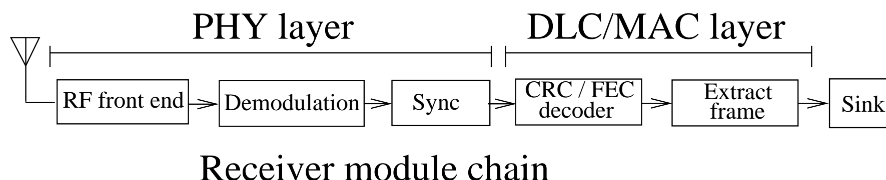



Figure 2.7: Block diagram of a wireless communication system: receiver

across the physical link. The task of the framer is to pack the information into smaller units called frames. In addition to the payload a frame contains a header and a trailer. The header contains amongst others the address of the destination and of the sender. On the DLC layer frames are used to avoid that the whole data has to be sent again when an error occurs during transmission. The next block adds redundancy for error detection to the frames. Error detection and Forward Error Control (FEC) codes are often used when data has to be transmitted over an error-prone channel. Error detection helps to verify the correctness of the received data. Cyclic Redundancy Check (CRC) is one method to realize error detection. As we know, wireless channels are committed by interferences. Without a FEC, transmission would accomplished with a minimum of success. FEC is accomplished by adding redundancy to the transmitted information using a predetermined algorithm. The two main categories of FEC are block coding and convolutional coding. Block codes work on fixed-size blocks of bits with a predetermined size. Reed-Solomon, BCH and Hamming are block codes that are used often. Convolution codes work on bit streams of arbitrary length and are often decoded with the Viterbi algorithm. Another important task of the DLC is the Media Access Control (MAC). MAC handles the transmission of the frames and decides when a frame is sent. Therefore special protocols exist. Those control for example if a frame has been arrived at the receiver side and send possibly frames again. The next block in the chain adds a preamble of bits to the data. These bits are needed to detect the beginning of a frame. In order to send the digital data over the wireless channel, the data has to be modulated to an analog signal. Modulation is a method of varying a carrier signal, typically a sinusoidal signal, in order to use that signal to convey information. One of the three key characteristics of a signal is usually modulated: its phase, frequency or amplitude. A device that performs modulation is known as a modulator. On the end of the transmission the signal is transformed back by a demodulator to its original digital signal. Many different modulation types exist. At the end of this chain these signals are emitted by the radio frequency (RF) front end.
On the receiver side the electromagnetic waves are received by the RF front end. The demodulator transforms the analog signals back to digital data. To find the beginning of the useful data, the data-stream must be synchronized. Therefore a synchronization pattern is searched in the demodulated data-stream. After that the redundancy is taken from the data and if it is possible, errors were repaired. Then the checksum of the frame is proven. At the end the header and payload of the frame is extracted. If the frame has reached the correct destination the data

get to the sink (for e.g. an application).
In the following sections some basic functions of a wireless communication system are specified.

### 2.3.1 Modulation

The amplitude-shift keying (ASK) is a form of modulation which represents digital data as variations in the amplitude of a carrier wave. The simplest and most common form of ASK uses the deflection of a carrier wave to indicate a binary one and its absence to indicate a binary zero.
Frequency-shift keying (FSK) uses diverse frequencies for data transmission. The simplest form of FSK is binary frequency-shift keying (BFSK). Thereby the frequency of a constant carrier signal is switched between two values. These two values correspond to a binary 1 or 0.
Another modulation is the phase-shift keying (PSK). PSK is a technique which shifts the period of a wave. A phase shift represents one binary state. Figure 2.8 illustrates these modulations.

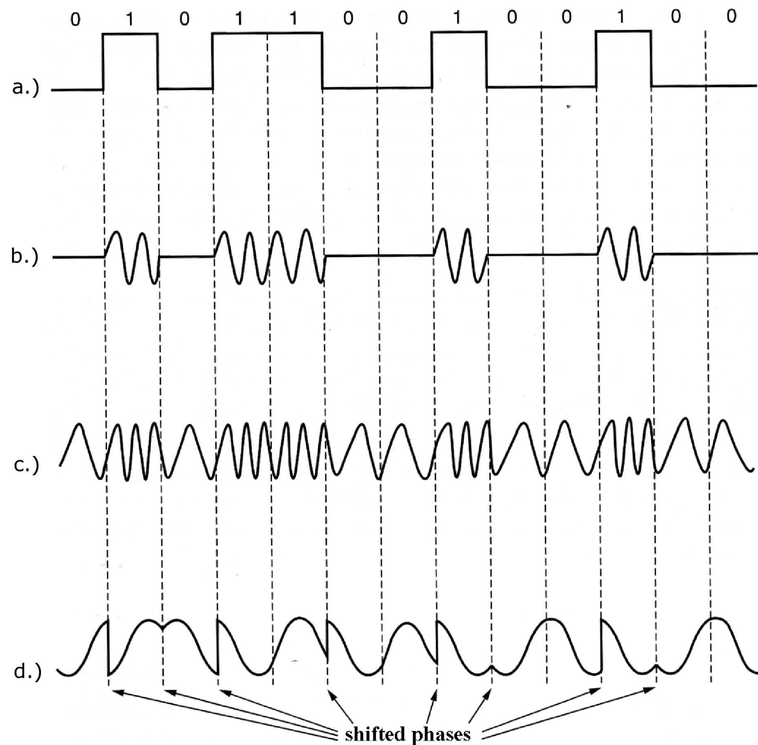



Figure 2.8: Modulation schemes: (a)Binary signal, (b)Amplitude modulation, (c)Frequency modulation, (d)Phase modulation [21]

Continuous phase modulation (CPM) is a method for modulation of data. This modulation uses a continuous phase, so there are no abrupt changes in phase. CPM has good spectral efficiency for a non-coherent modulation technique, and is less expensive to produce and more power efficient than coherent modulation schemes. It requires filtering to achieve good spectral efficiency. Most common filter choices are Gaussian or Raised cosine.
Minimum Shift Keying (MSK) is a type of continuous phase FSK. Another representative of continuous phase FSK is Gaussian Minimum Shift Keying (GMSK) [18]. GMSK is a modulation scheme in which the phase of the carrier is instantaneously varied by binary signal. A bit-stream of 0/1 is transformed into symbols of $-1/+1$. This $-1/+1$ signal is then filtered to a shaped $-1/+1$ pulses and then transformed into Gaussian shaped signals. The Gaussian signals are then frequency modulated.

### 2.3.2 Synchronization

In order to receive bits in the first place, the receiver must be able to determine when it has received a symbol and how fast the symbol was sent. Therefore the analog signal has to be sampled over a range. The sampler must be justified until it detects all incoming symbols. This phase is called synchronization. After the symbols are transformed back into bits, these bits must be interpreted to meaningful information. On wireless transmission it can happen that the receiver demodulates and interprets signals to a digital information, not knowing that the signal come form a source of a interfering transmitter. So there has to be an arrangement between sender and receiver to detect if the receiving signals contain useful information. Normally the sender and the receiver make use of a preamble that indicates the useful data. When the receiver detects this special bit pattern in the demodulated digital signals, he knows that he receives a potential message.

### 2.3.3 Forward error correction: Convolution codes

Convolution codes are often used in digital transmission systems, when the communication channel is error-prone. These codes allow the receiver to correct transmission errors. A convolution encoder increases the length of the original message by adding redundancy to the message, normally to a multiple of the original length. Figure 2.9 illustrates the structure of a convolution encoder.

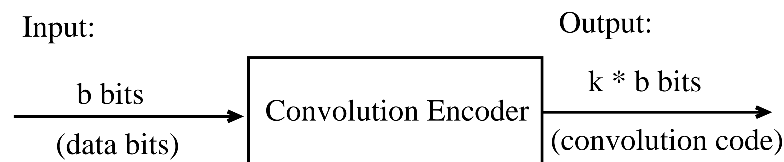



Figure 2.9: Principle of a convolution encoder

The input of the encoder is a block or a stream of data bits with a size of b. After the convolution the generated code has a multiple size of $b$. So the rate between input size $b$ and output size $c$ is

$$R = \frac{b}{c}. \tag{2.1}$$

$R$ is called the code rate. Common code rates are $\frac{1}{2}$ and $\frac{1}{3}$. For the computation of the output the convolution needs a number of information bits over which the convolution takes place, called the constraint length $k$. These $k$ bits are stored in memory, e.g. a shift register and are used to compute the output bits.
For example, we want to transmit 2 bits for every information bit. Our constraint length should be 3. Then the encoder would send out 16 bits for every 8 bits of input. Because of the constraint length of 3, each output pair depends on the present and the former 2 input bits. Thereby the information of each input bit is spread over 6 transmitted bits. This redundancy helps the receiver to reconstruct the correct input even when several transmission errors have affected the input.
In Figure 2.10 a typical implementation of a convolution encoder is shown. This encoder produces a code rate of $\frac{1}{2}$. For each bit two bits are generated. The sequence of the output bits is controlled by a selector.
The most important technique for decoding convolution codes is the Viterbi algorithm. This algorithm was developed by Andrew Viterbi. The idea of a Viterbi decoder is to find the most likely sequence of hidden states which belong to a sequence of observed events. The sequence of hidden states are called the Viterbi path. The basics of the Viterbi algorithm come from the "hidden Markov model" [17]. The Viterbi algorithm is also used in information theory, speech recognition, keyword spotting, computational linguistics, and bio-informatics. For example, in speech-to-text speech recognition, the acoustic signal is treated as the observed sequence of events, and a string of text is considered to be the "hidden cause" of the acoustic signal. The Viterbi algorithm finds the most likely string of text given the acoustic signal. The disadvantage of the Viterbi algorithm is that its implementation is quite complex which costs power (e.g. in mobile communication

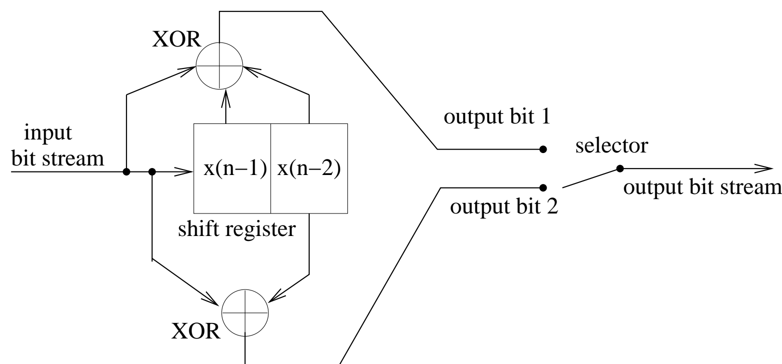



Figure 2.10: Implementation of a convolution encoder using shift registers

systems). An alternative to Viterbi are turbo decoder. A power efficient turbo decoder for SDRs has been implemented by B. Kang et al. [3].

### 2.3.4 Error control by Cyclic Redundancy Check

Cyclic Redundancy Check is a type of hash function used to produce a checksum. This checksum is computed over a block of data bits, for e.g. a frame. A bit string can be presented as a polygon with coefficients $x$ of 0 and 1. A block of bits with the length of $k$ can be illustrated as

$$x^{k-1} + x^{k-2} + x^{k-3} + ... + x^3 + x^2 + x + 1. \tag{2.2}$$

The bit string 101101 can be illustrated as polynomial $x^5 + x^3 + x^2 + 1$. Such a polynomial is divided by a generator polynomial over the integers modulo 2. At the end of this computation a remainder is left. This remainder is appended to the original message. After the transmission the message gets verified by dividing the message with the remainder by the same generator polynomial, used on the sender side, over the integers modulo 2. If there is no remainder, the data was received without an error or with a small probability an error has occurred. Errors can not be detected, if the polynomial of a erroneous frame is a factor of the generator polynomial.

### 2.3.5 Framing

A frame consists of a header, the payload and a trailer. The header has a fixed length of bytes and contains information to identify the destination and sender of the frame and the type of the frame (e.g. a data or acknowledgment frame). Sometimes also a frame number is part of the header which is used when more than one frame is sent. The trailer acts as the delimiter of the frame and is implemented often with a simple byte also called *flag-byte*. Between header and trailer is the payload which has a fixed or variable length of bytes. The bit pattern of the delimiter byte can also be a part of the payload bytes. In order to divide the payload from the delimiter a *Byte Stuffing* method can be used. Thereby an special byte called *escape-byte* is prefixed to each byte in the payload which contains the bit pattern of the flag-byte. The frame delimiter is pointed out because it has no escape-byte prefixed. The filled escape-byte have to be removed later on the receiver side. A problem occurs when the escape-byte already exist in the payload. This problem can be solved by prefixing an escape-byte to all matching bytes. Figure 2.11 illustrates the idea of the byte stuffing method.

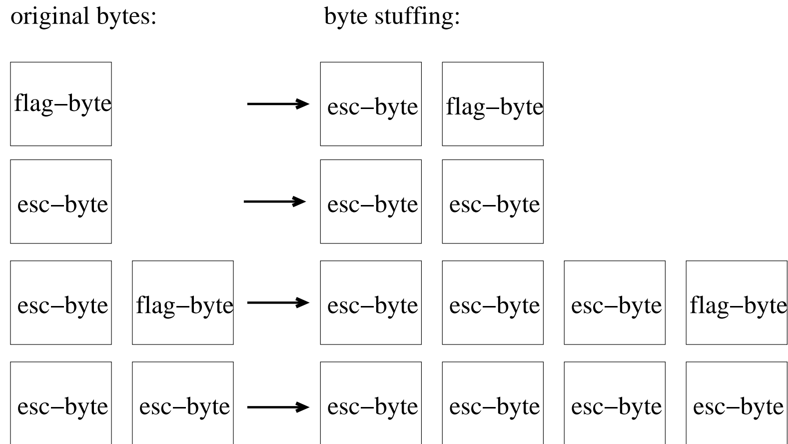



Figure 2.11: On the left side are payload bytes which contains the flag or esc-byte. On the right side these bytes are filled by the byte stuffing method

### 2.3.6 MAC protocols

The Media Access Control (MAC) provides functions to control the transmission of frames. Because of the problems in wireless transmission (described in 2.2), protocols are needed which take account characteristics of a wireless channel. The pure ALOHA is the first protocol which was intensively studied since introduction in 1970 at the University of Hawaii [21]. Pure ALOHA was used in the ALOHA-NET of wireless connection of serval stations. The principle of this protocol is quite simple. It neither coordinates the medium access nor does it resolve contention between other sending stations. Each sender can access the medium at any time. Thereby collisions with other sending stations can happen, so that frames get destroyed (Figure 2.12). In order to detect that a frame does not arrived the receiver, an Automatic Repeat-reQuest (ARQ) protocol is used. Different types of ARQ protocols exist. The simplest is the *Stop-And-Wait ARQ*. The sender only transmits one frame at a time and waits for an acknowledgment from the receiver. If the receiver does not answer in a fixed time, the sender assumes that the frame has not reached the receiver or that the message has been destroyed. So the sender transmits the frame again until it gets an acknowledgment. Other ARQ protocols are *Go-Back-N ARQ* and *Selective Repeat ARQ*, which provide more than one packet to be sending. They are thus more efficient regarding channel utilization and complicated.

But how efficient is the channel utilization using pure ALOHA? We assume that a frame has a constant length $L$ measured in bits and needs a transmission time of $X = \frac{L}{R}$, where $R$ is the bit transfer rate bits per seconds (bps). A frame starts with transmission time $t_0$ and is successfully transmitted if no other frame collides with it. Any other frame transmission that reach into interval $[t_0, t_0 + X]$ leads to collision. So we have a vulnerable period that lies between the interval $[t_0 - X, t_0 + X]$ (Figure 2.13).

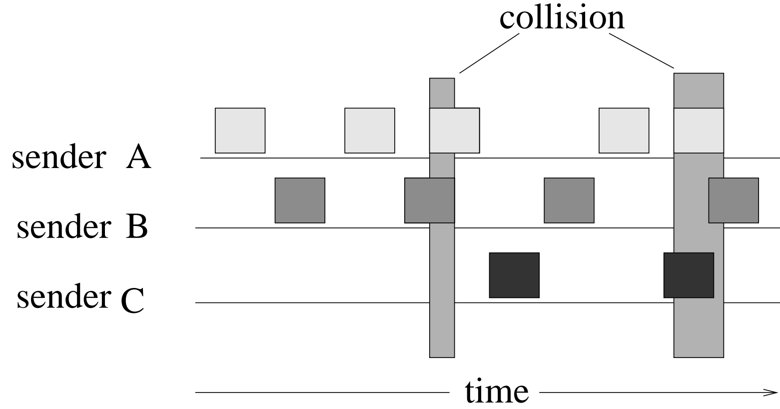



Figure 2.12: The pure ALOHA protocol with multiple access

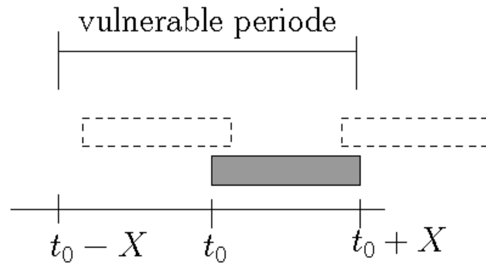



Figure 2.13: Vulnerable periode of frames

In other words the vulnerable period has a length of two frame transmission times. We consider the throughput $S$ of pure ALOHA. The throughput is the number of successful frame transmissions per time unit. $G$ is the load and is defined as the expected number of transmission and retransmission attempts from all users per time unit. The probability of a successful frame transmission should be $P_{succ}$. The throughput $S$ can be calculated by

$$S = P_{succ}G. \tag{2.3}$$

We assume a infinite number of users and thus we can assume that the load is as Poisson distributed with rate $G$. So the possibility that $k$ frames are generated by all users in a frame transmission time is:

$$Pr[k] = \frac{G^k e^{-G}}{k!}. \tag{2.4}$$

The probability $P_{succ}$ that a frame not suffers by collision is equal to the probability that no frame is generated during two frame transmission times:

$$P_{succ} = \frac{(2G)^0 e^{-2G}}{0!}. \tag{2.5}$$

So the throughput is:

$$S = G \cdot P_{succ} = G \cdot e^{-2G}. \tag{2.6}$$

In Figure 2.14 the throughput of the pure ALOHA protocol is illustrated. The maximum throughput lies at $G = 0.5$ with $S = \frac{1}{2}e$ and is 0.184. So we have a maximum channel utilization of 18.4%. This means that 81.6% of the total available bandwidth is basically wasted due to users trying to talk at the same time.

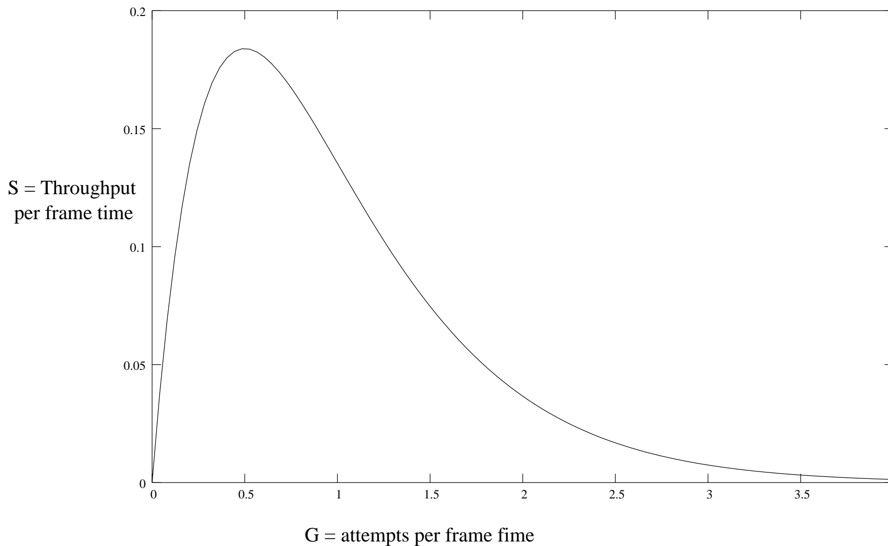



Figure 2.14: Throughput of the pure ALOHA protocol

An improvement to the original ALOHA protocol was Slotted ALOHA [18]. In the slotted ALOHA protocol a sender is not allowed to transmit anytime, but only in time slots. Therefore all users have to be time synchronized and have to know when a time slot begins and ends. Collision happens only when two or more senders use the same time slot. Thus with slotted ALOHA a throughput is reached of 0.368, that is twice as much as the throughput of pure ALOHA.

## 2.4 Software-defined radios

After we have discussed some basic functions of communication systems we want to consider classic radio communication systems and a technology named Software Defined Radios (SDR). Classical radio systems were built to accomplish often just one task, for e.g. receiving TV, AM/FM broadcast radio or for mobile communication networks(e.g. WLAN, GSM). Each radio service has its own fixed frequency and its own protocols. Two radio systems with different waveforms according to different modulation types, different protocols or error detection schemes are not able to communicate with each other [19]. An example, the communication equipment of the US military branches could not talk to each other. Even the police and fire fighters could not coordinate missions over radio, because they use different

radio frequencies. A communication system was required, that could changed its behavior by changing its software. Software defined radio uses software for the modulation and demodulation of radio signals and for the protocols. Figure 2.15 and Figure 2.16 illustrate this difference between classic communication systems and an SDR. In Figure 2.15 a classic communication system is shown. The PHY and DLC are implemented in hardware and cannot be changed after construction. In an SDR system (Figure 2.16) functions of the PHY and DLC can be changed. In the PHY the modulation and demodulation of signals can be controlled by software. Just few very time intensive functions are still implemented in hardware, for example the sampling of signals and the synchronization between component parts. We can take influence of the complete DLC layer. Error Control and MAC protocols can be switched on demand. This makes SDR very flexible. An SDR can theoretical morph into a cell phone using GPRS, into a wireless communication system using IEEE 802.11, into a AM/FM or HTDV receiver or into a navigation system using GPS.

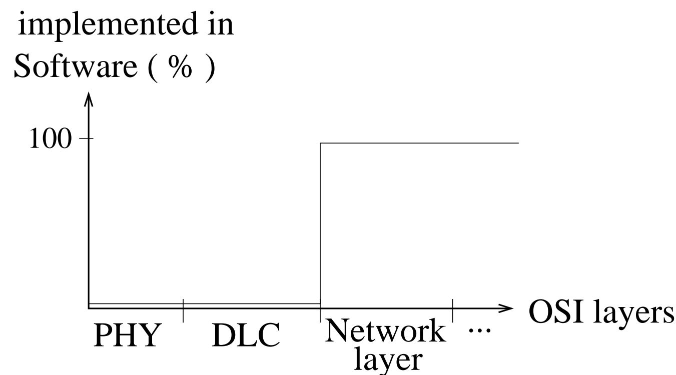



Figure 2.15: Classic communication systems

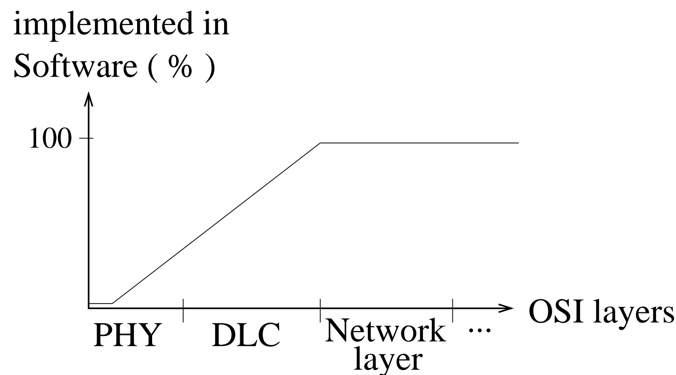



Figure 2.16: A software-defined radio

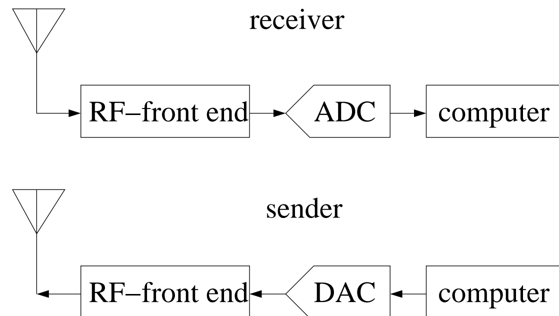



Figure 2.17: The basic architecture of an SDR

The basic architecture of an SDR is illustrated in Figure 2.17 A radio signal is received by an RF front end. This signal is digitized by an analog-to-digital converter (ADC) and can be handled by a computer. On the digital output signal processing can be performed to extract useful information. At the sender side of an SDR, a digital signal is generated by a computer. These signals are transformed by a digital-to-analog converter (DAC) to analog signals which can be transmitted by an RF front end. The signal processing is performed by a computer or by reconfigurable hardware which act as a Digital Signal Processor (DSP).
An ideal SDR can receive and send in all frequencies and provides every communication protocol. Such a SDR is not yet available.
In spite of this, there are several SDR implementations. One SDR implementation is the *SPEAKeasy* system, which was built for military use in the US [15]. SPEAKeasy allows to communicate over a wide range of frequencies and to change the modulation techniques, data encoding methods, cryptographic types and other communication parameters. After the success of SPEAKeasy other SDR projects follows like *SPEAKeasy II*, the Airborne Communication Node (ACN) and the Joint Tactical Radio System (JTRS) [19][2]. These SDR have been invented for military aims, but there are also SDR for civil or research like the sandbridge SDR communication platform [11], the virtual radio [22] and the GNU Radio.

## 2.5 The GNU Radio SDR platform

The *GNU Radio – The GNU Software Radio* is an open source project which is reasonably hardware-independent. It provides a free software toolkit for learning about, building and deploying software radios [4]. Main part is a library of signal processing blocks which can be used to build SDRs or to create new signal processing blocks. In order to implement an SDR, at first a signal graph has to be created. In this graph the vertices are signal processing blocks and the edges represent the data flow between them. The architecture of such a block is shown in Figure 2.18.

Signal blocks process infinite streams of data flowing from their input port to their

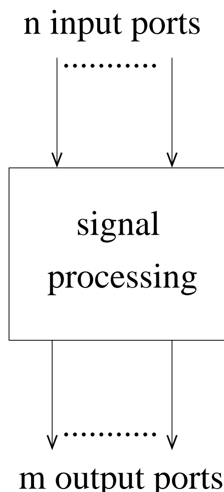



Figure 2.18: A signal block has n input ports and m output ports

output ports. Blocks have a number $n$ of input ports and and a number $m$ of output port. Another attribute is the type of data that flows through the block. The type of input and output data can be different. Valid types are char, short, integer, double, float and complex. Signal blocks can also have different input/output (I/O) rates. For example, a block which calculates a convolution code with a code rate of $\frac{1}{2}$ produces for each input data unit two output data units. The data a block may consume on each input stream can have a different rate, but all output streams must produce data at the same rate. In order to allocate enough memory to buffer input and output data streams and to schedule the data flow through the graph an approximate I/O rate can be set. If this rate is not explicit set a default I/O rate of 1.0 is used.
The signal processing blocks are written in C++. There are different predefined signal blocks in GNU Radio library which can be used to implement own blocks. Own blocks inherit from a predefined signal block functions (e.g. the signal processing function) and characteristics (e.g. consuming a special number of data per processing). These functions can be adapted to solve wanted tasks and the characteristic of the block can be arranged new.
Every signal block has a function *work* where the signal processing tasks place. In this function the output data stream is calculated e.g. by a given input stream. This function is called by the scheduler of the block when enough input items has been buffered. The work function gets not the whole data input at once, but in small portions which depends on the buffered data. So the work function is called several times until no more buffered data exists.
A special kind of signal blocks are sinks and sources. Source blocks have only output ports and serve as the source of a data stream which can be e.g. a vector, a file or a ADC. Sink blocks have only input ports and are the last link in the signal processing chain. Sinks can write data streams e.g. to memory, a file, a DAC, sound-card or to a graphic. GNU Radio provides a lot of predefined signal processing blocks as sources, sinks, filter, de-/modulation-blocks, etc.

As mentioned, signal blocks can be connected to a signal graph. This is handled in the script language Python. In order to use the signal blocks written in C++, they have to be wrapped into Python objects. Therefore, GNU Radio uses the Simplified Wrapper and Interface Generator (SWIG) [14].
Python is not only used to build the signal graph, also to control the signal processing on a higher level. Signal flow can be started or stopped, information can be evaluated from signal blocks or signal blocks can be replaced. The following example wants to give an impression how a signal graph is implemented in Python. The 2.5 program generates a sine wave that is outputted to a sound card.

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

from gnuradio import gr
from gnuradio import audio

sampling_freq = 32000
ampl = 0.1

fg = gr.flow_graph ()
src = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 550, ampl)
dst = audio.sink (sampling_freq)
fg.connect (src, dst)

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

In each GNU Radio application a flow graph object instance has to be created by the *gr.flow_graph()* call. This object provides functions to connect and disconnect signal blocks, to allocate buffer size for the blocks and to schedule the signal processing flow. The beginning of each graph is a sink block. In this example an instance of *gr.sig_source_f* is used which generates a sine wave of a frequency of 550 Hz. The end of the signal graph is an instance of the *audio.sink* object. This block outputs the received signals to the sound-card. In order to build a signal graph both block instances have to be integrated into the graph. Therefore the *connect* function of the flow graph object is used. It takes at least two signal blocks as parameters where the first parameter marks the start of a connection and the last parameter the end of a connection. Between start and end block other signal blocks can be placed. After building the simple graph with two vertices the signal flow can be executed by running the *start()* function of the graph object. At this point signals which are generated by the source block flow through the graph to the sink block. In this example a continuous 550 Hz tone is outputted by the sound-card until the *Enter*-key is pressed. Then the *stop* function of the graph object is called and the signal processing stops.

## 2.6 The Universal Software Radio Peripheral hardware front-end

In order to receive and send radio waves, GNU Radio needs additional hardware. One hardware peripheral which works with GNU Radio is implemented by the National Center for Supercomputing Applications (NCSA) [1]. NCSA have extended the GNU Radio receiver design into 900 MHz narrow-band SDR receiver.
Another hardware peripheral is the *Universal Software Radio Peripheral* (USRP) board which is used in this student research project. The USRP is developed by *M. Ettus* et al. [6]. The USRP consists of a main board and up to four daughter boards. Figure 2.19 shows the USRP with four daughter boards. The main board incorporates AD/DA converters and a Field Programmable Gate Array (FPGA).

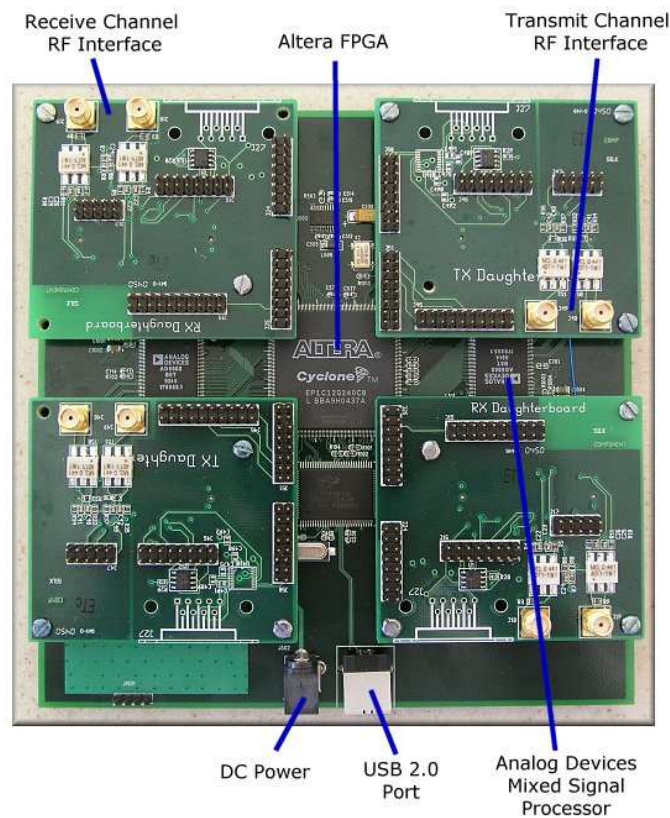



Figure 2.19: The USRP main board with two TX and two RX daughter boards [5]

The daughter boards serve as RF front ends. Up to two transmission daughter boards (TX) and two receiver daughter boards (RX) can be used simultaneously. They exist in different versions depending on the frequency range which is used for communication. In this research project we are using the basic daughter boards which have no special transmission functions. There are four high-speed AD converters with a resolution of 12 bit and a sampling rate of 64 mega samples (MS) per second which can digitize a band as wide as about 32 MHz. Also there are

four high-speed DA converters with a solution of 14 bit which can generate signals up to 50 MHz. The FPGA is used for computationally expensive pre-processing of the input signal from the ADCs and DACs [20]. The standard configuration includes digital down converters (DDC). These DDC converts the digitized frequency range, which the receiver daughter boards passed to the ADC, down to the base frequency range, also called base band. Then the signals are decimated so that the data rate can be adapted by the USB interface. When the USRP is used to sent data, the base band signals coming from the computer are first interpolated by the FPGA. Then the signals pass a digital up converter (DUC) which exist as a special chip on the USRP main board. Here the signals are interpolated, converted digital up to a higher frequency range and finally sent to the DAC. The FPGA is connected to a USB2.0 interface chip that provides data transfer between the computer. About 32 MByte/sec can be transfered over the USB connection. The type of the information which is sent across USB is complex and consists of a 16 bit real part and a 16 bit imaginary part.
For using the USRP in GNU Radio applications the USRP Python module has to be included. The USRP can be used as a source or as a sink block in a signal graph. The following piece of code shows how a USRP sink block is initialized in Python.

```
....
#usrp is data source
src = usrp.source_c (0, 250)
src.set_rx_freq (0, 104.5)
src.set_pga(0,20)
....
```

With *usrp.source_c (0, 250)* an instance of an USRP source object is created. The first parameter specifies which USRP board is used. If only one USRP is connected to the computer this parameter is 0. The second parameter set up the decimation rate to the USRP board. This is used to reduce the digital signal, which is sampled by the ADC, to a data rate which can be handled by memory and CPU. As mentioned the ADC sample rate is $64 \cdot 10^6$ samples per second. In this example the decimation is chosen to be 250 so the resulting data rate is $256 \cdot 10^3$ samples per second. This data rate suffices to receive the spectrum information of a 256 kHz frequency band. In order to capture a frequency band the USRP needs a intermediate frequency. This is set with the function *set_rx_freq*. An intermediate frequency (IF) is a frequency to which a carrier frequency is shifted. In order to amplify the incoming signal the *set_pga* function can be used. PGA stands for *Programmable Gain Amplifier*, which is set to 20 decibel (dB) in the example. The USRP sink block has almost the same functions as the USRP source block.

```
...
dst = usrp.sink_c (0, 160)
u.set_tx_freq (0, 10e6) #10 MHz
u.set_pga (0, 20)
...
```

The only difference is that the second parameter of the *usrp.sink* call sets the interpolation rate of the FPGA and not the decimation factor.

# Chapter 3

# Implementing physical and data link control layer on the GNU software-defined radio platform

In this student research project some basic functions of the PHY and DLC/MAC layer are implemented on the GNU Radio platform with the aim to transfer packages from a sender to a receiver terminal. Some functions of the PHY and DLC layer are implemented as signal blocks. The protocols of the MAC layer are implemented in Python [7]. The main functions are the *receive*-function and the *send*-function. In the following sections the functionality of both functions are described.

## 3.1 Send function

The send function is implemented as a Python application and can be called from the command line by the command *send -d 33 -s 44 -c 0 -n 0 - f "data.dat"*. The parameters $d$ and $s$ set the destination and sender addresses, the parameter $c$ sets the type of the frame (0 is data, 1 is acknowledgment) and the $n$ is the package number. There are two different ways to retrieve payload. It is possible to read payload from a file by using the $f$ parameter with a filename or from a string using the $i$ parameter with a string. The task of the Python application is to build a frame with the payload and to send it over the USRP. Therefore, signal blocks are used and connected to the signal graph. The graph of the send function is illustrated in Figure 3.1. The signal graph can be divided into two different parts.

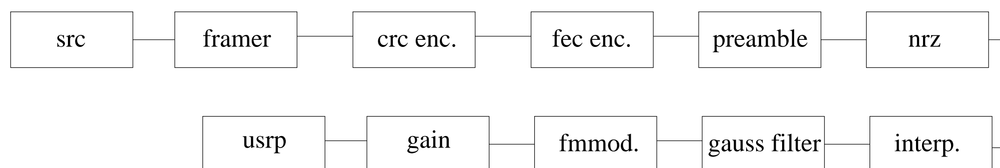



Figure 3.1: Signal graph of the sender

The first part contains all signal blocks which operate on the data type char and do signal processing on the byte or bit level. These are the source, framer, CRC encoder and the FEC encoder blocks. The second part contains all blocks that operate on data type float or complex. These data types are needed to calculate analog signals to perform signal processing as filtering or modulation. The blocks of the second part are NRZ, interpolation, Gaussian filter, fmmod and gain. They were provided by the GNU Radio libraries and are used for GMSK modulation. We used GMSK in this implementation because it is a part of the GNU Radio framework which is well tested. In the following the functions of the sender graph are described per signal block.

### 3.1.1 Source vector

The source block of the sender signal graph is the *vector_source_b* which GNU Radio provides. The block has two parameters, a vector of bytes and a boolean value. It generates with the vector a stream of bytes and outputs them. The boolean value tells the block if the vector should either passed once through the signal graph (value set to 0) or continuously (value set to 1). In this implementation the input vector is filled by a fixed number of bytes and is sent once.

### 3.1.2 Framer

The framer block generates a frame for the payload coming from the source block. The frame begins with a header which contains the address of the destination and the sender, a command and a package number field. The frame ends with a delimiter (EFD = End Frame Delimiter) in this case a simple byte. Figure 3.2 illustrates the architecture of the frame. The command field marks the type of the message. There are two types of possible messages, a data message (initialized to 0) and an acknowledge message (initialized to 1). In order to detect the end of the frame at the receiver side that is marked by a special byte which could also be contained in the payload, byte stuffing subsection 2.3.5 is used. A framer signal block is initialized with the addresses of sender and destination, the type of frame, a package number and the length of the payload.

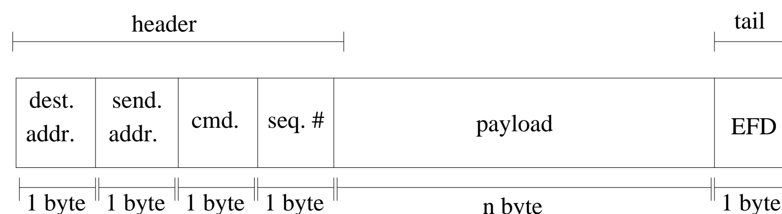



Figure 3.2: Architecture of the frame

### 3.1.3 CRC encoder

The CRC encoder block computes a 32 bit CRC checksum over the whole frame and adds this checksum to the end of the frame. At the end of the CRC checksum a second frame delimiter is placed (ECD) to detect the end of the checksum. This delimiter is set because CRC checksum of different lengths can be set (e.g. 16 bit). In Figure 3.3 the frame is shown after the checksum has been added. The checksum is calculated as described in subsection 2.3.3 using functions of the Booster library [23].

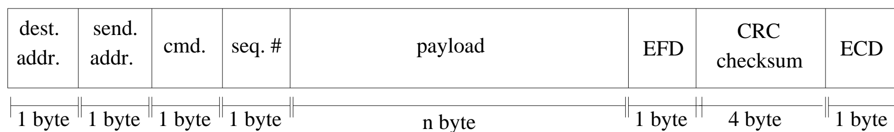



Figure 3.3: Architecture of the frame additional with CRC part

### 3.1.4 FEC encoder

Over the whole data stream a convolution code is computed. The code rate is close to be $\frac{1}{2}$, so the I/O rate of this block is also close to be $\frac{1}{2}$. The convolution encoding and decoding routines are based on a code example from [12]. In that example the convolution encoder generates for each bit of a byte two bytes, so the input/output rate was $\frac{1}{16}$. For this implementation the code was changed according to generate for each bit two bits instead of two bytes.

### 3.1.5 Preamble

This block prefixes a 64 bit synchronization word to the data stream. This synchronization word is used by the correlator signal block on the receiver side to detect the beginning of a frame.

### 3.1.6 Non Return to Zero (NRZ)

The NRZ block transforms a bit stream to a Non Return to Zero (NRZ) signal. Characteristic for NRZ signals is that they not fall back in a continuous interval to a zero potential. Typically the binary coded signals are represented through positive and negative potential [18]. In this implementation each bit of a byte is mapped to a floating point. Thereby a bit representing 0 is mapped to floating point of $-1.0$ and a bit representing 1 to $+1.0$. So this block produces for each input item of the type char eight output items of the type float. A graphical illustration of the signal output is shown in Figure 3.4.

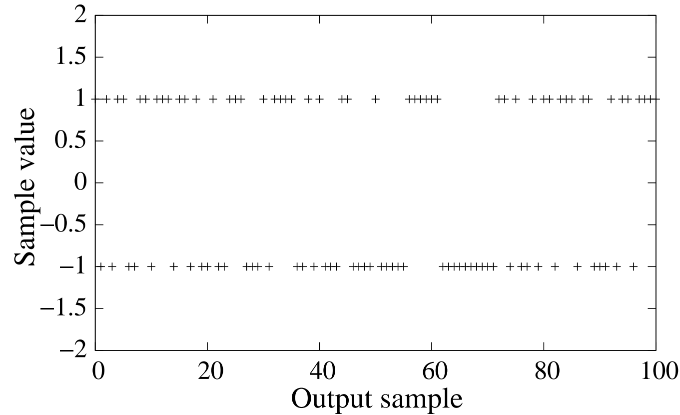



Figure 3.4: Output signals of the NRZ block

### 3.1.7 Interpolation

In order to transform the incoming data signals to a Gaussian signal, at first the signal has to be interpolated. Interpolation is used to construct from a discrete set of data values a continously function. This block transforms the input $-1/+1$ signal to shaped $-8/+8$ pulses using a FIR filter. An FIR filter is a digital filter which convolves the input signal with the digital filter's impulse response. In Figure 3.5 the operation of an FIR filter is shown. An input signal x(n) is convolved with FIR filter's impulse response h(n), resulting in a filtered output signal y(n). The output signal from a linear system is equal to the input signal convolved with the filter's impulse response. The operation $(x) * (h)$ is called convolution.
The filter used in the interpolation block multiplies values of a given vector (also called filter taps) with one input sample item. The result is then outputted. In this implementation the filter vector is $(8, 0, 0, 0, 0, 0, 0, 0)$. The input items are $-1$ or $+1$, so the output of the filter is $(-8, 0, 0, 0, 0, 0, 0, 0)$ or $(+8, 0, 0, 0, 0, 0, 0, 0)$. Thus, the I/O rate of this signal block is $\frac{1}{8}$. The output of the interpolation block is shown in Figure 3.6

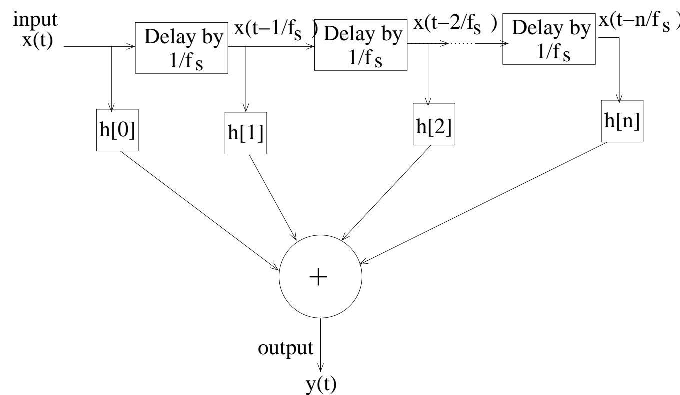



Figure 3.5: The operation of a FIR filter [8]

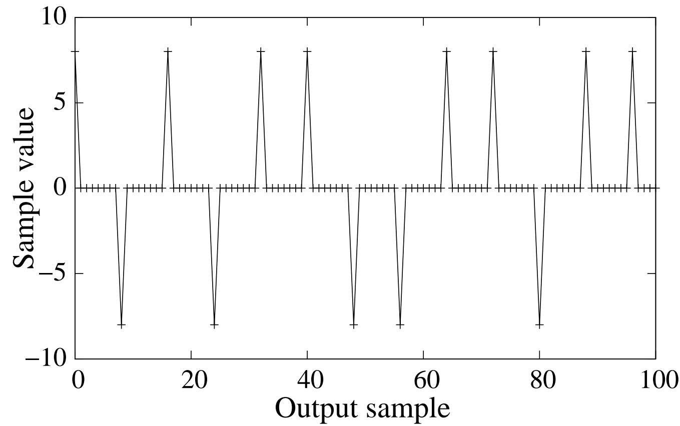



Figure 3.6: Output signals of the Interpolation block

There is a problem with the FIR filter implementation in GNU Radio. Filters in GNU Radio do not produce output unless they are entirely filled. That is, if a filter has N taps, it will not produce data unless there is at least N samples to work with [13]. To guarantee that the frame is sent completely we send additionally overhead bytes.

### 3.1.8 Gaussian filter

In this block the input signals are pre-modulated to Gaussian-shaped signals. Therefore a Gaussian filter is used. The pre-modulation Gaussian filter has narrow bandwidth and sharps cutoff properties which are required to suppress the high-frequency components. Figure 3.7 shows the output of this signal block. The Gaussian filter is implemented as an FIR filter.

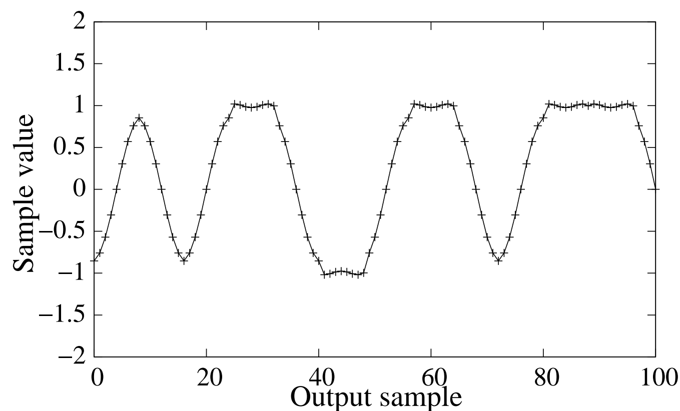



Figure 3.7: Output signals of the Gaussian filter block

### 3.1.9 Frequency modulation

This block is the frequency modulator. The input signals, which are of the type float, are modulated to a frequencies represented as complex values. The output signals of the frequency modulation block are shown in Figure 3.8.

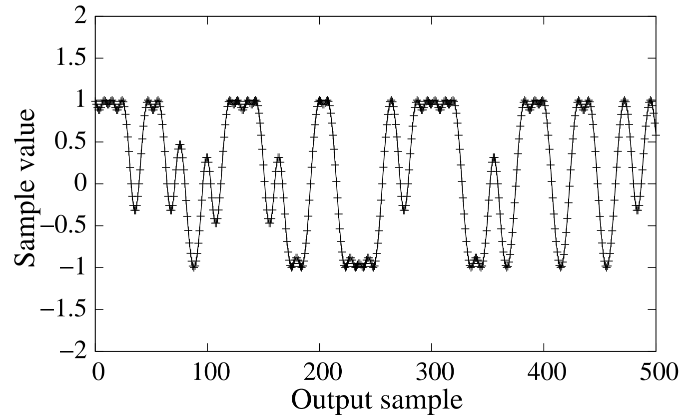



Figure 3.8: Output signals of the frequency modulation block

### 3.1.10 Gain

In the gain block each input value is multiplied to a constant factor to amplify the signals. The factor used in the implementation is an integer of 8000. The output is shown in Figure 3.9

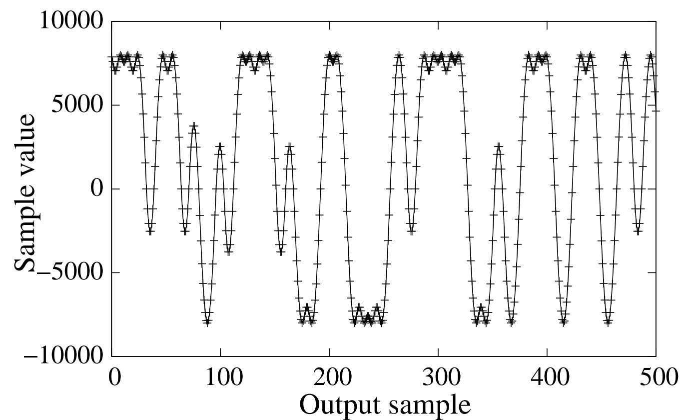



Figure 3.9: Output signals of the gain block. The input signals coming from the frequency modulation block are multiplied by a factor of 8000.

### 3.1.11 USRP TX

The last link in the sender signal chain is the usrp block. The data stream from the gain block is forwarded to the USRP hardware where finally the calculated signals are transformed to physical analog signals by the ADC.

## 3.2 Receiver function

As the sender function the receiver function also utilizes a GNU Radio signal graph. Figure 3.10 shows all signal blocks which are used in the receiver graph. Like the sender signal graph the receiver signal graph can be divided into two

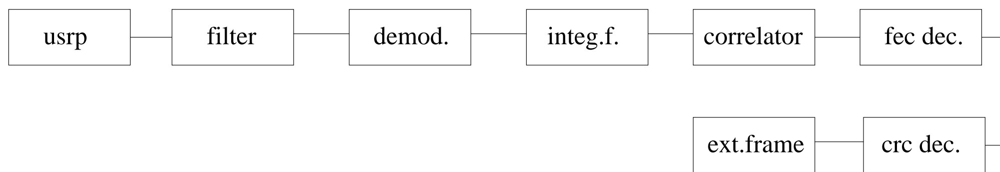



Figure 3.10: Signal graph of the receiver

parts. The two filter blocks, the demodulation and the correlator block preform signal processing over complex and float data types. These blocks are provided by the GNU Radio libraries and are unmodified except for the correlator block. The other blocks works on the char data type and were needed to implement. In the following all blocks of the receiver signal graph are described.

### 3.2.1 USRP RX

At the beginning of the receiver signal chain the usrp sink block is placed. It outputs the complex data coming from the USRP.

### 3.2.2 Filter

The signals coming from the usrp source block are filtered by a low-pass FIR filter before they get demodulated. This signal block filters frequencies which are higher than a cutoff frequency. The output of this block should be nearly equal to the amplified signal of the gain block.

### 3.2.3 Demodulation

The demodulation block demodulates the frequency modulated signal with quadrature demodulation. A quadrature demodulator shifts the modulated carrier signal by 90° at the center frequency. This phase shift is either greater or less than 90° depending on the direction of deviation. A phase detector compares the phase-shifted signal to the original to result in the demodulated baseband signal.

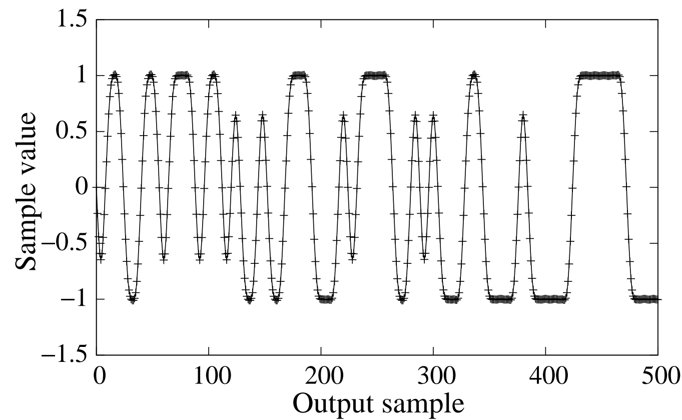



Figure 3.11: Output signals of the demodulation block

### 3.2.4 Integrate filter

The integrate filter smooths the incoming signal by averaging over eight samples. This filter is implemented as an FIR filter.

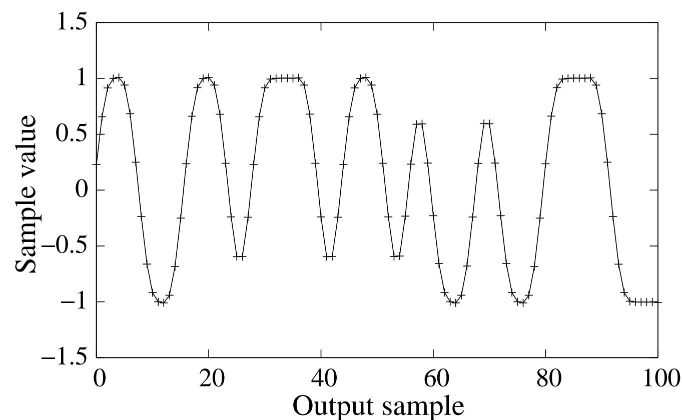



Figure 3.12: Output signals of the integrate filter block

### 3.2.5 Correlator

The correlator block has the purpose to detect bits from the incoming digitized signal and to output this bit sequence byte-wise. Therefore, the correlator at first has to identify the synchronization word from the input stream which is needed to synchronize the bit-stream and to detect the beginning of the data part of the frame. After the synchronization extracted bytes are outputted. The correlator block implements a fairly simple method to find the transmitted data. Basically it bit-slices the data into 0 and 1 and places each bit into one of eight unsigned integers. It calculates the Hamming distance between each of the unsigned integers and the synchronization word. The code remembers when the Hamming distance between any of the integers and the synchronization word falls below three and

then when it rises over three. It thus has a window where the correct 64 bit sync value was recognized. It estimates the best place to sample the bit as the center of the window. Instead of using the original correlator block from the GNU Radio libraries this block was changed to detect the length of the here used frames.

### 3.2.6 FEC decoder (Viterbi decoder)

This module receives a convolutional coded byte stream. To transform the coded information back to the original data a Viterbi decoder is used. Many implementations of the Viterbi algorithm exist. In this module we used the implementation by [12]. Because this implementation of the Viterbi decoder is byte-based, this code was changed to a bit-basis by removing redundancy. The decoder reduces the I/O rate of the signal block to the half of the input size.

### 3.2.7 CRC decoder

The CRC decoder block proves if the received frame is correct. Therefore, the CRC checksum is extracted from the end of the frame and is used to calculate the correctness of the whole frame. This signal block provides a function to get the result of the CRC calculation. This function is also used to detect if a frame arrived correctly. A value of 1 is returned if a frame has been detected and the CRC check was correct. If a destroyed frame has been received this value is set 0.

### 3.2.8 Extract frame

The last link in the receiver signal graph is the extract frame block. The task of this block is to extract the header of the frame and to remove the ESC-flags as explained in chapter two subsection 2.3.5 from the payload which were added by byte stuffing. This block provides functions to get the header information and to extract the whole payload.

## 3.3 MAC protocols

The send and receive signal graphs are only parts of the communication system which allow sending and receiving frames. In order to communicate protocols are needed. In this implementation the pure ALOHA and the send-and-wait ARQ protocol are used subsection 2.3.6. The receiver function and the MAC protocols are implemented both in one Python application. After the signal graph is initialized and started as shown in Figure 3.13, in an infinite loop is checked if a correct frame has been detected. Therefore, the CRC signal block function *get_checksum* is used. In order to prove if the frame has reached the correct receiver the *get_dest_addr*

function from the extract frame block is used. This function returns the destination address of the frame. If this address matches all other header information are extracted by functions provided by the extract frame block.
The further states of the protocol depend on the role of the terminal. If the terminal is the receiver, it checks if the command field in the header of the frame marks payload. In this chase the payload is extracted and stored. In this implementation the payload is stored in a file. After that the receiver terminal sends a message which contains an acknowledgment and the received frame number back to the senders address. Then the receiver waits for next coming frames.
At the sender terminal side also a receive signal graph is running. After a frame was sent to a destination the terminal waits for the acknowledgment. Therefore, it checks continuously if a frame has been arrived which contains the acknowledgment of the last sent frame. If in a fixed time no acknowledgment has been arrived the terminal sends the same frame again.

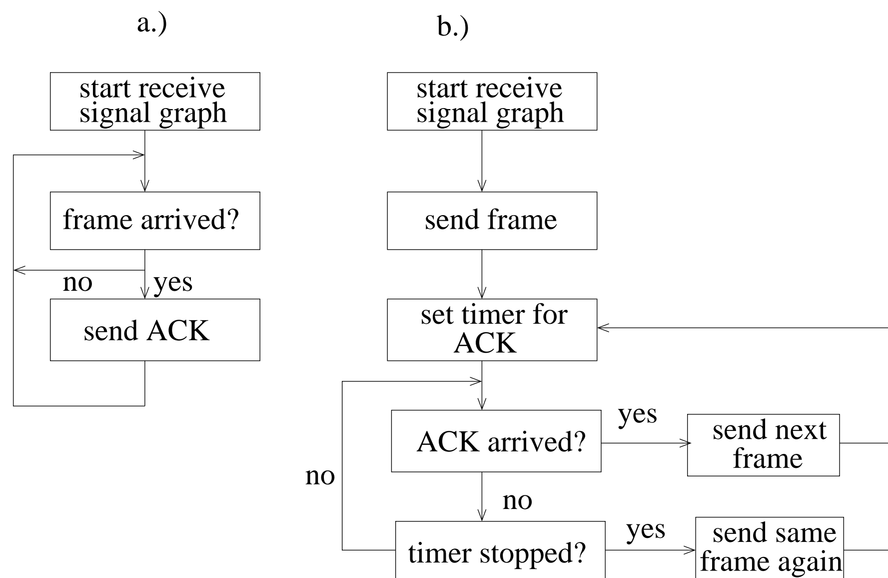



Figure 3.13: a.) Protocol of a sender terminal b.) Protocol of a receiver terminal

# Chapter 4

# Performance study

In the last chapter we discussed the implementation of sender and receiver functions. These functions have been tested in a simple communication system between two computers. On this system we have done different measurements. We have measured the RTT of a frame and the latency of each signal block. The results should help us to evaluate the performance of our implementation and should give information if the GNU Radio/USRP platform can be used for practicable communication. At the beginning of this chapter we describe the experimental setup. Then we introduce the methodology of our measurements and their results. At the end of this chapter we discuss our results.

## 4.1 Experimental setup and parameterization

In order to test our functions an end-to-end scenario is built between two computers, as illustrated in Figure 4.1. As we can see both computers use a separate USRP. An uni-directional connection is built via coaxial cable between a sender unit (TX) of one USRP to a receiver unit (RX) of the other USRP. This results in a bi-directional communication where each terminal can send and receive at the same moment. In our scenario computer $A$ sends packages containing payload to computer $B$. Computer $B$ receives these packages and answers with acknowledgments (ACK). Both computers have identic hardware and software configuration. Some specifications of the computers are listed in Table 4.1. The parameterization of our communication system is shown in Table 4.2.

| Component | Value |
|---|---|
| CPU | Intel(R) Pentium(R) 4 CPU 3.40 GHz |
| Memory | 1 GByte DDR2 SDRAM, 400 MHz |
| USB profile | EHCI (USB2.0) |
| USB controller on PC | Intel(R) 82801FB ICH6 I/O Controller Hubs [10] |
| USB controller on USRP | CY7C68013 EZ-USB FX(TM) Controller [9] |
| OS | std. Linux, kernel 2.6.11.4 SMP |
| GNU Radio core version | 2.5 |
| USRP version | REV.1 |

Table 4.1: Specifications of the test environment

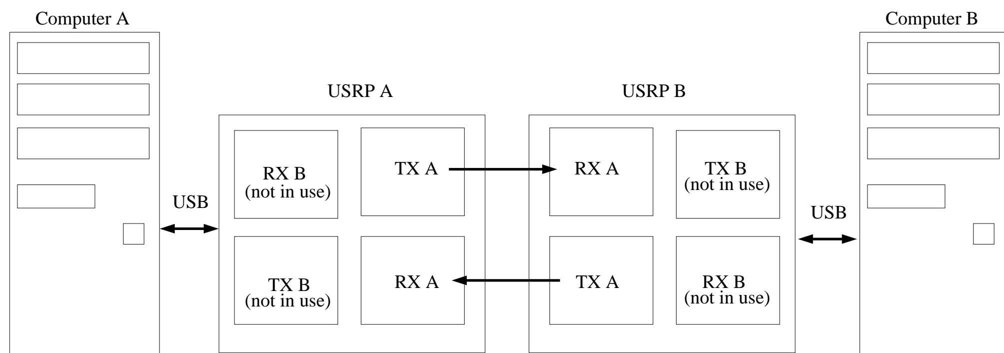



Figure 4.1: Experimental setup of the communication system

| Parameter | Value |
|---|---|
| Sender signal blocks | |
| frame size | 1001 byte |
| CRC-checksum | 32 bit |
| Preamble | 64 bit |
| Modulation scheme | GMSK |
| Symbol rate | $10^5$ |
| Gaussian filter bandwidth · symbol time | 0.3 |
| Bandwidth | 800 kHz |
| FEC code rate | $\frac{1}{2}$ |
| Gain-factor | 8000 |
| USRP interpolation factor | 160 |
| USRP IF | 29.32 MHz |
| Receiver signal blocks | |
| USRP receiver IF | 29.32 MHz |
| USRP decimation factor | 80 |
| MAC protocols | |
| max. time to wait for ACK | 4 sec. |

Table 4.2: Parameters of the

## 4.2 Measurement results for the round trip time

At first the round trip time (RTT) is measured. Typically the RTT is defined as the time that a frame needs to be sent from terminal A to terminal B and back again. In Figure 4.2 the RTT is illustrated. Using an ARQ protocol, we can measure the RTT by sending a frame from terminal A to terminal B and stop the time that is used until the acknowledgment of terminal B arrives at terminal A. In this implementation the data frame and the acknowledgment frame have each a fixed length of 1001 bytes. This length was chosen because of the mentioned filter problem (subsection 3.1.7). The RTT time starts at the moment the first byte passes the first sender signal block (the source signal block) of terminal A and stops when the first byte of the acknowledgment frame arrived in the last receiver signal block (extract frame block) of terminal A. The measured value is normalized to seconds per bit. Therefore, the measured time has been divided to the number of payload bits (512 bit) of a frame. The following histogram shows the average RTT of a bit in seconds (Figure 4.3). On the x-axis we have the RTT per bit measured in seconds. On the y-axis the number of samples is shown. While the total of 706 samples is collected most of the sample values fall in the interval between $2.9 \cdot 10^{-3} s$ and $3.3 \cdot 10^{-3} s$. The calculated mean of all samples is $3.14 \cdot 10^{-3} s$ while the standard deviation is $1.095 \cdot 10^{-4} s$ which confirms that the RTT deviates minimal from the mean RTT.

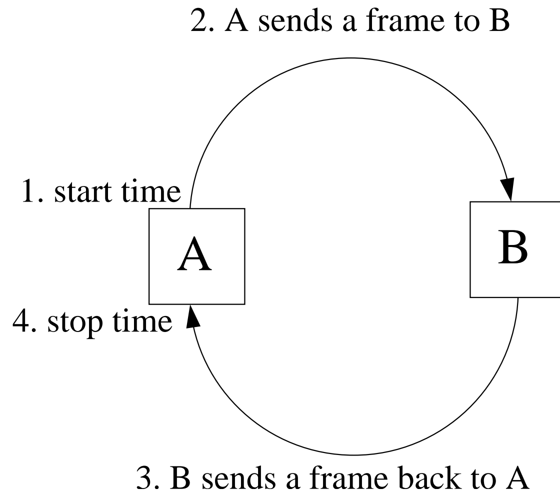



Figure 4.2: Round trip time: Terminal A stops the time that a frame needs from A to B and back to A.

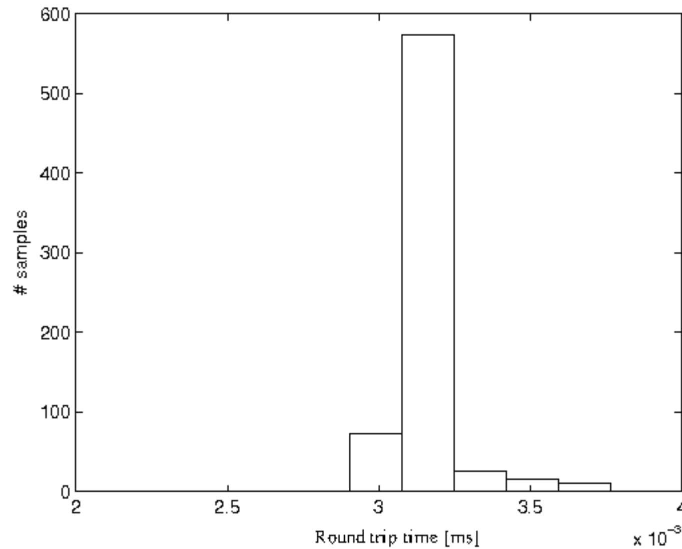
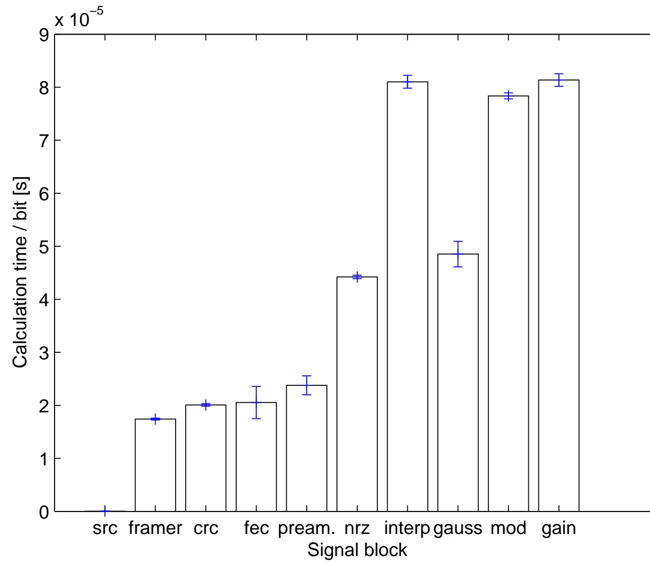


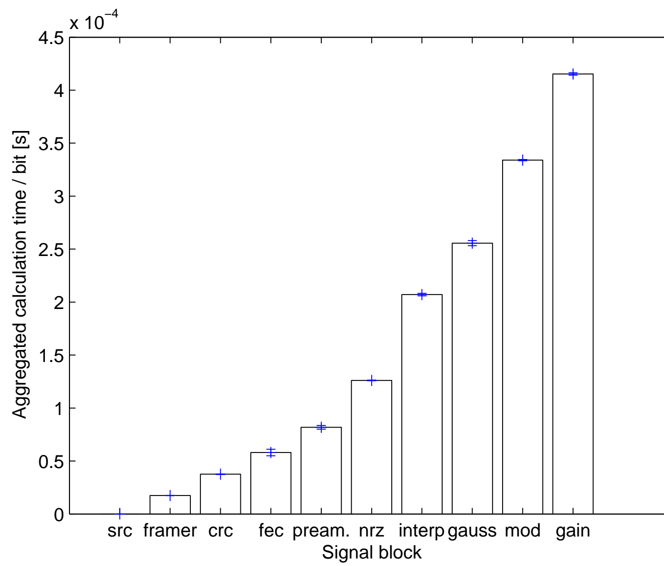

Figure 4.3: Histogram of the average RTT per bit

## 4.3 Measurement results for the calculation time per signal block

After measuring the RTT we are interested how much time of the RTT is spent in each of the signal blocks. Therefore, the calculation time of each used signal block was measured. With calculation time we mean the time that a frame needs to pass a signal block. This time is normalized to seconds per bit. Per signal block 1000 samples have been collected. The samples are used to calculate the average time per bit and the confidence intervals. In 4.4(a) the average calculated time per bit of all used sender signal blocks is shown together with its confidence intervals. As we see most of the time per bit is used by the nrz, interpolation, gauss filter, modulation and gain signal blocks. This is because those blocks operate over more data input as the framer or fec signal blocks. From these blocks the interpolation, modulation and gain signal blocks are the most time intensive blocks. The data rate of these blocks is the sixteen-fold of the original input data rate. Also in these blocks mathematic operations like multiplication are done which costs a lot of calculation time. The calculation time of the source block that is near zero is quite noticeable. This short calculation time results from the fact that the whole data is available at the start time of the signal block. Thus no scheduler is needed which has to buffer input data for processing and beside passing the I/O stream there are no operations done. In 4.4(b) we have aggregate the average calculation time started by the source block up to the gain block. This results in the average calculation time per bit for the whole sender signal graph of $4.154 \cdot 10^{-4} s$.




(a) Average calculation time per bit

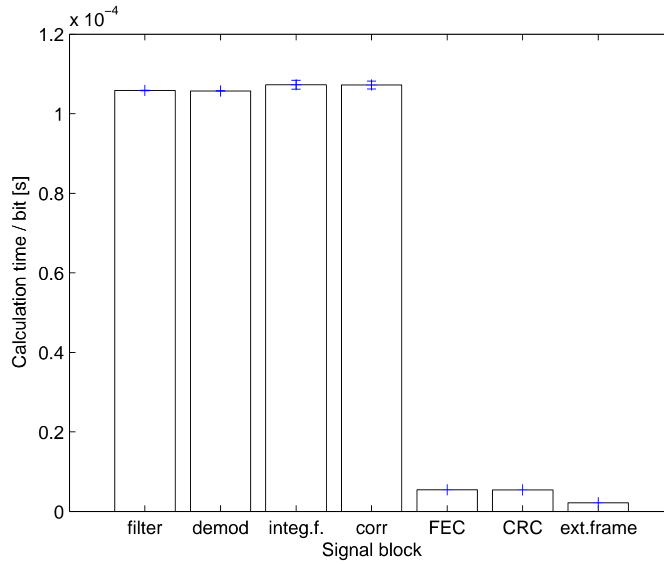



(b) Aggregation of the average calculation time per bit

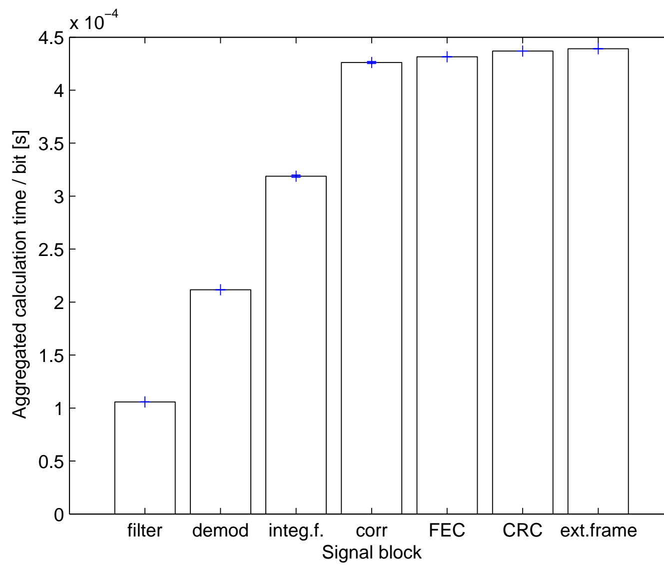
Figure 4.4: Measurements of the sender signal blocks.

After we have discussed the sender graph we consider the receiver graph. In 4.5(a) the average calculated time per bit of all used receiver signal blocks is shown together with the confidence intervals. As we can see most of the calculation time is spent on the filter, demodulation, integrate filter and the correlator blocks. In these blocks a lot of calculation is done which costs time (e.g. subsection 3.2.5). After this time consuming signal to bit mapping, the fec, crc and extract frame need a considerably shorter time for signal processing. These blocks have a smaller input data rate and the operations on the data is not so time consuming as it is in the other blocks.

In 4.5(b) we show the aggregated average calculation time starting with the filter block up to the extract frame block. The average calculation time per bit of the whole receiver signal graph is about $4.391 \cdot 10^{-4} s$.




(a) Average calculation time per bit




(b) Aggregation of the average calculation time per bit

Figure 4.5: Measurements of the receiver signal blocks.

## 4.4 Discussion

After we have studied the average time that a bit needs to pass the signal graphs we compare this time with the measured RTT. The result is that more than the half of the whole RTT (55.41%) was spent in the signal processing blocks. The difference of 45.59% of the total is used by MAC-protocols, the USRP and the GNU Radio scheduler. Although further analysis of these components of the system can be done, we focus on the parts of the system implemented in this student research project, e.g. the sender and receiver signal graphs.

The results of our measurements are helpful to further analyze the performance of an SDR using GNU Radio and USRP. The measured RTT per bit is $3.14 \cdot 10^{-3}s$ on the average. The half of the RTT $1.57 \cdot 10^{-3}s$ is approximate the time that is needed to send and to received one bit neglecting pipelining effects. This means that theoretical about 636 bits/s can be transfered sequentially. This data rate is not suitable for transmission which use a high data rate, but it is suitable for communication. It is possible to achieve higher transmission rates by fixing the FIR filter problem (subsection 3.1.7) and increasing the payload size. However, this rate is a lower bound for throughput since the pipeline effects are neglected. System performance in terms of throughput and good-put have not been studied in this work.

# Chapter 5

# Conclusion

In this work we have analyzed the suitability of a specific SDR platform, the GNU Radio, for prototyping of wireless communication systems. Therefore, we have implemented basic functions for wireless communication, consisting of standard PHY and DLC functions to provide reliable data transfer and MAC protocols. The performance of this system is measured in terms of latency.
In order to evaluate the suitability of the GNU Radio/USRP platform for wireless prototyping we consider this platform under the aspects of the usability and the performance. Since usability is not countable, a subjective impression of the GNU Radio framework is given here. The GNU Radio framework has a modular concept that allows to implement rapidly SDRs. Comparable to a construction kit the signal processing blocks are arranged to build a radio sender or receiver. GNU Radio provides nearly one thousand predefined signal blocks and allows to implement own signal processing blocks. Thus, the GNU Radio platform is suitable for rapid prototyping of wireless PHY and MAC functions.
But such a clear modular architecture results in overhead decreasing performance. The result of our RTT measurement is that in the mean $3.14 \cdot 10^{-3}s$ are needed to send and receive one bit neglecting pipelining effects. This means that about 636 bits/s can be transfered sequentially. This data rate is not suitable for high-speed communication. The measurements of the latency of the signal blocks result that 55.41% of the RTT is spent in the signal blocks. From this it follows that 45.59% of the RTT is used e.g. for the USRP and the GNU Radio scheduler.

As our implementation is the first version of a communication system, some implemented functions can be improved to get a more efficiently communication system. Especially the mentioned filter problem (subsection 3.1.7) has to be fixed. Also a better performance can be achieved by increasing the payload size. In future work we will improve these functions. Also further measurements of the system have to be done, e.g. throughput.

# Appendix A

# Tutorial of measurement

## A.1 How to measure a signal block

At first the USRPs have to be initialized. This can be done by running the USRP test programs in the USRP directory (e.g. `../usrp-0.8/host/apps/test_usrp_standard_tx`). After that, we have to extend the source code of the signal block which should be measured by time functions. These functions are used to set time stamps. One time stamp is set when the first byte passes the signal block. Further time stamps are set when data items are outputted. A time stamp is a string which contains the name of the signal block, a marker to identify if start or end time is measured and the system time in seconds, millisecond and microseconds. The time stamps are buffered in an array and are printed out to command line when the destructor of the signal block is called. This happens when the signal graph is stopped. The following source code A.1 shows how the time functions are placed in the source code:

```
...
#include <sys/time.h>
#include <string>
using std::string;

static struct timeval zeit;
static string str_arr[255];
static int cnt_str = 0;
static char str12[255];
...
gr_multiply_const_cc::~gr_multiply_const_cc ()
{
  //print the buffered time stamps out
  for (int i = 0; i < cnt_str; i++)
    std::cout << str_arr[i];
}

//main function for signal processing
int gr_multiply_const_cc::work (int noutput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
{
  gr_complex *iptr = (gr_complex *) input_items[0];
  gr_complex *optr = (gr_complex *) output_items[0];

  int   size = noutput_items;

  //set time stamp before the first byte has passed
  if (d_start_meas == 0)
  {
    gettimeofday(&zeit, NULL);
    sprintf(str12, "GAI\t%d.%d\n", zeit.tv_sec, zeit.tv_usec);
    str_arr[cnt_str++] = str12;
    d_start_meas++;
  }
  ...
  //here signal processing is done
  ...
 //save time stamps in array, till the last byte has passed
  gettimeofday(&zeit, NULL);
  sprintf(str12, "E_GAI\t%d.%d\n", zeit.tv_sec, zeit.tv_usec);
  str_arr[cnt_str++] = str12;

  return noutput_items;
}
```

The *work*-function of the signal block can be called several times section 2.5, so we prove by a variable if data has passed the first time the signal block. Therefore we have defined the variable *d_start_meas* which is initialized in the constructor function with a value of 0. We prove in the *work*-function before the first operation is done if the value of the variable *d_start_meas* is equal to 0. If the result is true we save the current system time in an array and increment the value of the variable *d_start_meas*. The incrementation is done to guarantee that we have only one starting time for our latency. At the end of the *work*-function we also save the system time in an array. So the last value in the array marks the end of the latency. The values of the array are printed out in the destructor function. After the source code has been compiled a measurement of the block can be done. Therefore, on the sender side we start the *sender_transceiver* function and on the receiver side we start the *receiver_transceiver*-function. The gain signal block is a part of the sender-graph. Each time data passes the signal graph, this happens when a frame is sent, the gain signal block will output time stamps. This information is used to calculate the latency of this block. In order to collect this information we pipe the output of the sender-function to a file (e.g. sender_output.txt). Here is an example A.1 of collected time stamps we have obtained from the gain signal block.

```
GAI     1131356620.996795
E_GAI   1131356620.996868
E_GAI   1131356620.997771
...
E_GAI   1131356621.5315
E_GAI   1131356621.7631
E_GAI   1131356621.8656
E_GAI   1131356621.9365
...
E_GAI   1131356621.039055
E_GAI   1131356621.039075
```

The first column contains the name of the signal block (GAI == gain signal block). A prefixed *E_* means that the time stamp is created at the end of the *work*-function. Only the first *GAI* and last *E_GAI* time stamp is relevant for the calculation of the latency. The difference between the *E_GAI* time stamp and the *GAI* time stamp is the latency of the gain signal block. The second column contains the measured system times in seconds, milliseconds and microseconds. As you can see the number of the decimal places is not always the same. This is because of the time function we have used for the decimal places. This time function returns the system time in microseconds without a special format. Before the measured times can be used, we have to convert them to the right format (six decimation places) by filling the milliseconds and microseconds up with zeros. This can be done manually or by a script. Here is the same example where we have edited the format of the system time.

```
GAI      1131356620.996795
E_GAI    1131356620.996868
E_GAI    1131356620.997771
...
E_GAI    1131356621.005315
E_GAI    1131356621.007631
E_GAI    1131356621.008656
E_GAI    1131356621.009365
...
E_GAI    1131356621.039055
E_GAI    1131356621.039075
```

After we have collected enough information we can analyze them by using the *separate.py* Python script. This script can work over a lot of collected information. Regular expressions are used to find the right row for calculation. Therefore the script has to be adapted to find the start and the end of the time stamp. The script calculates the latencies of the signal block and print out the results. These results can be piped e.g. into a file.

## A.2 How to measure round trip time

In order to measure the RTT we have to set two time stamps one in the source signal bock and one in the extract frame signal block. Both time stamps should return the system time when the first data unit has passed the signal block. This is equal to the measurement of the latency where we measure the starting time. Because the source and the extract frame signal blocks are in different signal graphs which are used in different programs, we collect the output of the time stamps in two files. Time stamps of the source block are collected in *sender_output.txt* and time stamps of the extract frame block are collected in *received_output.txt*. After converting the system time in the time stamps to the correct format (section A.1), we can calculate the RTT by using different scripts. The first script *auswertung.sh* is used to extract only the important information from both files and save them to a *sent* and a *received* file. The first column of the *sent* file is the system time when the first byte of a frame has been sent. The second column lists the package numbers and the third column lists how often a frame was sent before it has been acknowledged. Here is an example of a *sent*-file.

```
1131966483,638910        0        1
1131966629,683387        1        2
1131966689,439487        2        1
1131966715,435264        3        1
1131966722,646815        4        1
...
...
```

The *received*-file has only two columns. The first column lists the time when a frame has reached the extract frame signal block. The second column lists the package number. Here is an example of a *received*-file.

```
1131966485,229201        0
1131966636,949306        1
1131966691,162689        2
1131966717,059921        3
1131966724,221508        4
...
...
```

Ideally the *sent*-file and the *received*-file have the same number of rows and equal package numbers. If this is not the case the files have to be edited manually. Then we can merge both files together to calculate the RTT. This can be done by the *merge*-script. The script needs two parameters, the *received*-file and the *sent*-file. It reads the first column of a row of the *received*-file and stores this value. Then it reads the first column of a row of the *sent*-file and subtracts this value from the other value. The result is the RTT that is printed out together with the package number and the number how often a frame was sent. Here is the result of the *merge*-script:

```
1,590291        0        1
7,265919        1        2
1,723202        2        1
1,624657        3        1
1,574693        4        1
```

# Appendix B

# Acronyms

**SDR** Software-defined Radio

**USRP** Universal Software Radio Peripheral

**OSI** Open System Interconnection

**PHY** physical layer

**DLC** Data link control

**MAC** Media Access Control

**GSM** Global System for Mobile communication

**HF** high frequency

**FEC** Forward Error Control

**CRC** Cyclic Redundancy Check

**RF** radio frequency

**ASK** amplitude-shift keying

**FSK** Frequency-shift keying

**BFSK** binary frequency-shift keying

**PSK** phase-shift keying

**CPM** Continuous phase modulation

**MSK** Minimum Shift Keying

**GMSK** Gaussian Minimum Shift Keying

**ARQ** Automatic Repeat-reQuest

**ADC** analog-to-digital converter

**DAC** digital-to-analog converter

**DSP** Digital Signal Processor

**ACN** Airborne Communication Node

**JTRS** Joint Tactical Radio System

**SWIG** Simplified Wrapper and Interface Generator

**NCSA** National Center for Supercomputing Applications

**FPGA** Field Programmable Gate Array

**DDC** digital down converters

**DUC** digital up converter

**IF** intermediate frequency

**NRZ** Non Return to Zero

**FIR** Finite Impulse Response

**RTT** round trip time

# Bibliography


[1] A. Betts, M. Hall, V. Kindratenko, M. Pant et al. The GNU software radio transceiver platform. In *Procs. of 2004 Software Defined Radio Technical Conference (SDR Forum)*, November 2004.

[2] A. Haghighat. A review on essentials and technical challenges of software defined radio. In *Procs. of IEEE Military Communications Conference, VOL. 21, NO. 1*, October 2002.

[3] B. Kang, N. Vijaykrishnan et al. Power-efficient implmentation of turbo decoder in SDR systems. In *Procs. of IEEE International SoC Conference (SoCC)*, September 2004.

[4] E. Blossom, M. Ettus et al. GNU Radio – the GNU Software radio. Retrieved November 30, 2005 from `http://www.gnu.org/software/gnuradio`.

[5] E. Blossom, M. Ettus et al. GnuRadio: UniversalSoftwareRadioPeripheral. Retrieved November 30, 2005 from `http://comsec.com/wiki?UniversalSoftwareRadioPeripheral`.

[6] M. Ettus. Ettus Research LLC. Retrieved November 30, 2005 from `http://www.ettus.com`.

[7] Python Software Foundation. Python Programming Language. Retrieved November 30, 2005 from `http://www.python.org`.

[8] R. Frohne. FIR filters. Retrieved November 30, 2005 from `http://www.wwc.edu/~frohro/qex/sidebar.html`.

[9] Cypress Inc. CY7C68013 EZ-USB FX2(TM) USB Microcontroller High-speed USB Peripheral Controller. Retrieved November 30, 2005 from `comsec.com/usrp/CY7C68013.pdf`.

[10] Intel. Intel(R) I/O controller hub 6 (ICH6) Family. Retrieved November 30, 2005 from `ftp://download.intel.com/design/chipsets/datashts/30147302.pdf`.

[11] J. Glossner, M. Moudgill and D. Iancu. The sandbridge SDR communications platform. In *Procs. of IEEE Symposium joint IST workshop on mobile future and symposium on trends in communications*, June 2004.

[12] P. Karn. Forward Error Correcting Codes. Retrieved November 30, 2005 from `http://www.ka9q.net/code/fec`.

[13] J. Lackey. GMSK python modules for GNU Radio. Retrieved November 30, 2005 from `http://noether.uoregon.edu/~jl/gmsk/`.

[14] The University of Chicago. Simplified Wrapper and Interface Generator. Retrieved November 30, 2005 from `http://www.swig.org`.

[15] P. G. Cook and W. Bonser. Architectural Overview of the SPEAKeasy System. In *Procs. of IEEE Journal on selected areas in communications, vol. 17, no. 4*, April 1999.

[16] J. G. Proakis and M. Salehi. *Communication systems engineering*. Prentice-Hall, Inc., 2002.

[17] M. Purser. *Introduction to error-correcting codes*. Artech House, Inc., 1995.

[18] T. Rappaport. *Wireless Communications Principles & Practice*. Prentice-Hall, Inc., 2002.

[19] M. N. O. Sadiku and C. M. Akujuobi. Software-defined radio – a brief overview. In *IEEE Potentials Journal*, October/November 2004.

[20] D. Shen. Experimental Research. Retrieved November 30, 2005 from `http://www.nd.edu/~dshen/GNU`.

[21] A. S. Tanenbaum. *Computer networks*. Prentice-Hall, Inc., third edition, 1996.

[22] V. Bose, M. Ismert, M. Welborn and J. Guttag. Virtual radios. In *Procs. of IEEE Journal on selected areas in communications, vol. 17, no. 4*, April 1999.

[23] D. Walker. Boost CRC Libary. Retrieved November 30, 2005 from `http://www.boost.org/libs/crc`.

# Erklärung der Urheberschaft

Die selbstständige und eigenhändige Anfertigung der vorliegenden Studienarbeit versichere ich an Eides statt.

Ort, Datum　　　　　　　　　　　　　　　　　　Unterschrift