Implementation of uncoordinated direct sequence spread spectrum (U-DSSS) using software defined radios

Master Thesis

Author(s):
Meškovič, Saša

Publication date:
2008

Permanent link:
https://doi.org/10.3929/ethz-a-005633085

Implementation of Uncoordinated Direct Sequence Spread Spectrum (U-DSSS) using Software Defined Radios

Master Thesis

Saša Mešković - *sameskov@student.ethz.ch*
99-920-670

Supervising Professor: Prof. Dr. Srdjan Čapkun
Supervising Assistants: Christina Pöpper and Mario Strasser
Period: Oct 08, 2007 - Apr 08, 2008

System Security Group
Department of Computer Science
ETH Zurich

April 08, 2008

Acknowledgements

I'd like to thank my supervisors Christina Pöpper and Mario Strasser for supporting me with this master thesis. Especially for their help with questions, their ideas and their time.

Also I'd like to thank Prof. Dr. Srdjan Čapkun for his inspiring courses about Security of Wireless Networks and for giving me the opportunity to make this master thesis.

But most of all I'd like to thank my girlfriend and my daughter for their patience with me and their support during the last months.

For questions or comments concerning this thesis I can be reached at sasa.meskovic@gmail.com

Abstract

One of the major threats to wireless communications is jamming. Many anti-jamming techniques have been presented in the past. However most of them are based on the precondition that the communicating devices have a pre-shared secret that can be used to synchronize the anti-jamming scheme. E.g. for frequency hopping the secret could be used to derive the hopping sequence and for direct sequence spread spectrum the secret is used to derive the spreading codes.

But how can the devices bootstrap a jamming-resistant communication without having a pre-shared secret? Christina Pöpper and Mario Strasser propose as scheme for Uncoordinated Frequency Hopping (UFH) and Uncoordinated Direct Sequence Spread Spectrum (UDSSS) in their papers [1] and [2] respectively.

The goal of my project was an implementation of Uncoordinated Direct Sequence Spread Spectrum (UDSSS) using Software Defined Radios. The first version should serve as an easy to use and extendable proof of concept for the proposed scheme.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The following document is the final project report of my master thesis about the implementation of a system for uncoordinated direct sequence spread spectrum (U-DSSS).

1.1 Jamming-resistant communication

Currently one of the major threats to wireless communication is the fact that it can easily be intercepted or jammed. Especially jamming is a topic that has to be dealt with care because there is basically no protection against it. The attacker could potentially sit anywhere and is usually assumed to have huge but not infinite power and processing resources.

Current known anti-jamming techniques require the communicating devices to have a pre-shared secret that can be used as a secret spreading key. For frequency hopping based anti-jamming schemes this key is used to derive the hopping sequence. For DSSS based anti-jamming schemes the key is used to derive the codesequences.

1.2 Overview

My work is based on the papers [1] and [2] published by my supervisors Christina Pöpper and Mario Strasser. In order to be able to fully appreciate and see behind the scenes of the implementation the reader might need to read these papers first.

The papers aim at finding an answer to the following question: "how can two devices that do not share any secrets establish a shared secret key over a wireless radio channel in the presence of a communication jammer?". In the assumptions of this question no pre-shared secret exists so current anti-jamming techniques fail to accomplish this task. The proposed U-DSSS scheme however solves this issue in the following way: The jamming-resistance property of the channel is achieved as usual by choosing secret

DSSS codesequences. However since the two devices don't share any secrets that could be used to agree on a sequence, the sequences are chosen randomly from a given range. Although this channel is very error-prone and no fast or completely reliable communication is possible, the properties suffice to perform a key establishment protocol that will result in a key which then can be used to agree on a secret DSSS codesequence.

1.3 The system

1.3.1 Implementation

The system described in this document is a proof of concept for the proposed scheme. The declared aim was clearly an easy to use implementation that serves as a proof of concept, can easily be extended and is able to produce well-traceable results for further research. Performance and robustness issues were also taken into account but had a lower priority.

The main project delivery consisted of the system itself as a VMware image, including the full sourcecode, a detailed class documentation generated with Doxygen, short preliminary project reports and presentations, this final report, the measured results, many C++ and Python examples and many test files which were extensively used during the development phase.

Throughout this report I'll try to explain or at least provide links to everything the reader needs to know in order to be able to completely follow the implementation of the system. My aim is that, following this report and the provided links, the reader should actually be able to implement the system on his own or implement similar systems without having to waste too much time on gathering the necessary background information.

1.3.2 Environment

For the implementation I used *software-defined radios* (SDR), where the GNU Radio framework [3] probably is the current state-of-the-art. SDRs allow the developer to implement the whole signal processing blocks completely in software, giving him the freedom to code exactly the properties that are needed for the system. As RF front ends I used 2 USRPs [8] equipped with the RFX2400 daughterboards, having the following characteristics [9]:

- Frequency range between 2.3 GHz - 2.9 GHz
- Four 64 MS/s 12-bit analog to digital Converters
- Four 128 MS/s 14-bit digital to analog Converters
- Four digital downconverters with programmable decimation rates

- Two digital upconverters with programmable interpolation rates
- High-speed USB 2.0 interface (480 Mb/s)

DSP blocks

Although GNU Radio provides a big set of standard DSP (digital signal processing) blocks the overlap with what I needed was so small that I decided to implement the whole system from scratch without reusing the existing code. That way I was able to get rid of overhead like error correction or the GNU Radio way of packaging data into message blocks, which would have had an impact on the system performance and thus dilute the measured performance times. Also that way when running into problems I was quickly able to tweak exactly these parts of the system that needed to be changed.

Deployment details

All the coding and the tests were done inside a VMware Player [11] (Version 2.0.1 build-55017) running an Ubuntu 7.10 Gutsy Gibbon [12] guest OS and a manually installed GNU Radio 3.1.1 release. This way the deployment can be done very easily by burning the VMware image on a DVD which will run on any host OS that is supported by the VMware Player. The development installation ran on an IBM ThinkPad Lenovo T61.

1.4 Chapter overview

This report is split into 4 Parts: Background, The System implementation, Experiments and Conclusions and finally the Appendix. Throughout the whole report I've set great value on providing the reader with useful links and resources that I've found during the work on my project.

In the *background* part I'll try to cover and summarize all the important knowledge that I've learned during my work. There will be a chapter about DSP and one about SDRs and GNU Radio [3] in general. However since there are already very good resources available about DSP and GNU Radio and a detailed description would be outside the scope of this report, I'll just summarize the most important facts needed for this report and guide the reader to online resources for further information. Thus in the bibliography at the end of this report there will be a great collection of links about all the covered topics.

The part about *the system implementation* will cover the actual work I've done during the last few months. It'll contain an overview of the system on the whole and then go into the details of the individual blocks. I'll also explain the signal path of a datablock from DSSS spreading over binary phase shift keying to passing the USRP and show the different stages with figures that either display the FFT spectrum of the signal or the oscilloscope.

Experiments and Conclusions will describe the testing environment, the test cases and the results. Graphs will show the performance of the overall system and the effects of changing single parameters. Finally I'll conclude and mention possible enhancements and future projects.

Part I

Background

Chapter 2

GNU Radio Overview

2.1 Introduction

GNU Radio provides an open source framework for developing SDRs. Due to it's flexibility and motivated community it represents the current state-of-the-art for SDRs. Also the GNU Radio Development Team developed a big set of filters and applications that can easily be extended and adapted to the current needs.

To support the further development and to add a flexible open source RF front end to GNU Radio, Matt Ettus, a member of the GNU Radio Team, founded the Ettus Research LLC [8] and started to build the Universal Software Radio Peripheral (USRP). This device is built using a flexible, open source design that allows 4 different daugherboards to be connected to it, each daughterboard working on its own frequency range. Currently daughterboards are available for the ranges between DC up to 2.9 GHz. Probably the most used daughterboard is the RFX2400 which is a transceiver between 2.3 GHz to 2.9 GHz.

Figure 2.1 shows the USRP Motherboard.

2.2 A common SDR application

The information summarized in this chapter was found at [3], [4], [5], [6] and [7] which together represent a great resource for starting with SDR and GNU Radio.

2.2.1 RF front end

A common SDR system consists of the RF frond end that is connected to an ADC (analog to digital converter) which produces more or less highspeed data samples. These samples are completely processed in software, the application code and the DSP filters. For a GNU Radio application the RF

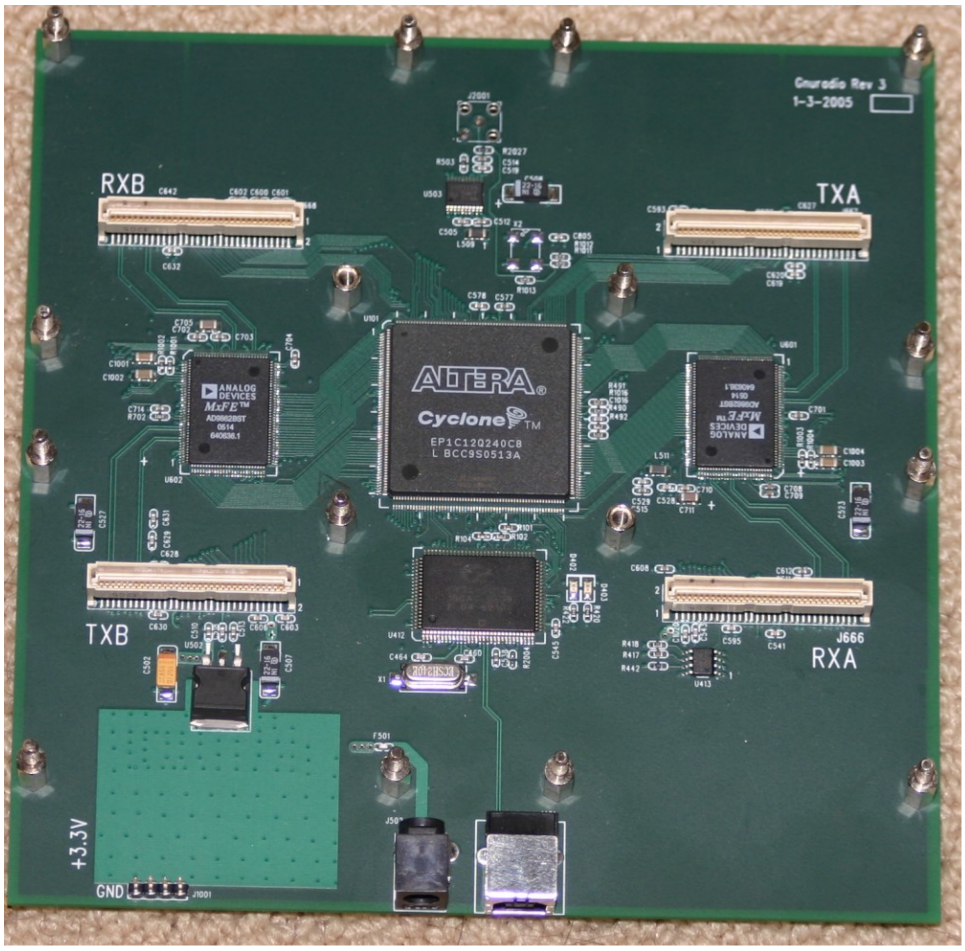


Figure 2.1: Picture of a USRP Rev. 3 Motherboard, taken from [8]. It illustrates the slots for the 4 daughterboards (RX/TX A and B), the USB 2.0 connector and the power plug

front end and the DAC/ADC is implemented in the USRP. It can be seen as a blackbox that has just 1 input which is the RX/TX center frequency. All data, that is sent to the USRP is modulated to (multiplied by) the carrier frequency which results in a translation of the baseband signal to the carrier frequency in the analog domain. However for USRPs this translation is not done directly but it's split into 2 distinct parts: the analog and the digital part.

The digital part of this translation is done by the DDC/DUC (digital down/up converter). This part basically does exactly the same as the analog, but this time it's done in the digital domain, which allows fine tuning, fast frequency changes within the bounded spectrum (bounded by the speed of the ADC/DAC) and especially it allows to decimate the signal, i.e. decimate the data stream to a datarate that can be sent over the USB cable. Figure 2.2 shows a schematic diagram of a DDC which consists of a local sine / cosine generator to translate the signal followed by a low pass filter and a downsampler to decimate the signal.

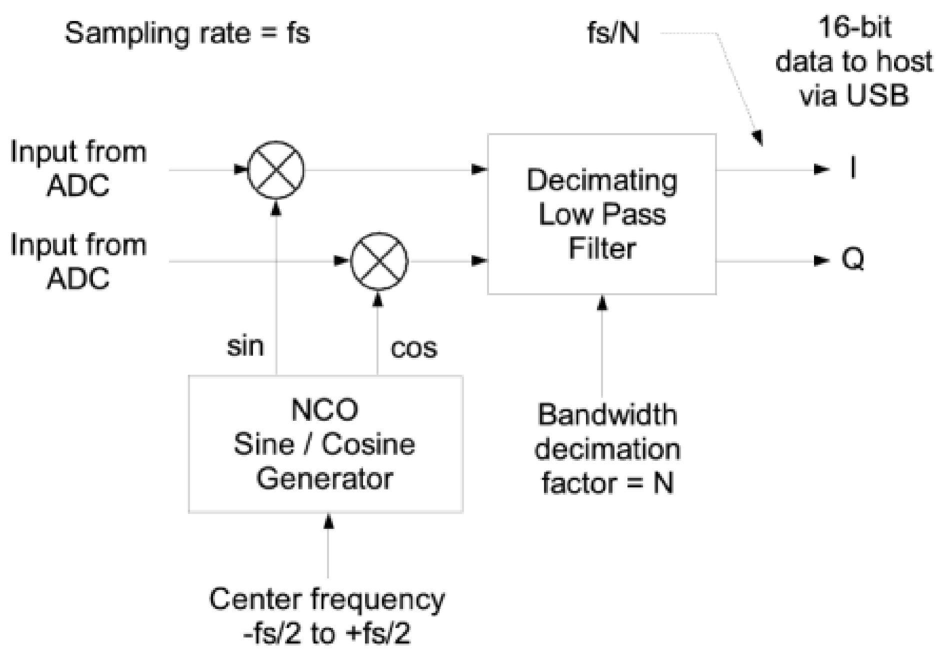

Sampling rate = fs
fs/N
16-bit data to host via USB
Input from ADC
Input from ADC
Decimating Low Pass Filter
I
Q
sin
cos
NCO Sine / Cosine Generator
Bandwidth decimation factor = N
Center frequency -fs/2 to +fs/2

Figure 2.2: Digital Down Converter Block Diagram, taken from [4]. It consists of a local sine/cosine generator followed by a decimating and low pass filter.

So if the USRP antenna catches a signal at *carrier frequency* (RF) it gets translated down to the *intermediate frequency* (IF) in the analog world.

That's where the ADC digitizes the signal and forwards the samples to the DDC. The DDC decimates the signal, applies a lowpass filter and translates it down to baseband before sending it over USB to the software world.

2.2.2 The software world

Now the GNU Radio framework takes care of the software world, i.e. it provides interfaces to the USRP, cares about data buffering and linking the custom DSP filters together.

The GNU Radio framework is designed as a two layer architecture. This is the design layer and the signal processing layer. In the upper layer Python [13] [14] is used to build and run a graph which represents the DSP blocks and the dataflow between them. A DSP block is implemented in C++. It must extend the gr_block baseclass and follow certain naming conventions. Input and output buffers link the different DSP blocks to each other. Other than that, a DSP block is pretty free to do to the signal whatever is needed to do. [6] provides a very good tutorial on how to write a signal processing block, how to compile it properly and how to link it to other blocks using Python.

Every block that is used for the application needs to be created and linked in Python, while constructing the flowgraph. The GNU Radio framework then handles the creation of input and output buffers, starting and stopping of the threads and the forwarding of data from one block to the other according to the definitions in the flowgraph that has been defined in Python. Of course not all blocks can be connected to each other. The distinction between sources (blocks that just produce data), sinks (blocks that just consume data and don't have any output) and other blocks is done via naming conventions and the input/output signatures. These signatures are created when initializing the block in its constructor and they also define which sort of input/output buffer elements (e.g. floats or complex items) are expected.

2.2.3 Helpful tools

During my work I found the following two tools that are provided by GNU Radio to be very helpful in designing DSP filters and analyzing the signals. These tools are the fft (fft_sink) and the oscillograph (scope_sink). Both blocks support float and complex input buffer items and thus can be connected to almost any other block to analyze its output behaviour in the frequency or the time domain. Most of the screenshots that illustrate this report, especially the ones that explain the pathway of a databit, show either the frequency spectrum or the signal in the time domain.

Chapter 3

DSP - As much as we need of it

3.1 Introduction

As soon as a signal passes the ADC it enters the digital world where we get a stream of samples which represent the original signal. This stream can now be freely processed the way we want before we usually forward it over the DAC back to the analog world. Thus this processing is called DSP and is done in distinct blocks that are called digital filters. I'll use the terms *DSP block*, *DSP filter*, *filter*, *block* and *class* interchangeably throughout the next chapters. Figure 3.1 shows a schematic view of a usual DSP system that gets the signal from the antenna over an ADC to the software world, where the signal gets processed and sent back over the DAC to the antenna.

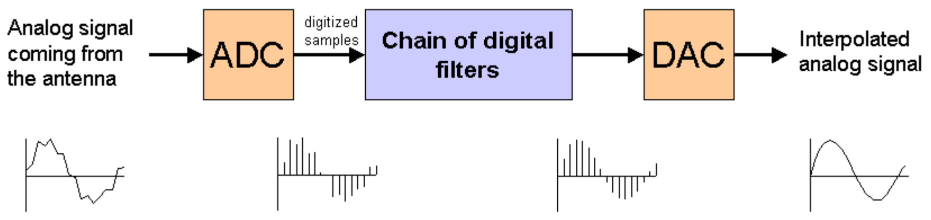

Analog signal coming from the antenna
ADC
digitized samples
Chain of digital filters
DAC
Interpolated analog signal

Figure 3.1: Schematic view of a DSP system [15] that starts with the analog signal passing the ADC followed by a chain of DSP filters and finally passing the DAC.

During my work I found the following links and tutorials to be very helpful. [15], [16] and [17] are great online resources and introductory tutorials for learning DSP. At [18] there is a complete and very detailed book that not only explains a lot of the fundamentals needed to write DSP applications but also demostrates the knowledge with many examples and goes into deep details on signal analysis. Also an excellent book on DSP systems and Digital communication is [19] by John G. Proakis.

3.2 Digital filters

The function of a filter in general is, as the name suggests, to filter a signal. This filtering can be anything from smoothing the signal, removing unwanted frequencies or noise or evaluating a certain function over the signal on the whole. Thus a filter is usually defined over its input/output behaviour. A digital filter processes the signal in the digital domain, working on the discretized sample stream. Figure 3.2 shows a schematic view of a filter.

raw stream of samples coming from the ADC or another digital filter
digital filter
filtered samples for further processing

Figure 3.2: A schematic view of a DSP filter that filters the raw incoming sample stream.

With the evolution of todays CPU processing speeds almost every function can be implemented as a digital filter without the need of using its analog counterpart.

3.2.1 DSP Filters in GNU Radio

As mentioned in the chapter about GNU Radio, the framework provides an easy, buffer-based interface to write own DSP filters and to link them together by defining the flowgraph in Python. GNU Radio connects the filters by adding input and output buffers to them. These buffers can contain any standard basetype available in C++ like complex, floats or integers. In order to connect one filter to another the items of the first filters output buffer are then forwarded to the input buffer of the second filter.

The developer is free to choose one of the many standard filters or may write his own one. When following some naming conventions and guidelines, defined in [5], writing an own filter consist mostly of only implementing the main processing function called *work*. This function gets the input and output buffers as parameters and is free to perform any functions on this data.

3.2.2 Standard filters

The GNU Radio framework provides a very easy way to define a set of standard filters. Among these filters there are also the well-known standard ones like low pass, high pass, band pass or band reject. What makes these filters so valuable is the fact that their behaviour can be easily changed in software while the system is initialized. Also their inherent properties like cut-off frequency, gain, transition width, ... can freely be defined at startup.

3.3 Translating a signal

DSP filters usually work with the raw signal at baseband. However a signal when it's transmitted over the antenna is usually not transmitted at baseband because that way different applications would collide with each other and the usable spectrum would be very limited. To solve these issues the data signal is used to modulate a much higher frequency signal: the carrier signal. This action will translate the data signal from baseband to a frequency band around the carrier frequency. As already mentioned in the GNU Radio chapter, for our application this translation is done in the USRP and the DDC/DUC. Figure 2.2 showed the schema of a DDC that multiplies the data signal with a locally generated sine / cosine wave in order to translate it from the intermediate frequency (IF) to baseband.

3.3.1 How does multiplication with a sine wave translate a signal?

Multiplication of 2 signals with frequencies $f1$ and $f2$ will result in a signal that contains the frequencies $f1 + f2$ and $f1 - f2$. Applying a low pass filter will get rid of the signal at $f1 + f2$ leaving just the $f1 - f2$ signal as a result. That's exactly what happens when the DDC translates a signal from IF to baseband.

If one of the 2 original signals is just a single sine wave, i.e. a single-frequency signal (f1), then the above equations imply that all frequencies of the other signal get translated by $f1$. Figure 3.3 displays this. It shows the result of a multiplication of 2 sine waves (*red* and *blue*) with the same frequency. This multiplication results in a new sine wave (*green*) with frequency $2 * f1$, because the other part of the resulting signal $f1 - f2$ is zero.

Figures 3.4 and 3.5 show the FFT of a data signal that consists of the following sequence 00110101. The figures show the data signal before and after multiplication with a sine wave at 100 kHz, i.e. before and after translation.

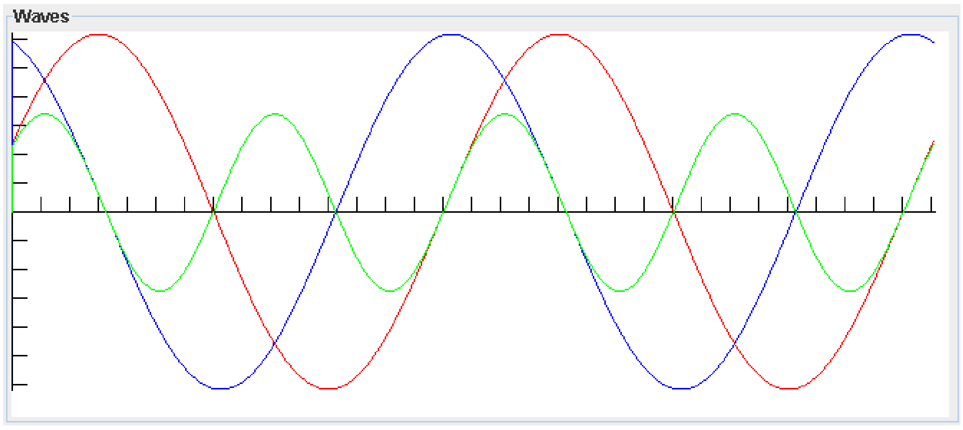


Figure 3.3: Multiplication of two sine waves with the same frequency $f1$ (red and blue) results in a signal (green) containing the two frequencies $2 \cdot f1$ and 0.

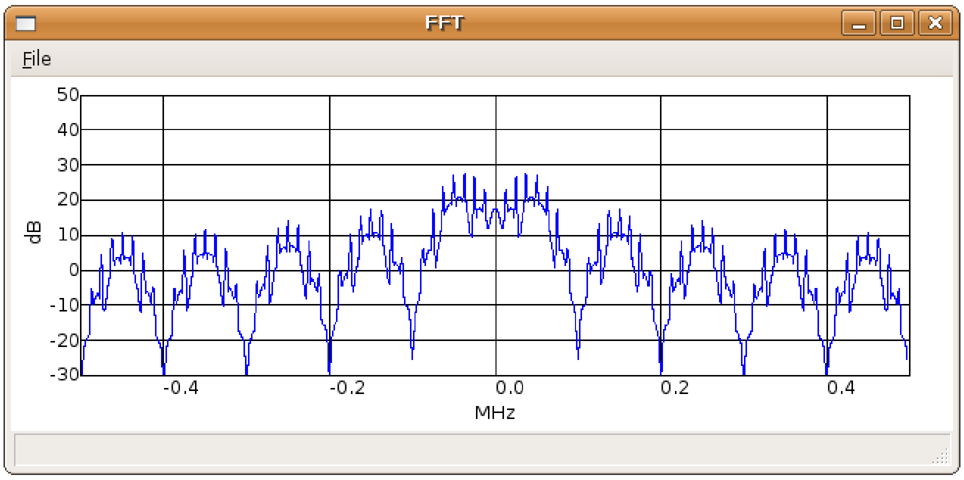


Figure 3.4: FFT of the repeated data signal 00110101 before translation, i.e. in baseband

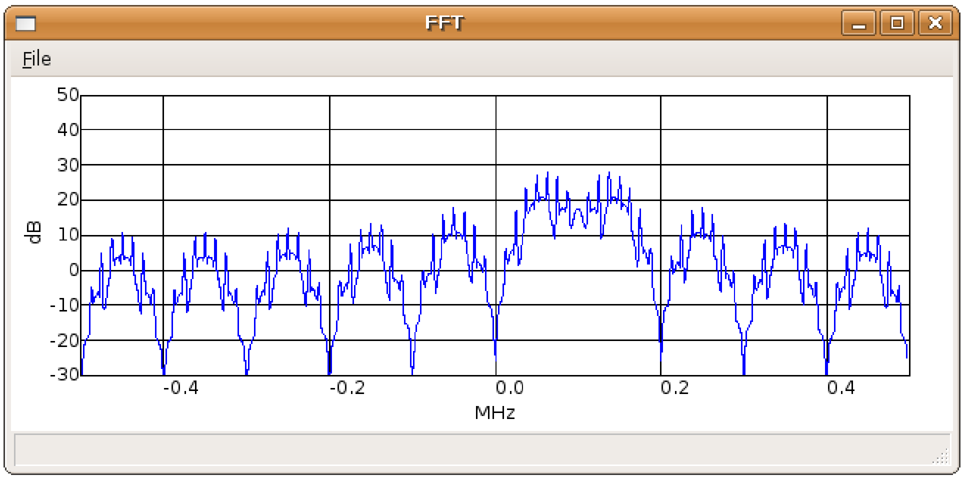


Figure 3.5: FFT of the repeated data signal 00110101 after translation by (multiplication with) a carrier at 100 kHz i.e. after carrier modulation. The multiplication results in a shift of the baseband signal in the frequency spectrum by 100 kHz.

Frequency- and phase offset

What happens now, when the DDC can't tune exactly to the desired frequency or when the senders carrier doesn't send at the exact frequency? I.e. what if there is an offset between the sender and the receiver frequency? And why is this important?

Following the equations from above the resulting signal will not be in baseband but the processing DSP filter will see a signal that is shifted by $f1 - f2$. So instead of working on a nice and clear baseband signal it works on a signal centered around $f1 - f2$. This has to be taken into account the one or the other way when working with the sampled data. Examples of this will follow in the chapter *System implementation* when talking about the PSK modulation filter.

The same issues hold for the phase as well. However there it shows up differently. It will be offset by the difference of the senders phase and the receivers phase which is important as well for some modulation schemes such as *phase shift keying* modulation (PSK). When there is a phase offset in PSK the resulting data signal after demodulation will seem to have a lower amplitude compared to the maximal amplitude that is possible. Thus it will be harder to recognize it. Please refer to [19] for a detailed explanation of this effect.

Part II

The System implementation

Chapter 4

System Overview

4.1 The bird's eye view

As already seen in section 2.2 a SDR receiver application consists of a RF front end connected to an ADC which forwards the sampled data to the software world. For GNU Radio applications these parts are represented by the USRP and the DDC for the RF front end and the ADC and the GNU Radio framework for the software part. Figure 4.1 shows the receiver part of the implemented system.

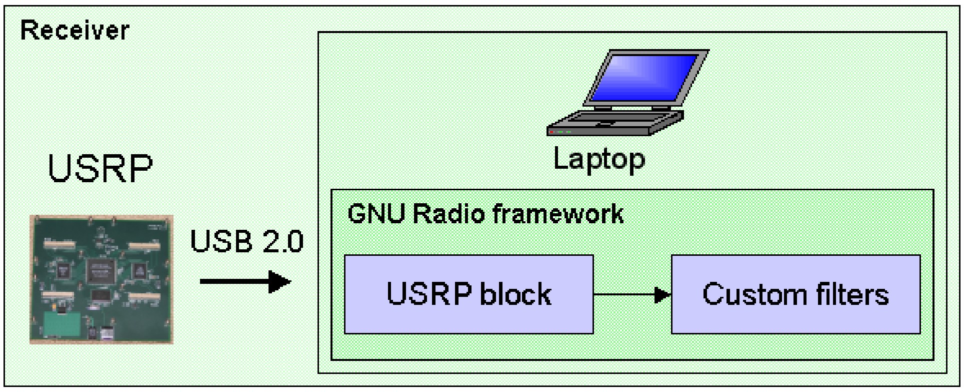


Figure 4.1: A usual SDR receiver system based on GNU Radio consists of the USRP connected over USB to a laptop that is running the GNU Radio framework.

The data samples that are coming from the USRP over USB are forwarded directly to the USRP block. This block is a DSP filter provided by the GNU Radio framework to handle basic access to the USRP device. It allows the developer to send and receive data, set properties and get information about the connected device such as the possible frequency range or the number and types of connected daughterboards.

In my system I used 2 USRPs for the sender and the receiver. Each of them is connected to a separate VMware machine running on the laptop. As daughterboards I used the RFX2400 devices which transmit in the frequency range of 2.3 GHz - 2.9GHz.

4.2 The signal pathway

Once the block diagram is clear, connecting the blocks in Python is very straightforward. That's why I'm not going to explain the Python world any further but instead show figures of the corresponding flowgraphs.

4.2.1 Sending data

The sending application starts with the *Sender* block. This block generates random test data and packs it into packages that are recognizable by the receiver. These packages contain the *message id*, the *payload data* and the *crc* which is used by the receiver to verify the correct transmission of the data. I'll show the exact package format in figure 5.2.

Such a packet is forwarded to the *DSSS* block. There each databit gets spread using the corresponding code from the randomly chosen codesequence. For each packet a random codesequence is chosen and the whole message is spread with this sequence. The codesequences are chosen from a list in a codefile as shown in figure 4.2. A definition of the codefile format will follow in the next chapter.

Now the spread signal is forwarded to the *PSK* block for modulation and finally it's sent over the USRP block to the air. The sender flowgraph is shown in figure 4.2.

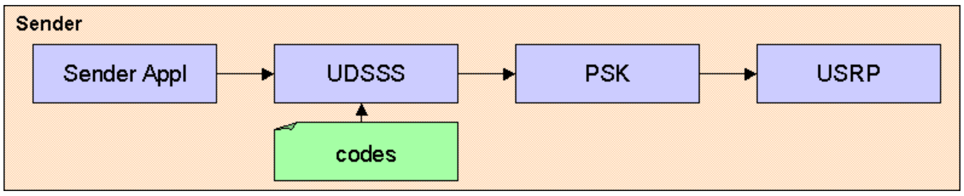


Figure 4.2: The flowgraph of the sender application. It shows the sender blocks Sender Appl, UDSSS, PSK and USRP and the codesequences file.

4.2.2 Receiving data

The receiving application is the counterpart to the sender. After passing the USRP block the data enters my code where the whole processing starts. I chose *differential binary phase shift keying* (DBPSK) modulation for the system due to its simplicity and straightforward implementation. BPSK modulates the signal by changing the carriers phase. Binary implies that there are only 2 distinct phases, 1 and −1.

So the first block after the USRP block is called *DePSK*. It demodulates the incoming BPSK signal and performs chip synchronization. Since for the test layout and results it didn't matter if the signal was weaker or stronger than the noiselevel I could make it so strong that it could easily be distinguished from the noise. That way the chip synchronization could be based on analyzing the incoming signal directly without having to despread the signal first. This of course has a big impact on the overall system performance as we will see when talking about the test results.

After demodulating the signal it needs to get despread. This is done in the *DeDSSS* block which of course needs to find the correct DSSS code sequence first before it despreads the signal. The crucial part here is the number of possible codesequences that need to be searched in order to find the right code. This value also directly impacts the overall performance of the system. However since the chip synchronization is already done at this point, the time to find the right codesequence is linear in the number of sequences.

Finally the *Receiver* block receives the data, performs a checksum verification to check if the data is valid and was correctly transmitted. It also handles the time measurement for the test cases if the data verification was successful.

The flowgraph that needs to be defined in Python is shown in figure 4.3. It represents the DSP blocks as rectangles and the linkage between them with arrows.

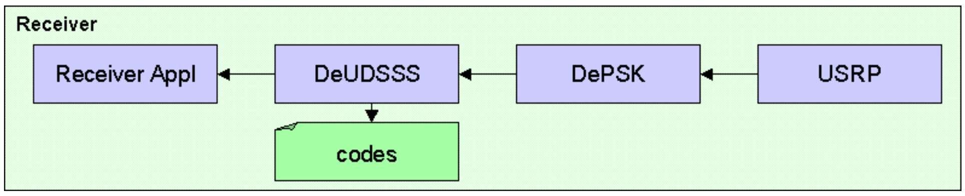


Figure 4.3: The flowgraph of the receiver application. It shows the 4 receiver blocks USRP, DePSK, DeUDSSS and Receiver Appl and the file that contains the DSSS codesequences.

Chapter 5

System implementation

The last chapters provided a good overview of what the final system looks like. In the introductory chapters GNU Radio and DSP we saw how the GNU Radio framework can help to implement a SDR application. Also we saw an overview on how DSP filters can be implemented and how they can be linked together to a complete application. System design decisions were explained in the last chapter which together formed the foundation for the detailed system design explained in this chapter.

This chapter follows the signal pathway from the sender application to the receiver and will explain every block on this path in detail. Where it helps the understanding I'll add snippets of the sourcecode or of the detailed class documentation that was generated with Doxygen [20]. Every block documentation is arranged the following way: It starts with a block overview that explains the processing result from a signal and design perspective and the input/output behaviour. Then a detailed class description will follow that includes the most important functions and their interface, if it supports the understanding. Finally I'll conclude the description by displaying how the signal changes either in the frequency or in the time domain which is probably the most interesting part from a DSP point of view.

5.1 The sender application

As has been seen in the system overview both the sender and the receiver applications run inside separate VMware machines. However all blocks are installed on both machines. Actually the development is done on one machine which is then copied and renamed to a different directory on the host machine before starting a second VMware Player and running the tests.

The sender application consist of the *Sender Appl*, *UDSSS* and the *PSK* blocks and the Python file that creates them and links them together. Figure 4.2 shows the corresponding flowgraph. These blocks together generate the test data and send it over the USRP to the receiver application. Time measurement which is needed for the tests is done by using timestamps. Since both VMware machines run on the same host environment the time measurement could easily be done by comparing timestamps directly without having to synchronize the two applications first.

5.2 The receiver application

A schematic overview of the receiver application has been shown in figure 4.3. It consist of the blocks *DePSK*, *DeUDSSS* and *Receiver Appl* which together decode the test data, perform checksum verification and store the measured times for the statistics. This processing can either be done online or offline. When the processing is done offline then the Python file replaces the USRP block by a *Filesource* and connects it to the DePSK.

5.3 Blocks in Detail

5.3.1 Codesequence input file

When looking at the sender and receiver flowgraphs one can see another block called *codes*. This block is not a DSP filter but a file that is used by the two DSSS classes. This file defines the different codesequences that can be used by the DSSS classes. Both classes read the file at startup and cache all the defined sequences in a local variable for later use.

Now what is a codesequence?

As usual for DSSS each input bit is spread by a code. For U-DSSS the choice of the current code is random. But instead of using a random code from a list the U-DSSS chooses a random codesequence where the codesequence defines exactly which bit of the input message is spread by which code. This means for example that for a message with a length of 1024 bits the codesequence must contain 1024 codes.

File format

A codesequence file defines the number of codesequences(k), the bitlength of a message(n), the chiplength of a code(N) and finally the codesequences themselves. The single chips are written by '+'/'1' or '-'/'0'. A chip is represented as $< cknN >$ in the file format specification that is shown in figure 5.1.

5.3.2 Class skeleton

Of course a block that wants to be supported by the GNU Radio framework needs to implement certain functions and fulfil certain criteria. Please refer

Figure 5.1: Codesequence file format

```
<nr. of codesequences(k)> <message bitlen(n)> <code bitlen(N)>
<c111><c112>...<c11N>
<c121><c122>...<c12N>
...
<c1n1><c1n2>...<c1nN>
<c211><c212>...<c21N>
<c221><c222>...<c22N>
...
<c2n1><c2n2>...<c2nN>
...
... ...
<ck11><ck12>...<ck1N>
<ck21><ck22>...<ck2N>
...
<ckn1><ckn2>...<cknN>
```

to [6] for a complete listing of these criteria.

To ease the development the framework provides a number of base classes that can easily be extended to implement the desired DSP functionality. The most commonly used base classes are *gr_block* and *gr_sync_block* where the main difference between them is that *gr_sync_block* assumes that there are always as many output items as there are input items.

When extending from *gr_sync_block* all that at least needs to be implemented is the function *work* which directly handles the data processing. When extending from *gr_block* there are the functions *general_work* and *forecast* which need to be implemented where *general_work* has basically the same scope as *work* and *forecast* is needed to forecast the number of input items needed when the number of output items is given.

What follows is a listing of a headerfile template that I used during my development phase. The listing should demonstrate the reader how small a minimal DSP class could be while still obeying the guidelines for a block usable by the framework. Comments that start with ! contain tags, e.g. brief used by Doxygen to produce a javadoc-like documentation in html.

Listing 5.1: A headerfile template for a class extending from gr_sync_block. It demonstrates the minimum requirements for a DSP block in GNU Radio

```
#ifndef TIBITS_CLASS
#define TIBITS_CLASS

#include <gr_sync_block.h>
```

```
class tibits_class;

typedef boost::shared_ptr<tibits_class>
   tibits_class_sptr;

tibits_class_sptr tibits_make_class();

/*!
 * \brief brief description of the class
 *
 * detailed description of the class, including
 * its purpose and the processing in the DSP
 * world.
 */
class tibits_class : public gr_sync_block{
private:
    //----- class constants -----------
    //----- instance variables ---------
    /*! \brief enable debug for this class */
    bool d_debug;
    //----- private functions ----------
    /*!
     * \brief used by the SWIG framework.
     * Will create a single instance of this class
     */
    friend tibits_class_sptr tibits_make_class();
    /*!
     * \brief constructor. initialize class variables
     * and call baseclass constructor to provide the
     * input/output buffer signatures.
     */
    tibits_class();

public:
    //----- initialization -------------
    /*!
     * \brief empty destructor
     */
    ~tibits_class();
    /*!
     * \brief dynamically enable or disable debug for
     * this class.
     * \param enable    true to enable
     */
    void enable_debug(int enable){ d_debug = enable;}

    //------ overrides gr_sync_block ----
    /*!
     * \brief main processing function
     *
     * Called by the framework to perform the
     * processing.
     * \param noutput_items  number of output items
     * \param input_items    ptr to the input buffer
     * \param output_items   ptr to the input buffer
     * \returns the number of produced output elements
     */
    int work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items);
};
#endif
```

5.3.3 Sender blocks

The overview of the sender blocks was shown in figure 4.2. Here is the class description of the three classes.

Sender Appl

Let's start with the following data (4 bits) that needs to be transmitted: 0110. The Sender Appl class will wrap this payload to a message that can be recognized by the receiver. It'll add the static *package header* $0x98765432$, a unique *message id*, a *package id* and the generated *checksum* as figure 5.2 shows. However since for our usage the class doesn't have any input but just produces output it's called a *source* in GNU Radio semantics and stands at the top of the flowgraph.

$0x98765432$	*messageid*	*packageid*	*payloaddata*	*crc*

Figure 5.2: The package format that is used by Sender Appl and Receiver Appl to send and receive data. Each package has a static package header, a unique message and package id, the payload data and a checksum.

The *work* function does exactly this. Since for the test cases there is no data that needs to be transmitted, it generates random data, creates a valid message in the above format and sends it to the output buffer using the native C *memcpy* function.

To start the pathway of the signal tour, figure 5.3 shows it in the time domain. High-values represent a logical 0 and low-values a logical 1.

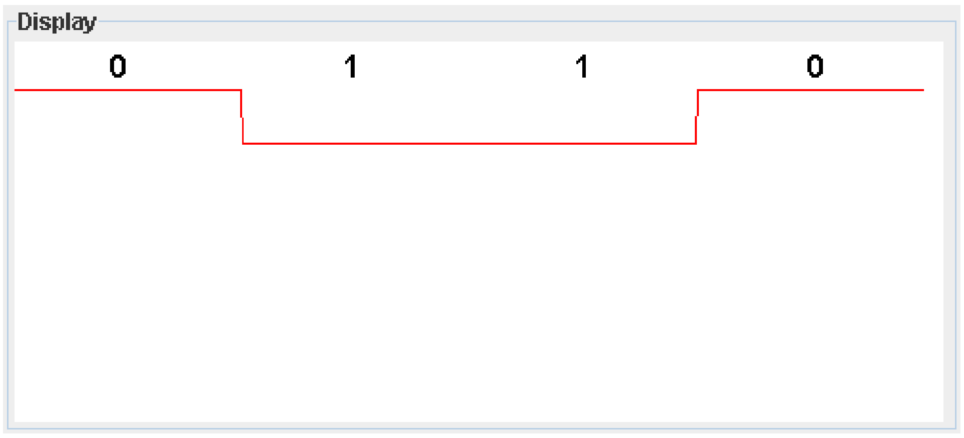


Figure 5.3: The data 0110 in the time domain displayed in red. Binary ones are represented as low-values, zeros are represented as high-values.

UDSSS

This class expects a stream of data bytes on its input that needs to be spread. If a new message starts then it will choose a new random codesequence from the file and start spreading the message with this sequence. Since we are working in the digital world spreading means just sending a complete code to the output buffer. So this class reads from the input buffer bit by bit and sends the current code from the current sequence to the output buffer either inverted, if the current input bit is a 1 or plain if the bit is a 0. The inversion comes from a logical XOR of the code with the 1 at the input.

This functionality is implemented in the *general_work* function of the class *tibits_dsss_bb*. It loops through the complete input and spreads it bit by bit by calling the functions *send_zero* or *send_one* respectively as shown in the following code snippet. These functions both take a pointer to the output buffer and the bit position within this buffer as their parameters.

Listing 5.2: The mainloop in general_work of the class tibits_dsss_bb. It loops bit by bit through the input and sends the DSSS code for a 0 or a 1 respectively to the output.

```
// number of input items of the first input stream
int ctr = ninput_items[0];
```

```
for(i=0; i<ctr; i++){
  // range check. we need at least 1 full byte
  if((bitctr >> 3) > (noutput_items-2)) break;
  // extract single bits from current databyte
  byte andmask = 0x80;
  // spread 1 byte
  for(int j=0; j<8; j++){
    if((input[i] & andmask) == 0)
      bitctr += send_zero(output, bitctr);
    else
      bitctr += send_one(output, bitctr);
    andmask >>= 1;
    // next code from current sequence
    d_curchipseq++;
  }
  // if current message is fully spread
  if(d_curchipseq >= d_msgbitlen){
    // message finished -> choose new sequence
    d_curchipseq = 0;
    d_curcodeseq = rand() % d_ncodes;
  }
}
```

However a GNU Radio application is purely stream-based and there is no direct communication between the different classes. There is just data buffers that connect them. So e.g. there is no possibility to ‘tell’ the next block that a new message has started. Strictly speaking the function *general_work* gets a chunk of data in the input buffer, whatever this data is. That’s why there is no concrete notion of a message in this class.

Since GNU Radio handles the buffering, the original message could be split into multiple parts and subsequent calls to *general_work* would each get one of these parts. Or it’s also possible that there could be multiple messages within the same buffer which is passed to the function in one single call.

Thus the decision on when to start a new codesequence is based on the number of bits that were already spread. So it starts spreading with a new codesequence after every n-th bit.

Following the pathway to the next block, the data bits 0110 are now spread with a random codesequence. Assuming the chosen codesequence with $k = 1, n = 4, N = 4$ was 0100 0101 1001 1100 then the signal would look as figure 5.4 shows. The red signal is the original data, the blue one is the spreading code and green is the resulting signal.

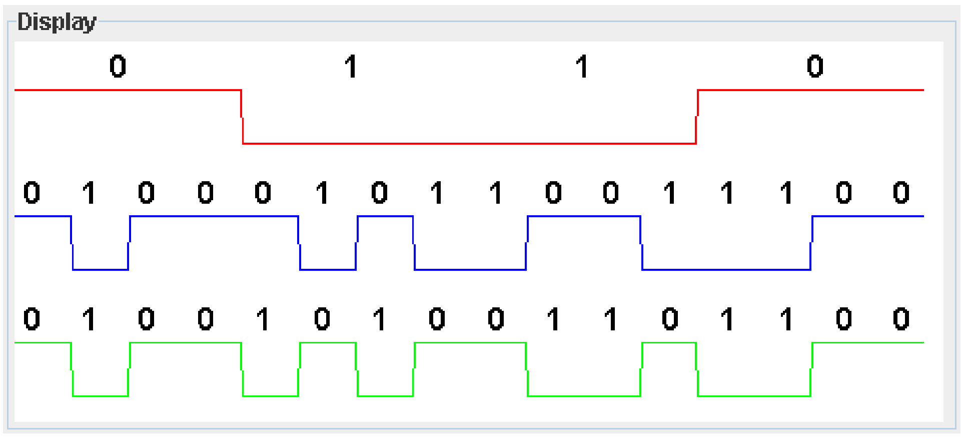


Figure 5.4: The data 0110 displayed in red after spreading with the sequence 0100 0101 1001 1100 (blue) results in the transmitted chip sequences (green).

PSK

PSK takes the spread signal as input and performs a binary phase shift keying modulation over it where binary means that there a the two phases 0 and π. From the signal point of view shifting the phase by π is the same as inverting the signal, i.e. multiplying by -1.

Thus in the analog world this filter would have to generate a sine wave, obeying the correct frequency, and invert the output whenever the input bit changes. However we are in the digital world and if you remember the section about signal translation in the DSP chapter, the input to the USRP is multiplied by the carrier frequency. So all this class has to do is to change the input bits from 0 and 1 to 1 and -1 which, when multiplied by the carrier, will set the carriers phase to 0 or π respectively.

The class *tibits_psk_bc* expects the samplerate and datarate as its parameters. These parameters are used by *general_work* to produce $\frac{samplerate}{datarate}$ output items (either 1 or -1) for each input bit. This way the carrier signal changes its phase obeying the correct datarate.

Figure 5.5 shows parts of the modulated carrier signal in the time domain. For the sake of this example I chose a very high data rate such that the carrier phase changes quickly which is very well visible in the figure. The red and blue lines represent the real and the imaginary parts of the signal.

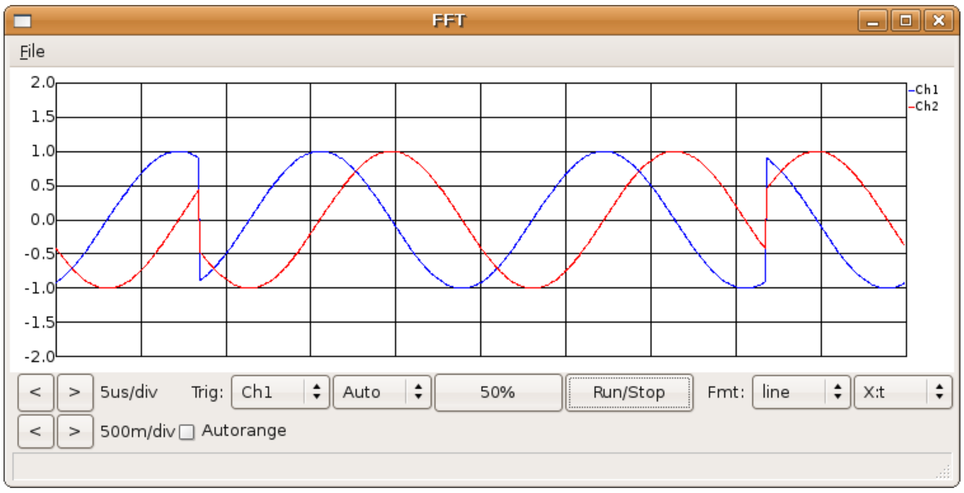


Figure 5.5: The oscillograph of a modulated carrier signal with high data rate and visible phase shifts. The red and blue lines are the real and imaginary components of the signal.

5.3.4 Receiver blocks

DePSK

When a signal passes the USRP and is sent over the air there are many effects that distort it. Usually there is not only the free space attenuation that will lower the received signal power but there is also noise that adds up. Obstacles weaken the signal through absorption and reflection and multipath effects also interfere with each other. These effects arise from the different paths that a signal can take besides the direct line-of-sight path before it gets caught at the receivers antenna. This leads to multiple copies of the same signal that overlap each other and thus weaken the received signal power. Another issue that the *DePSK* needs to care about is the frequency offset, that was explained in the DSP chapter. Two USRPs can never be perfectly synchronized, so there will always be a frequency offset between them. This leads to the fact that the resulting signal after being translated to baseband by the USRP will not look the same as it was before modulation (as shown in figure 5.4).

There is a number of error correction mechanisms and timing and carrier recovery techniques that can be applied in order to cope with the signal distortion. Among them are the *Mueller and Muller algorithm* or the *Gardner algorithm* for timing recovery and *(Digital) phased locked loops (DPLL)* or *Costas loop* for carrier recovery.

However luckily for our testing purposes we can get rid of multi-path effects and some of the noise so we don't need to apply any of these sophisticated error correction and signal recovery techniques. However the frequency offset will remain. Figures 5.6 and 5.7 show the oscillograph and FFT of how a received signal could look like when coming from the USRP as input to this class.

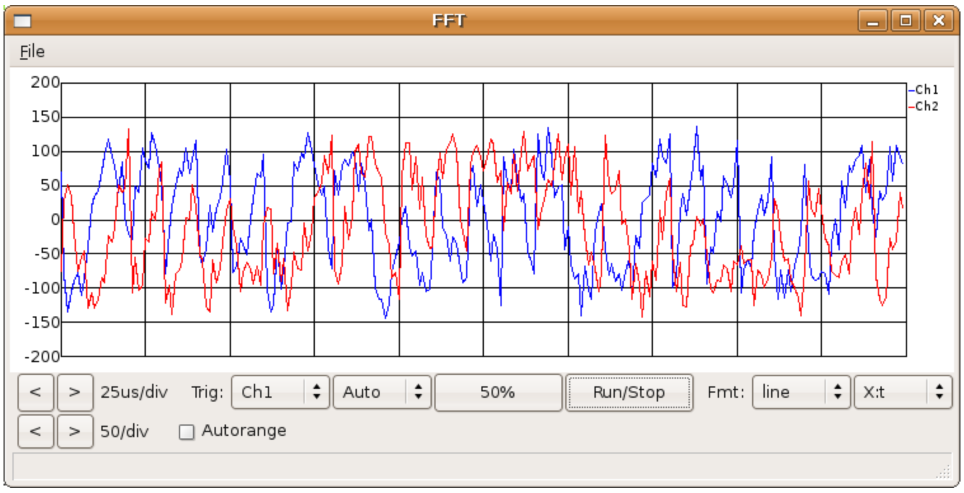


Figure 5.6: The oscillograph of a distorted signal coming from the USRP with noise, frequency offset and other attenuation. The red and blue lines are the real and imaginary components of the signal.

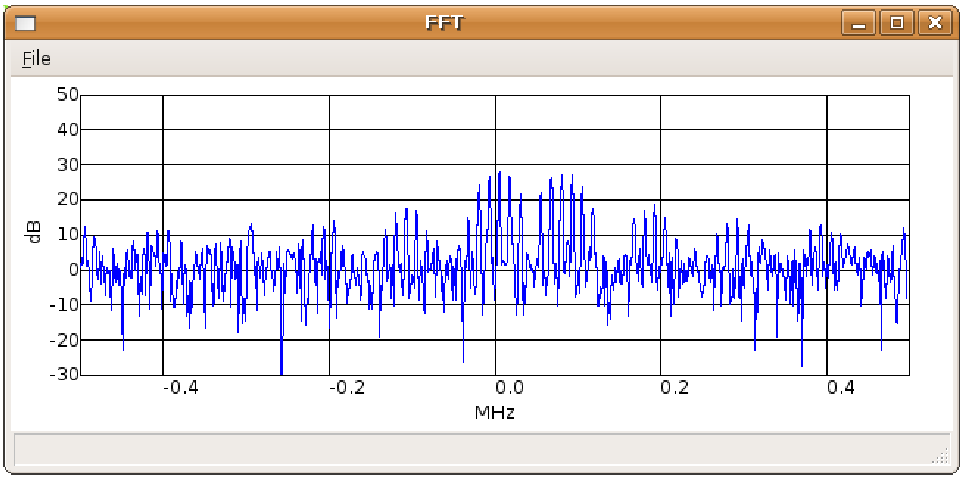


Figure 5.7: FFT of a distorted signal coming from the USRP with noise, frequency offset and other attenuation.

In order to cope with the frequency offset and some low noise the function *tibits_depsk_cb.general_work* gets rid of it by computing the angle of each pair of subsequent samples from the input buffer. If this angle is larger than a predefined threshold this means that there was a phase jump and thus the original data must have had a bit change from 0 to 1 or vice versa. More precisely the function starts with the bit 0 and scans the input for phase jumps in chunks of size $\frac{samplerate}{datarate}$ bytes. If one was found the current bit changes to 1 and is appended to the output buffer. If there was no phase jump in the current chunk then the bit 0 is appended. This way also chip synchronization is performed implicitly and doesn't have to be done in the *DeUDSSS* class.

After this class has processed the input the signal on the output should be the same again as the green line in figure 5.4.

DeUDSSS

Similarly to the discussion of UDSSS the despreading can be simplified here as well. This class expects a stream of codes at its input and generates the original data at the output. Since this is a streaming application there will be large chunks of noise coming from *DePSK* that don't contain any valid data in them. So this class will scan the input for the beginning of a message by trying out all possible codesequences that were read from the codefile. If the start of a codesequence could be found then this class will despread the complete message with this codesequence. If not then the data is simply dropped.

Despreading is done by comparing the current chunk (N bits) of the input with the current code of the codesequence. When they are equal then a 0 bit is appended at the output. If the chunk equals to the inverse of the current code then a 1 is appended. Otherwise the despreading fails and the input data is dropped.

The function *tibits_dedsss_bb.general_work* starts by calling the function *find_startpos* which searches for the start of a codesequence from a given bitposition on. When found the function *read* is called iteratively to despread 1 byte of data until a message has completely been despread or an error occurs, e.g. the comparison failed due to a bit error in the input. The same as has been told about messages in *UDSSS* holds here as well. There is no concrete notion of a message in this class. It just searches for the start of a codesequence and starts despreading the rest of the input data bit by bit until n data bits have been despread. I.e. the first N bits of the input are compared to the first code of the codesequence, then the next N bits are compared with the 2nd code, ... Then after having created n data bits (i.e. after successfully having processed $n \cdot N$ input bits) the function starts again by searching for the start of the next codesequence.

Comparing a chunk of N input bits with a code is done in the function *is_code_at* which is the most central function of this class. It expects the pointer to the input buffer, the bitposition within the buffer to start the comparison and a pointer to the code itself in its parameters. Then it starts comparing the input with the code (or its inverse) while taking care of the fact that the code can start at any position within the input and doesn't necessarily need to start at a full byte. That's why the 2nd parameter is the bitposition within the input and not the byte position.

Listing 5.3 shows this function to demonstrate how the bit-wise comparison is done and how the data bits are 'created'.

Listing 5.3: Function is_code_at from the class tibits_dedsss_bb. It compares the given chunk of input bits with the given code.

```
int tibits_dedsss_bb::is_code_at(const byte *input,
                                 int curbitpos,
                                 const byte *code)
{
  // index to input[current byte]
  int curbyte = curbitpos >> 3;
  // startbit position within the current byte
  int bitpos = curbitpos & 0x0007;
  // inverse of bitposition
  int nbits = 8 - bitpos;
  // bit masks for shifted byte comparison
  byte andmask_inv = (1 << bitpos) - 1; // eg 00000111
  byte andmask     = ~andmask_inv;      // eg 11111000

  int inv = 0;  // return value
  for(int n=0; n<d_codelen; n++){
    byte tmp; // shifted input byte for comparison
    if(bitpos == 0)
      tmp = input[curbyte+n];
    else
      tmp = ((input[curbyte+n] << bitpos) & andmask) |
         ((input[curbyte+n+1] >> nbits) & andmask_inv);
    // if comparison failed -> no code found
    if((code[n] != tmp) && (code[n] != (tmp^0xFF)))
       return 0;
    if(code[n] == (tmp^0xFF))   inv = -1;
    else                        inv = 1;
  }
  return inv;
}
```

After passing this block the original data should be restored completely again and can be passed to the receiver.

Receiver Appl

Finally after demodulation and despreading the *Receiver Appl* can buffer the messages, verify the checksums and perform time measurement.

More precisely it expects data in form of messages as defined in figure 5.2. So it scans the input for the unique *package header* to find the start of a new package. When found it extracts the *message id* and *package id* fields, buffers the *payload data* and *crc* performs a checksum verification and if successful stores the received message locally and stores the measured times.

Every time when the main function *tibits_appl_receiver.general_work* is called it sits in one of three states: Either there is no current message, so it'll search for the start of a new message in the input buffer which is done in the function *find_pck_start*. Or it's currently buffering payload data or it's storing and verifying the crc.

When a message was successfully received the time is stored locally and written to *standard output* (or a file) to allow further statistics over the processing performance.

5.4 Summary

In this chapter the reader saw how the data is created and how it flows through the system. We saw how the different sender and receiver blocks process the data, how the signal changes after each block and how we can easily cope with such things as frequency offset and noise. All these explanations were backed up by illustrative examples, figures and code listings.

Now let's have a look at the system in action when explaining the experiments, measurements and the testbed layout.

Part III

Experiments and Conclusions

Chapter 6

Experiments and Measurements

6.1 Testbed layout

For the tests I used 2 VMware virtual machines running the sender and the receiver on a laptop running Ubuntu 7.10 with GnuRadio 3.1.1 installed. Each VM was connected to a USRP which were about 3m apart.

Different code sequence files were provided to measure the receivers despreading performance. The files contained between 50 and 10000 sequences, with chip lengths of 32 to 1024 and a fixed message lenght of 2048 bit.

The processing was done offline, i.e. the produced sender signal was sampled using a 2nd USRP and the samples were stored to a file. This file was then used as the input to the receiver which started the despreading.

The testbed itself was built of the following components:

- IBM ThinkPad Lenovo T61 equipped with an Intel Core 2 Duo CPU running at 2 GHz and 2 GB Ram
- Microsoft Windows XP SP 2 as the host operating system
- 2 USRP devices connected through USB 2.0 to the laptop
- VMware Player, Version 2.0.1 build-55017
- 2 identical VMware images running a standard desktop installation of Ubuntu 7.10 Gutsy Gibbon with GNU Radio 3.1.1 installed

Figure 6.1 shows a schematic overview of the testbed.

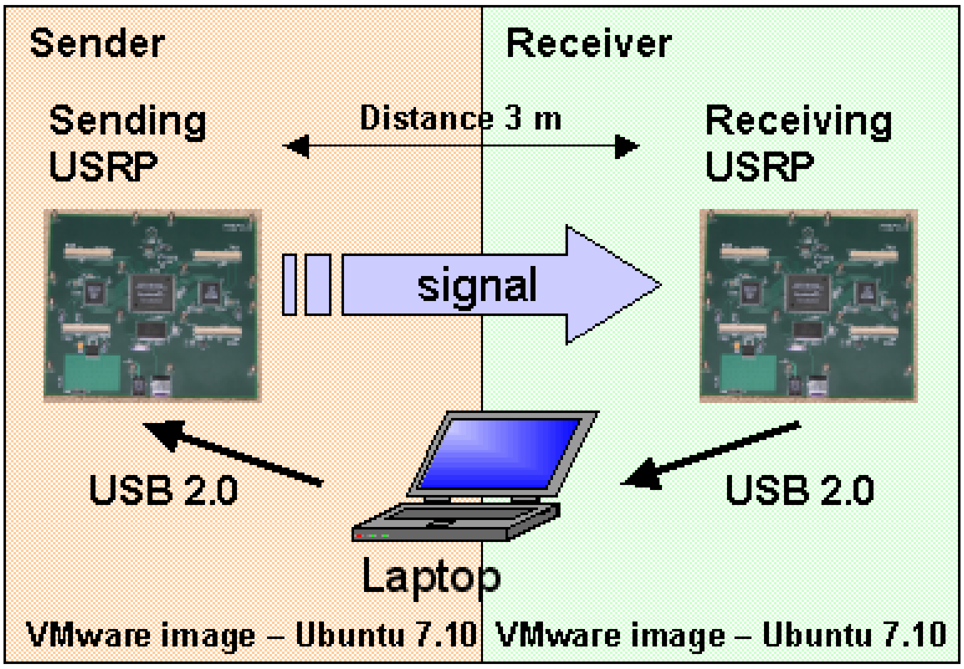


Figure 6.1: The testbed used for the measurements. Sender and receiver are 3m apart and each connected over USB to a laptop that is running 2 VMware machines simultaneously.

6.2 Test cases

Since the system is meant as a proof of concept for the U-DSSS scheme the main interest was to measure the time used by the receiver to despread the signal. Based on this the following criteria were defined:

1. Minimal, maximal and average time to despread the first bit of a message
2. Min, max and avg time to despread the whole message after the first bit was found

Let me quickly explain what these values mean. The *first bit time* is the time used by the receiver from start of the application to when the first bit could successfully be despread. So the incoming data that is read from the file is searched for any occurrence of a valid first code of one of the defined codesequences. If such a code could be found in the input then this time is measured, independently of the successful despreading of the whole message. Thus this time stands for the processing time that is used by the functions to search through the input buffer plus the time used by the GNU Radio framework to call and run the blocks, e.g. to copy output buffers from preceding blocks to input buffers of the current block.

The second value *message time* on the other hand stands really for two values. Firstly the processing time used by the block *DeUDSSS* to despread a whole message, i.e. to compare the codes from the codesequence bit by bit with the input, after the first bit was found. This is purely based on the speed of the CPU and the length of the message. Secondly as *first bit time* it also depends on how fast the GNU Radio framework handles the buffering.

All tests were repeated between 20 and 100 times in order to get meaningful average times and min-max intervals. However since running on Windows the time granularity was restricted to 10ms. Also a lot of the time that is measured is not coming from the system itself but, as we will see in the next section, from the framework that e.g. needs time to read the sampled data from the input file, create and handle buffering between the blocks, ...

6.3 Results

What follows are some of the charts that were generated out of the exhaustive test data.

6.3.1 First bit times per Nr. of Codesequences

Figures 6.2 and 6.3 compare the first bit times and their min-max intervals for different ks (number of codesequences) and code bitlengths of 32 or 1024 bits respectively. On the x-axis we see the number of codesequences and on the y-axis the milliseconds used to despread the first bit. From the charts we see that even if we had over 1000 different codesequences this doesn't have a big impact on the overall system performance. Furthermore the time used to despread the first bit is more or less constant.

However chip synchronization plays an important role as well when analyzing the system performance. As described in the last chapter chip synchronization is done at signal demodulation level already in the class *DePSK*. So searching in the input buffer for the start of a codesequence is simply a linear function in the number of codesequences because the synchronization is pretty much fixed and is not a 'free variable'. If the system was implemented differently and synchronization wasn't fixed (i.e. the DSSS signal level would be below the noise level) then the *first bit time* would more likely be a quadratic function.

The huge difference of the absolute values when comparing the two charts against each other (about 400ms for 32-bit codes and about 12,5s when working with 1024-bit codes) comes from the fact that there is much more data to be read from the file when using longer codes. Since the data reading speed is constant the amount of data to be read directly impacts the overall time used to despread the first bit.

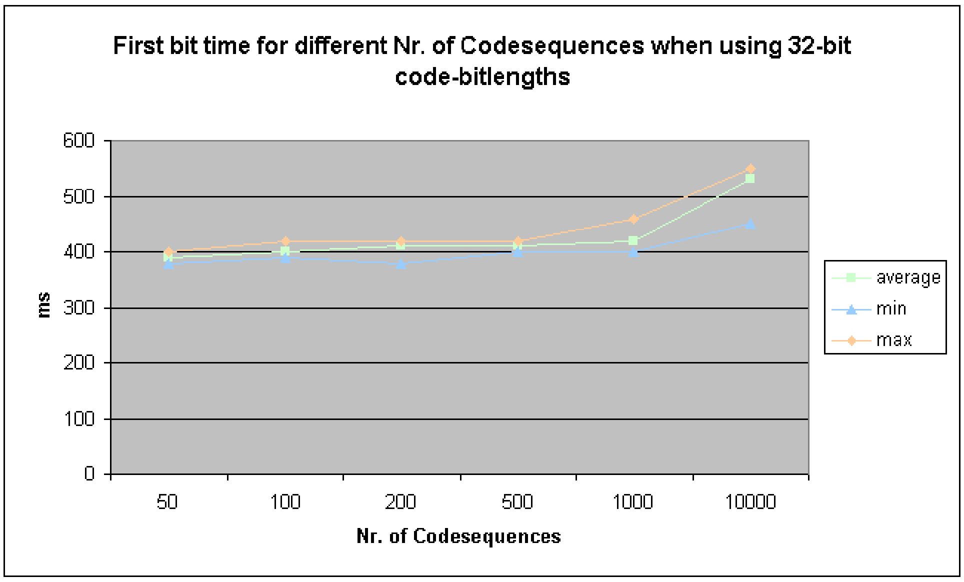


Figure 6.2: Comparison of first bit times against number of codesequences when using 32-bit codes. Averaged over 100 measurements. The average value is represented in green, the max value in red and the min value in blue.

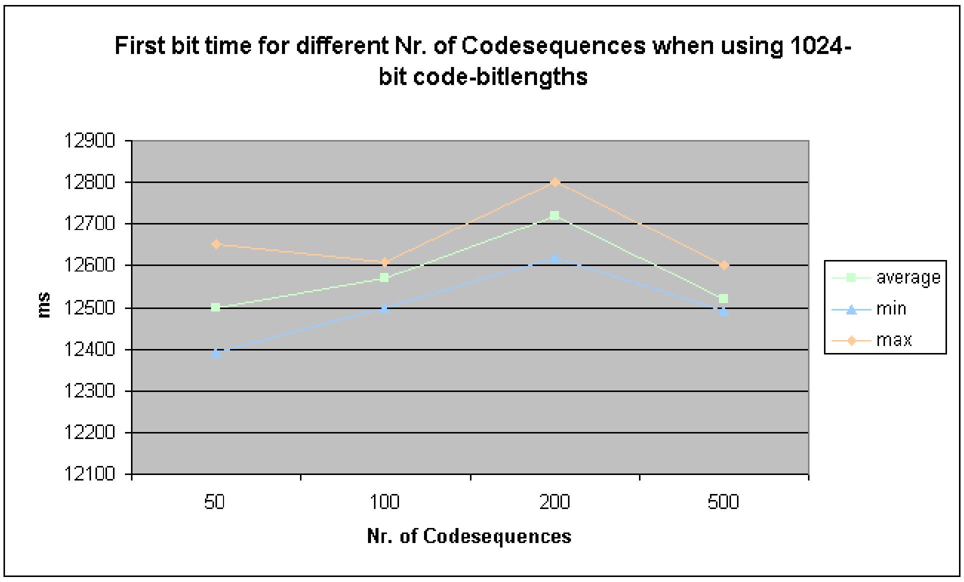


Figure 6.3: Comparison of first bit times against number of codesequences when using 1024-bit codes. Averaged over 20 measurements. The average value is represented in green, the max value in red and the min value in blue.

6.3.2 Message times per Nr. of Codesequences

Figure 6.4 compares the times to despread a complete message against different number of codesequences and 32-bit codes. As expected these values are constant, meaning that the processing time to despread a complete message after the first bit was found doesn't depend on the number of codesequences used but only on the processing speed of the CPU and the GNU Radio framework.

6.3.3 Other results

Due to the fact that reading from a file is at constant speed which impacts directly the measured times (as explained in the last subsection) it's not possible to make further conclusions about how the chosen code bitlength would impact the first bit times or the message times. The time spent for reading the data is so much bigger that the other values aren't measurable.

Also comparing this U-DSSS system against a pure DSSS system with 1 predefined code would just produce interesting results for very large ks. However since the code sets are stored in a file according to the defined format a codesequence file quickly gets very big and thus it's currently not feasible (mostly due to HD space restrictions) to generate codefiles with more than about 40000 codesequences when using the minimal code bitlength of

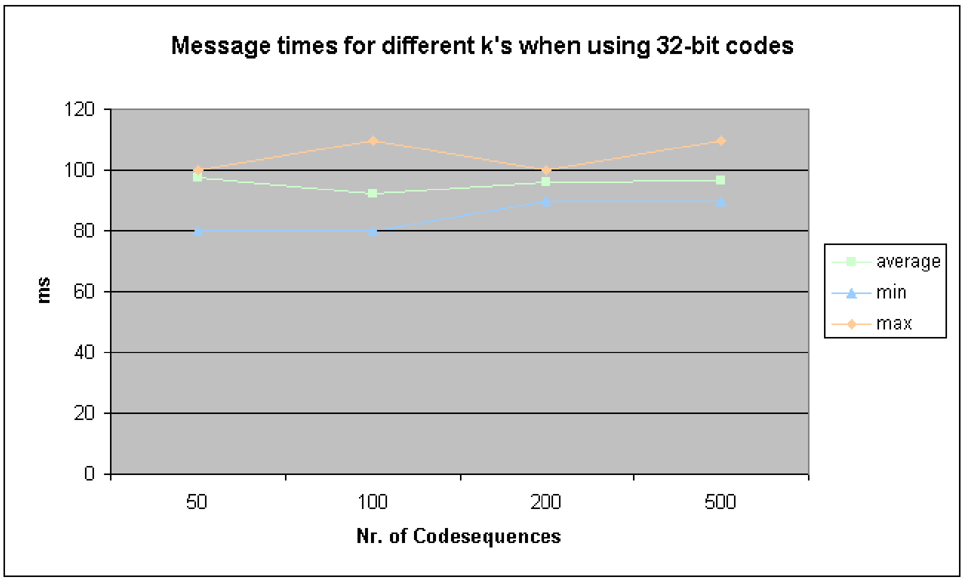


Figure 6.4: Comparison of message times against number of codesequences when using 32-bit codes. Averaged over 100 measurements. The average value is represented in green, the max value in red and the min value in blue.

8-bit codes.

So for further tests with this system it would be interesting to implement another codefile format and time measurements that don't depend on the speed of how fast the data is read.

Chapter 7

Conclusions

7.1 Review

The aim of my master thesis was to design and implement a system for Uncoordinated Direct Sequence Spread Spectrum (U-DSSS). This document started by explaining the necessary backgrounds like the GNU Radio framework and added a short introduction to what we need to know about digital signal processing. We saw what a common SDR application looks like, what a RF front end is and the corresponding parts of it in the GNU Radio world. We also learned what digital filters are and how to write and link them in GNU Radio.

Then this report proceeded to define the system implementation starting with a quick overview that showed the system on the whole and then going into details, explaining each block in detail. There we saw the sender and the receiver flowgraphs and how they communicate with each other over USRPs. Also how spreading and modulation is done in the digital world and what needs to be considered when a signal passes over the air and gets distorted. Where necessary, the design decisions were explained and why certain things were implemented this way and not the other. Finally after every block we followed the pathway of a data signal that flows through the system shown with illustrative figures.

In the third part we saw the system in action. The testbed was defined, test cases were generated and finally the produced results were illustrated with graphs.

7.2 Lessons learned

The project started with evaluating different platforms that could be used to implement the proposed system. Choosing an open source solution, GNU Radio, was generally a good choice because it leaves very much freedom of what the implementation finally will look like. Also it didn't restrict me in any sense such that there would have been something that wasn't possible to implement. Although I sometimes had to work around some limitations like the fact that the framework is purely stream-based and thus doesn't provide a simple way for the different blocks to exchange control data.

However spending more time in evaluating other systems could have been interesting. Especially when considering performance issues hardware based systems that already provide basic DSSS functionality would be interesting to work with.

But after all I found GNU Radio to be very easy and great fun to use so I'll surely do further projects with it in the future.

7.2.1 Project management

During the project a lot has changed (see the project schedule in the appendix). For a next project I'd certainly concentrate more on project management, i.e. planning, designing, thinking about the proposed ideas, scheduling more meetings with the 'customers' in order to get a clearer view of what the final system should look like. However due to the fact that the 'idea' of the final system evolved as well during the project, this wasn't necessarily possible this time.

7.3 Future work

The current system was implemented as a proof of concept only. Thus many design choices were made in order to get fast and reliable results. Although it provides everything that we defined to be absolutely necessary for the first version there's certainly much more that could have been done with U-DSSS.

Here are just a few of the many examples of what further development could look like:

- One of the major characteristics of DSSS is that the signal 'hides' below the noise level and thus is not visible to the attacker. For the current implementation of *DePSK* to work, the signal must however be stronger than the noise. Extending this implementation would make the system more secure. Although it would also have a big impact on the system performance because chip synchronization couldn't be done at PSK stage anymore but would have to be done at the DSSS level.

- Currently the system can only recognize one sender and processes the input stream only one time in order to find the strongest signal. Further development would need to be done to recognize multiple senders that overlap each other using different codesequences.

- Error correction and signal recovery mechanisms could be applied when the signal enters the system in order to make it more robust to noise and signal distortion. These mechanisms could consist of chip timing recovery, carrier signal synchronization, application of matched filters, ...
- Having a more robust system, different attacker models can be implemented to get a better insight of how an attacker can harm the system performance.
- Support fragmentation of a message, i.e. split a message into packets of same length and spread each packet with an own codesequence.
- Improving the time measurements to get closer to the time that really is spent by the system.
- Implement the key exchange protocol that is propsed in [1] and let it run against the different attacker models.

However probably the most interesting enhancement to this system would be to implement it ‘in-the-small’. Where ‘in-the-small’ means that it’s either integrated into small devices or provides an open easy to use interface such that other systems can seamlessly use it without having to make a lot of changes. This solution would allow other systems to easily bootstrap their jamming-resistant communication by exchanging keys or other information using U-DSSS.

Part IV

Appendix

Appendix A

Project Schedule

A.1 Review

Due to the nature of the project it was clear at project start that it wasn't possible to define a fixed project schedule. I still tried to define at least the most important milestones and some rough key-dates in order to have an overview of what steps will lead to the final system.

During the project evolvement a lot has changed, rendering the original dates mostly useless. E.g. the implementation changed from UFH to U-DSSS, also the measurements, results and some extensions to the system became more important than a sophisticated attacker model, so that this and other milestones got dropped.

However I think it's important to see and analyse the differences between the planned and the actual schedule of any project in order to learn for future projects and to improve.

A.2 Planned schedule

The first meeting was on Wed, 19th of Sept 2007 where we decided about my master thesis topic and fixed an official starting date, the 8th of October 2007.

Figure A.1 is a mind map that shows the different planned steps, the defined milestones and the rough due dates. This map was created right after our kickoff meeting on Oct 10, 2007.

Table A.1 explains the defined milestones in more detail.

A.3 Actual schedule

After the first 3 defined milestones were reached the project plan changed and we defined new objectives that rendered the old milestones invalid. So the actual project schedule looked as follows:

Table A.1: Project milestones

Milestone	Due date	Description
1. Gather knowledge	Oct 17, 2007	Read through the provided material, GNU Radio documentation and evaluate different destination platforms
2. Platform decision	Oct 18, 2007	Meeting with supervisors to decide which platform we will choose
3. First demo	Nov 21, 2007	Implement a first running demo on the chosen destination platform
4. Proof of concept	Dec 19, 2007	Implement the proof of concept for UFH without an attacker or jammer
5. First Measurements	Jan 07, 2008	Install and run 2 systems, define test cases and run the tests
6. Simple attacker	Jan 15, 2008	Add a simple attacker to the system, e.g. a randomized jammer, and run the defined test cases again
7. Sophisticated attacker	Jan 23, 2008	Make the attacker more sophisticated, e.g. inserting fake messages to the channel, and rerun the test cases
8. Extensions	Mar 19, 2008	tbd, e.g. further development, extend the system, performance measurements, ...
9. Presentation	Apr 02, 2008	Finish up the project, submit the final results, write the final report and prepare for the final presentation.

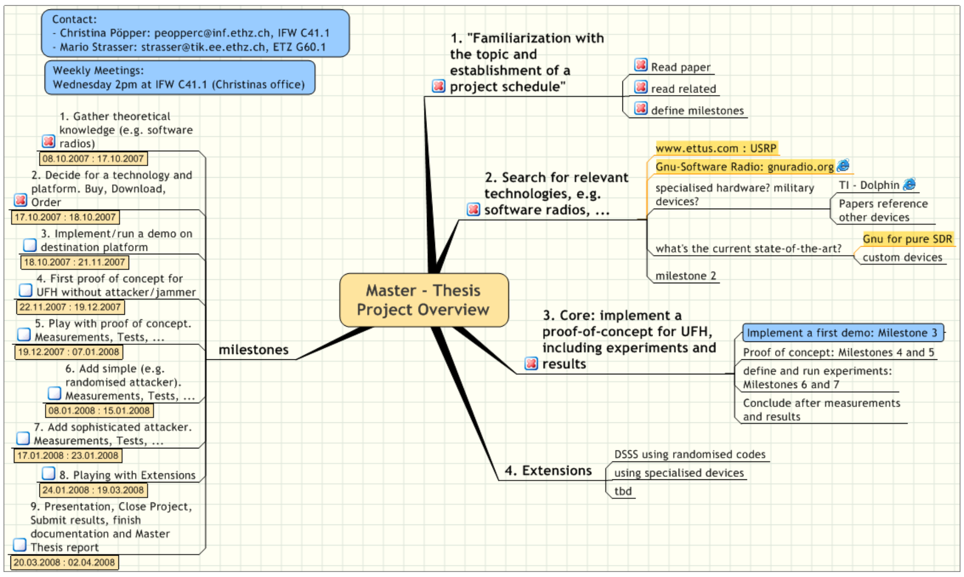

Contact:
- Christina Pöpper: peopperc@inf.ethz.ch, IFW C41.1
- Mario Strasser: strasser@tik.ee.ethz.ch, ETZ G60.1
Weekly Meetings:
Wednesday 2pm at IFW C41.1 (Christinas office)
1. Gather theoretical knowledge (e.g. software radios)
08.10.2007 : 17.10.2007
2. Decide for a technology and platform. Buy, Download, Order
17.10.2007 : 18.10.2007
3. Implement/run a demo on destination platform
18.10.2007 : 21.11.2007
4. First proof of concept for UFH without attacker/jammer
22.11.2007 : 19.12.2007
5. Play with proof of concept. Measurements, Tests, ...
19.12.2007 : 07.01.2008
6. Add simple (e.g. randomised attacker). Measurements, Tests, ...
08.01.2008 : 15.01.2008
7. Add sophisticated attacker. Measurements, Tests, ...
17.01.2008 : 23.01.2008
8. Playing with Extensions
24.01.2008 : 19.03.2008
9. Presentation, Close Project, Submit results, finish documentation and Master Thesis report
20.03.2008 : 02.04.2008
milestones
Master - Thesis Project Overview
1. "Familiarization with the topic and establishment of a project schedule"
Read paper
read related
define milestones
2. Search for relevant technologies, e.g. software radios, ...
www.ettus.com : USRP
Gnu-Software Radio: gnuradio.org
specialised hardware? military devices?
TI - Dolphin
Papers reference other devices
what's the current state-of-the-art?
Gnu for pure SDR
custom devices
milestone 2
3. Core: implement a proof-of-concept for UFH, including experiments and results
Implement a first demo: Milestone 3
Proof of concept: Milestones 4 and 5
define and run experiments: Milestones 6 and 7
Conclude after measurements and results
4. Extensions
DSSS using randomised codes
using specialised devices
tbd

Figure A.1: Original project schedule including milestones. Generated using MindManager.

Table A.2 explains the actual schedule along with the due and meeting dates.

Table A.2: The actual project schedule

Until	Description
1. Oct 17, 2007	Read through the provided material, GNU Radio documentation and evaluate different destination platforms
2. Oct 18, 2007	Meeting with supervisors to decide which platform we will choose. Decision: GNU Radio with 2 USRPs.
3. Dec 3, 2007	Presentation of the first tests and implementations with GNU Radio. Meeting date was shifted from Nov 29 due to timing collisions. Objectives changed: We will implement a system for U-DSSS instead of UFH. The exact conditions will follow.
4. Dec 12, 2007	Meeting with Prof. Dr. Capkun and supervisors discussing open questions concerning DSP, FFT, ... Preliminary definition of what the minimal requirements for a system are that will suffice as a proof of concept for U-DSSS.
5. Jan 16, 2008	Meeting with Prof. Dr. Capkun and supervisors showing a first prelimiary presentation of the implemented system. [DRAFT] The presentation slides can be found in the appendix.
6. Jan 18, 2008	Delivery of a VMware image that contains the current system running on Ubuntu.
7. Feb 13, 2008	Enhancements to the system and implementation of sender and receiver application to support time measurements, crc, ...
8. Feb 18, 2008	First definition of test cases and measurements.
9. Mar 05, 2008	Final definition of test cases, start running the test and produce the results for Christinas and Marios paper about U-DSSS.
10. Mar 12, 2008	Delivery of test results and system short description.
11. Apr 8, 2008	Enhancements to the system. Deadline for the final master thesis report.

Appendix B

Preliminary presentation Jan 16, 2008

U-DSSS – System overview

Saša Mešković

16-Jan-2008

Outline

- Today after the presentation
- GnuRadio overview
- System overview
- Blocks in detail
- Open questions
- Next steps

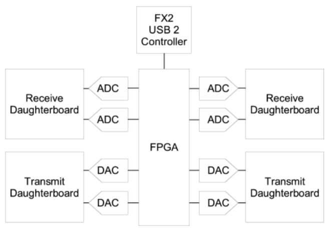
After the presentation

- VMWare image (password: stibit)
- Code is very well documented (doxygen). Have a look if you want to.
- you‘ll get new souces in a regular basis
  - copy c++/.h files to src/lib
  - copy .py files to src/python
  - run in source-root folder:
    - make clean // to clean the Doxygen-docs)
    - make // build sources and SWIG linkage
    - sudo make install // install to /usr/local/...

GnuRadio overview

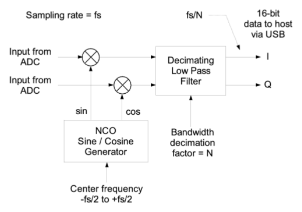
- Pure Streaming application
- Design digital filters in C++ and connect them with Python (pure C++ in next version)
- No direct communication between C++-classes
- Which values constrain our system?

USRP – Overview and constraints



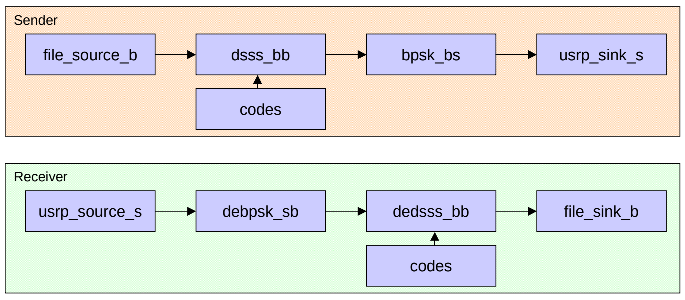
USRP

- ADC: 32MB/s
- USB: 16MB/s
- Short (16bit) vs. Complex (2 * 16bit)
- Freq-Range RX/TX



DDC

System overview



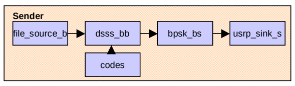
- How to do measurement?
- pure streaming application
- No communication between classes

```
Namingconventions:
<package>_<blockname>_<inform><outform>

Bufferformats (in/outform):
b   byte    f   float
s   short   c   complex
```

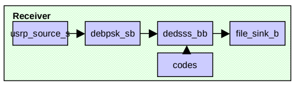
Blocks in detail – DSSS codes

- Codefile format currently:
  - <ncodes> <ncodebitlength>
  - code1...
  - code2...
- Codelenghts limited to multiple of 8 (DeDSSS)
- should be 32-bit pre-/ in- and suffix free
  - i.e. it shouldn't be possible to find 32 bit of a code in a sequence of different codes. Might lead to recognition of a wrong code and loss of data.



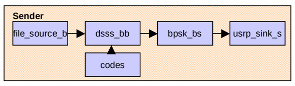
Blocks in detail – DSSS

- Read codes from codefile
  - constrains: length = multiple of 8 bit
- Spreads the input using the codes from the codefile
- operates byte-wise, not bit-wise
- for each input bit, transmit complete chip-sequence (0 as it is, 1 inverted)
- change current code after each byte



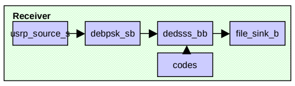
Blocks in detail – DeDSSS

- Read codes from codefile
- Code-Synchronization (functions: *findcode* and *readdata*)
- *findcode*
  - at least 32 bits (configurable) must successfully be decoded to accept a code
- *readdata*
  - at least 1 full byte must be read to be accepted
- other data is discarded (i.e. even if 7 bits could successfully be decoded)



Blocks in detail – BPSK

- Modulates the DSSS signal to BPSK
- Uses a cached sin-wave generator as basis
- transmit 1 sin-wave for each inputbit, either normal or inverted
- Parameters
  - sample-rate (rate that comes/goes from the usrp)
  - frequency: e.g. 1Mbps (speed)
  - amplitude: can be used to scale the output, when using short instead of float output



Blocks in detail - DeBPSK

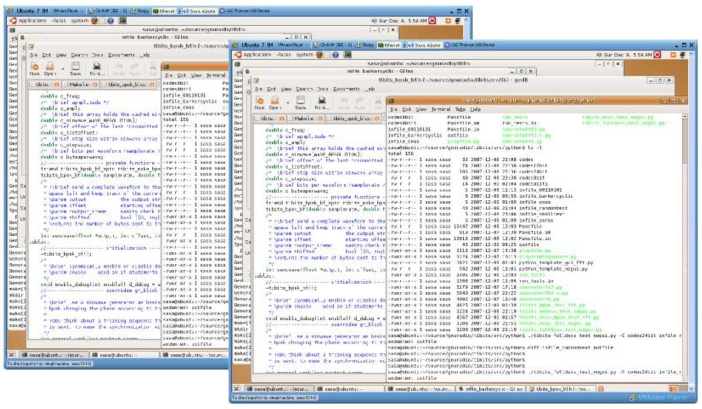
- Converts modulated signal to bit-stream
- Based on a cached sin-wave generator
- Use ‚Zero-Search‘ to synchronize local sin-wave with signal in the inputstream
- Multiply local sin-wave with inputstream
- current version has some bit errors with average noise

à needs further discussion

à How to synchronize with lower oversampling

à could a ‚trainingsequence‘ help? à no insecure

A few examples ...



Open questions & next steps

- DeBPSK issues
- How to integrate this streaming blocks into an ‚application' with time measurement, crc, ... ?

à Evaluate QPSK (native GnuRadio)

à Feedback from lower blocks to DeBPSK

à Integrate all receiver blocks into 1 class **à** similar to offline processing

Bibliography

[1] Mario Strasser and Christina Pöpper. Jamming-resistant Key Establishment using Uncoordinated Frequency Hopping. In Proceedings of IEEE Symposium on Security and Privacy, 2008, to appear.

[2] Christina Pöpper and Mario Strasser. Uncoordinated Direct Sequence Spread Spectrum. Not yet published.

[3] GNU Radio homepage http://gnuradio.org/trac/

[4] Eric Blossom. Exploring GNU Radio. http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

[5] Eric Blossom. How to Write a Signal Processing Block. http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[6] J. Nicholas Laneman. SDR Documentation. http://www.nd.edu/~jnl/sdr/docs/

[7] Eric Blossom. Listening to FM Radio in Software, Step by Step. http://www.linuxjournal.com/article/7505

[8] Ettus Research LLC http://www.ettus.com/

[9] USRPv4 specification: A flexible, extensible RF front end mainly developed for use with GNU Radio. http://www.ettus.com/downloads/usrp_v4.pdf

[10] Preliminary binary installers for GNU Radio on Windows. http://www.olifantasia.com/projects/gnuradio/mdvh/mingw/binary-installer/gnuradio-with-usrp/

[11] Virtualization software for most current operating systems. VMware homepage http://www.vmware.com/

[12] Community developed, Linux-based operating system. Ubuntu homepage http://www.ubuntu.com/

[13] Python online tutorial http://www.python.org/doc/current/tut/

[14] Dive Into Python `http://www.diveintopython.org/`

[15] Short DSP tutorial `http://www.dsptutor.freeuk.com/`

[16] dspGuru. Digital Signal Processing Central. `http://www.dspguru.com/`

[17] Bores signal processing. Introduction to DSP. `http://www.bores.com/courses/intro/`

[18] Steven W. Smith. The Scientist and Engineer's Guide to Digital Signal Processing. `http://www.dspguide.com/`

[19] John G. Proakis. Digital Communications. ISBN-10: 0071181830, ISBN-13: 978-0071181839. `http://www.mhhe.com/engcs/electrical/proakis/`

[20] Doxygen. Source code documentation generator tool. `http://www.doxygen.org/`