Martin Hinteregger, BSc

Evaluating a 2.4GHz Free-Space Data Link and associated Multirate Digital Signal Processing using Software-Defined Radio

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Michael Gadringer

Institute of Microwave and Photonic Engineering

Leica Geosystems AG

Graz, February 2020

This master's thesis was supported by
Leica Geosystems AG

Zusammenfassung

Wir leben in einer Zeit der Vernetzung, immer mehr elektronische Geräte kommunizieren miteinander. Dieser Kontakt kann nicht immer über Kabel gebundene Systeme realisiert werden. Eine Drahtlose Kommunikation bietet sich sowohl über kurze, als auch über weite Distanzen als populäre Alternative an. Der Entwicklungsaufwand mag höher sein, jedoch sprechen sich Anwendbarkeit und Kostenaufwand oft für eine Freiraum-Übertragung aus. Ein großer Nachteil einer Kabel gebundenen Kommunikation ist die geringe Flexibilität. Kaum ein drahtloses Konzept kontert dieses Problem wie konfigurierbare Wireless Systeme, so genannte *Software-Defined Radio*. Die Flexibilität wird durch eine Zusammenführung von *digitaler Signalverarbeitung*, meist realisiert durch *Field Programmable Gate Arrays*, und analoger Schaltungstechnik erreicht. Der Anteil an analogen Komponenten ist dabei so gering wie möglich, meist nur mehr Verstärkung und Filterung im Basisband und mischen mit der Trägerfrequenz.
Konfigurierbare Wireless Systeme sind in den letzten Jahren vermehrt im Einsatz, sowohl in der Industrie, als auch kostengünstig im Amateurfunkbereich. Für letztere Anwendungen kommen vor allem lizenzfreie Frequenzbänder ins Spiel, und die Werkzeuge für das schnelle entwickeln einer digitalen Signalverarbeitung bei dieser Anwendung sind oft frei für jeden verfügbar. Für eine industrielle Anwendung wird die Plattform jedoch selten von einem PC, sondern eher von einem eingebettetem System aus gesteuert. Weiters muss ein großer Entwicklungsaufwand in die konfigurierbare Digitaltechnik investiert werden.
Deshalb wird in dieser Masterarbeit das Augenmerk auf *digitale Multiraten-Signalverarbeitung* gelegt. Ein großes Vorwissen in die notwendige, und vor allem optimale, digitale Signalverarbeitung spart Entwicklungszeit und -kosten. Um dieses Wissen zu untersuchen werden unterschiedliche Werkzeuge genutzt. Die Auswertung erfolgt zum einen mithilfe von Software, sowie Messtechnisch mit geeigneten Instrumenten.
Nach einer theoretischen Einführung in ausgewählte Themen befasst sich diese Arbeit mit der Konfiguration des Systems sowie einem Auswahlverfahren für mögliche SDR Plattformen. Ein Konsolen-basiertes Nutzerprogramm zum kennenlernen und einarbeiten war der erste Schritt im Entwicklungszyklus. Die folgenden Kapitel behandeln GNU Radio (ein Entwicklungstool für digitale Signalverarbeitung das frei zugänglich ist), sowie die zusätzlich programmierten Funktionen und die entwickelten Flussdiagramme. In weiterer Folge wird zur Evaluierung von Symbolen im Zeit- und Freuqeunzbereich ein selbst erstelltes MATLAB Analyseprogramm vorgestellt. Schlussendlich werden die Flussdiagramme analysiert und eine Zusammenfassung der Arbeit präsentiert.
Diese Masterarbeit wurde am Institut für Hochfrequenztechnik der Technischen Universität Graz geschrieben und in Kooperation mit der Firma Leica Geosystems AG in Heerbrugg, Schweiz, entwickelt. Die Firma Leica Geosystems AG fungierte im Zuge dieses Projekts sowohl als Entwicklungsplatz als auch als Sponsor der notwendigen Hardware. Benutzte Software Pakete sind entweder unter GNU General Public License lizenziert oder wurden selbst entwickelt.

Abstract

Nowadays a high number of connected electronic devices has been entering our every day life. These contacts can't always be realized via a wired connection. Wireless communication is a popular alternative for both short and long distances. The development cost might be higher, but applicability and expense often favor a free-space communication. A big disadvantage of a wired communication is the low flexibility. One free space concept that counters this problem are re-configurable wireless systems, so-called *Software-Defined Radio*. The flexibility is achieved by combining *digital signal processing*, mostly realized by *Field Programmable Gate Arrays*, and analog circuitry. The amount of analog components is often as low as possible, only consisting of amplification and filtering in baseband and mixing with the carrier frequency.
Re-configurable wireless systems are well-engineered and get deployed with rising popularity in industry and also cost-efficient in ham radio implementations. For these applications, most systems operate in license free frequency bands and can be implemented using open source tool kits for fast development of digital processing. However, for industrial use, a SDR platform often cannot be operated by a host PC, but rather by an embedded device. Also, a big part of the development process is getting the re-configurable digital circuitry right for the preferred application.
Therefore, in this master's thesis, the main focus is on *multirate digital signal processing*. A strong background knowledge about the necessary and optimal digital signal processing chain saves development time and cost. In order to get an understanding about this mechanism, the evaluation of the different processing steps will be conducted with software tools and with appropriate measurement instruments.
After an initial theoretical background, this thesis will discuss the setup of the system in use as well as a selection procedure of possible SDR platforms. A console based user program to get acquainted with the design environment was the first step of the development process. The following chapters will deal with GNU Radio, an open source tool kit for digital signal processing, as well as the additionally independently programmed processing steps and the developed flow graphs. To evaluate symbol data, a MATLAB tool was created illustrating data in both time and frequency domain. Finally, the evaluation of the different flow graphs will be documented with an appending conclusion of the different development and processing steps.
This master's thesis was created at the Institute of Microwave and Photonic Engineering at Graz University of Technology in cooperation with Leica Geosystems AG in Heerbrugg, Switzerland. Most of the development process was conducted at Leica Geosystems AG, which also sponsored the necessary hardware. Used software packages in this thesis are either licensed under GNU General Public License or were programmed by the author.

Acknowledgement

First, I would like to thank my supervisor Ass.Prof. Dipl.-Ing. Dr.techn. Michael Gadringer at Graz University of Technology for his competent guidance during my progress through this master's thesis, and for always keeping me on track.

I extend my gratitude to the company Leica Geosystems AG, represented by R&D Director Electronics Marco Landert, for giving me the opportunity to research this topic backed by an industry partner. A special thank you to my advisor in his team, Senior Engineer Electronics Michael Klammer, who helped me through the stages of this thesis with his many years of experience.

Further thanks to the Institute of Microwave and Photonic Engineering, a professional workplace for many years with competent colleagues and friends. Particular gratitude to my employer Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Erich Leitgeb and my tutor at the institute Dipl.-Ing. BSc Ing. Michael Vorderderfler.

For helping to guide me through the synchronization challenges, a special thanks to Assoc.Prof. Dipl.-Ing. Dr.techn. Wilfried Gappmair at the Institute of Communication Networks and Satellite Communications.

Lastly, a big thank you deserve my parents Kurt and Renate for supporting me through my years of study, and my sisters Carolin and Verena, who always had an open ear to my challenges.

Graz, February 2020 Martin Hinteregger

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ADC	Analog-to-Digital Converter
AFSL	Atmospheric Free Space Link
AGC	Automatic Gain Control
API	Application Programming Interface
ARM	Advanced RISC Machines
AUT	Antenna Under Test
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BPF	Band-Pass Filter
BPSK	Binary Phase-Shift Keying
CEG	Constellation Evaluation GUI
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSI	Channel State Information
DAC	Digital-to-Analog Converter
DC	Direct Current
DCR	Direct-Conversion Receiver
DCS	Digital Communication System
DD	Decision-Directed
DSB	Double-Sideband
DSP	Digital Signal Processors
DSSS	Direct-Sequence Spread Spectrum
DUT	Device Under Test
EIRP	Effective Isotropic Radiation Power
EM	Electromagnetic
EN	European EN Standard
ESA	European Space Agency
EVM	Error Vector Magnitude
FDD	Frequency-Division Duplexing
FDM	Frequency-Division Multiplex
FFT	Fast Fourier Transform
FHSS	Frequency-Hopping Spread Spectrum
FIFO	First In, First Out
FIR	Finite Infinite Response

FPGA	Field Programmable Gate Array
FPRF	Field Programmable Radio Frequency
FSPL	Free-Space Path Loss
GCC	GNU Compiler Collection
GFIR	General-Purpose FIR filter
GNU	GNU General Public License
GPIO	General-Purpose Input/Output
GPL	see GNU
GPS	Global Positioning System
GPU	Graphics Processing Units
GRC	GNU Radio Companion
GUI	Graphical User Interface
HDD	Hard Disk Drive
IC	Integrated Circuit
IDE	Integrated Development Environment
IF	Intermediate Frequency
IID	Independent and Identically Distributed
ISI	Intersymbol Interference
ISM	Industrial, Scientific and Medical Radio Bands
JTAG	Joint Test Action Group
LED	Light-Emitting Diode
LMS	Least Mean Square
LO	Local Oscillator
LOS	Line-of-Sight
LPF	Low-Pass Filter
LSB	Least Significant Bit
MAP	Maximum a Posteriori Probability
MCU	Micro Controller Unit
MIMO	Multiple-Input and Multiple-Output
ML	Maximum-Likelihood
MLSE	Maximum-Likelihood Sequence Estimation
MSB	Most Significant Bit
NI	National Instruments
OFDM	Orthogonal Frequency-Division Multiplexing
OS	Operating System
PAM	Pulse-Amplitude Modulation
PDF	Probability Density Function
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express
PCM	Pulse-Code Modulation
PLL	Phase-Locked Loop
PPA	Personal Package Archive

PPM	Parts-Per Million
PRBS	Pseudo-Random Binary Sequence
PSD	Power Spectral Density
PSK	Phase-Shift Keying
QA	Quality Assurance
QAM	Quadrature Amplitude Modulation
RAM	Random-Access Memory
RC	Raised-Cosine
RF	Radio Frequency
RISC	Reduced Instruction Set Computer
RMS	Root Mean Square
RRC	Root-Raised-Cosine
RSSI	Received Signal Strength Indicator
RX	Receive, also Receiver
SDR	Software-Defined Radio
SI	International System of Units
SIM	Subscriber Identity Module
SISO	Single-Input and Single-Output
SMA	Subminiature version A
SNR	Signal-to-Noise Ratio
SPI	Serial Peripheral Interface
SRC	Sample-Rate Conversion
SSD	Solid-State Drive
SWIG	Simplified Wrapper and Interface Generator
TDD	Time-Division Duplexing
TDM	Time-Division Multiplex
TED	Timing Error Detector
TRX	Transceiver
TSP	Transceiver Signal Processing
TX	Transmit, also Transmitter
UHD	USRP Hardware Driver
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VCO	Voltage-Controlled Oscillator
VHDL	Very High Description Language
WAT	Wireless Analysis Tool
XML	Extensible Markup Language
ZIF	Zero Intermediate Frequency

Chapter 1

Introduction

Wireless communication is a key component in a majority of everyday electronic products. Initial inventions in the field of electrical telecommunication date back to the early eighteen-hundreds. The beginning of the semiconductor age in 1950 revolutionized designs, succeeded in a boost of wireless technology for the past 30 years. This process lead to a shift from wired to wireless and from pure analog to a mixture of analog and digital systems in RF (**R**adio **F**requency) communications.

By now the variety and options that wireless communications enable seem almost limitless, especially considering the high amount of frequency ranges in the electromagnetic spectrum and their individual applicability. From low to high frequency ranges, from infrared over the visible spectrum: With enough work put into it, every desired form of wireless communication is feasible. However, the process from the beginning of development to the final product can mean a lot of effort, and a sufficient initial knowledge of both analog and digital theory can conserve cost and save development time.

Recent trends also shifted from a fixed frequency communication to a more adjustable approach. **S**oftware-**D**efined **R**adios, abbreviated as SDR, combine the flexibility of digital signal processing with the power of analog front ends. This enables products with a multitude of functionality and re-usability. The most recent example being the OPS-SAT satellite deployed by the European Space Agency (ESA)[1] in cooperation with Graz University of Technology. This satellite employs, among other components, a re-configurable wireless hardware, enabling a variety of different experiments without the need of extra satellites.

In this thesis, emphasis will reside on an optimum configuration of digital signal processing using the *2.4GHz* frequency band. Different technologies already operate at this frequency range, which makes it crowded, yet also one of the most well researched bands there is. Once a software-defined radios' digital processing has been configured at this carrier, efforts to switch to similar bands are low.

A typical development process first starts with configuration (a) illustrated in Figure 1.1.

The host is used to program a re-configurable digital signal processing hardware, mostly realized in form of a FPGA (**F**ield **P**rogrammable **G**ate **A**rray). Once this system is fully functional, final efforts need to be taken to replace the host with an embedded hardware to uncouple the final product from a fixed processing platform and make it a standalone mobile wireless communication system.

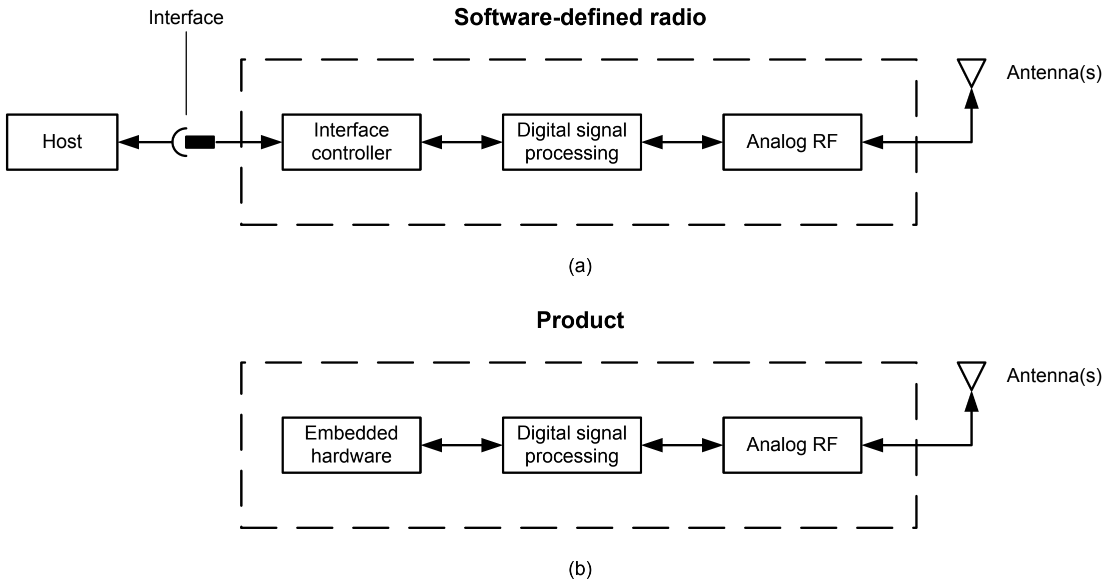


Figure 1.1: During a typical development process, the system set-up might be similar to structure (a). Once the system was configured successfully, the host gets replaced by embedded hardware (b) in the final product.

Knowledge on an optimum digital-signal processing chain, which this thesis will cover, gives developers a good starting point. The employed digital signal processing in the final product can either still be re-configurable or configured for specific uses, for example in form of an ASIC (**A**pplication-**S**pecific **I**ntegrated **C**ircuit).

In order to enable a stable communication, different concepts of wireless techniques can be used. A few important ones will be subjects in this thesis, including antenna diversity, multirate digital signal processing techniques and synchronization algorithms.

1.1 Structure

This thesis is structured into 9 different chapters. After this introduction, **2. Theory** necessary for later discussion is presented. These topics include an introduction into digital communication system design, filter theory including sampling rate conversions, diversity techniques, channel models, synchronization algorithms and finally an introduction to SDR. The different topics are supported by literature references, mainly originating from books by the topic of wireless communications and SDR design.

Chapter **3. System Overview** first discusses the parameters and limitations of the system to be developed. As a preparation to this thesis a case study was conducted to compare different possible hardware options. The results of this is presented in sections 3.2 and 3.3. As an analog front end serves a FPRF (**F**ield **P**rogrammable **R**adio **F**requency) integrated circuit, the very basic concepts of the specific chip is documented in section 3.4.

The first step was getting to know the design environment, and in this course a console based **4. Wireless Analysis Tool** was developed. After declaring the goals of this tool, section 4.2 shows how the tool was implemented. The following section will give a few use cases of selected commands, finally giving a conclusion of the tool in section 4.4.

Main development tool for the digital signal processing block chain is introduced in chapter **5. GNU Radio Framework**, together with the graphical version of the environment. Section 5.3 gives an overview of newly created and modified blocks.

Chapter **6. GNU Radio Flow Graphs** documents the structure of the digital signal processing. It starts with the primary concept design of the whole system. However, to ease computational load when processing huge amounts of data, the system was divided into several parts. The structure of each one of these is documented in the following sections.

In order to support later measurements, chapter **7. Constellation Evaluation GUI** describes a MATLAB tool created to import GNU Radio constellation data and process it. This tool illustrates the constellation diagram, samples in time domain and the frequency representation of the signal.

The evaluation of the system is documented in chapter **8. Design Evaluation**. It first describes how a frame is structured and what kind of binary test cases were used. Following is a theoretical prove of concept and a step by step evaluation of the design. These steps are supported by measurement equipment and by software. Additionally, this chapter also evaluates different symbol timing recovery algorithms.

Finally, the last chapter **9. Conclusion and Outlook** summarizes the work accomplished in this thesis and gives a short outlook on additional topics and how this system could be further improved.

Chapter 2

Theory

This chapter will cover the theoretical aspects necessary for later topics. The first part is a basic introduction in digital system design. Following sections will move on to special theories about filters, propagation diversity and an overview of different channel models. The chapter will concluded with an overview of synchronization mechanisms and an introduction to the topic of software-defined radio.

2.1 Digital Communication System

The block diagram of a typical DCS (**D**igital **C**ommunication **S**ystem) is illustrated in Figure 2.1[2], containing signal processing blocks and displaying the signal flow through the system. Goal of a DCS is to transmit information between two devices. The information can either be an analog signal or a finite numbered and discrete in time digital signal.

Top blocks show the connection from the information source to the wave emitter. Next is the transmission channel, which is the physical medium that the propagating signal must traverse to arrive at the receiving part of the system. This physical medium may be a wire line or a free-space link, to name two possibilities. Illustrated at the bottom of the Figure is the receiver part. Here the blocks are in a reversed order: from the receiving component to the information sink.

Since some of these blocks are design specific and may not be relevant for this thesis, reference to more detailed literature will be provided if necessary. Also keep in mind that the order of the blocks shown in Figure 2.1 can vary for different kinds of systems.

This section will discuss the different processing blocks, the next section 2.2 will describe the information formats passed from one block to another.

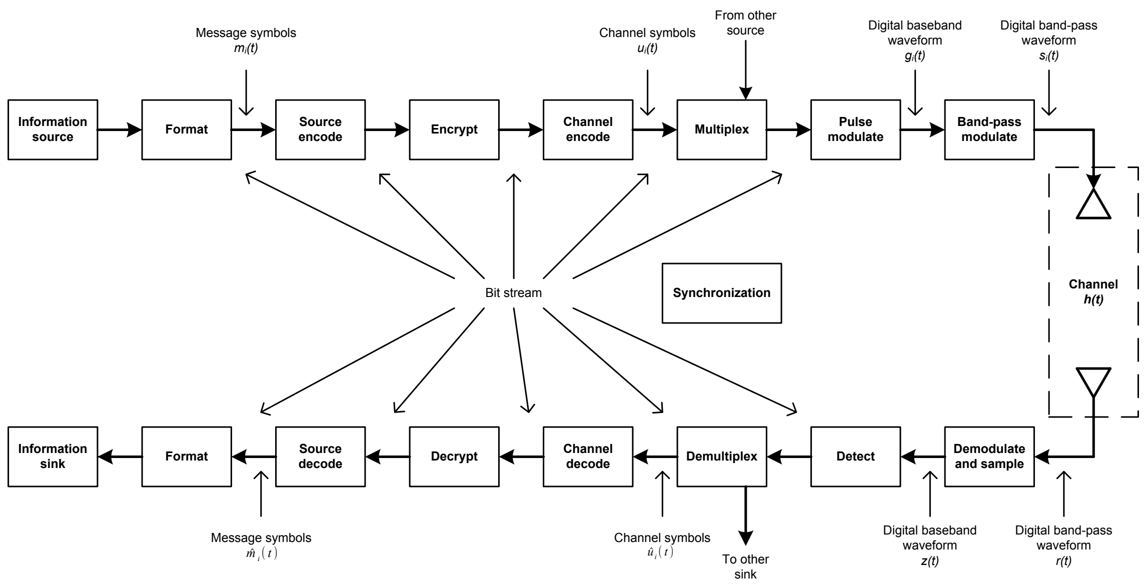


Figure 2.1: Block diagram of a typical digital communication system.

2.1.1 Transmitting

The *information source*, as stated above, can be any form of appropriate analog signal. If the following blocks need a discrete signal, the analog signal could first pass through a *format* block to sample and quantize the information into binary digits (bits).

The goal of *source encoding* is to reduce redundancy to allow more information per second to traverse the system. Note that the previously decribed format block can be part of the source encoding block. *Encryption* adds coding for privacy reasons[2]. The result is redundant neutral information compared to the output of the source encoder.

A *channel encoder* aims to reduce the probability of error in the channel by adding redundant information, for example error correction coding. This reduces the effect of noise but in turn affects the speed of the system in a negative way. If additional signals need to be concatenated with the current stream of information, a *multiplexer* can combine these. Note that multiplexer blocks can be at different locations, even multiple, depending on the system design.

Modulation is the last step in the transmit path and it consists of two parts: A *pulse modulator* transforms the binary information representation to a **baseband waveform**. Baseband is a signal with a bandwidth from DC to a finite frequency value. *Band-pass modulation* is necessary whenever the transmission medium will not support the propagation of pulse-like waveforms. This step requires a **band-pass waveform**, which means that the baseband waveform gets frequency translated to a much higher frequency, the carrier frequency. The carrier frequency must be higher than the bandwidth value of the baseband waveform.

2.1.2 Wave Emitter

Different transmission mediums require different transmission components. In case of a free-space transmission, one or more antennas, radiating either linear or circular polarized waves, can be used. Antennas follow the reciprocal theorem, meaning they can both transmit (refereed to as TX) and receive (refereed to as RX) signals. A developer can either use the same type of antennas as TX and RX, or a combination of different antennas.

If the system operates in the visible light spectrum (or fairly close to it), the TX side usually sends its information with a LED (**L**ight-**E**mitting **D**iode) or laser. Receiving the signal is realized with a photo diode.

2.1.3 Transmission Channel

There are different types of transmission mediums. The most common is an ASFL (**A**tmospheric **F**ree **S**pace **L**ink), where the signal is propagating as an EM (**E**lectro**m**agnetic) wave. As the name suggests, EM waves consist of an electric and a magnetic part which are connected via the wave impedance Z_0 and the Poynting radiation vector S.

$$\vec{S} = \vec{E} \times \vec{H} \tag{2.1}$$

$$Z_0 = \sqrt{\mu_0 \epsilon_0} \tag{2.2}$$

Assuming to traverse vacuum, both the permeability μ_r and the permittivity ϵ_r values are equal 1. From a linear polarized antenna, the emitting magnetic wave is shifted by 90 degree in space to its electric counterpart as illustrated in Figure 2.2. Depending on the used band-pass frequency and the distance between TX and RX, the electric and magnetic part is either in phase (further distance, far field) or phase shifted by 90 degrees (smaller distance, near field, inductive or capacitive coupling).

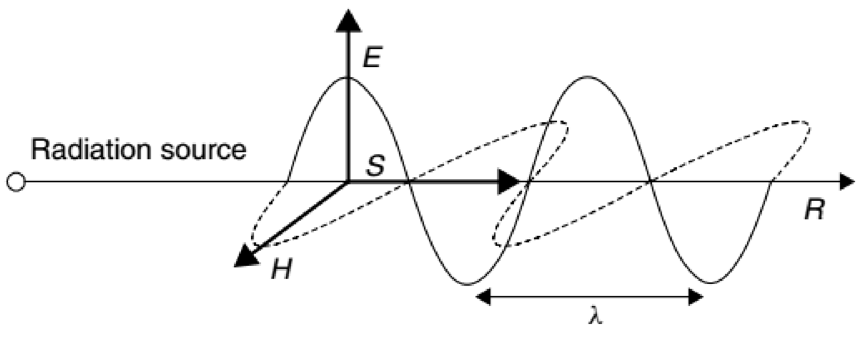


Figure 2.2: The Poynting radiation vector $\vec{S}$ as the vector product of $\vec{E}$ and $\vec{H}$[3].

The channel adds noise to the transmitted signal. This topic will be further discussed in section 2.5.

Using the visible spectrum, the channel can also be a free-space optical atmospheric link or a fibre glass connection. Other possibilities include wireline channels, waveguide channels, underwater acoustic channels and storage channels[4].

2.1.4 Receiving

On the RX side of the system, after the receiving wave emitter (e.g. antenna), the signaling blocks are in reversed order. However, there can be additional blocks not necessary on the transmitter side. Typical examples that are relevant for this thesis are *synchronization* blocks. An in depth analysis of additional blocks on receiver side will be discussed in sections 2.6 and 6.4.

2.2 Information Formats

After discussing the general design of a DCS it is now necessary to talk about nomenclature and nature regarding the information formats being passed from one block to another in Figure 2.1.

This thesis will restrain to talk about selected fundamentals of digital communications, for example the sample theorem, aliasing, ISI (**I**nter**s**ymbol **I**nterference) and SNR (**S**ignal-to-**N**oise **R**atio). There are a lot of in-depth descriptions of fundamentals in literature, and basic knowledge of these topics will be assumed[2][3][4][5].

2.2.1 Binary Representation

Bit Stream
Assuming the data is already quantified in binary digits (bits) by the format block. If a continuous stream of bits is transmitted from one block to another, this is called a *bit stream.*

Message Symbols
Single bits can be grouped together to get *message symbols* m_i:

$$m_i = \{m_1, m_2, \dots, m_M\} \tag{2.3}$$

$$M = 2^k \tag{2.4}$$

where k is defined as the number of bits (i.e. length) of the message symbol, and M as the number of possible unique message symbols. An example of this is the 7-bit ASCII character code: The length of one symbol is $k = 7$ bit, resulting in a $M = 128$ sized alphabet of symbols.

This is all assuming fixed-length code words, which means that the number of binary digits per symbol is always k.

Symbol Rate

Symbol rate f_S is the amount of symbols per second that can be passed from one block to another. *Symbol duration time* T_S is the reciprocate of this value and states the length of a symbol.

$$T_S = \frac{1}{f_S} \tag{2.5}$$

Note that different blocks can change the symbol by adding or removing redundant information. If the symbol rate changes in a system, this is refereed to as a *multirate* system.
Since all symbols still consist of a sequence of bits, a sequence of symbols can also be referred to as the previously mentioned bit stream.

Channel Symbol

While it is still valid to refer to symbol rate, the output of a channel encoder may rather be called *channel symbols* u_i. Channel symbols got more redundancy due to the nature of a channel encoder.

Data Rate

Since every symbol consists of at least one bit and all other necessary parameters have been discussed, a *data rate* in bits per second is defined as:

$$R = \frac{k}{T_S} = \frac{\log_2(M)}{T_S} \tag{2.6}$$

2.2.2 Waveforms

Since it's not possible to send a binary representation of information before preparing it (binary data has no "physical" property), it is common to transform the bit streams to digital band-pass waveforms for a RF (**R**adio **F**requency) application. In order to modulate, the system processes the message symbols to waveforms that are able to traverse the channel. This process consists of two steps.

Pulse Modulation

A *pulse modulator* transforms the binary format representation to a baseband waveform $g(t)$[2].

$$g_i = \{g_1, g_2, \ldots, g_M\} \tag{2.7}$$

Which means that there is a unique baseband waveform for every possible message or channel symbol.

Baseband is a signal with a bandwidth from DC to a finite frequency value. The resulting binary waveforms are called PCMs (**P**ulse-**C**ode **M**odulations). There are a variety of possible pulse modulators, yielding different pulse shapes.

Band-pass Modulation

If the transmission medium will not support the propagation of pulse-like waveforms, band-pass modulation needs to be implemented. A band-pass waveform $s_i(t)$ is a baseband waveform translated by a carrier wave $c(t)$ to a frequency that is much larger than baseband[2]. The baseband gets multiplied or heterodyne with the carrier.

$$c\left(t\right) = \cos\left(2\pi f_c t\right) \tag{2.8}$$

$$s_i\left(t\right) = g_i\left(t\right) c\left(t\right) \qquad i = 1, 2, \ldots, M \tag{2.9}$$

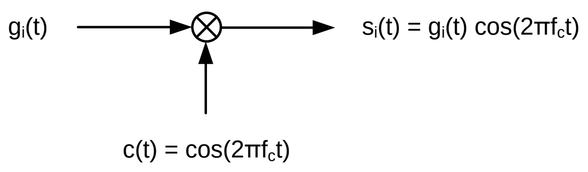


Figure 2.3: Baseband translation to higher band-pass using mixer.

The Fourier frequency shifting theorem states that this operation splits the spectrum to one part at positive and one part at negative f_c. This is called a DSB (**D**ouble-**S**ide**b**and) modulated signal.

In digital modulation, mapping symbols to a continuous waveform is accomplished by manipulating the amplitude, frequency or phase according to every specific message symbol in the pool of possible symbols M. However, these waveforms might also differ in a combination of the previously mentioned signal parameters. This mapping is done during each symbol duration time T_S and has the advantage that every unique waveform represents a specific symbol, negating the need to map every single bit and therefore improving data rates.

This results to constellation symbols in a complex plane: Data containing a real part called in-phase **I** (the cosine wave part) and an imaginary part called quadrature carrier **Q** (the sine wave part). Data is mapped to fixed points in the constellation diagram, depending on the modulation scheme, representing different message symbols. Most common modulation schemes are PSK (**P**hase-**S**hift **K**eying) and QAM (**Q**uadrature **A**mplitude **M**odulation).

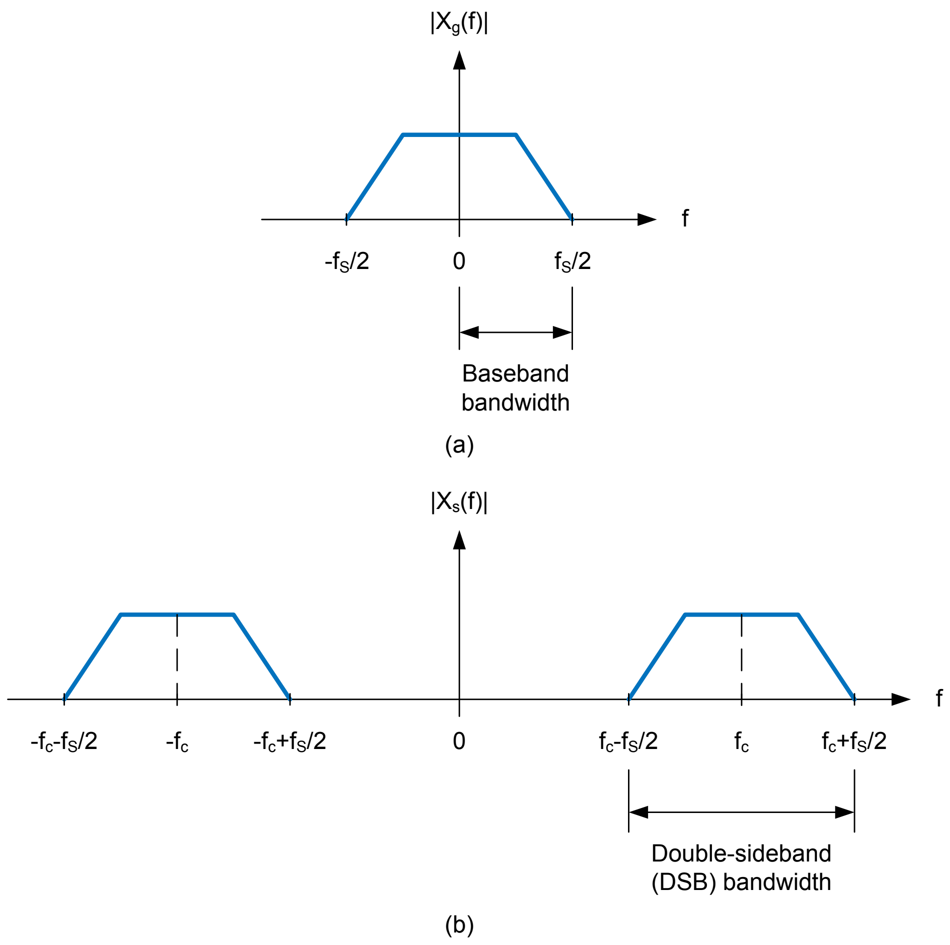


Figure 2.4: Baseband translation to higher band-pass in frequency domain. (a) represents a baseband waveform (b) represents a band-pass modulated signal around a carrier frequency.

Figure 2.5 shows three different modulation schemes in the complex plane: (a) BPSK is a binary scheme where the amplitude stays the same and the symbols are distinguished by a phase jump. (b) represents a higher order phase modulation with 4 different symbols. Note that the amplitude is still constant, only the phase differs. (c) shows a 16QAM modulation: Both amplitude and phase can be different for different symbols.

While lower order schemes like BPSK have a higher distance between symbols, higher order schemes like 16QAM have higher data rates since they posses a higher amount of unique message symbols.

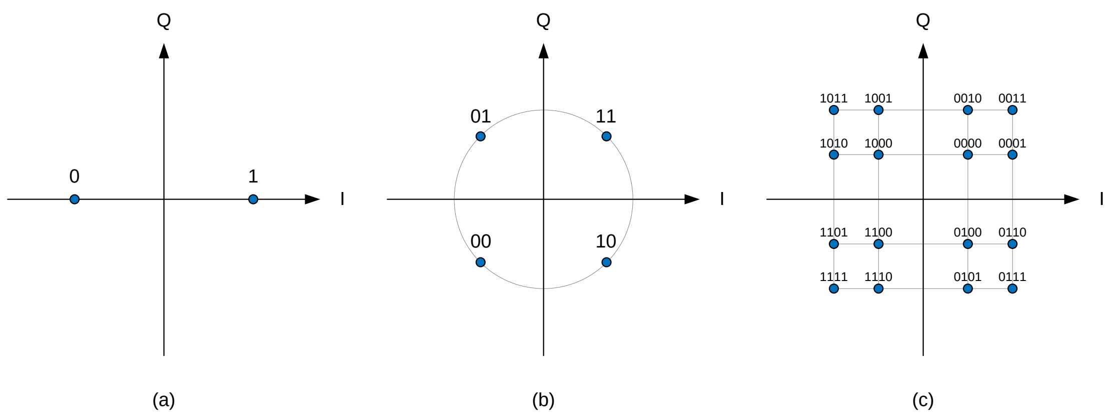


Figure 2.5: Different constellation diagrams. (a) BPSK (b) QPSK (c) 16QAM.

2.2.3 Receiver Side

After traversing the channel, the band-pass waveforms were influenced by two parameters: The channel impulse response $h_c(t)$ and the additive noise $n(t)$ imposed by the channel.

$$r\left(t\right) = s_i\left(t\right) * h_c\left(t\right) + n\left(t\right) \qquad i = 1, 2, \ldots, M \tag{2.10}$$

These channel effects will be discussed in section 2.5.

The formats between the blocks are in reversed order at the receiver side. At first the blocks will pass constellation data between them, as they synchronize the noise afflicted constellation data and map it to expected constellation points. These will then be demodulated and detected to get possible channel symbols $\hat{u_i}$ and message symbols $\hat{m_i}$, resulting in the correct data arriving at the information sink, and therefore completing the transmission.

2.3 Filters and Sampling

Filtering is common case in RF communication. There are various filters throughout the system. Especially for narrow-band applications, filters constrain signals to the bandwidth they should operate in.

Filters usually have three main goals. First, make the signal suitable to transmit over the physical channel. Second, increase SNR to reduce propagation errors. And finally third, reduce ISI from a multipath channel. Instead of discussing basics, this section will deal with ideal and Nyquist filters, as well as basics to digital filtering and sample-rate conversion.

2.3.1 Linear System Transmission

Hardware and software blocks, if processing analog information, are assumed to be linear and time-invariant. This means that blocks map an input $x(t)$ or $X(f)$ to an output $y(t)$ or $Y(f)$ with its operator $h(t)$ or $H(f)$, as visualized in Figure 2.6. The mapping is calculated mathematically with the convolution operator in time domain and by multiplication in frequency domain. Note that $h(t)$ refers to the impulse response.

$$y\left(t\right) = x\left(t\right) \, * h\left(t\right) = \int_{-\infty}^{\infty} x\left(\tau\right) h\left(t-\tau\right) d\tau \tag{2.11}$$

$$Y\left(f\right) = X\left(f\right) H\left(f\right) \tag{2.12}$$

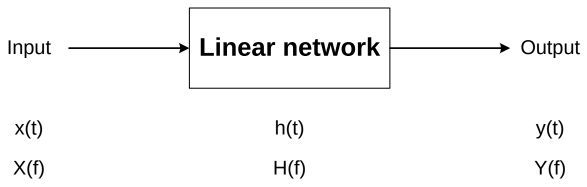


Figure 2.6: Linear System with its parameters.

An impulse response in time domain can be transformed into frequency domain using either the Laplace transform or Fourier transform. Getting from frequency to time domain is achieved with the inverse Laplace or Fourier transform.

$$H\left(s\right) = \mathcal{L}\left\{h\left(t\right)\right\} \tag{2.13}$$

$$h\left(t\right) = \mathcal{L}^{-1}\left\{H\left(s\right)\right\} \tag{2.14}$$

$$H\left(f\right) = \mathcal{F}\left\{h\left(t\right)\right\} \tag{2.15}$$

$$h\left(t\right) = \mathcal{F}^{-1}\left\{H\left(f\right)\right\} \tag{2.16}$$

2.3.2 Ideal Filters

When talking about ideal filters, assumptions are that the transfer function has a constant magnitude of 1 for the bandwidth were the frequencies should pass (pass-band) and 0 magnitude for all other frequencies (stop-band). This filter can operate at the theoretical minimum system bandwidth without ISI. Relevant to this thesis are the transfer functions of LPFs (**L**ow-**P**ass **F**ilters) and BPFs (**B**and-**P**ass **F**ilters), see Figure 2.7.

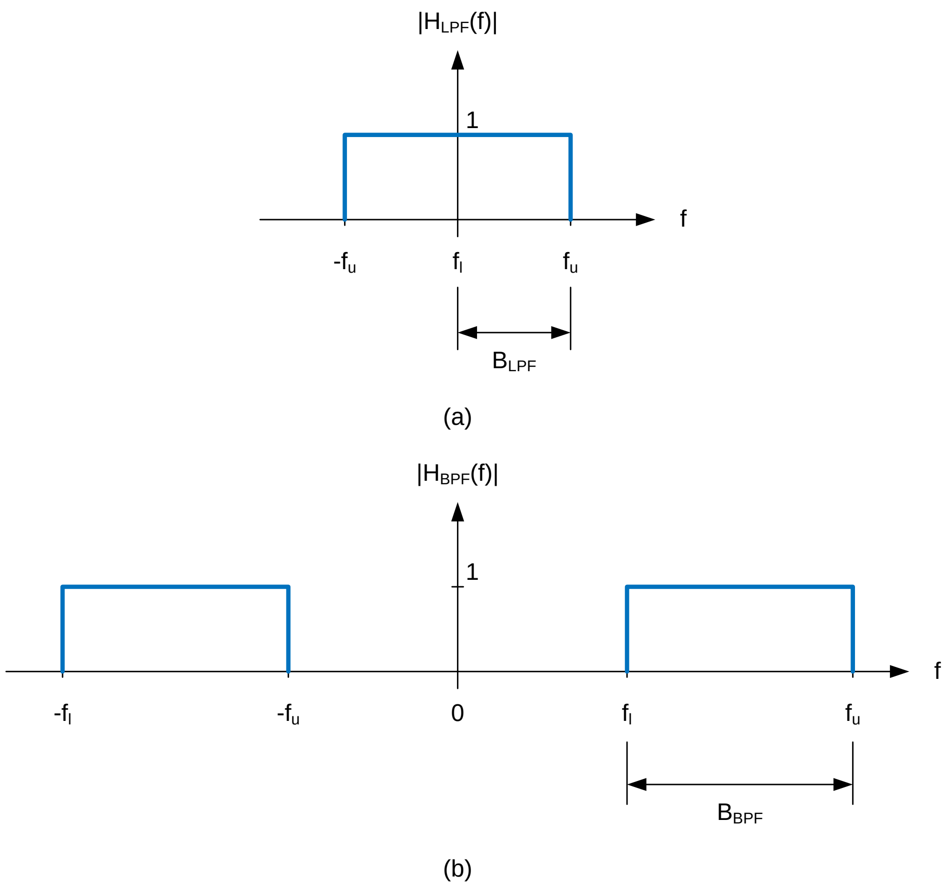


Figure 2.7: Transfer functions of ideal LPF and BPF filters.

For simplicity purpose, whenever working with an example filter during this thesis, it will be a LPF. The transfer function matching to the previous representation of a LPF is defined as

$$H\left(f\right) = \left|H\left(f\right)\right| e^{-\jmath\Phi(f)} \tag{2.17}$$

containing the frequency dependent phase $\Phi(f)$ and the frequency dependent magnitude $|H(f)|$ where

$$\left|H\left(f\right)\right| = \begin{cases} 1 & \text{for } |f| < f_u \\ 0 & \text{for } |f| \geq f_u \end{cases} \tag{2.18}$$

Using equation 2.16 to calculate the impulse response of an ideal LPF in time domain, the result is a function in the form of

$$sinc(\frac{t}{T}) = \frac{\sin\left(\frac{t}{T}\right)}{\frac{t}{T}} \tag{2.19}$$

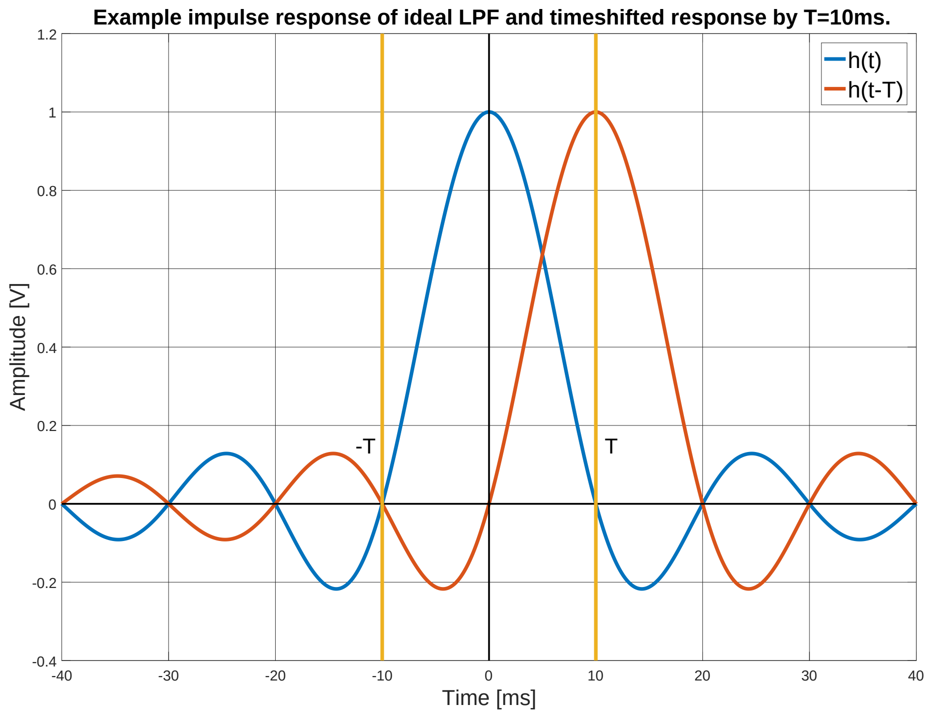


Figure 2.8: Impulse Response of an ideal LPF.

This is called the ideal Nyquist pulse, since two successive pulses time shifted by T ($h(t)$ and $h(t-T)$) show that the first one is zero at the second' ones peak, e.g. its sampling timing. Likewise all other zero crossings overlap.

The Nyquist pulse-shaping criterion states that to satisfy the described condition

$$h\left(nT\right) = \begin{cases} 1 & \text{for } n = 0 \\ 0 & \text{for } n \neq 0 \end{cases} \tag{2.20}$$

it is necessary for the Fourier transform to be equal to

$$\sum_{m=-\infty}^{\infty} H\left(f + \frac{m}{T}\right) = T \tag{2.21}$$

However, looking at the rectangular shape of the transfer function and its infinite impulse response, it is clear that such kind of filters are not realizable (acausal). Instead, a filter that is narrow in frequency and therefore wide in time (frequency-time inverse relationship), but not infinite, and simultaneously fulfills the Nyquist condition, is needed.

2.3.3 Nyquist Filter

In order to get an approximation filter with the previously discussed parameters, the general consent is to use a filter with a frequency transfer function that can be represented by a rectangular function convolved with any real even-symmetric frequency function. This type of filter is called a Nyquist filter[2].

The goal is to compress the bandwidth of the pulses to just a minimal higher value than the Nyquist minimum (see the Nyquist sampling theorem) of 2 symbols/s/Hz, which is accomplished with pulse shaping using a Nyquist filter. This is also refereed to as an equalizing filter to compensate distortion of both transmitter and the channel. For further details to equalization see subsection 2.6.4.

Equivalent Channel

It is common practice to design a TX and RX filter according to the given channel. These three parts can be combined to get the *equivalent channel* model illustrated in Figure 2.9

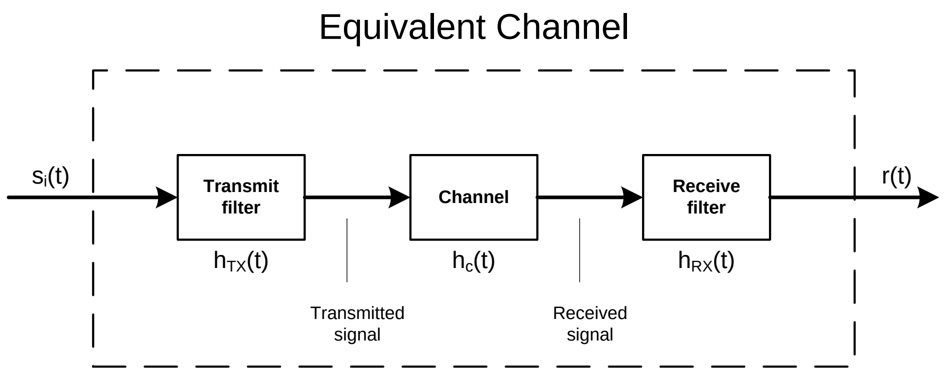


Figure 2.9: Components of an equivalent channel chain.

where $h_{TX}(t)$ and $h_{RX}(t)$ denote the impulse response of the transmitting and receiving filter respectively and $h_c(t)$ is the channel response. These three can be combined to get the impulse response of the equivalent channel

$$h\left(t\right) = h_{TX}\left(t\right) * h_c\left(t\right) * h_{RX}\left(t\right) \tag{2.22}$$

The reason being is to take the Nyquist pulse-shaping theory into account to get ISI free transmission, therefore including pulse-shaping filters when designing a system. In this configuration, the TX filter is pulse-shaping the signal, while the RX filter performs matched filter operations[6].

Raised-Cosine Filter

A RC (**R**aised-**C**osine) filter can be expressed by the frequency transfer function[4]

$$H\left(f\right) = \begin{cases} 1 & \text{for } 0 \leq |f| \leq \frac{1-\beta}{2T_S} \\ \frac{1}{2}\left[1 + cos\left(\frac{\pi T_S}{\beta}\left(|f| - \frac{1-\beta}{2T_S}\right)\right)\right] & \text{for } \frac{1-\beta}{2T_S} \leq |f| \leq \frac{1+\beta}{2T_S} \\ 0 & \text{for } |f| > \frac{1+\beta}{2T_S} \end{cases} \tag{2.23}$$

where β is the roll-off factor, which represents the excess bandwidth. This defines the steepness of the filter roll off and can be calculated using the absolute bandwidth B and the minimum Nyquist bandwidth B_0, resulting in the definition that $B - B_0$ is the additional bandwidth to the Nyquist minimum.

$$\beta = \frac{B - B_0}{B_0} \tag{2.24}$$

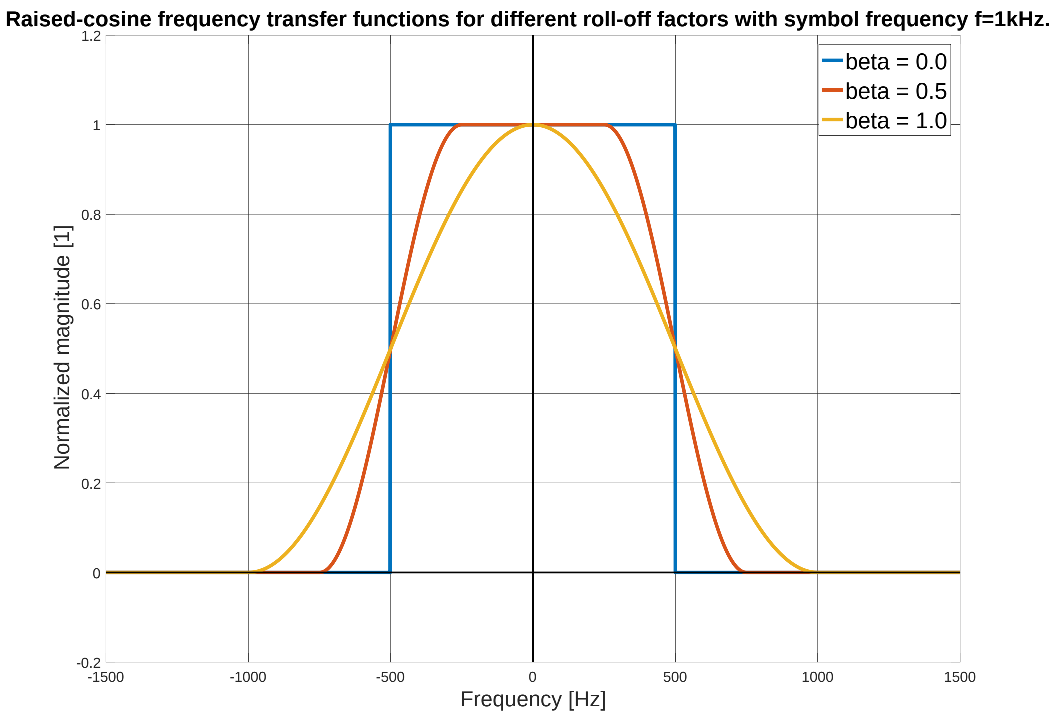


Figure 2.10: Frequency transfer function of a raised-cosine filter.

A value of $\beta = 0$ relates to a rectangular shape while $\beta = 1$ relates to an excess bandwidth of *100%*. Larger valued roll-off factors mean less sensitivity to timing errors at the cost of more excess bandwidth, while a smaller roll-off filter increases signal rates by smaller bandwidth, at the cost of larger pulse tails and amplitudes, i.e. sensitivity to timing errors. Transforming the frequency transfer functions for different roll-off factors to time domain results in the representation of Figure 2.11.

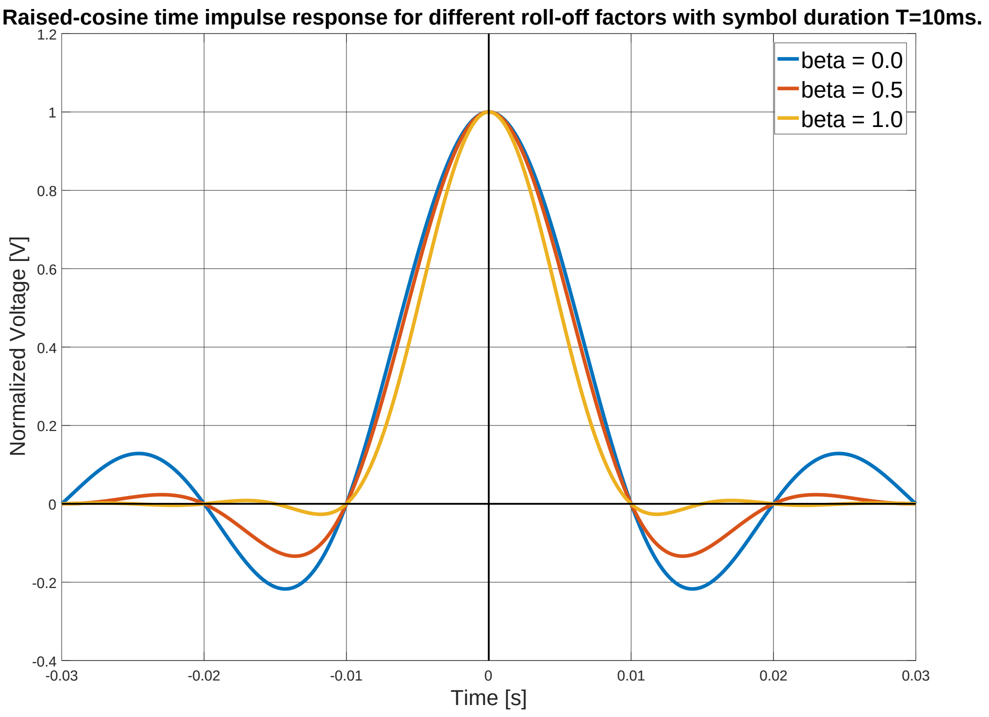


Figure 2.11: Impulse response function of a raised-cosine filter.

A general relationship between the bandwidth B and the symbol rate f_S that satisfies the Nyquist bandwidth is defined with formula

$$B = \frac{1}{2}(1+\beta)\,f_S \tag{2.25}$$

where a roll-off factor of $\beta = 0$ refers to the minimum bandwidth. Band-pass modulated signals like PSK require twice the transmission bandwidth and are refereed to as DSB signals. Therefore, the formula modifies to

$$B_{DSB} = (1+\beta)\,f_S \tag{2.26}$$

Usually RC filters are used as receiving filters, to filter out the unwanted effects of the channel. But there are also implementations where a RC filter is on the TX side, just before transmission.
Since the type of impulse response in Figure 2.11 is still not realizable (the impulse response is not zero for $t < 0$), the impulse response needs to be time shifted by a value t_0 to the right. This time shift is crucial.

Root-Raised-Cosine Filter

A RRC (**R**oot-**R**aised-**C**osine) filter has a similar appearance, but it's spectrum does not decay as rapidly. The combination of two of these filters, also known as a square-root-raised-cosine filter, is equal to the previously mentioned RC filter.

$$H_{RC}(f) = H_{RRC}(f)\, H_{RRC}(f) \tag{2.27}$$

and therefore

$$H_{RRC}(f) = \sqrt{H_{RC}(f)} \tag{2.28}$$

A perfect assumption is that the channel is ideal ($H(f){=}1$ for $|f|{\le}B$). The reason being that RRC filters do not have zero ISI. However, if both TX and RX side deploy RRC filter, the product of these transfer functions result in a RC filter as shown in equation 2.27, which result in ISI free transmission. To prove this, just modify the Fourier transformation of formula 2.22 regarding the equivalent channel.

$$H(f) = H_{TX}(f)\, H_c(f)\, H_{RX}(f) = H_{RRC}(f) \cdot 1 \cdot H_{RRC}(f) = H_{RC}(f) \tag{2.29}$$

2.3.4 Digital Filter

Digital filter deal with discrete-time signals. Therefore the convolution formula must be discretized and, using the z-transform, transformed into discrete-frequency domain.

$$y[n] = x[n] * h[n] \tag{2.30}$$

$$Y(z) = X(z)\, H(z) \tag{2.31}$$

"Since ideal brick wall filters are not achievable in practice, we limit our attention to the class of linear time-invariant systems specified by the difference equation:

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k], \tag{2.32}$$

where $y[n]$ is the current filter output, the $y[n-k]$ are previous filter outputs, and the $x[n-k]$ are current or previous filter inputs. This system has the following frequency response:

$$H(z) = \frac{\sum_{k=0}^{M} b_k e^{-z}}{1 + \sum_{k=1}^{N} a_k e^{-z}} \tag{2.33}$$

where the a_k are the filter's feedback coefficients corresponding to the poles of the filter, and the b_k are the filter's feed-forward coefficients corresponding to the zeros of the filter, and N is the filter's order.
The basic digital filter design problem is to approximate any of the ideal frequency response characteristics with a system that has the frequency response, by properly selecting the coefficients a_k and b_k." From literature[6], chapter 2.6.4.

2.3.5 Sample-Rate Conversion

Sample-rate conversion (abbreviated as SRC) is the process of changing the time interval between adjacent elements. Increasing the interval is called decimation, which reduces storage and computational requirements. Decreasing time intervals is called interpolation and it preserves fidelity[6]. This subsection will also define the term *resampling.*

Decimation

This is usually a two step process, containing a low-pass anti-aliasing filter and a down-sampler as illustrated in Figure 2.12.

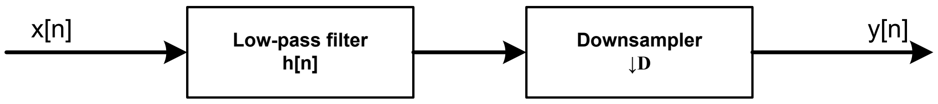


Figure 2.12: Decimation process block chain.

This results in ignoring every Dth sample as is visible in Figure 2.13 and leads to the following equation

$$x_D\left[n\right] = x\left[Dn\right] \quad D = 1, 2, \ldots \tag{2.34}$$

where $x[n]$ is the original signal, $x_D[n]$ is the downsampled signal and D is the decimation rate. It can be shown[6] that frequency values get transformed by

$$f_D = \frac{f}{D} \tag{2.35}$$

where f is the original sampling frequency and f_D is the decimated sampling frequency. It is obvious, since time and frequency are inversely related, that decimation *stretches* the spectrum and adds copies of the original frequency transfer function. The implementation of the mentioned LPF is therefore crucial to avoid anti-aliasing.

Figure 2.16 shows the downsampling process in frequency domain, with visible aliasing effects. This representation is the result of a convolution of the original signal with the downsampled impulse train.

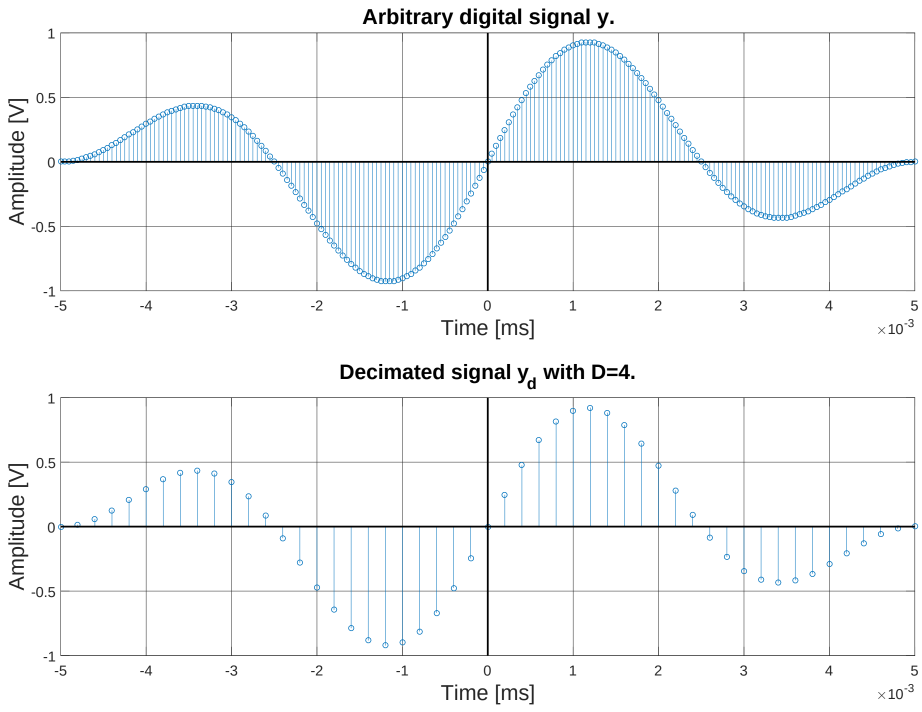


Figure 2.13: Discrete decimation example. Top is the original signal, bottom is the decimated signal with decimation factor $D = 4$.

Interpolation

Interpolation happens in two steps: First the signal gets ”stretched” by adding a defined number of samples with a value of 0 between every original sample. This process is refereed to as *upsampling*. Next the signal gets smoothed using a LPF, which means that the inserted zeros get interpolated to an expected curvature trend.

$$y[n] = \begin{cases} x\left[\frac{n}{I}\right] & n = 0, \pm I, \pm 2I, \ldots \\ 0 & \text{otherwise} \end{cases} \tag{2.36}$$

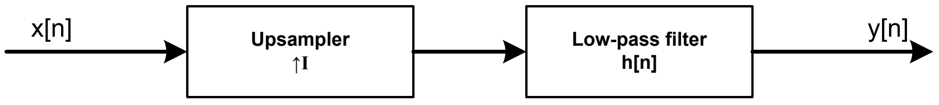


Figure 2.14: Interpolation process block chain.

I is the interpolation factor and must be a positive integer value. The sampling frequency gets therefore changed reciprocally to the equation of a decimator[6].

$$f_I = If \tag{2.37}$$

Assuming the decimated signal from Figure 2.13, an interpolation process will reverse the decimation operation, as is illustrated in Figure 2.15.

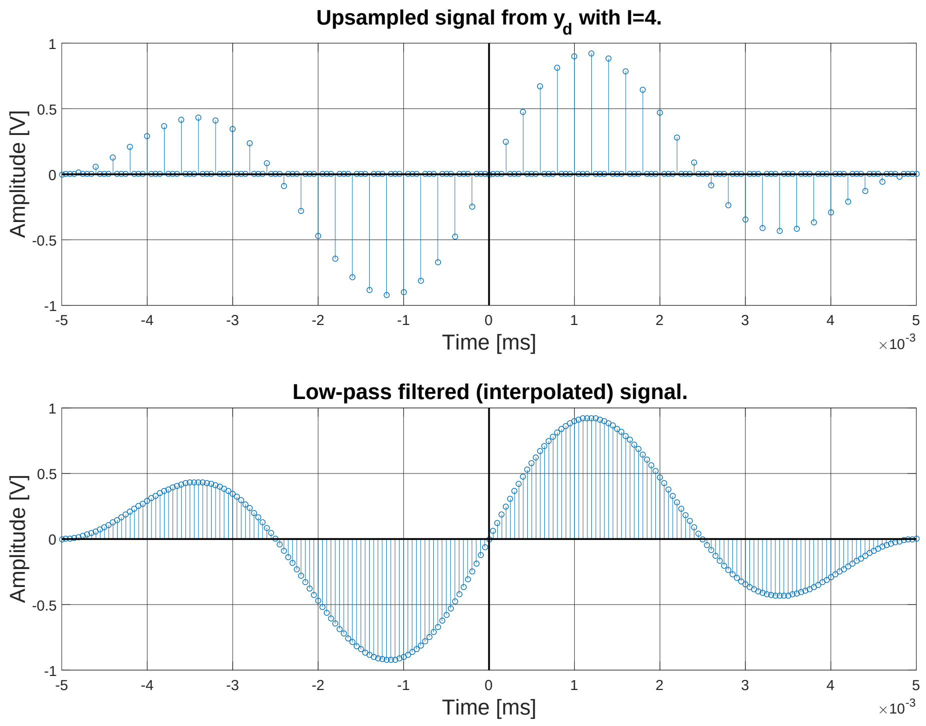


Figure 2.15: Interpolation of the previously decimated signal. Top is the upsampled signal with $I = 4$, bottom is the low-pass filtered signal and clearly the same as the original arbitrary one.

Again it is obvious that this process *compresses* the original frequency signal. It also creates copies of this signal, therefore the design of the LPF is crucial. Figure 2.16 shows the upsampling process in frequency domain. Same as with downsampling, the result of upsampling is a convolution of the original signal with the corresponding impulse train.

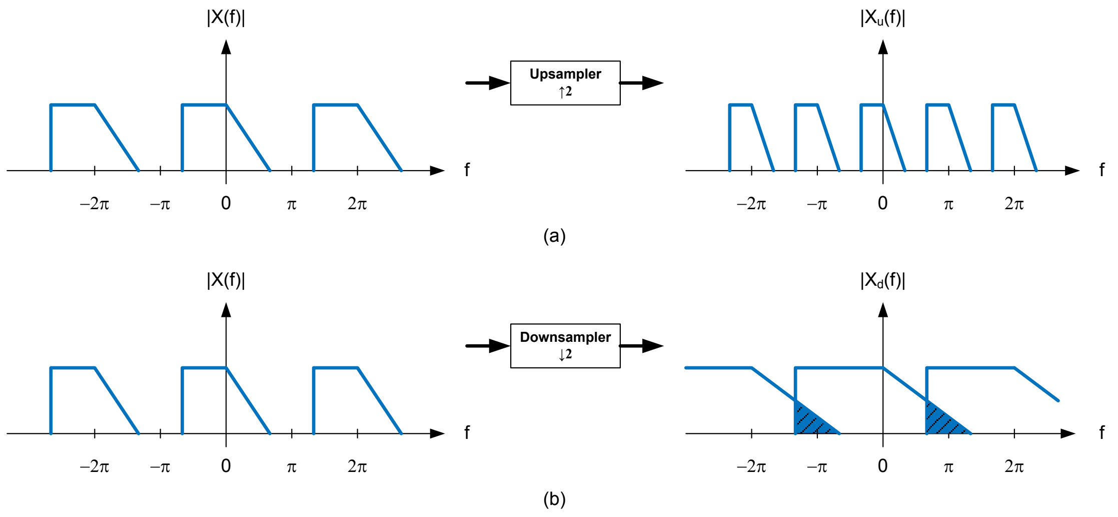


Figure 2.16: Upsampling and downsampling in frequency space, both by a factor of 2. (a) upsampling operation compresses the spectrum (b) downsampling operation stretches the spectrum, with visible aliasing.

Resampling

It's important to note, that the term *resampling* in general refers to a combination of the previously discussed interpolation and decimation process. In other word, while decimation and interpolation work with an integer factor, a resampling operation can also change the sampling rate by a floating value. This is done by combining an upsample with a subsequent downsample operation, or vice versa, with additional filters.

Irrational factor SRC is illustrated in Figure 2.17. First the incoming signal gets upsampled by L and filtered. Afterwards it passes through a continuous time integrate and dump circuit with the goal of integrating the input sample until the output picks up the current content of the accumulator and clearing it[7]. Finally, the decimation process takes place where the signal is first filtered and then downsampled by M.

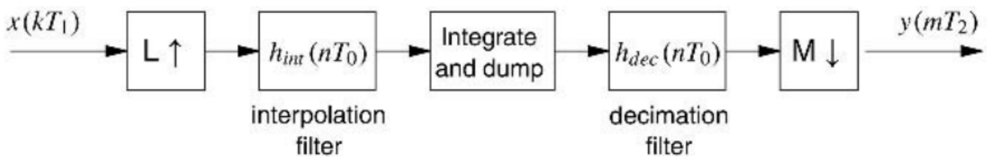


Figure 2.17: Irrational factor SRC[7].

The factor of this operation is referred to as *irrational factor* ν and can be defined by the following equation[7]

$$\nu = \frac{T_1}{T_2} = \frac{L}{M} + \epsilon \tag{2.38}$$

where T_1 and T_2 are symbol duration times and ϵ is an approximation error, which is an irrational number. There is a restriction to this resampling factor, in this case it must be limited to still meet the Nyquist criteria. The irrationality of this design relates to the fractional value of the symbol duration times in equation 2.38 resulting in an irrational number. However, if the fraction of these two result in a rational value, this design is instead defined as a *rational factor SRC* and the formula modifies to[7]

$$\nu = \frac{T_1}{T_2} = \frac{L}{M} \tag{2.39}$$

where both L and M are positive integers.
If one of these two factors has a value of 1, then the design modifies to a *integer factor SRC*, i.e. a interpolation or a decimation process.

An advantage of this system is that it can ease a filtering operation. The order can be changed to a downsampling first followed by an upsampling operation. By placing a filter in between these two, the filter can then operate at a way lower frequency bandwidth, which makes the magnitude transfer function easier to design. Of course in this case, both factors M and L must be the same to return to the correct sample rate. Care must be taken not to get aliasing effects during downsampling.

2.4 Diversity Techniques

This section will discuss the topic of diversity in a DCS: Separating a signal over time or frequency, or transmitting the signal through multiple antennas to get multiple independent or highly **uncorrelated** signal paths. Diversity techniques like polarization diversity or angle-of-arrival diversity will not be subject of this section[5].
Note that these subsections assume basic knowledge of multipath propagation and fading channels.

2.4.1 General

The goal is to reduce the depth and duration of fades at the receiver side to provide significant link improvements with little added cost. This is done by exploiting the nature of a wireless propagation system, where if one path undergoes a deep fade, another path may have a peak. In other words, uncorrelated signals will experience different fades.

Diversity systems are popular if there is no clear LOS (**L**ine-**o**f-**s**ight) connection between two transceivers (abbreviated as TRX, meaning devices which can both send and receive signals). Using diversity techniques a designer is able to mitigate both small-scale and large-scale fading problems.

2.4.2 Time and Frequency Diversity

If a receiving device has a probability p of any one signal to fade below some threshold where it can no longer be detected, then p^N is the probability that N independently fading copies of the same signal will fade below the detection threshold[2]. Therefore, more independent copies of the original signal reduce the error probability significantly, even if the channel attenuation is large.

One way to employ diversity is *frequency diversity*, where the same information is transmitted using N different carriers. Important aspect is that the frequency bandwidths do not overlap and are separated by a defined value. Here the term *coherence bandwidth* of the radio channel comes into play: these are the frequencies at which the channel can be considered "flat", meaning that frequencies in this range experience an equivalent amount of amplitude fading. Frequency diversity is often employed as multiplex mode (FDM = **F**requency-**D**ivision **M**ultiplex) in LOS links.

The other possibility is to employ *time diversity* by sending the same signal over N different time-slots. Again it's important to separate the different time slots by a given time value. This is refereed to as *coherence time*, the time in which the phase is predictable. An example is TDM (**T**ime-**D**ivision **M**ultiplexing).

This kind of system is defined as repeating or as a form of block-interleaving. Consequently there are also systems that use time and frequency diversity together.

2.4.3 Antenna Diversity

Antenna diversity, also refereed to as *space diversity* or *spatial diversity*, usually means spacing out base stations to achieve uncorrelated signals in a mobile radio system. In other words, a system that uses two or more antennas to improve link stability. This is also called *diversity gain*.

However, the scope of this thesis will distinguish between two principles:

1.) **Spatial antenna diversity** - Here all TX antennas transmit the same signal. By spacing the antennas far enough (at least one wavelength on a mobile unit, ten wavelengths between base stations) the propagation of all signals through the channel will be non coherent. Link stability will improve, but care must be taken to process the receiving signal correctly.

    One could also, as an example, use multiple RX antennas that all receive the same signal. The next step would be to implement a detection algorithm that only forwards the "best" of the received signals, for example by comparing SNR values.

2.) **Spatial antenna multiplexing** - With this principle multiple antennas send multiple parts of a symbol stream. The different streams could be created by de-interleaving on TX side and interleaving on the RX side of the system. Although this may sound like pure time diversity, keep in mind that these systems transmit multiple streams of different data at once, instead of just one. This system would not result in a more stable communication, but would lead to higher data rate as a whole.

There are different processing techniques available on the receiving side of the system. The already described selection algorithm (based on best SNR) would route only one of the RX antennas to the following blocks. Another possibility would be to combine all RX antennas directly (equal gain combining) or weighted & added coherently (maximal ratio combining).

Multiple-Input and Multiple-Output

Note that the described **spatial antenna multiplexing** technique is one of the three categories of MIMO (**M**ultiple-**I**nput and **M**ultiple-**O**utput). The two other categories (precoding and diversity coding) will not be subject of this thesis.

MIMO systems can work with accurate CSI (**C**hannel **S**tate **I**nformation). CSI refers to information about the channel properties like fading and scattering, and can be measured by alternating in the transmission of known data (for example just one *"1"* magnitude level) through every TX antenna. All receiver antennas note how well this maximum magnitude level traverses to them, therefore giving the information which TX antenna to RX antenna stream is the best.

Figure 2.18 shows the principle of a basic MIMO system. It is apparent that n transmitting antennas and m receiving antennas (usually $n \geq m$) result in $n \cdot m$ different streams that experience different channel impulse responses. This topic will be discussed further in subsection 2.5.4.

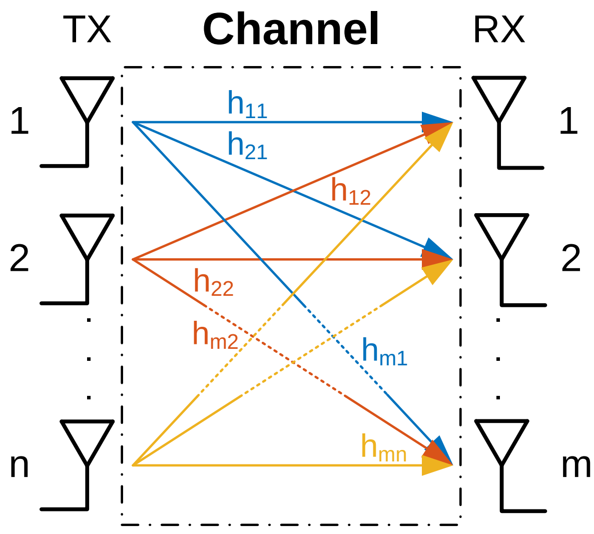


Figure 2.18: MIMO channel and stream principle.

2.5 Channel

After emitting from the TX antenna, the propagating signal s gets influenced by the channel impulse response $h(t)$. Mathematically this operation equals a convolution. Furthermore, simple channel models dictate an additional AWGN (**A**dditive **W**hite **G**aussian **N**oise) $n(t)$.

$$r\left(t\right) = s\left(t\right) * h_c\left(t\right) + n\left(t\right) \tag{2.40}$$

As already discussed in section 2.3.1, the impulse response of the channel is connected by a mapping of input and output using the convolution operator as in equation 2.11. However, an impulse response is defined as the function $h(t)$ of a system if the input is a unit impulse $\delta(t)$, given the system is causal.

In frequency domain the complex channel impulse response contains its magnitude and its phase over the frequency.

$$H_c\left(f\right) = \left|H_c\left(f\right)\right| e^{-\jmath\Phi_c(f)} \tag{2.41}$$

These following subsections will assume basic knowledge of calculation using dB.

2.5.1 Noise

The term noise refers to unwanted electrical signals and can result from thermal noise, interference from other transmitters and interference from circuit switching transients. Overall this noise causes detection errors on the receiving side, together with ISI.

The mentioned thermal noise is one of the natural sources of noise and refereed to as *Johnson noise.* This noise is caused by the motion of single electrons, the same electrons responsible for electrical conduction, in dissipative components (resistors, wires, ...). Johnson noise can be described by a Gaussian random process[2].

$$p\left(n\right) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{n}{\sigma}\right)^2} \tag{2.42}$$

Therefore $n(t)$ is a Gaussian process where the random function n at any arbitrary time t is statistically characterized by a Gaussian PDF (**P**robability **D**ensity **F**unction) $p(n)$ with a variance of σ^2.

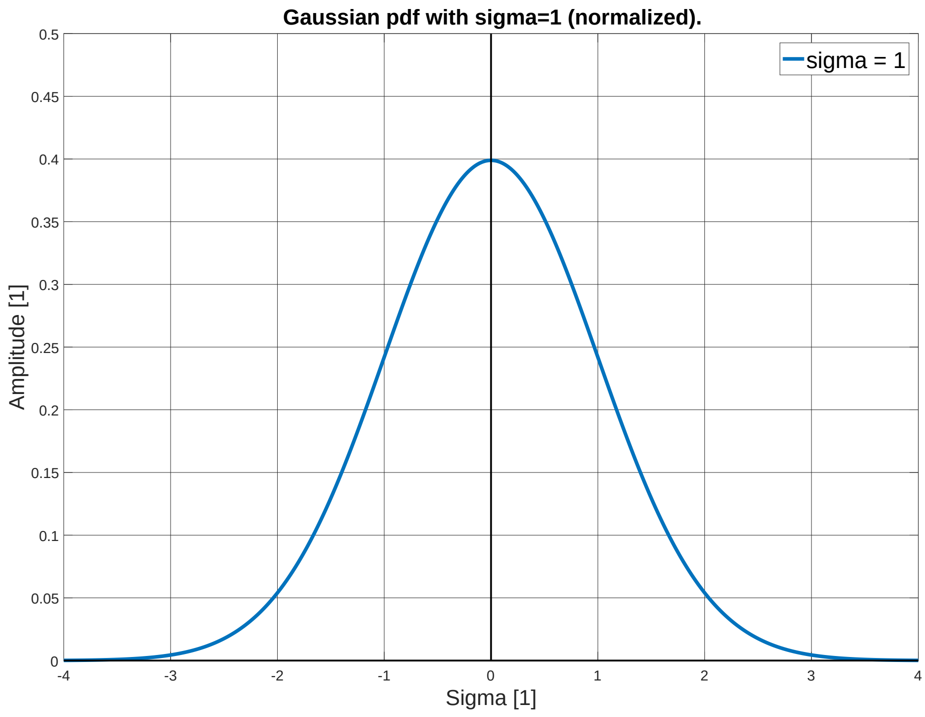


Figure 2.19: Normalized Gaussian distribution function.

The power spectral density for all frequencies of interest is the same in thermal noise, therefore referring to as "white", and is denoted as

$$G_n\left(f\right) = \frac{N_0}{2} \tag{2.43}$$

where N_0 is called the noise power per units of bandwidth with the SI unit *watts/hertz*. The division by 2 denotes a double sided white noise. N_0 can be calculated using the temperature T and the Boltzmann's constant k

$$N_0 = kT \tag{2.44}$$

or by adding the channel bandwidth Δf, resulting in the Johnson-Nyquist noise

$$P_N = kT\Delta f \tag{2.45}$$

Notice that the SI unit of this value is *watts*. A *memoryless channel* is a channel where the AWGN noise affects each transmitted symbol independently.

2.5.2 Link Budget

The term link budget defines the received power, given a known wireless communication system. It contains a multiplication of all gains and losses through this wireless system: Starting with the transmitter, losses in the channel and finally values through the receiver.

In order to simplify the calculations, all these values can be logarithmized and summarized. This can lead to the simplified link budget formula

$$P_{RX,dB} = P_{TX,dB} + G_{TX,dB} - L_{TX,dB} - L_{Ch,dB} - L_{M,dB} + G_{RX,dB} - L_{RX,dB} \tag{2.46}$$

where $P_{RX,dB}$ is the receiving power, $P_{TX,dB}$ is the transmitted power, $G_{TX,dB}$ and $G_{RX,dB}$ are the transmitter and receiver antenna gains respectively, $L_{TX,dB}$ and $L_{RX,dB}$ are the total losses in the transmitter and the receiver respectively, $L_{Ch,dB}$ are losses in the channel and $L_{M,dB}$ are miscellaneous losses.

Most of these losses are grouping together individual values. For example $L_{TX,dB}$ and $L_{RX,dB}$ could be comprised of band limiting losses, modulation losses, efficiency losses, pointing losses and polarization losses[2].

Path Loss

The FSPL (**F**ree-**S**pace **P**ath **L**oss) is a wavelength dependent value defining the free-space loss over the channel. It can be calculated by[4]

$$L_S = \left(\frac{\lambda}{2\pi d}\right)^2 \tag{2.47}$$

where d is the distance between transmit and receive antenna. Logarithmized the formula modifies to

$$L_{S,dB} = 10n \log \frac{\lambda}{2\pi d} \tag{2.48}$$

while log being the common logarithm. n is the path loss exponent and usually 2 in free-space, but gets higher in more crowded channels, like dense urban areas.

Friis transmission equation

The Friis transmission equation is a variant of the link budget equation under simplified conditions. It implements the path loss into the link budget formula.

$$P_{RX,dB} = P_{TX,dB} + G_{TX,dB} + G_{RX,dB} - 10n \log \frac{\lambda}{2\pi d} \tag{2.49}$$

2.5.3 Rayleigh and Ricean Fading

This subsection will take a short look in statistical channel models. In other words, taking multipath into account, a large number of signal paths makes it possible to apply the central limiting theorem[8]. Therefore, these models can be described with probability density functions.

Rayleigh Fading Distribution

> "In mobile radio channels, the Rayleigh distribution is commonly used to describe the statistical time varying nature of the received envelope of a flat fading signal, or the envelope of an individual multipath component. It is well known that the envelope of the sum of two quadrature Gaussian noise signals obeys a Rayleigh distribution. [...] The Rayleigh distribution has a probability density function (pdf) given by
>
> $$p\left(r\right) = \begin{cases} \frac{r}{\sigma^2} e^{-\frac{r^2}{2\sigma^2}} & \left(0 \leq r \leq \infty\right) \\ 0 & \left(r < 0\right) \end{cases} \tag{2.50}$$
>
> where σ is the rms value of the received voltage signal before *envelope detection*, and σ^2 is the time-average power of the received signal *before* envelope detection." From literature[5], chapter 5.6.1.

Rayleigh fading is a typical model for tropospheric and ionospheric wireless signal communication and used when there is no dominant LOS component in the communication. It is also very popular in urban communication systems. In Rayleigh fading, transmitting each symbol from every antenna simultaneously is equivalent to using a single transmit antenna[9].

Ricean Fading Distribution

Ricean fading models are similar to Rayleigh models, however they get primarily used when there is a strong direct path component between transmitter and receiver (often refereed to as *specular path*). Imposed multipath components summarize to a smaller valued amplitude than the LOS component. The Ricean PDF distribution is given as

$$p\left(r\right) = \begin{cases} \frac{r}{\sigma^2} e^{-\frac{r^2+A^2}{2\sigma^2}} I_0\left(\frac{Ar}{\sigma^2}\right) & \left(A \geq 0, r \geq 0\right) \\ 0 & \left(r < 0\right) \end{cases} \tag{2.51}$$

where A denotes the peak amplitude of the dominant signal and I_0 is the 0th order modified Bessel function of the first kind[5].

If the dominant LOS component grows smaller, the Ricean model approaches a Rayleigh model. For simplification, introduce the factor K, the ratio between deterministic signal power and the variance of the multipath.

$$K_{dB} = 10 \log \frac{A^2}{2\sigma^2} \tag{2.52}$$

K is known as the *Ricean K factor*[8] and it specifies the Ricean distribution: The smaller the dominant LOS component A, the higher valued K will get, until it equals a Rayleigh distribution at $K = -\infty dB$.

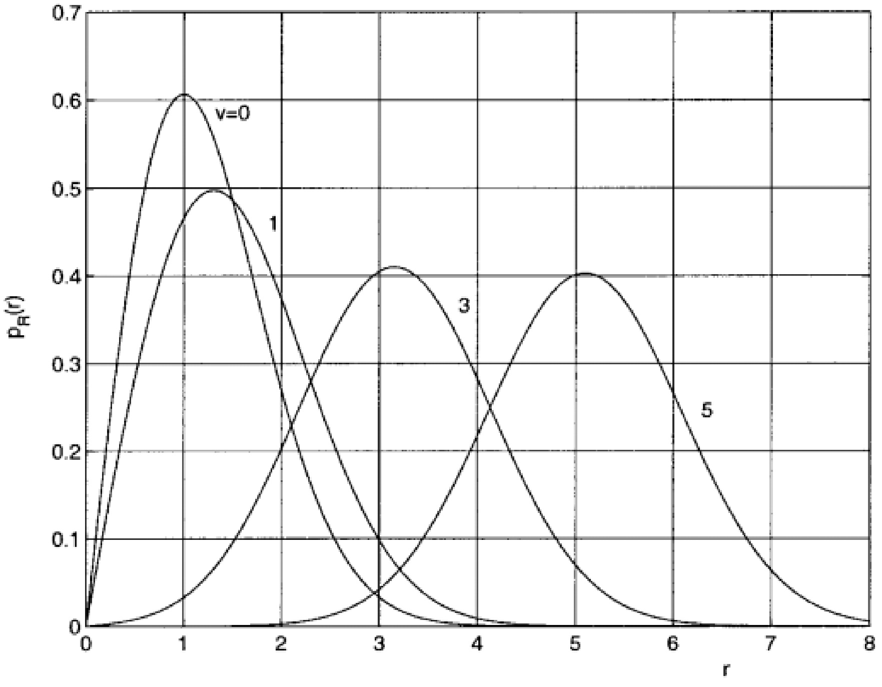


Figure 2.20: Example Ricean pdf distributions with different A value ($v \Longrightarrow A$) with $\sigma^2 = 1$[10].

2.5.4 MIMO Channel

For the sake of completeness it shall be noted that the channel model for a typical MIMO system illustrated in Figure 2.18 can be defined with the formula

$$\vec{y} = \mathbf{H}\vec{x} + \vec{n} \tag{2.53}$$

where $\vec{y}$ is a vector $m \times 1$ of the received signals, $\vec{x}$ is a vector $n \times 1$ of the transmitted signals, $\vec{n}$ is a vector $m \times 1$ for the thermal noise (IID = **I**ndependent and **I**dentically **D**istributed, circularly symmetric complex Gaussian with unit variance) and $\mathbf{H}$ is a matrix $m \times n$ containing all the different impulse responses that the signal paths take[9].

2.6 Synchronization

DCSs need different types of synchronization blocks: Phase synchronization, carrier synchronization, frame synchronization and network synchronization[2]. This section will discuss a few concepts that will be relevant in later chapters of this thesis.

Figure 2.21 shows the different synchronization blocks on the receivers side. Blue colored blocks are for timing synchronization. Matched filters were discussed in section 2.3.3. Orange Blocks are frequency synchronization blocks for the carrier. Frame Synchronization is depicted in yellow and equalization illustrated in purple.

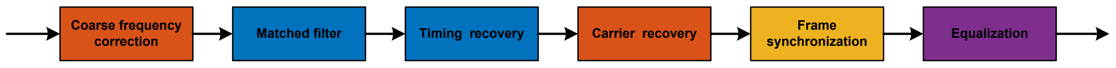


Figure 2.21: Receiver synchronization block chain.

2.6.1 Timing Synchronization

Timing synchronization, also called *timing recovery*, is responsible for the correct sampling time instances. This is one of the most critical functions performed in the receiver. The receiver must know the frequency at which the outputs of the matched filters are sampled and also where to sample each symbol during its interval.

In some systems both the TX and RX synchronize their clocks (for example hardwired), in which case the timing recovery only has to account for the time delay of the signal through the channel. This will be relevant later, when connecting the TX output of one device via a cable to the RX input of the same device. Another possibility would be to add information about the clock frequency (or a multiple of it) to the transmitted signal. This can be done with the previously mentioned SRC, which will also be demonstrated in later chapters.

Phase-Locked Loop

Some of the following algorithms use a so-called PLL (**P**hase-**L**ocked **L**oop). It consists of three main components: A multiplier, a loop filter and a VCO (**V**oltage-**C**ontrolled **O**scillator). Figure 2.22 shows how these come together.

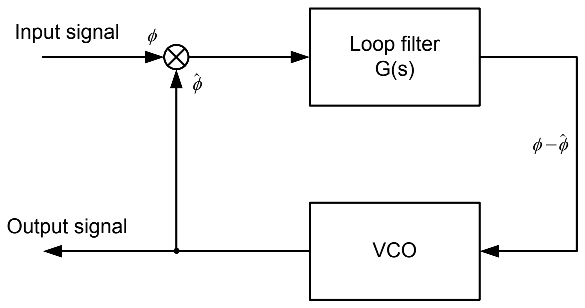


Figure 2.22: Basic components of a PLL.

Let's assume that the input signal is a sinusoid $\cos(2\pi f_c t + \phi)$ and the VCO outputs $\sin\left(2\pi f_c t + \hat{\phi}\right)$. $\hat{\phi}$ represents the estimate of the phase ϕ. Calculating the product of these two and simplifying the result using trigonometric identities leads to

$$\begin{aligned} e(t) &= \cos(2\pi f_c t + \phi)\sin\left(2\pi f_c t + \hat{\phi}\right) \\ &= \frac{1}{2}\sin\left(\hat{\phi} - \phi\right) + \frac{1}{2}\sin 4\pi f_c t + \left(\hat{\phi} + \phi\right) \end{aligned} \tag{2.54}$$

The loop filter is designed as a LPF that lets the term $\hat{\phi} - \phi$ pass. This term is zero if the estimated phase is correct. If it's not zero, the VCO will correct its value. An example transfer function of the LPF could be

$$G(s) = \frac{1 + \tau_1 s}{1 + \tau_2 s} \tag{2.55}$$

where $\tau_1, \tau_2;\ \tau_1 > \tau_2$ are design parameters. Lastly the VCO can be modelled as an integrating function, e.g.

$$G_{VCO}(s) = \frac{K}{s} \tag{2.56}$$

where K is a scalar gain factor. Using the formula for the closed loop it is obvious to arrive at a second-order transfer function. After simplification the formula modifies to

$$H(s) = \frac{\left(2\zeta\omega_n - \omega_n^2/K\right)s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{2.57}$$

where ζ is the loop damping factor and ω_n is the natural frequency of the loop[4].

Early-Late

A very simple algorithm for timing recovery is called *Early-Late symbol timing recovery.* Figure 2.23 shows the block diagram of this symbol timing algorithm[11] and Figure 2.24 illustrates the sampling in time domain.

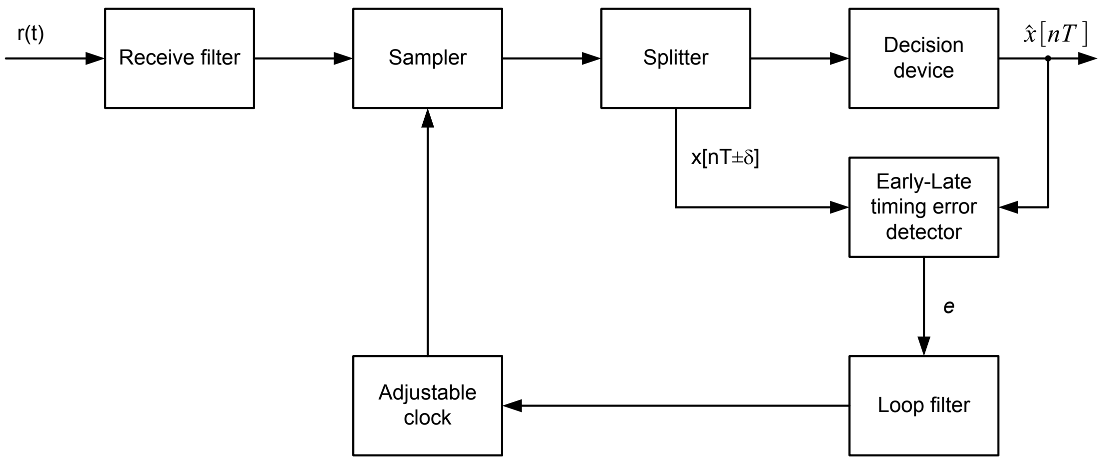


Figure 2.23: Early-late symbol recovery block diagram.

This algorithm samples the signal at a timing nT, together with two additional samples, equidistant apart from nT by a small value δ. $nT-\delta$ is called the early symbol, $nT+\delta$ is the late symbol. The timing error is evaluated by comparing the two equidistant samples: The difference of amplitude of these two defines if the timing point must be moved. If the amplitudes are the same, a perfect timing was achieved.

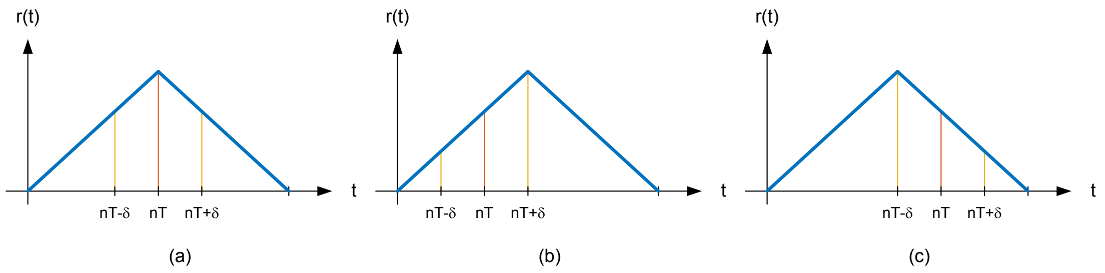


Figure 2.24: Early-late symbol recovery sampling. (a) is perfectly timed (b) sampling timing to early (c) sampling timing to late.

The real timing error, which is the output of the TED (**T**iming **E**rror **D**etector), is computed as follows

$$e = \{x\,[nT+\delta] - x\,[nT-\delta]\}\,\hat{x}\,[nT] \tag{2.58}$$

where $\hat{x}$ is the decision. See also literature[11][12] for a more detailed evaluation of this parameter. Note that the imaginary part was neglected, to work only with the amplitude, for simplicity purpose. The same applies for other error formulas in this section. Depending on the value of this timing error, it is either perfectly sampled ($e=0$), or it needs alignment ($e<0$ or $e>0$).

Gardner and Zero Crossing

In principle, the Gardner symbol timing recovery is very similar to the block diagram of the Early-Late timing recovery in Figure 2.23. However, this algorithm needs an additional zero crossing. When sampled perfectly, the amplitude at point $nT - T/2$ is zero. If the sampling instance was wrong, the numerically controlled oscillator needs to be adjusted, depending on the sign of the amplitude. Figure 2.25 shows this concept.

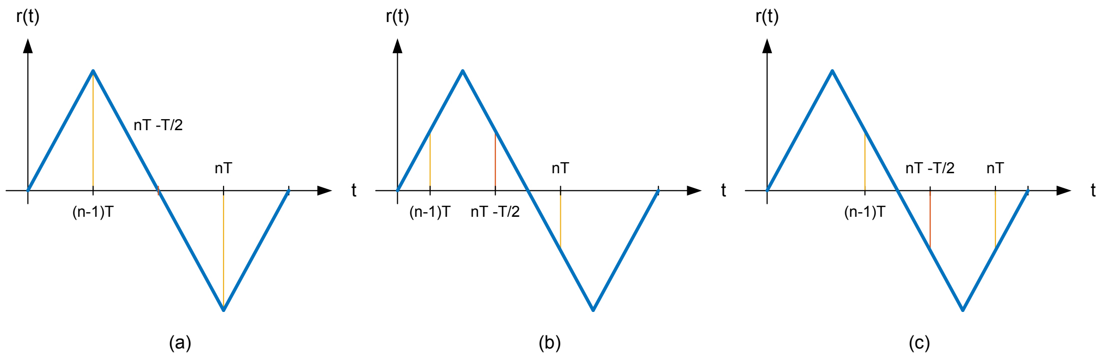


Figure 2.25: Gardner symbol recovery sampling. (a) is perfectly timed (b) sampling timing to early (c) sampling timing to late.

The timing error output from TED gets computed by[12]

$$e = \{x\,[nT] - x\,[(n-1)T]\}\,\hat{x}\,[nT - T/2] \tag{2.59}$$

Again, if this value is not zero, the sampling timing was wrong. The Gardner algorithm should work better with larger roll-off factors in the implemented matched filters.

A close relative to Gardner is the Zero Crossing algorithm. A big difference is that while Gardner is non-data-aided, Zero Crossing is actually a decision-directed.

Mueller and Müller

Mueller and Müller synchronizer is a decision-directed optimum timing algorithm. Other then the previous described Early-Late algorithm, this synchronizer only requires one sample per symbol and knowledge of the previous symbol to estimate the timing error. It's block diagram is depicted in Figure 2.26 and illustrates its decision feedback architecture. This synchronizer combines the PAM signal with the error samples, which in turn is used to bring the timing clock in synchronization[11].

One interesting aspect of this concept is that the Mueller and Müller synchronizer, different to other synchronizers, actually performs better with a lower roll-off factor in the Nyquist filter[12]. This means that the receiving filter should either have a very low roll-off factor or even keep it at zero.

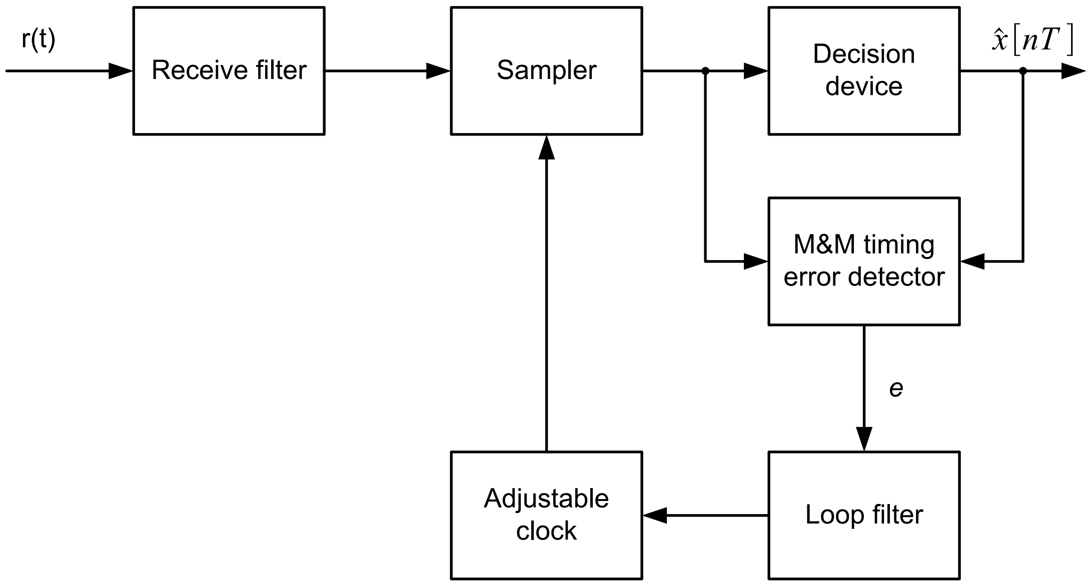


Figure 2.26: M&M symbol recovery block diagram.

Figure 2.27 shows how this algorithm functions in time domain. Different then the previous symbol recoveries, M&M operates on T-spaced samples and adjusts its symbol sampling depending on the distance between these samples.

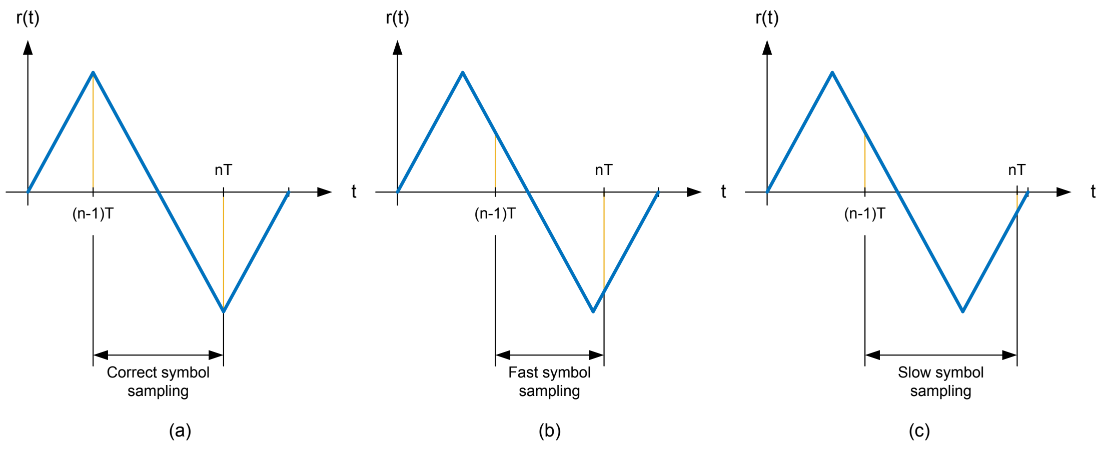


Figure 2.27: M&M symbol recovery sampling. (a) is perfectly timed (b) sampling timing to fast (c) sampling timing to slow.

The error output from the TED gets computed by

$$e = \{x[(n-1)T]\}\,\hat{x}[nT] - \{x[nT]\}\,\hat{x}[(n-1)T] \tag{2.60}$$

Maximum-Likelihood

In order to construct an optimal receiver, the goal should be to detect the symbol sequence with minimal probability of detection error. It is known that this is accomplished when the detector maximizes the a posteriori probability for all sequences a. This is refereed to as MAP (**M**aximum **a** Posterior **P**robability).

$$\hat{a}_{MAP} = arg \max_{a} p\left(a|r\right) \tag{2.61}$$

The a posteriori probability can be rewritten using Bayes's rule.

$$p\left(a|r\right) = \frac{p\left(r|a\right)p\left(a\right)}{p\left(r\right)} \tag{2.62}$$

Since $p(r)$ is only a normalization parameter, it can be ignored. The problem then simplifies to maximizing the likelihood function $p(r|a)$. This process is refereed to as Maximum-likelihood (abbreviated as ML).

It is obvious, that to maximize this likelihood function, a derivative is needed, where a zero crossing of this derivative means an extremum. This concept is exploited by sgn(y[n]y'[n]) ML symbol timing recovery. As the name of this algorithm suggests, it uses a derivative of the matched filter together with this matched filter itself to recover the symbol timing. This makes this approach very efficient, but also very complex to implement, which is why it gets used very rarely. Instead, easier-to-implement algorithms like the previously described ones are more popular.

2.6.2 Carrier Synchronization

Carrier synchronization complements the previous phase synchronization and it is necessary to maintain wireless links with separate LOs (**L**ocal **O**scillators) in TX and RX. Relative frequency offsets will exist between these two LOs due to natural effects, even dynamic drifts must be taken care of. Commercial oscillators provide the frequency offset in PPM (**P**arts-**P**er **M**illion) which can be translated into a maximum carrier frequency drift:

$$f_{o,max} = \frac{f_c \cdot PPM}{10^6} \tag{2.63}$$

This is an important parameter, since this value dictates some design criteria. There are multiple examples of decision-directed and non-decision-directed loops that perform carrier locking. In the scope of this thesis, only **Costas loops** are of interest. Figure 2.28 shows the block diagram of a Costas loop.

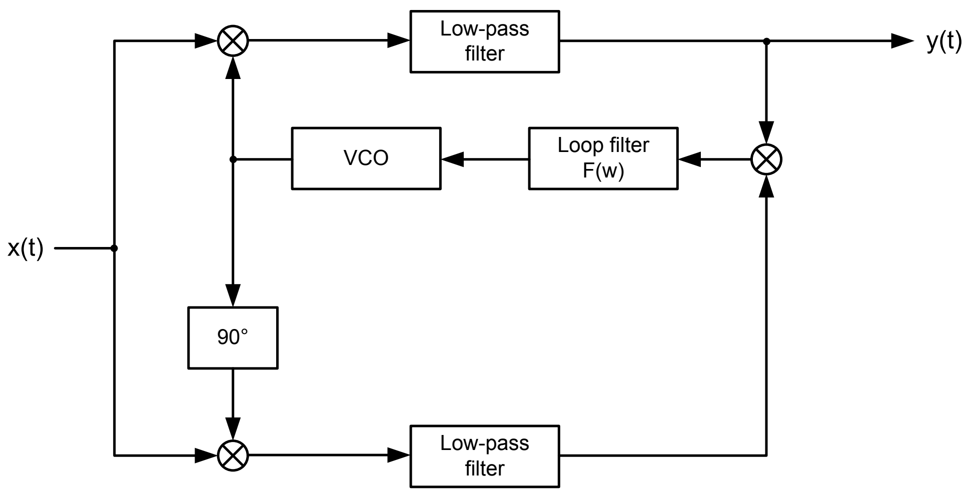


Figure 2.28: Costas loop block diagram.

This PLL based structure multiplies the input signal with the sinusoid output from the VCO sin $\left(2\pi f_c t + \hat{\phi}\right)$ and a 90 degree phase shifted signal cos $\left(2\pi f_c t + \hat{\phi}\right)$. The following LPFs eliminate high frequency parts $(2 \cdot f_c)$. Both results are being multiplied to get an error signal, this error signal then controls the VCO and therefore the frequency. Important to notice is that the two LPFs must be perfectly matched[2][4].

2.6.3 Frame Synchronization

For the sake of completeness it shall be noted that many communication systems have uniformly sized groups of bits that can be refereed to as a *frame*.

Frame synchronization guarantees that the receiver has a synchronized data stream around the frame structure. This is usually accomplished with the aid of some special signalling procedure in the transmitter. For example, these frames can also come with a synchronization sequence to indicate the start of the frame. The receiver needs to be aware of the correct pattern and the injection interval. Correlating the known pattern with the incoming data stream, achieving frame sync once the correlation was correct[4].

2.6.4 Equalization

In order to get ideal transmission characteristics over the channel, the magnitude of the channel $|H_c(f)|$ in equation 2.41 must be constant and the phase $\Phi_c(f)$ must be a linear function of the frequency. In other words, the delay must be constant for all spectral signal components.

Because attenuation in the magnitude is not constant, amplitudes will be distorted. If the phase is not linear, the signal will have phase distortion. These affects usually even occur together and it's refereed to as *smearing* (pulse is not well defined). Overlapping smearing effects are also known as ISI.

There are two main groups of equalizers:

1.) **MLSE** (**M**aximum-**L**ikelihood **S**equence **E**stimation) - Measures the channel frequency response $h_c(t)$ and provides means to adjust the receiver to the physical transmission environment. The symbols are not modified in any way, instead the MLSE receiver adjusts itself to deal with distorted symbols. A very popular example is the *Viterbi equalizer.*

2.) **Equalization with filters** - Means compensating distorted pulses using filters. The goal is to forward ISI free symbols to the demodulator blocks. The filters are either *transversal equalizers* (linear and only containing feedforward elements) or *decision feedback equalizers* (nonlinear containing both feedforward and feedback elements). In other words, preset or adaptive filters.

"The performance of an algorithm is determined by various factors which include:

- **Rate of convergence** - This is defined as the number of iterations required for the algorithm, in response to stationary inputs, to converge close enough to the optimum solution. A fast rate of convergence allows the algorithm to adapt rapidly to a stationary environment of unknown statistics. Furthermore, it enables the algorithm to track statistical variations when operating in a nonstationary environment.

- **Misadjustment** - For an algorithm of interest, this parameter provides a quantitative measure of the amount by which the final value of the mean square error, averaged over an ensemble of adaptive filters, deviates from the optimal minimum mean square error.

- **Computational complexity** - This is the number of operations required to make one complete iteration of the algorithm.

- **Numerical properties** - When an algorithm is implemented numerically, inaccuracies are produced due to round-off noise and representation errors in the computer. These kinds of errors influence the stability of the algorithm." From literature[5], chapter 7.8.

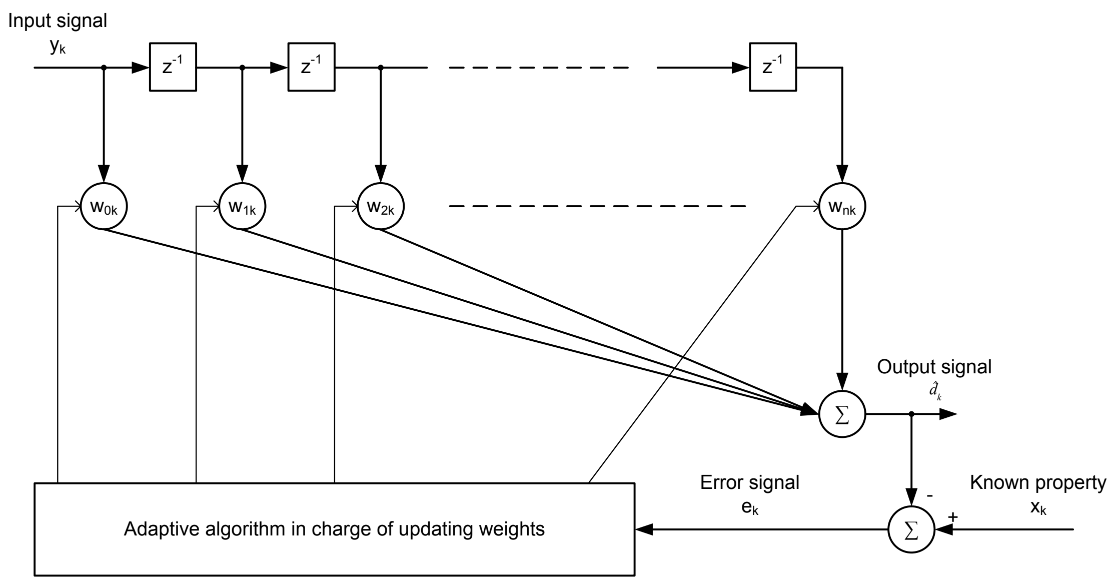


Figure 2.29: Block diagram of a basic adaptive equalizer.

Figure 2.29 shows the basic structure of an adaptive equalizer. For this thesis the relevant equalizer is a decision-directed LMS (**L**east **M**ean **S**quare) algorithm. The prediction error is given as

$$e_k = x_k - \hat{d}_k \tag{2.64}$$

where the indices k refers to the time instant and e_k refers to the error between the equalizer output $\hat{d}_k$ and the known property x_k, for example the modulation constellation points. The LMS algorithm computes the mean square error by computing $|e_k|^2$ and minimizing this error by updating the weights w_{nk} accordingly[5].

$$\zeta = E\left[e_k^* e_k\right] \tag{2.65}$$

2.7 Software-Defined Radio

In wireless communication it is very common to combine digital processing with analog RF components. A **S**oftware-**D**efined **R**adio (SDR) is a wireless device where much of the digital signaling operations and baseband functionality can be implemented using software rather than fixed hardware circuits[13]. This gives the developer a big degree of flexibility, makes these devices re-configurable and very powerful.

SDR technology is nothing new, the term *software radio* first appeared in 1984[6]. Similar to many other technological advances, SDRs have their roots in military. *SPEAKeasy* was a United States Military project in 1991 with the goal to develop a single radio system that could be used to communicate with 10 different types of military radios. Today there is a range of military grade SDR hardware available, very powerful devices aimed to search for enemy communication frequencies and combined with powerful computational power to fast crack encryption. Some systems even work with computational intelligence and neuronal networks (signal intelligence). However, there is also a consumer market for low-cost high performance radios, aimed for industrial or amateur use. Cellular options are also very popular, to send and receive different radio protocols in real time, and to create base stations. Some SDRs even find use in radio astronomy.

The software part that replaces hardware blocks like mixers, filters and amplifiers is usually implemented on a PC (**P**ersonal **C**omputer) or embedded device. However, most of the hardware boards come with dedicated re-configurable ICs (**I**ntegrated **C**ircuits) systems, specially DSPs (**D**igital **S**ignal **P**rocessors). These integrated circuits can be FPGAs (**F**ield **P**rogrammable **G**ate **A**rrays), GPUs (**G**raphics **P**rocessing **U**nits) or ARMs (**A**dvanced **R**ISC **M**achines). An ideal basic receiver design of an SDR consists of an antenna, an analog-to-digital converter and the digital signal processing, see concept principle in Figure 2.30. The system needs enough computational capabilities to achieve realtime operations in order to implement this big signal processing part.

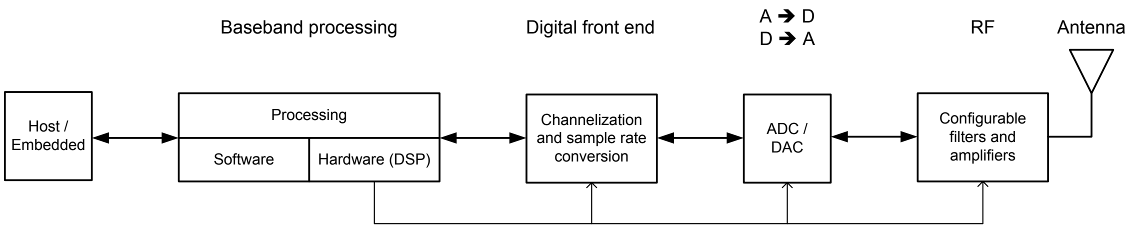


Figure 2.30: Ideal receiver principle of a software-defined radio.

As already mentioned, a typical SDR has an onboard configurable IC. However the heart of every SDR is its FPRF (**F**ield **P**rogrammable **R**adio **F**requency) integrated circuit. These ICs usually integrate the whole analog and digital RF signal chain on one piece of silicon, therefore minimizing process variations. They often contain duplicates of their signal processing paths, enabling them to to implement antenna diversity.

A SDR is the perfect device to integrate all the previously mentioned topics in this chapter. It can set up a digital communication system, process the information formats from section 2.2 through the blocks mentioned in the first section of the theoretical chapter, employ filters for ISI free communication (section 2.3, but instead of static filters, tune-able filters are implemented), use diversity technologies as described in section 2.4 to improve link stability and correct negative channel effects, and lastly handle the synchronization of data by employing the technologies from section 2.6.

2.7.1 Architecture

Single or dual mixing stage superheterodyne architecture RF is a very popular and commonly used architecture in radio frequency communication systems. The benefits include configurable channel bandwidth and selectivity thanks to IF (**I**ntermediate **F**requency) filters, low spurious emissions and trade-off between optimizing noise figure and linearity thanks to gain distribution across the stages[6][13].

Figure 2.31 shows a typical dual stage superheterodyne signal block chain[14]. It consists of different filters and amplifiers, mixers where the VCO frequency is mixed with the signal and finally the digital part of the system, interfaced by **A**nalog-to-**D**igital **C**onverters (ADC) and **D**igital-to-**A**nalog **C**onverters (DAC) respectively.

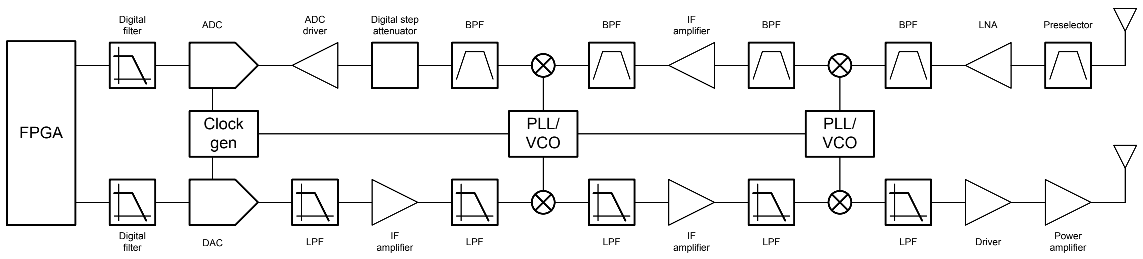


Figure 2.31: Dual stage superheterodyne signal chains. Top is receive part, bottom shows transmit path.

A more popular architecture in recent years is the ZIF (**Z**ero **I**ntermediate **F**requency) architecture, which utilizes a single frequency mixing stage with a local oscillator. A ZIF receiver is also refereed to as DCR, **D**irect-**C**onversion **R**eceiver. This LO is directly set to the interesting frequency, translating the signal down to baseband. Figure 2.32 shows how after this translation, when phase shifting before by an angle of 90 degree, the result is the in-phase I and quadrature Q part of the signal. The structure strongly resembles the principles discussed in section 2.6.

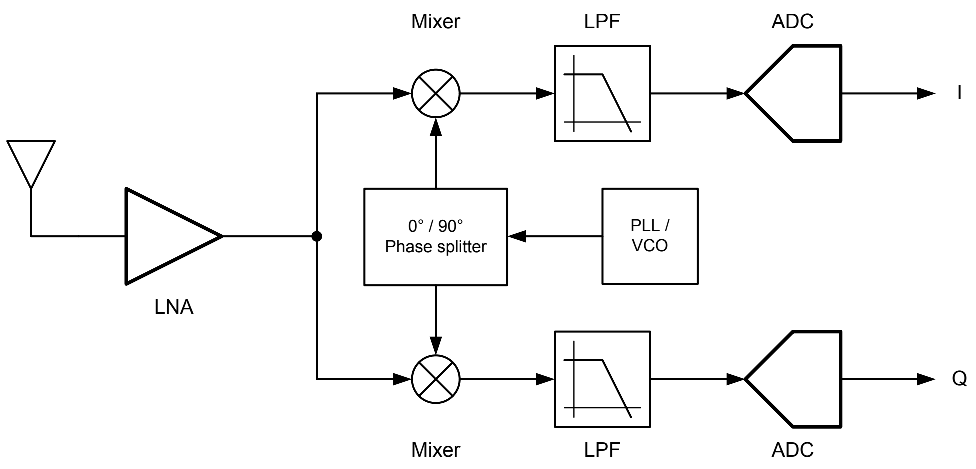


Figure 2.32: Zero intermediate frequency architecture splitting incoming signals into I and Q parts.

The positive aspect about this design is that all analog filtering takes place at baseband level. This lower frequency makes it easier to design components, reduces sampling times and makes components cheaper. Since components are non-perfect, care must be taken to correct the I/Q imbalance between the two paths. Therefore calibration is an important part of every SDR as bad calibration can lead to reduced performance. Most SDR boards equip a microprocessor to perform the calibration.

2.7.2 Automatic Gain Control

Automatic gain control, abbreviated as AGC, is a regulating technique used to maximize signal amplitudes on the output of an amplifier circuit to its maximum value. In other words, given a range of input signal values, a AGC circuit modifies the gain to give output signal levels the maximum range. Usually SDR platforms employ AGC circuits that regulate the gain of a chain of amplifiers dynamically. This is of course done on the RX path of the transceiver. Automatic gain control is used in a variety of systems, the implementation varies for different use cases.

2.7.3 Software Environment

The software part of a SDR can start after an ADC and end at a DAC. A SDR is fully configurable, however the most important parameters are center frequency, bandwidth, sample rate, gains and type of modulation. In a development process, the system first gets configured on a host PC or embedded device: Configure and calibrate the device correctly, develop, tune and optimize the modulation and demodulation of the signals. Doing this on a host makes it easier to debug and visualize the signals. The second step would be to write the developed technique in a high-level language in floating point and implement it on a production worthy environment, making it a truly standalone software-defined device.

For the first step it is important to chose a software environment that is both powerful in processing huge amount of data and flexible enough to implement the goals of the project. One of these environments is MATLAB, which is a technical computing environment and programming language. This is especially handy if the communication systems toolbox is available. Another possibility is the open-source GNU Radio software toolkit. GNU Radio employs a variety of *C++* libraries for digital communications and signal processing and combines these libraries with *SWIG* (**S**implified **W**rapper and **I**nterface **G**enerator, an open source software wrapper, connecting *C/C++* programs with scripting languages like *Python*). More about GNU Radio will be discussed in chapter 5.

As mentioned, the real implementation (once the project has been developed and is functional) would be written in a higher level language like Very High Speed Integrated Circuit Hardware Description Language, often abbreviated as VHDL. However, the scope of this theses will only focus on the development process and the lessons and results this will lead to.

Chapter 3

System Overview

This chapter will first focus on system parameters, limitations and possible development platforms. Afterwards a short design decision on which SDR hardware to use will be presented. When choosing between different SDR boards, it is also very important to keep the software environments for the development process in mind. An additional look on the performance of the antennas follows an short summary of a few selected analog and digital hardware parts. Finally a overview on the host configuration will be provided.

3.1 Parameters and Limitations

3.1.1 General Concept and Parameters

Figure 3.1 shows the concept of this thesis. An unidirectional link between two SDR over a variable distance d_{FS} using a bandwidth B_C at a center frequency f_c (and therefore wavelength λ). Even though it is an unidirectional communication, the SDR will include TRX FPRF chips, meaning that both devices can send and receive information. Therefore, it's technically a bidirectional communication. Additionally, both SDR shall have multiple antennas for transmission and reception, in other words they implement antenna diversity.

The transmission channel is assumed to have LOS or near LOS (meaning that the link sight can be sporadically disrupted by moving objects like cars or people) propagation conditions, in an urban environment. The maximum distance between two devices where communication is still possible is not specified. Higher possible communication distances are better, though having a stable communication at a very short distance would be viewed as proficient, as increasing the range would just be a matter of increasing the radiated power. The data rate shall be up to *10MBit/s* with a minimum of *1MBit/s*.

Given these first parameters it is obvious why to choose SDR hardware (see also section 2.7). High data rates over the channel mean that higher order modulation schemes come in handy. However, if the distance is getting longer and the noise level over the channel rises, it would make sense to decrease the amount of symbols (i.e. change the modulation scheme) to guarantee a stable communication at a reduced data rate. With a SDR this kind of modification is possible. Using the discussed spatial antenna diversity technique, the link stability can be improved even further.

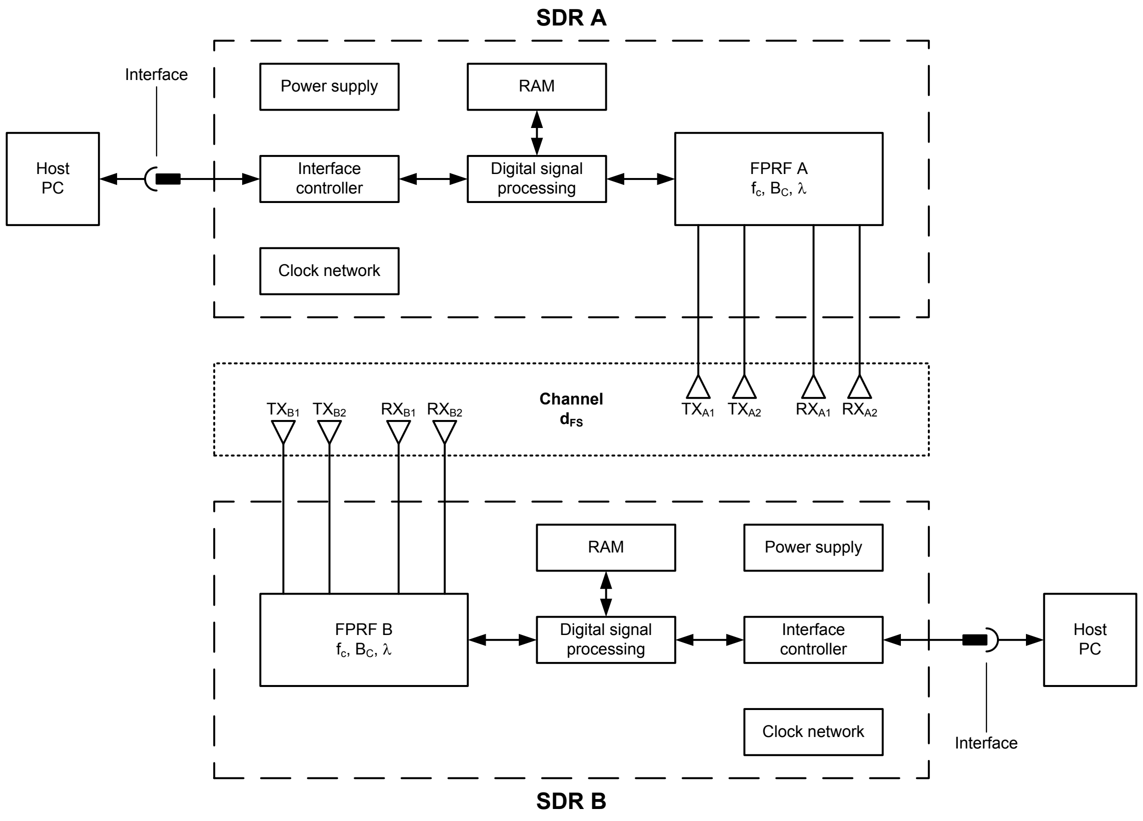

SDR A
Interface
Host PC
Power supply
RAM
Interface controller
Digital signal processing
FPRF A f_c, B_C, λ
Clock network
Channel d_{FS}
TX_{B1} TX_{B2} RX_{B1} RX_{B2}
TX_{A1} TX_{A2} RX_{A1} RX_{A2}
RAM
Power supply
FPRF B f_c, B_C, λ
Digital signal processing
Interface controller
Host PC
Clock network
Interface
SDR B

Figure 3.1: Common development concept for this thesis.

3.1.2 Norm Limitations

Chosen frequency band for this thesis is the ISM (**I**ndustrial, **S**cientific and **M**edical) radio band at *2.4GHz*. The system to design will be non adaptive, and the specification for this frequency band for European standards can be looked up in the ETSI EN 300 328[15] document. This norm specifies a few of the system parameters:

- **Frequency Range** - Describes the start and stop frequency values in which both transmitting and receiving signals must operate. For this ISM band, the frequency range is the same for TX and RX, and it ranges from *2400MHz* to *2483.5MHz*. Also note the term *occupied channel bandwidth*, which is defined as the bandwidth in which 99% of the signal power must resign.

- **Operation Mode** - There are two different modes. Adaptive, meaning that the system automatically scans its radio environment and chooses the best frequency range or channel to operate in. And non-adaptive mode, which just transmits on its predefined frequency range and does not react to its radio surrounding.

- **Spread Spectrum** - The EN also distinguishes between spread spectrum communications, in this case FHSS (**F**requency-**H**opping **S**pread **S**pectrum), and non spread spectrum communications. Note that other technologies with multiple dedicated frequency bands or carriers in the frequency range (for example DSSS and OFDM) are refereed to as non spread spectrum communications.
- **Maximum RF Power** - Is the maximum power value of the transmission radiating from the equipment, summarized over the whole area around the antenna. This is also refereed to as EIRP (**E**ffective **I**sotropic **R**adiation **P**ower) and the value is given in dBm. The exact value depends on the operational mode and if it is spread spectrum technology.
- **Power Spectral Density** - Defines how much power can be transmitted for a given frequency range. Therefore, the unit is given as dBm/MHz, by default normalized to *1MHz*.
- **Other Specifications** - Include duty cycle, medium utilization factor, receiver blocking and adaptivity.

To simplify the overall implementation it was decided to avoid adaptive or spread spectrum approaches. This results in the following limitations of the system: The hardware will stay in the occupied channel bandwidth and, therefore confine itself to the maximum bandwidth given by *20MHz*. If this value is reached there can still be 4 different channels active at the same time. The maximum RF power is limited by a value of *100mW* or *20dBm* EIRP and a maximum power spectral density of *10dBm/MHz*. Most of the other specifications become invariant with this given setup or are not relevant for further discussions.

3.2 Hardware Design Options

In preparation of this thesis and after the initial definition of the previously described system parameters, a case study has been conducted which SDR hardware and software pair would be optimal to reach the desired goals.

First of all the host will be a standard issue PC or Laptop. The only requirements are to have sufficient computational power for the signal processing (i.e. fast processor and preferably fast hard discs like SSD. RAM should not be that crucial) and an appropriate interface to the SDR with suitable data speed.

As of the SDR, three possible platforms were evaluated.

3.2.1 USRP B210 / USRP-2901

Hardware

USRP B210[16] is a SDR kit from Ettus Research providing a dual channel transceiver board, meaning it supports antenna diversity with two TX and two RX antennas. USRP stands for **U**niversal **S**oftware **R**adio **P**eripheral. It has a frequency range from *70MHz* to *6GHz* and supports a bandwidth of around *30MHz* when both transmission channels are active. As for digital signal processing, the board employs a Xilinx Spartan 6 FPGA and connects to a host via a USB 3.0 interface. The transceiver FPRF is an AD9361 from Analog Devices applying 12 A/D and D/A converters at the digital interface.

However, since Ettus was fairly recently acquired by National Instruments (NI), a contact for a sale offer to NI concluded in the company offering the USRP-2901[17], which has the same characteristics as the B210. In detail, NI offered the Comm Sys MIMO Teaching Bundle[18], which includes two USRP-2901 devices together with cables and courseware.

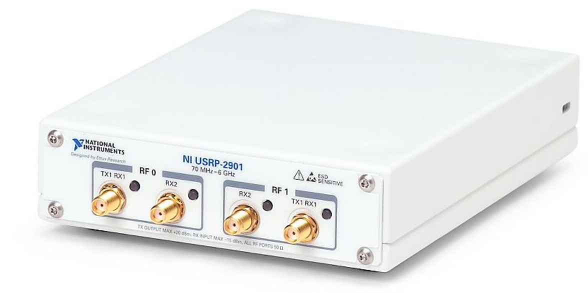


Figure 3.2: USRP-2901 product picture from NI homepage[17].

The front face of the device has the analog connections to the antennas, where as the back side includes interfaces to the host and a plug for power supply. The FPGA employs 147.000 logic elements. Note that NI has an even wider collection of possible SDR devices[19].

Software

This product comes with the choice of two different software tool chains.

The first one is the NI-USRP driver, which is programmed using NI's development toolkit LabVIEW. This is a data flow programming style toolkit, where both the host and the FPGA can be programmed, therefore no VHDL skills are needed. The second option is the USRP Hardware Driver (abbreviated as UHD), which is published under open source license and uses *C/C++* libraries. It offers cross-platform support for development environments like GNU Radio, MATLAB and *Python*. Both drivers work under Windows and Linux.

3.2.2 LimeSDR

Hardware

LimeSDR[20] is a SDR platform from Lime Microsystems. The FPRF is one of Limes' own chips, the LMS7002M, a transceiver IC which employs dual antenna diversity. LMS7002M has a built in Microcontroller that handles the calibration and tuning, a 12 bit interface to its analog front-end and a frequency coverage between *100kHz* and *3.8GHz* at a bandwidth of around *30MHz* in dual mode. Furthermore, it provides a digital transceiver signal processing block and adjustable analog and digital filters.

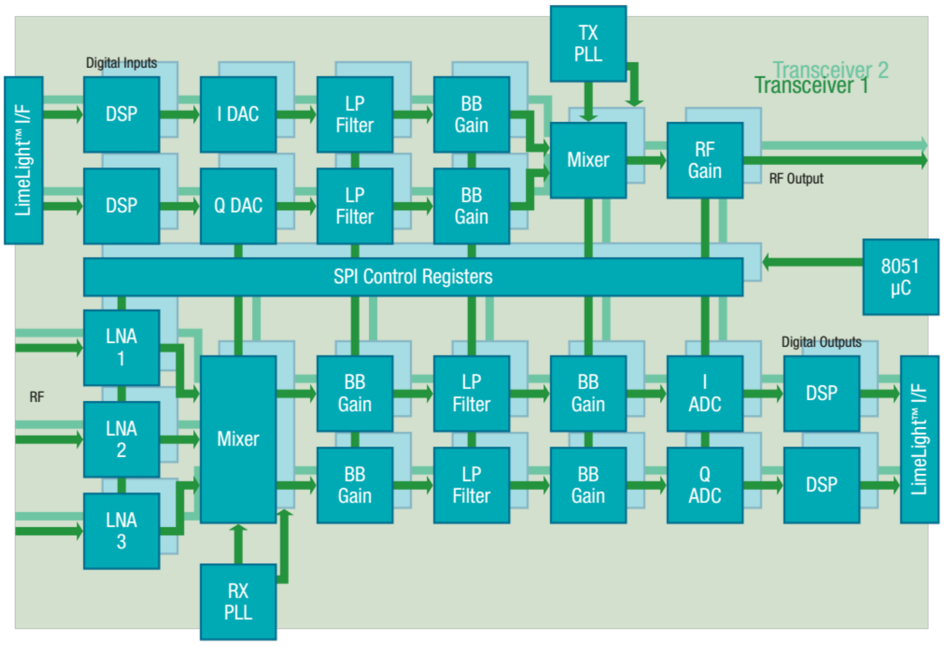


Figure 3.3: Block diagram of LMS7002M with dual transceiver topology[21].

Figure 3.3 illustrates the internal structure of the LMS7002M FPRF transceiver. It clearly shows the dual transceiver topology and visualizes the transmission part on the top half and the receiving part on the bottom half. The transmission chain first gets the samples from the FPGA, further processes them, converts them to analog domain before they are filtered, amplified and mixed with the carrier frequency. In the receiving part, the signals are processed in reversed order. However, there are three different pre-selections on received signal frequency range and therefore three analog paths: Low-, high- and wideband frequency range. The different blocks are controlled via SPI (**S**erial **P**eripheral **I**nterface).

As for the digital signal processing, the SDR uses an Altera Cyclone IV FPGA with 39.600 logic elements. The interface controller is a Cypress USB 3.0 controller. Figure 3.4 shows the different components on the LimeSDR board and how these are connected to each other. Note that there is also a Lime board with a different interface (PCIe) and a SDR called LimeSDR mini using the already mentioned older LMS6002D.

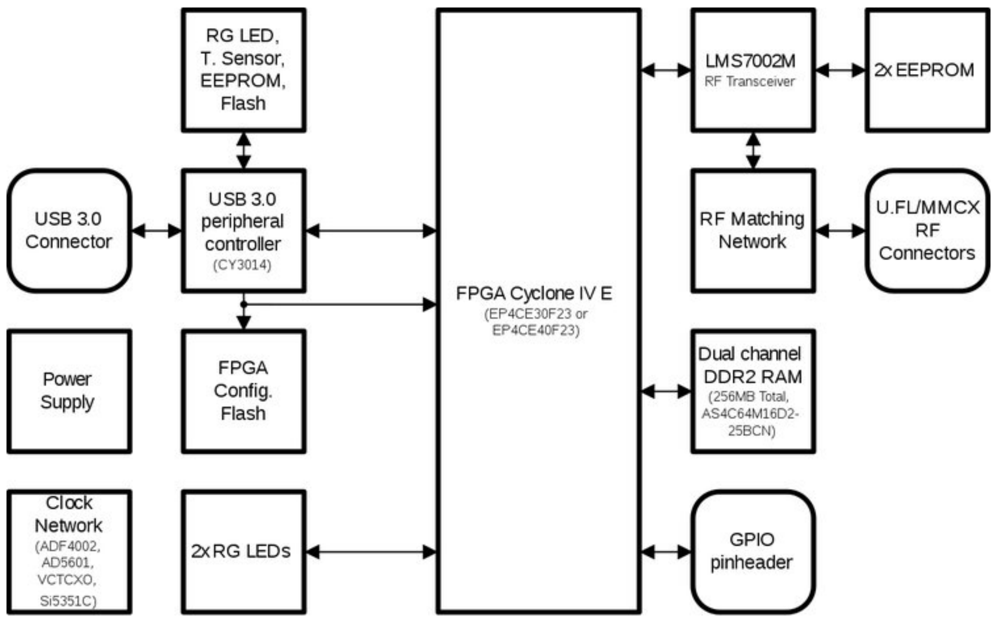


Figure 3.4: LimeSDR components block diagram[22].

Software

As for the software, Lime Microsystems offers a full open source project and its own driver for its devices[20]. Designers have access to board schematics and layouts, FPGA gateware projects, interface controller firmware and all data sheets.

The open source driver is called LimeSuite, which includes *C/C++* API library and an additional MATLAB API, with comes with limited functionality. After installing the driver, LimeSuite also deploys a GUI (**G**raphical **U**ser **I**nterface). This makes it easy to fast set-up a communication or to sniff for frequency bands. The digital signal processing on the host can be implemented using GNU Radio, Pothos, SoapySDR or UHD. As for operating systems, both Linux and Windows are supported.

3.2.3 XTRX

Hardware

The XTRX platform[23] uses the same FPRF as the already mentioned LimeSDR, the LMS7002M, see Figure 3.3. Therefore, it has a similar frequency range from *30MHz* to *3.8GHz*, and of course a dual transceiver architecture enabling antenna diversity. Digital-to-analog interface has a width of 12 bits.

As for the digital part, XTRX deploys a Xilinx Artix 7 FPGA. There are two configurations of the platform, CS and Pro. The first one comes with a smaller FPGA in terms of logic gates. The hardware block diagram is illustrated in Figure 3.5 and it shows the connection between the components and also the interface to the host, a mini PCIe edge. This interface has a very high bandwidth. XTRX also comes with a SIM Card reader and an on-board GPS disciplined oscillator. Similar to the previous discussed SDRs the platform also has GPIO (**G**eneral-**P**urpose **I**nput/**O**utput) connectors.

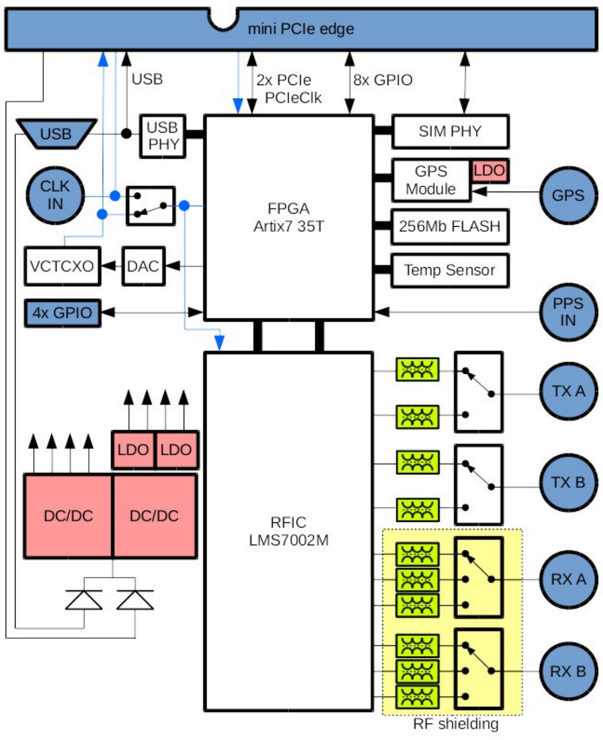


Figure 3.5: XTRX block diagram showing all hardware components[24].

If embedding the device in a host PC using the PCIe interface is not possible, the producers also sell a USB 3.0 adapter with aluminum enclosure.

Software

All the needed software is open source and can be found on Github[25]. It includes FPGA gateware and the driver which implements low-level API and a SoapySDR interface. This interface can be implemented in the already mentioned UHD environment. Note however, that as of the writing of this thesis, the driver is only supported by Linux, although a Windows version is in development.

3.2.4 Other Options

There are a variety of other hardware platforms that could be used for this thesis, especially considering the option of using multiple single transceiver boards to set up a antenna diversity system. However, most options had a general fraud like being under powered, to expensive or with insufficient software support.

3.3 Hardware Design Decision

After summarizing the advantages and disadvantages of the different SDR platforms, a design decision was made. First of all, the XTRX, even though it has sufficient hardware capabilities, had one major drawback: At the time of deciding which hardware platform to use, this board was in an early stage of development. The platform versions that would have been available were still prototypes and the software support was lacking at that time.

As of the other two possibilities, both boards have a USB 3.0 interface and were very similar in terms of computational power. They also had sufficient bandwidth, dual transceiver capabilities and they covered the desired frequency range. However, the USRP platform has a way higher maximum frequency than is actually needed. Both devices provide not enough output power to utilize the maximum output power as constrained by the ETSI EN 300 328[15]. As for the price range, LimeSDR is a much more affordable SDR. On the software support side, both platforms can be used with GNU Radio and UHD and have full FPGA support. USRP can also be operated using LabVIEW. Both drivers work with Linux and with Windows.

Comparing these two devices, it is obvious that USRP is the more powerful option. When keeping in mind that the advantages of the USRP are not necessary for this thesis (for example the higher frequency range), they both become very similar again. And this does not justify the higher price of the USRP, coming in at least five times the amount compared to the LimeSDR platform. The LimeSDR also has the advantage of having a total open source software environment.

So it was concluded to work with the LimeSDR platform in this thesis. Over the course of development, a total of four LimeSDR boards were ordered, two of which were bought with the aluminum kit as shown in Figure 3.6 and additional antennas and USB interface cables. Additional material include:

- Two USB 3.0 type A male/female cables with a length of one meter for a more flexible deployment of the boards.
- A USB 3.0 Hub to further extend flexibility of deployment.
- Four QPF7200 evaluation boards from Qorvo. These are power amplifiers with selective bandwidth filters and *37dB* of TX gain. With these it was planned to extend the maximum radiated power of both devices to meet the maximum allowed transmission power of the EN.

Figure 3.6: Ordered LimeSDR in aluminum kit.

For simplicity purpose the four devices were named by different colors. The corresponding serial numbers of the devices are

- Green, 0009070602451726
- Red, 0009070105C50D11
- Blue, 0009072C02881C16
- Yellow, 0009070602461225

3.4 LMS7002M Structure

For later discussions, it is necessary to know about some internal structures of the LMS7002M dual transceiver FPRF. Therefore, this section will cover the parts that will become relevant later on. Note that most information will be taken from the current newest version of the data sheet[26]. Some information will also link hardware structures to their usage in the LimeSuite driver implementation and API calls.

Section 3.2.2 already mentioned that the LMS7002M connects to a DSP, in this case to a FPGA. The interface between these two components is called LimeLight. It is TDD (**T**ime-**D**ivision **D**uplexing) and FDD (**F**requency-**D**ivision **D**uplexing) capable. After this interface, LMS7002M first employs a digital circuit chain refereed to as TSP (**T**ransceiver **S**ignal **P**rocessing), followed by DAC/ADC components and the analog chain which contains filters, amplifiers and mixers. All of the analog parts are adjustable.

The concept how these parts are connected is illustrated in Figure 3.7 for both the analog and digital part. It's also clearly visible that there are 3 RX paths, optimized for different frequency ranges, and 2 different TX paths. Since the chip has dual transceiver architecture, all these paths are doubled.

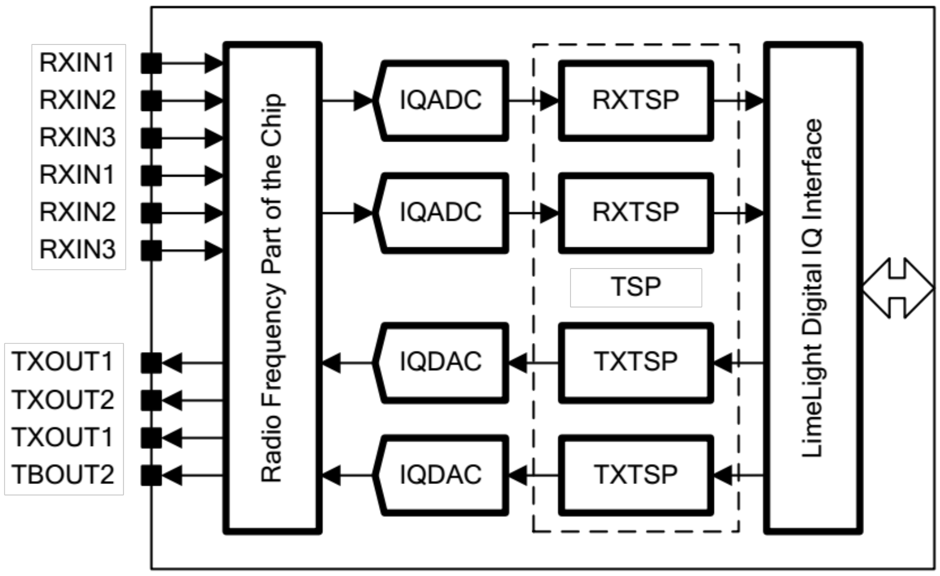


Figure 3.7: LMS7002M interface, digital and analog concept[26].

3.4.1 Gain Control

The transmitter path has two amplifier stages: One provides digital gain control and another one handles the programmable gain of the RF signal. On the receiving side of the system are three stages of gain control. Their maximum gain ranges from *12dB* to *32dB* and is most times being controlled by AGC to get the optimum range of signal magnitude into filter stages or into the ADCs. There is also an additional optional digital gain control step.

Gain values can be changed via the LimeSuite API call *LMS_SetGaindB* for both directions. Notice that when using this function, the individual amplifier gains will be set automatically, only a gain value in dB must be specified.

3.4.2 Low-Pass Filters

There are selective LPF in both transmit and receive path. All filters have programmable pass-band frequency values, however, exact ranges vary between TX and RX filters. Two kinds of filters complement each other: Analog and digital filters combine their performance in high flexibility and very good adjacent channel rejection.

The API call to change transition widths of filters is called *LMS_SetLPFBW*. Similar to the API call for the gain, it sets and tunes all relevant filters automatically.

3.4.3 Synthesizer

LMS7002M has two low phase noise synthesizers to mix the carrier frequency. Both of these can operate up to *3.8GHz* and drive the IQ mixers on the transmit and the receive path.

In the LimeSuite driver, using the API call *LMS_SetLOFrequency* lets the user define the LO frequency by which the baseband signal is mixed.

3.4.4 Interface Lengths

As previously discussed, the LMS7002M comes with 12 bit D/A and A/D converters. This is also the interface width on the LimeSDR between the LMS7002M FPRF and the Altera Cyclone IV FPGA. LimeSDR has some additional interfaces like GPIO and JTAG which are not relevant for this thesis.

To modify the sampling rates of the ADC and DAC, the user can call *LMS_SetSampleRateDir* and provide a sampling rate in floating point notation as well as an oversample value.

3.4.5 Transceiver Signal Processing

As obvious from the last few subsections, LMS7002M has a multiple staged digital path. Figure 3.8 illustrates the digital processing in the receive path and Figure 3.9 illustrate digital processing in the transmitting path of the FPRF.

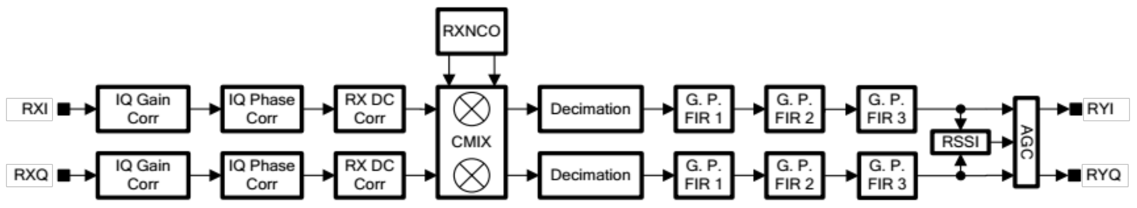


Figure 3.8: LMS7002M RX digital path[26].

Both paths have correction blocks: DC correction and phase/gain correction in the complex symbol plane. The TX path also employs an inverted sinc filter, which compensates for $\sin(x)/x$ amplitude roll of by the DAC. A mixer structure, designed to correctly mix complex IQ signals together, processes the incoming signal with a numerically controlled oscillator, which provides high resolution sine and cosine waveforms. Further blocks are decimation and interpolation in the receive and transmit path respectively and 3 freely configurable digital GFIR (**G**eneral-Purpose **FIR** filter) filter blocks. Finally, the RX path also employs AGC control.

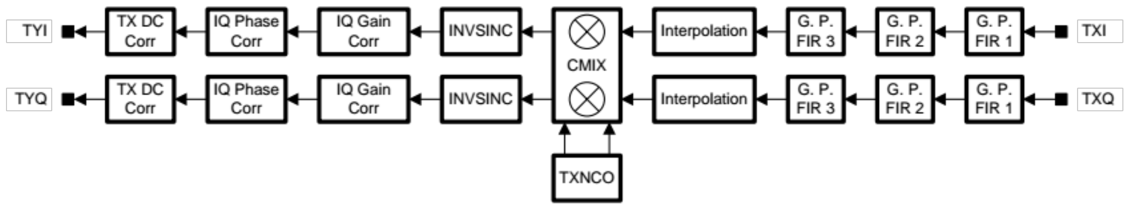


Figure 3.9: LMS7002M TX digital path[26].

3.5 Antennas

The mentioned LimeSDR aluminum kit comes with extra antennas to connect to the SMA (**S**ub**m**iniature version **A**) connectors. These are dipole antennas, transmitting a strong linear polarized signal along their horizontal plane and low power into the vertical axis. Figure 3.10 shows the ideal radiation pattern of a dipole antenna in the horizontal plane.

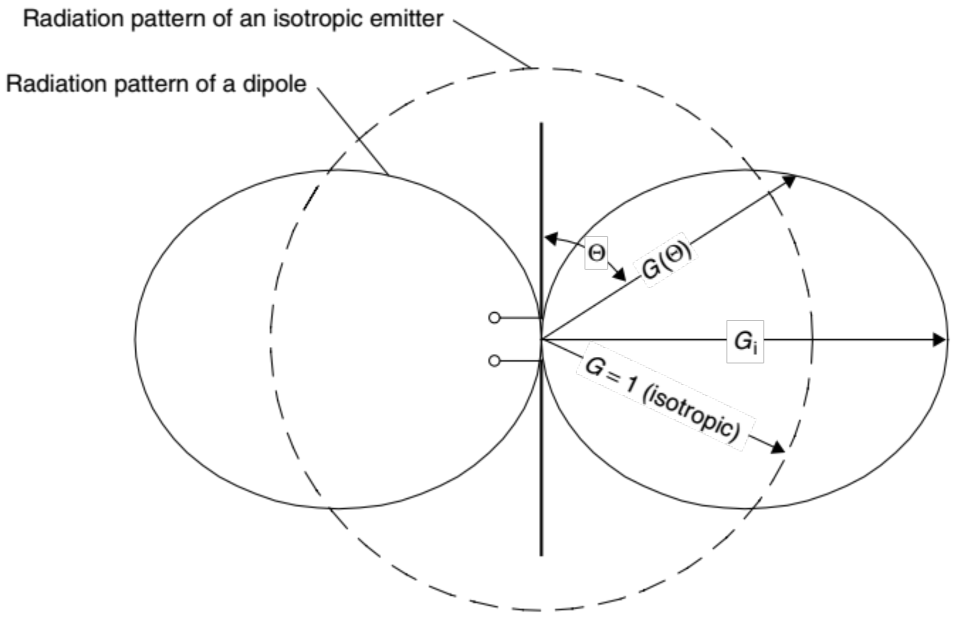

Radiation pattern of an isotropic emitter
Radiation pattern of a dipole
Θ
G(Θ)
G_i
G = 1 (isotropic)

Figure 3.10: Horizontal radiation pattern of a dipole compared to an isotropic emitter[3].

For the sake of completeness, these antennas were characterized for the thesis in the anechoic chamber of the institute. The Institute of Microwave and Photonic Engineering has an antenna measurement chamber.

First of all, the measurement setup had to be calibrated. For this purpose a Rohde & Schwarz ZVA 67 Vector Network Analyzer was connected to a Schwarzbeck SBA 9113 reference antenna. Figure 3.11 shows the already calibrated measurement setup with one of the antennas to be characterized.

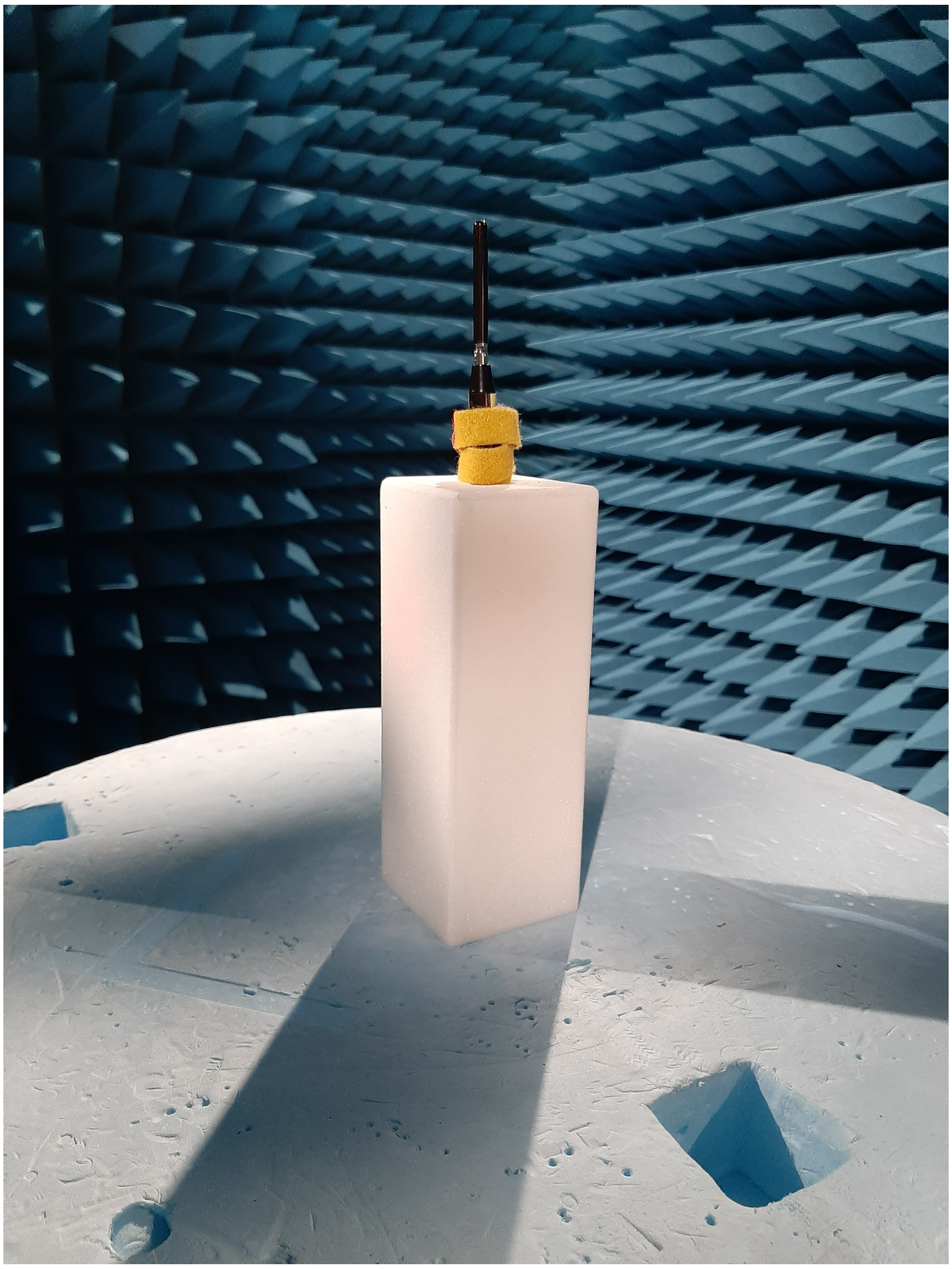
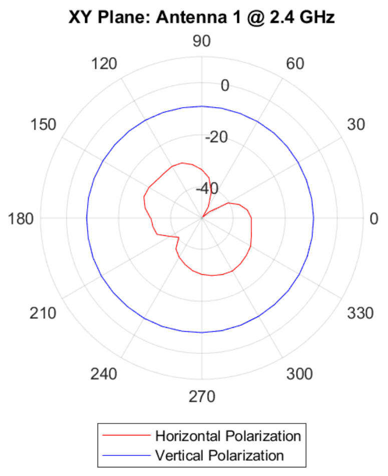


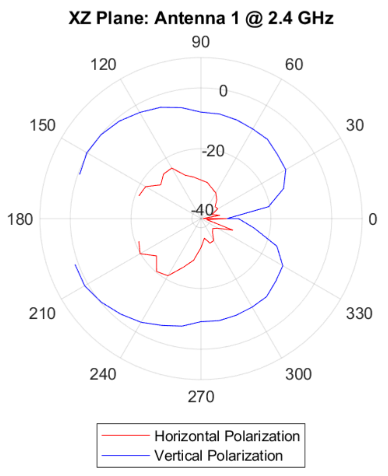
Figure 3.11: Anechoic chamber with connected AUT.

Clearly visible to the right is a circular disc which rotates and measures the radiating power from the AUT (**A**ntenna **U**nder **T**est). The AUT sits on a rotating desk which can turn by a full 360 degree angle.

A MATLAB script was responsible to evaluate the measurement results and plot them respectively. Figure 3.12 shows the results, with a clearly visible radiation pattern resembling the ideal depicted in Figure 3.10 in the XY plane. However, the XZ plane measurements could not complete a full 360 degree turn due to obstructing material, and it shows a very bad radiation power in the zero degree direction.



(a) Radiation in XY plane.



(b) Radiation in XZ plane.

Figure 3.12: Polar radiation graphs for LimeSDRs antennas, both for vertical and horizontal polarization.

Finally, the MATLAB tool also evaluated the matching of the antenna for a frequency range up to *10GHz*, illustrated in Figure 3.13. It is clearly visible that the dipole shows rather mediocre results. At the interesting frequency range between *2.4GHz* and *2.5GHz* the reflection coefficient ranges between *-7dB* and *-9dB*. These measurements were accomplished for only one of the antennas, quick checks with a calibrated ZVL Network Analyzer showed that all other antennas have similar characteristics.

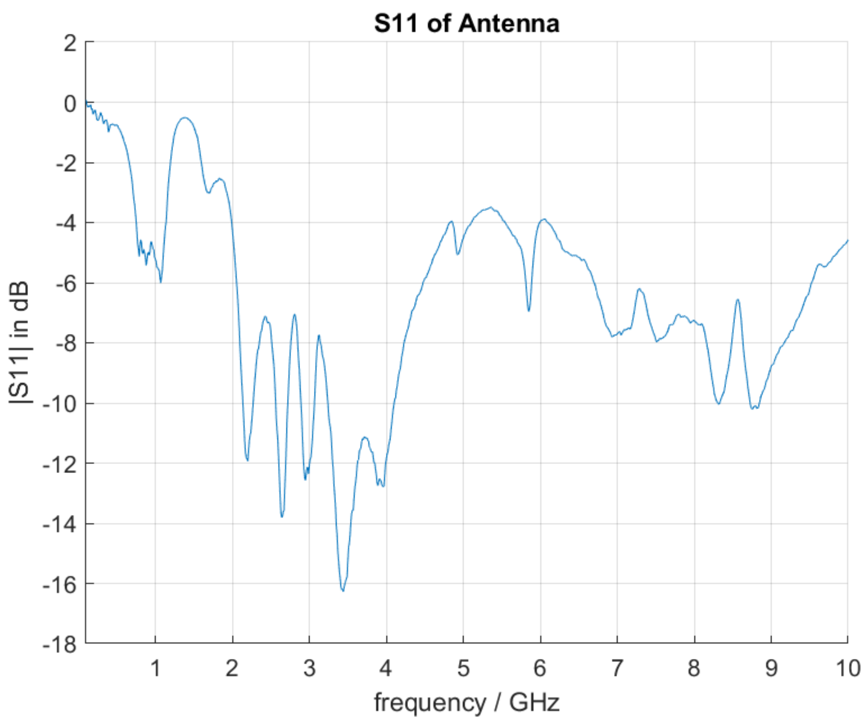


Figure 3.13: Reflection coefficient over a defined frequency range.

3.6 Host Setup

This system will either work with one host connecting to both devices for short range transmissions, or use two hosts with each connecting to one device for longer distance channels. Using two hosts also reduces the workload and should be considered when processing large amounts of data over a small period of time.

3.6.1 Hardware

The host will be a standard issue PC or Laptop. Minimum requirements consist of at least 4GB RAM, an Intel i5 Processor (or equivalent or better) and USB 3.0 interface connectors. Additionally it is suggested to use a SSD for faster read and write operations.

3.6.2 Software

Operating System

LimeSDRs driver supports both Windows and Linux OS (**O**perating **S**ystems). However, it was decided to use Linux as OS. The reason being that it is a good platform for software development and compiling of source code is simplified.

For simplicity the chosen Linux based OS was Ubuntu 16.04 LTS. Over the course of this thesis the version changed though, since close at the beginning of development Ubuntu was updated to 18.04 LTS. Therefore this documentation will focus on version 18.04.

Additional Packages

There are a multitude of additional packages that should be installed to Ubuntu. Chapter **Host Packages** in the appendix includes a list of packages and which releases to use. It also mentions how to install the development environment.

Chapter 4

Wireless Analysis Tool

The first step to familiarize with the software environment of the open source driver called LimeSuite and the hardware platform LimeSDR was to create a console based **W**ireless **A**nalysis **T**ool (abbreviated as WAT) for simple communications between two devices. This chapter will document the goals of the tool, how WAT was set up and what additional software libraries were used, its structure, how it implemented the LimeSuite driver API, some use cases and why the tool was discontinued.

Chapter **Project Setup** in the appendix documents how to set up the development environment to modify and compile this tool.

4.1 Goals

Besides the goal of familiarizing with the development environment, console based WAT also aimed at the following:

- Incorporate the driver API.
- Operate on a single host.
- Open interface connection with multiple devices.
- Configure devices independently. There are multiple parameters to configure, including selecting antenna ports, setting gain values, sampling rates, LPF and LO frequencies.
- Save configuration files from a connected SDR and load configuration files into a connected SDR.
- Choose between different modulation schemes.
- The ability to calibrate devices.
- Either select two devices to set up a transmission, or select a single device and have the transmission between its transmit and receive paths. Also select which

binary data file should be transmitted, and if it should only be transmitted once or continuously.

- Furthermore start a stream environment. This includes importing the chosen data file, modulate and prepare it accordingly, transmitting the data and receiving it on the RX side. The received data is then demodulated and saved in a separate file. Also add a visualization of the data by using appropriate software tools.
- The mentioned stream environment should be controllable by the user. This means that the transmission shall be interrupt able, stopping the stream and giving the user the ability to change different parameters of the device or of the stream environment. This includes changing the modulation scheme, turning AGC on or off and configuration of all the previously mentioned device parameters, for example the gain. The user will even be able to read and write individual SPI registers.
- Finally, once the communication is finished or the user cancels the transmission, close the stream environment in a controlled manner, free up memory and close all opened descriptors.

The last steps would be to add signal processing to the data to correct for synchronization issues, add error correcting codes to counter BER (**B**it **E**rror **R**ate) and divide it to a multiple host communication tool. However, these functions were not implemented into WAT, the reasons for this being discussed in the conclusion of this chapter.

4.2 Implementation

Figure 4.1 shows the simplified block diagram of WAT. This is the latest release (Jan. 2020) of the project. Not all member variables and member functions are listed. In some cases, like multiple *get functions*, only representatives are listed. Blue blocks are external code, orange ones are global files needed across the whole project and yellow blocks are object classes.

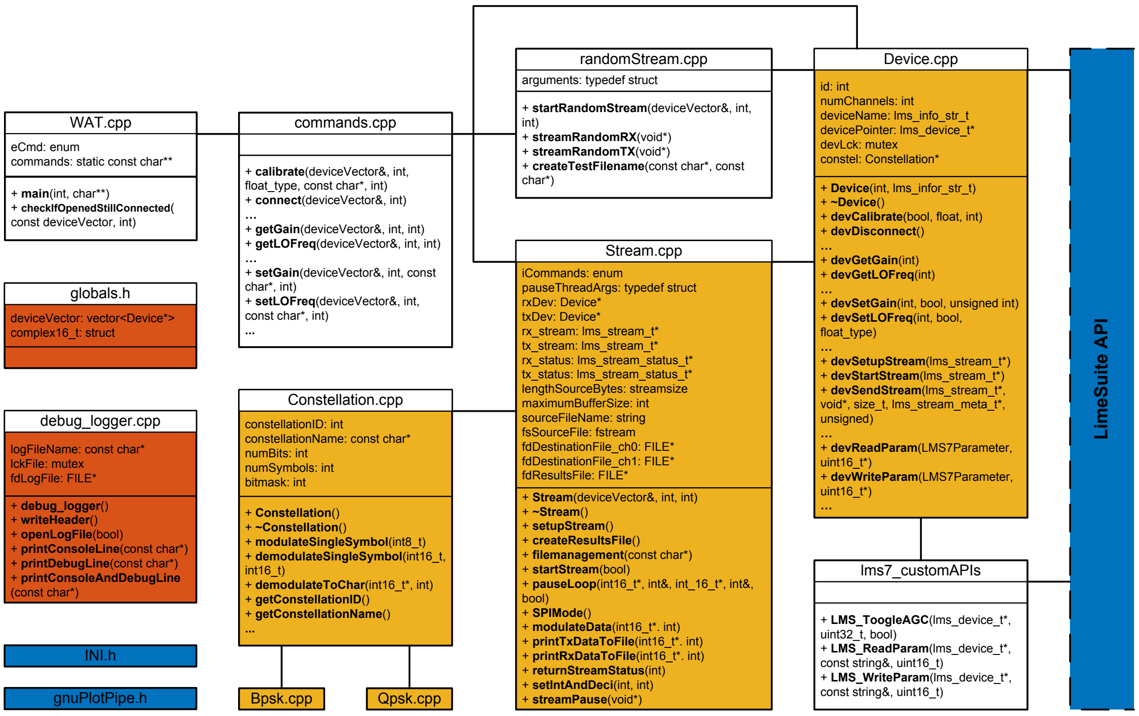

WAT.cpp
eCmd: enum
commands: static const char**
+ main(int, char**)
+ checkIfOpenedStillConnected(const deviceVector, int)
globals.h
deviceVector: vector<Device*>
complex16_t: struct
debug_logger.cpp
logFileName: const char*
lckFile: mutex
fdLogFile: FILE*
+ debug_logger()
+ writeHeader()
+ openLogFile(bool)
+ printConsoleLine(const char*)
+ printDebugLine(const char*)
+ printConsoleAndDebugLine(const char*)
INI.h
gnuPlotPipe.h
commands.cpp
+ calibrate(deviceVector&, int, float_type, const char*, int)
+ connect(deviceVector&, int)
...
+ getGain(deviceVector&, int, int)
+ getLOFreq(deviceVector&, int, int)
...
+ setGain(deviceVector&, int, const char*, int)
+ setLOFreq(deviceVector&, int, const char*, int)
...
Constellation.cpp
constellationID: int
constellationName: const char*
numBits: int
numSymbols: int
bitmask: int
+ Constellation()
+ ~Constellation()
+ modulateSingleSymbol(int8_t)
+ demodulateSingleSymbol(int16_t, int16_t)
+ demodulateToChar(int16_t*, int)
+ getConstellationID()
+ getConstellationName()
...
Bpsk.cpp
Qpsk.cpp
randomStream.cpp
arguments: typedef struct
+ startRandomStream(deviceVector&, int, int)
+ streamRandomRX(void*)
+ streamRandomTX(void*)
+ createTestFilename(const char*, const char*)
Stream.cpp
iCommands: enum
pauseThreadArgs: typedef struct
rxDev: Device*
txDev: Device*
rx_stream: lms_stream_t*
tx_stream: lms_stream_t*
rx_status: lms_stream_status_t*
tx_status: lms_stream_status_t*
lengthSourceBytes: streamsize
maximumBufferSize: int
sourceFileName: string
fsSourceFile: fstream
fdDestinationFile_ch0: FILE*
fdDestinationFile_ch1: FILE*
fdResultsFile: FILE*
+ Stream(deviceVector&, int, int)
+ ~Stream()
+ setupStream()
+ createResultsFile()
+ filemanagement(const char*)
+ startStream(bool)
+ pauseLoop(int16_t*, int&, int_16_t*, int&, bool)
+ SPIMode()
+ modulateData(int16_t*. int)
+ printTxDataToFile(int16_t*. int)
+ printRxDataToFile(int16_t*. int)
+ returnStreamStatus(int)
+ setIntAndDeci(int, int)
+ streamPause(void*)
Device.cpp
id: int
numChannels: int
deviceName: lms_info_str_t
devicePointer: lms_device_t*
devLck: mutex
constel: Constellation*
+ Device(int, lms_infor_str_t)
+ ~Device()
+ devCalibrate(bool, float, int)
+ devDisconnect()
...
+ devGetGain(int)
+ devGetLOFreq(int)
...
+ devSetGain(int, bool, unsigned int)
+ devSetLOFreq(int, bool, float_type)
...
+ devSetupStream(lms_stream_t*)
+ devStartStream(lms_stream_t*)
+ devSendStream(lms_stream_t*, void*, size_t, lms_stream_meta_t*, unsigned)
...
+ devReadParam(LMS7Parameter, uint16_t*)
+ devWriteParam(LMS7Parameter, uint16_t*)
...
lms7_customAPIs
+ LMS_ToogleAGC(lms_device_t*, uint32_t, bool)
+ LMS_ReadParam(lms_device_t*, const string&, uint16_t)
+ LMS_WriteParam(lms_device_t*, const string&, uint16_t)
LimeSuite API

Figure 4.1: Simplified block diagram of WAT.

Main project file is called **WAT.cpp**. It holds an *enum* which enumerates the different available *commands*. The main function is basically an endless *while* loop that waits for commands, then probes the user for different parameters dependent on the instruction. As of global files, **globals.h** defines a custom *vector* type of the *Device* class, while **debug_logger.cpp** manages a debug file and defines functions to print data to the file, to the console or to both. The debug file is automatically created (log.txt) in the folder where the binary gets called.

A very important file is **Device.cpp**, which defines a class called *Device* that gets created once for every connected device and then stored in a *deviceVector* list in *main*. It contains different specific variables for the device and also a mutex to make calls to this class thread safe. Thread safe calls were implemented for future extensions of the program, not all of which got implemented. This *Device* object is the main interface to the LimeSuite driver API, all functions that are implemented use at least one API call. The functions either are basic instructions (calibrate, initialize and reset the device), set or get different device parameters (gain, LPF bandwidth, LO frequency, ...), handle stream data (setup stream, send/receive data, ...) and calls to read/write SPI registers. Special API calls necessary for this project are defined in the file **lms7_customAPIs**. These functions mimic or slightly alter given API calls.

The **commands.cpp** file holds functions for every different command. The functions receive command specific information as well as a *deviceVector* list by reference, containing all opened *Device* objects from the main function. These functions then process the received data and either call the specific public functions of the selected *Device* classes or they set up a *Stream* class and start transmission. Table 4.1 shows a list of commands.

ID	eCmd enum	Explanation
0	ANTENNA	Get / Set specific Antenna ports.
1	CALIBRATE	Calibrate a device using a specified bandwidth.
2	CONNECT	Open to all connected devices. Already opened ones will be disconnected first.
3	CONSTELLATION	Swap the constellation (modulation scheme) of a opened device.
4	DEVICES	List all connected and, if available, all opened devices.
5	DISCONNECT	Disconnect all opened devices.
6	ENABLE	Enable specific TX / RX channel.
7	EXIT	Will clean up and exit the program.
8	GAIN	Get / Set relative TX or RX gain.
9	HELP	Print a list of available commands into console.
10	INIT	Load opened devices with default configuration.
11	LO	Get / Set LO Frequency.
12	LOAD	Load a configuration file.
13	LPBW	Configure low-pass bandwidth.
14	QUIT	Will clean up and exit the program.
15	RESET	Resets opened devices.
16	SAMPLE	Get / Set sampling rate.
17	SAVE	Save a configuration file.
18	STREAM	Will set up a stream object and start stream procedure between user defined devices and with a defined test file. User can pause stream by pressing *p* and issue commands.
19	WFMPLAYER	Specify a device to send a waveform through the FPGA waveform player.

Table 4.1: Possible console commands.

There are two possibilities to stream data: Either select a random data stream by using the functions in **randomStream.cpp**. This file starts a very basic stream with randomized information. The special thing about this stream environment is that it sets up two threads, one for TX and one for RX. It uses the typical *pthread_create* system calls to create threads, *pthread_join* to wait for thread termination and *yield* to pass thread execution to the next thread. However, this stream option should not be used, as it was written more as a tutorial for the author to write a much better stream object later on. Also, the code was written when the *Device* class was not yet thread safe, making this two threaded system very complicated using *chronos* high resolution clock and global *bools* to wait and synchronize threads.

The user can also set up a proper stream by using the **Stream.cpp** class, once the device parameters were set up properly. First create a *Stream* object, then call the member functions *setupStream* and *fileManagement*. It used to be that the function *setupStream* configured devices automatically, as this was faster then setting all device parameters via console every time WAT was restarted. Now it will only configure the device for stream operations. The second member function *fileManagement* will open the source file with the data to be modulated and sent, and then set up a source file with the received and demodulated data.

After the initial setup is complete, a call to the *startstream* member function will prepare data structures, read the data in the source file, modulate this data and start the stream threads. These are not the only threads however, an extra *pauseThread* gets created which will probe user input. The reason behind this: once the data gets sent and received by the devices, the user will have the possibility to press *p* or *P* on his keyboard whereby the pause thread will completely stop the stream and probe the user for commands.

The function *pauseLoop* gets called, which is structured very similar to the loop in the main function, asking the user for commands and subsequent parameters. However, the commands will not be called via characters, but rather via integer input. See Table 4.2 for a list of possible commands, what they do and how they are used. In this case, ID refers to the respective command integer input. Notice how this interface primarily works with numbers: The letter *i* symbolizes that the probed parameter should be an integer, *f* assumes a float and *b* assumes a bool (1/0) value.

ID	**iCommands enum**	**Explanation**
1	iCONTINUE	Will resume the communication.
2	iQUIT	Exit the stream environment and return to main.
4	iNEWDESTFILE	Delete the results file with the demodulated and received data and create a new, empty file.
5	iCHANGECONSTELLATION	Change the modulation scheme. This will modulate the data new. 1 is for BPSK, 2 for QPSK.
10	iPRINTTXDATA	Print all I/Q data points of the TX data into console.
11	iPRINTRXDATA	Print all I/Q data points of the RX data into console.
15	iPRINTSTREAMDATA	Get information about the LimeSuite internal *lms_stream_t* struct of the stream.
16	iPRINTDEVICEINFO	Print device ID and name.
18	iPRINTTXDATATOFILE	Take TX buffer data and print it to a file.
19	iPRINTRXDATATOFILE	Take RX buffer data and print it to a file.
20	iCHANGELPBW	Change the LPF bandwidth. Assumed input is *float* in MHz.
21	iGETLPBW	Return the LPF bandwidth.
22	iCHANGESAMPLING	Change the sampling rate. Assumed input is *float* in MHz and afterwards the oversampling value as *integer*.
23	iGETSAMPLING	Return the sampling rate.
24	iCHANGEGAIN	Change the gain value. Assumed input is *integer* for the gain value and afterwards *bool* for TX (1) or RX (0) direction.
25	iGETGAIN	Return the gain value.
26	iCHANGECLOCK	Change a specific clock. The first query assumes the clocks' ID (see iGETCLOCK), the second one the value in *float* (MHz).
27	iGETCLOCK	Return the frequency values for the clocks LMS_CLOCK_SXR (1), LMS_CLOCK_SXT (2) and LMS_CLOCK_CGEN (3).
28	iTUNECLOCK	Similar to iCHANGECLOCK, but this will tune the clock instead of setting a value.

30	iCHANGELO	Change the LO frequency. Assumed input is *float* in MHz, then a *bool* for the direction (TX/RX).
31	iGETLO	Return the LO frequency.
32	iCHANGEANTENNAPORT	Change the antenna port. First select the ID of the port, then the *bool* for direction (TX/RX). See also iGETANTENNAPORT.
33	iGETANTENNAPORT	Return the antenna ports. There are different antenna bands: **H**igh frequencies, **l**ow frequencies and **w**ideband frequency band.
34	iSETGFIRLPF	Set GFIR frequency value. This is a digital filter in the LMS7002M digital path, see also the data sheet[26]. First parameter is the direction (TX/RX), second one a *bool* that decides to enable (1) or disable (0) the filter and the third parameter is the bandwidth *float* in MHz.
36	iSETINTANDDEC	Set interpolation and decimation values in LMS7002M digital path, see also the data sheet[26]. The first *integer* is the interpolation value, the second one is the decimation value.
100	iONETONE	This will replace the TX data with a one tone sinusoidal using the formula $\sin(2\pi i/n)$, where i is the sample number and n is the chosen number by the user.
200	iSPIMODE	This will start SPI mode.

Table 4.2: Possible commands during stream pause and how they are used.

In SPI mode, the user can read and write individual SPI registers. The user can choose between read (0)/ write (1) a register or search for a register (3), see also the LMS7002M programming guide[27]. Further examples on how to use this mode are described in subsection 4.3.5.

The object **Constellation.cpp** is a virtual class. It holds virtual functions for modulation and demodulation that have no implementation, instead they get implemented by the files **Bpsk.cpp** and **Qpsk.cpp** that inherit the parent object.

The headers **INI.h** and **gnuPlotPipe.h** are external files. Note that the **API** library used was a locale one, which means that the LimeSuite source code was copied into the project. This included not all LimeSuite files, rather picking together the necessary source code and header files. The commit that was used is, as already mentioned, from 2018-08-16.

4.3 Use Cases

This section will focus on use cases of this tool and showcase a few commands.

4.3.1 Startup

To start the tool, simply run the binary in the terminal. If there are *Permission denied* errors thrown by *libusb*, it is useful to start the tool as administrator using *sudo*.

```
./WAT
```

```
WAT version 0.3, will init and start prompt.
Warning: Most inputs will not be checked for type or validity.
=>
```

Listing 4.1: Starting prompt of WAT.

If no devices are connected, the program will only give a prompt and wait for further commands. If the user enters *help* he will be presented with a list of available commands. The commands *exit* or *quit* will close the tool again, see Listing 4.2.

```
WAT version 0.3, will init and start prompt.
Warning: Most inputs will not be checked for type or validity.
=>help
Available commands (case sensitive):
antenna:       Get / Set specific Antenna ports active.
calibrate:     Calibrate a device for a specified bandwidth.
connect:       Open to all connected devices. Already opened ones will
    be disconnected first.
constellation: Swap the constellation (modulation scheme) of a opened
    device.
devices:       List all connected and, if available, all opened
    devices.
disconnect:    Disconnect all opened devices.
enable:        Enable specific TX / RX channel.
exit / quit:   Will clean up and exit the program.
gain:          Get / Set relative TX or RX gain.
init:          Load opened devices with default configuration.
lo:            Get / Set LO Frequency.
load / save:   Load / Save a configuration file.
lpbw:          Configure low-pass bandwidth.
reset:         Resets opened devices.
sample:        Get / Set sampling rate.
stream:        Will set up a stream object and start stream procedure
    between user defined devices and with a defined test file. User can
    pause stream by pressing "p" and issue commands.
wfm:           Specify a device to send a waveform through the FPGA
    waveform player.

Tip: Use -1 when prompted with deviceID, channel or similar to select
    all available.
=>exit
```

Listing 4.2: Help and exit command.

If devices are already connected, the tool will open communication with them as the program starts. However, this can also be done by issuing the *connect* command. Typing *devices* gives a list of devices and tells the user if the interface is already opened, while *disconnect* closes the communication. Listing 4.3 shows an example on how to use these commands.

Note that the tool will inform the user of a gateware mismatch! The reason being, as already discussed, that the tool uses an older local LimeSuite API library. This warning can be ignored, especially if the gateware version on the device is higher than the expected one.

```
Gateware version mismatch!
  Expected gateware version 2, revision 17
  But found version 2, revision 21
  Follow the FW and FPGA upgrade instructions:
  http://wiki.myriadrf.org/Lime_Suite#Flashing_images
  Or run update on the command line: LimeUtil --update

Estimated reference clock 30.6587 MHz
Reference clock 30.72 MHz
WAT version 0.3, will init and start prompt.
Warning: Most inputs will not be checked for type or validity.
Found number of devices: 1
Opened devices:
0: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
=>disconnect
=>devices
Connected devices:
LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
No opened devices.
=>connect
Gateware version mismatch!
  Expected gateware version 2, revision 17
  But found version 2, revision 21
  Follow the FW and FPGA upgrade instructions:
  http://wiki.myriadrf.org/Lime_Suite#Flashing_images
  Or run update on the command line: LimeUtil --update

Estimated reference clock 30.6587 MHz
Reference clock 30.72 MHz
Found number of devices: 1
Opened devices:
0: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
=>devices
Connected devices:
LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
Opened devices:
0: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
```

Listing 4.3: Disconnect device that is connected on tool startup and connect it again. Ensure connection using devices command.

4.3.2 Device Configuration

Once a device is connected it can be configured using specific commands. This subsection will show a few examples. The ID of the different devices to use is visible using the *devices* command, on the far left. Note that "*...*" means that debug lines were cut from this representation. Listing 4.4 shows how to use the *init* command.

```
=>device
Connected devices:
LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
Opened devices:
0: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
=>init
Init all devices? (y/n)
=>init=>n
Specify device ID to init.
=>init=>0
...
```

Listing 4.4: Init command example.

The next example Listing will set the LO frequency of device 0 to *2400MHz* for both channels.

```
=>lo
Get/Set LO frequency? (get, set)
=>lo=>set
Specify device ID.
=>lo=>0
Specify Channel (0/1).
=>lo=>0
TX, RX or both? (tx, rx, both)
=>lo=>set=>both
Device: 0. LO Frequency range: RX = 0.100000MHz - 3800.000000MHz, step
    0.000000. TX = 0.100000MHz - 3800.000000MHz, step 0.000000
Device: 0. LO Frequency: RX = 1200.000000MHz, TX = 1249.999995MHz.
    Channel: 0
Specify desired LO frequency in MHz for channel 0.
=>lo=>set=>tx=>2400
...
Device: 0. LO Frequency range: RX = 0.100000MHz - 3800.000000MHz, step
    0.000000. TX = 0.100000MHz - 3800.000000MHz, step 0.000000
Device: 0. LO Frequency: RX = 1200.000000MHz, TX = 2400.000000MHz.
    Channel: 0
Specify desired LO frequency in MHz for channel 0.
=>lo=>set=>rx=>=>2400
```

```
...
```

Listing 4.5: LO command example.

Note that in this context *channel* refers to the analog paths to the antennas. LimeSDR has two integrated transceiver paths, therefore two channels. The user can also choose *-1* if probed for a multiple choice answer to select all. This includes for example channels or choosing multiple devices. In the next Listing, the LPF bandwidth of two devices will be changed to *10MHz* on both channels using the *-1* command.

```
=>devices
Connected devices:
LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009072C02881C16, BLUE
LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
Opened devices:
0: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009072C02881C16, BLUE
1: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
=>lpbw
Set low-pass bandwidth of which device?
=>lpbw=>-1
TX, RX or both? (tx, rx, both)
=>lpbw=>tx
Specify Channel (0/1).
=>lpbw=>-1
Device: 0. LPBW Range: RX = 1.400100MHz - 130.000000MHz, step
    0.000000. TX = 5.000000MHz - 130.000000MHz, step 0.000000
Device: 0. Currently LPBW set to RX = 5.000000MHz, TX = 5.000000MHz.
    Channel: 0
Device: 0. LPBW Range: RX = 1.400100MHz - 130.000000MHz, step
    0.000000. TX = 5.000000MHz - 130.000000MHz, step 0.000000
Device: 0. Currently LPBW set to RX = 5.000000MHz, TX = 5.000000MHz.
    Channel: 1
Specify bandwidth in MHz (0 to exit) for channel 0.
=>lpbw=>set=>tx=>10
...
Specify bandwidth in MHz (0 to exit) for channel 1.
=>lpbw=>set=>tx=>10
...
Device: 1. LPBW Range: RX = 1.400100MHz - 130.000000MHz, step
    0.000000. TX = 5.000000MHz - 130.000000MHz, step 0.000000
Device: 1. Currently LPBW set to RX = 5.000000MHz, TX = 5.000000MHz.
    Channel: 0
Device: 1. LPBW Range: RX = 1.400100MHz - 130.000000MHz, step
    0.000000. TX = 5.000000MHz - 130.000000MHz, step 0.000000
Device: 1. Currently LPBW set to RX = 5.000000MHz, TX = 5.000000MHz.
    Channel: 1
Specify bandwidth in MHz (0 to exit) for channel 0.
=>lpbw=>set=>tx=>10
...
```

```
Specify bandwidth in MHz (0 to exit) for channel 1.
=>lpbw=>set=>tx=>10
...
```

Listing 4.6: LPBF and -1 commands example.

As mentioned there are also *get functions*. Listing 4.7 shows how to return the available antenna ports. This also shows which antenna ports are active.

```
=>antenna
Get/Set antenna ports? (get, set)
=>antenna=>get
Specify device ID.
=>antenna=>0
Specify Channel (0/1).
=>antenna=>-1
ID| RX             | TX
0: NONE            | NONE
1: LNAH            | BAND1
2: LNAL            | BAND2
3: LNAW            |
4: LB1           |
5: LB2           |
Device: 0. Acitve ports: RX 1, TX 1. Channel: 0. DeviceID: 0
ID| RX             | TX
0: NONE            | NONE
1: LNAH            | BAND1
2: LNAL            | BAND2
3: LNAW            |
4: LB1           |
5: LB2           |
Device: 0. Acitve ports: RX 1, TX 1. Channel: 1. DeviceID: 0
```

Listing 4.7: Antenna command example.

The user can also export the configuration of a device by using the *save* command and choosing a file name.

```
=>save
Save Config from which device?
=>save=>0
Where to save the Config File?
=>save=>conf.ini
```

Listing 4.8: Save command example.

As a last example, Listing 4.9 will show how to calibrate a specific transceiver path.

```
=>calibrate
Specify device ID to calibrate.
=>calibrate=>0
Specify bandwidth in MHz.
=>calibrate=>10
Specify Channel (0/1).
=>calibrate=>0
```

```
TX, RX or both? (tx, rx, both)
=>calibrate=>both
...
Tx | DC  | GAIN | PHASE
---+-----+------+------
I: | -154 | 1963 | -15
Q: | 150 | 2047 |
...
RX | DC  | GAIN | PHASE
---+-----+------+------
I: |   8 | 1663 | 87
Q: |   8 | 2047 |
...
```

Listing 4.9: Calibration command example.

4.3.3 Stream

In order to send and receive data (binary data, for example a text file), the user needs to issue the *stream* command, as shown in Listing 4.10. First select RX and TX device (this can also be the same device) and then select a file to send. The user can either select the file including its path in the file system, relative to the binary, or for simplicity purpose copy the file in the same directory as the binary.

```
...
0: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070602451726, GREEN
1: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108,
    serial=0009070105C50D11, RED
=>stream
RX device?
=>stream=>1
TX device?
=>stream=>0
Specify file to stream in local folder.
=>stream=>cafe.txt
...
```

Listing 4.10: Stream command example.

After this the stream environment will be set up. If this is successful, the last query will be if the data should be transmitted once or in a continuous mode. If the binary was compiled with the *gnuPlot* symbol set, there will be a visualization of the data. For TX the constellation data will be shown only once, for RX it will update every second.

```
Continuous stream? (0/1)
=>stream=>
Tx: 8.016 MB/s
Rx: 8.037 MB/s
Tx: 7.979 MB/s
Rx: 8.028 MB/s
Tx: 8.040 MB/s
```

```
Rx: 8.045 MB/s
```

Listing 4.11: Selecting continuous stream.

In order to stop a continuous stream, press *p* or *P* and hit the *enter* key. This will stop the communication and start the pause loop. Type in a *2* to stop the stream environment. Other possible options will be discussed in subsection 4.3.5.

An example configuration: Set antenna ports to BAND1 on TX side and LNAH on RX side, both LO to *2.4GHz* and a bandwidth of *10MHz*, with a sampling rate of *1MHz* at 4 times oversampling and a gain value of 40 for both devices. Furthermore, use QPSK modulation scheme.

Figure 4.2 shows the TX constellation data using *gnuplot*. Since QPSK has 4 constellation symbols with equal amplitude but different phase, the symbols are visible at the edges of the constellation diagram. Of course multiple symbols lie on the same spot.

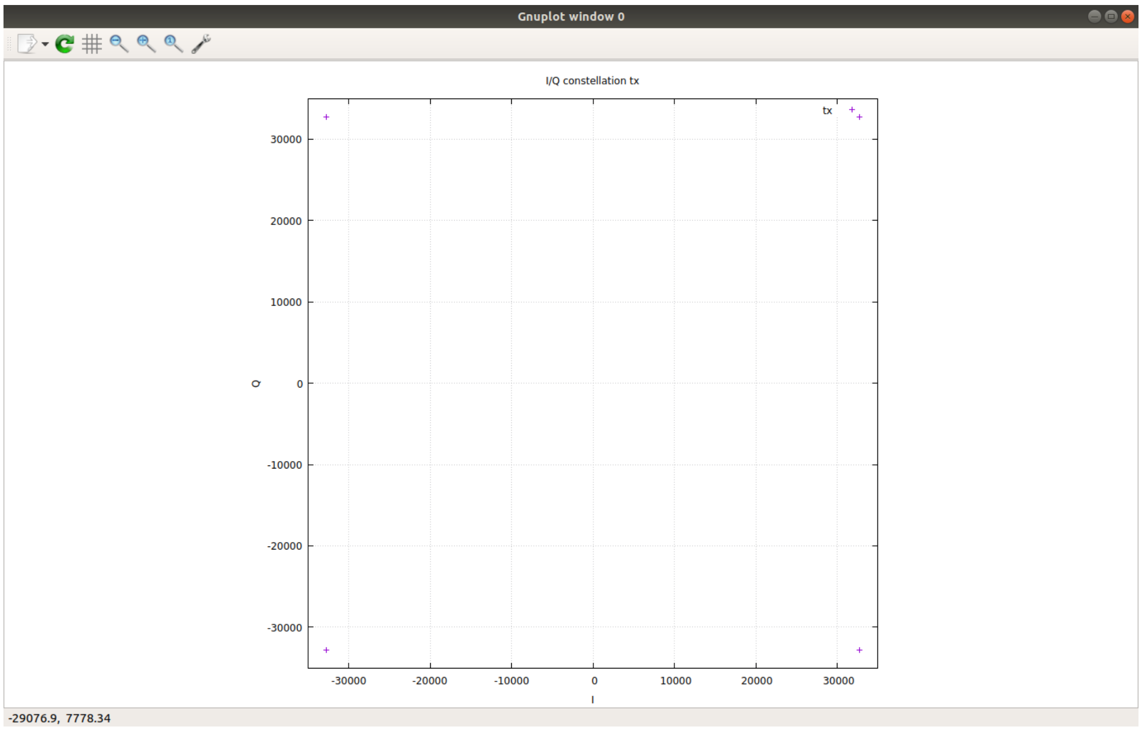


Figure 4.2: TX constellation data for QPSK modulation. Symbols are visible in the edge areas.

On the other hand, Figure 4.3 shows the received symbols, or rather one batch of received symbols. It is clearly visible that the symbols experienced additional noise. Since there is no carrier locking, the symbols rotate around the origin point, and a lack of clock synchronization means that the symbols get sampled at wrong intervals, therefore variate in amplitude. Both of these effects were visible during execution of the transmission.

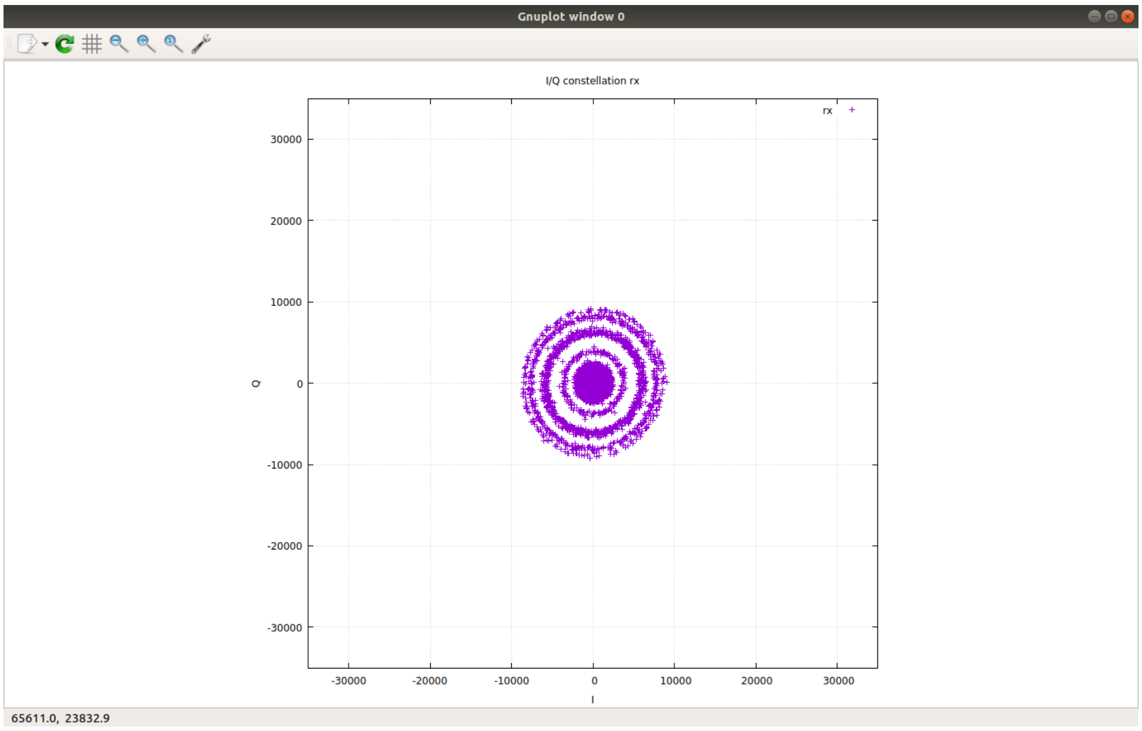


Figure 4.3: RX constellation data for QPSK modulation with clearly visible synchronization problems.

4.3.4 AGC Control

In order to change the AGC value of the RX device, the user just needs to input a number in the console **during** the communication and hit the *enter* key. This will call a custom API call in the **lms7_customAPI.cpp** file which will calculate the desired RSSI (**R**eceived **S**ignal **S**trength **I**ndication) with the formula

$$RSSI(x_{AGC}) = \frac{87330}{10^{\frac{3+x_{AGC}}{20}}} \tag{4.1}$$

where x_{AGC} is the numerical input from the user. The formula closely resembles the used formula in the LimeSuite driver GUI.

4.3.5 Stream Pause Controls

As already mentioned, once the user pauses a stream environment, he can modify different parameters. The different integer commands and how to use them were described in Table 4.2. As an example, choosing 1 continues the stream, a input of 2 quits the stream and returns to the main loop and 5 lets the user change the constellation, i.e. the modulation, method. Listing 4.12 shows how to change the constellation to QPSK.

```
...
Channel: 0                  TX |          RX
Active:                      0 |           0
Dropped Packets:             0 |           0
```

```
FIFO Filled Count:       2720 |      68000
FIFO Size:              87040 |      87040
Link Rate:           0.000000 |   0.000000
Overruns:                   0 |          0
Timestamp:               3113 |    3098250
Underruns:                  0 |          0

Channel: 1                 TX |         RX
Active:                     0 |          0
Dropped Packets:            0 |          0
FIFO Filled Count:          0 |      17680
FIFO Size:              87040 |      87040
Link Rate:           0.000000 |   0.000000
Overruns:                   1 |          0
Timestamp:               3113 |    3098250
Underruns:                  0 |          0

Halt stream confirmed. Ready for integer commands.
paused=>5
paused=>i5=>2
paused=>
```

Listing 4.12: Stream pause menu, changing the constellation to QPSK.

As a further example, the next Listing shows how to change the sampling rate to *2MHz* and the oversampling factor to 4.

```
paused=>22
paused=>f22=>2.0
paused=>i22=>4
...
paused=>2
```

Listing 4.13: Change the sampling rate to 2MHz and oversampling to 4.

All other commands work very similar and are described in the mentioned Table. Only one is really different, this being the SPI mode. This mode can be activated with the integer 200, after which the user can choose between reading a register (0), writing a register (1), find a specific register by a string (3) or showing all available registers in the LimeSuite drivers *LMS7parameterList* vector, see also the programming guide[27]. Press *c* to exit at any time and return to normal pause mode.

For example, Listing 4.14 shows how to print out the different SPI registers and how to search for a specific one, in this case all GFIR related registers. The first value shows the address, next are the registers MSB and LSB, the default value, the name of the register and an optional tooltip.

```
paused=>200
Entering SPI Mode, careful from now on. Write "c" to quit at all time.
paused=>SPI=>rd(0)/wr(1)/help(2)/find(3)=>2
0x0020 / 15:15/  1; LRST_TX_B, Resets all the logic registers to the
    default state for Tx MIMO channel B
```

```
0x0020 / 14:14/  1; MRST_TX_B, Resets all the configuration memory to
    the default state for Tx MIMO channel B
...
0x0641 /  6: 0/ 32; RSSIDC_DCO1, Value of RSSI offset DAC1
0x040c /  8: 8/  0; DCLOOP_STOP, Stops RxDC tracking loop
paused=>SPI=>rd(0)/wr(1)/help(2)/find(3)=>3GFIR
0x0205 / 10: 8/  0; GFIR1_L_TXTSP, Parameter l of GFIR1 (l =
    roundUp(CoeffN/5)-1). Unsigned integer
...
0x040c /  5: 5/  1; GFIR3_BYP_RXTSP, GFIR3 bypass
0x040c /  4: 4/  1; GFIR2_BYP_RXTSP, GFIR2 bypass
0x040c /  3: 3/  1; GFIR1_BYP_RXTSP, GFIR1 bypass
paused=>SPI=>rd(0)/wr(1)/help(2)/find(3)=>
```

Listing 4.14: Starting SPI mode, printout all SPI registers and search for a specific one.

This next Listing shows how a simple read command first queries the name of the register, if the user wants RX or TX device, and then returns the value plus address. In this case its the desired AGC value in the RX path. A write command for the same register calls a readout of the register first. After setting the register, another readout is performed to check if the value was set correctly. However, the user can also check this with another read command.

```
paused=>SPI=>rd(0)/wr(1)/help(2)/find(3)=>0
paused=>SPI=>rd=>Name=>AGC_ADESIRED_RXTSP
paused=>SPI=>rd=>rx(0)/tx(1)=>0
AGC_ADESIRED_RXTSP(0x0409): 0x00/0
paused=>SPI=>rd(0)/wr(1)/help(2)/find(3)=>1
paused=>SPI=>wr=>Name=>AGC_ADESIRED_RXTSP
paused=>SPI=>wr=>rx(0)/tx(1)=>0
AGC_ADESIRED_RXTSP(0x0409): 0x00/0
paused=>SPI=>wr=>Value=>1
AGC_ADESIRED_RXTSP(0x0409): 0x01/1
paused=>SPI=>rd(0)/wr(1)/help(2)/find(3)=>0
paused=>SPI=>rd=>Name=>AGC_ADESIRED_RXTSP
paused=>SPI=>rd=>rx(0)/tx(1)=>0
AGC_ADESIRED_RXTSP(0x0409): 0x01/1
```

Listing 4.15: Read and write operations on single SPI register.

4.4 Conclusion

The goals described in section 4.1 were all met. However, further development of this tool was not possible for one reason: Since it was basically a single threaded user program, it scraped the limits of the hosts computational power. Of course there were three additional threads running during stream (the pause thread and the two LimeSuite driver threads for both devices), but these were not prioritized by the scheduler. Therefore, as soon as there was any form of additional data processing, for example trying to synchronize the received data as seen in section 4.3.5, the threads for handling the devices and API calls were neglected. This meant that data did not pass through FPGA and FPRF fast enough, FIFO (**F**irst **I**n, **F**irst **O**ut) buffers were filled and packets were dropped.

One solution to this problem would be to use a combination of *fork*, *exec* and *pthread_setschedparam* system calls in *unistd.h C/C++* library to create a multi threaded user program that shares data, for example via the *pipe* command. However, this would still not guarantee a stable communication and would also exceed the scope of this master thesis.

Therefore the decision was made to explore other possibilities, which lead to the conclusion to use GNU Radio as further development environment.

Chapter 5

GNU Radio Framework

This chapter will focus on an introduction to GNU Radio. It will cover what GNU Radio is about, an introduction to its graphical tool and what additional processing blocks were developed in the course of this thesis.

Keep in mind that the GNU Radio release version is important if the intention is to further develop this thesis. Refer to subsection 3.6.2 and appendix chapter **Host Packages** for details. For a detailed explanation on how to develop a custom block to be added to the environment, refer to chapter **Adding Custom Blocks** in the appendix.

5.1 About GNU Radio

GNU Radio is an open source development toolkit, specialized in digital signal processing and incorporating SDR hardware via host interfaces. Using hardware is optional though, GNU Radio can be used as a pure signal processing simulation. It's an ideal development environment to quickly set up basic transmissions between radios or send/receive with a single SDR. Furthermore, it can be used to develop signal processing chains that will get integrated in a DSP in later stages of projects.

This toolkit offers a huge variety of so-called *blocks*, which process, absorb or create digital data. It can also be expanded by programming custom blocks, for which it offers tools to create bare-bone file structures to incorporate these blocks into the environment. GNU Radio installations come with filters, encoders and decoders, modulators and demodulators, synchronizers and many other useful blocks. Furthermore, it has blocks to illustrate data in various domains (time, frequency, constellation, ...). All blocks get connected via so-called *ports*, for which there are different data types available. These ports usually pass the data types as data streams. Included to the environment is the GNU Radio scheduler, which manages what blocks process data at which time, and also how the data is passed from block to block.

The GNU Radio application as a whole incorporates different programming languages. Many of the blocks are written in *Python*, especially when they perform rather simple tasks. For more complex implementations, blocks can also be written in *C++*, which gives the opportunity to develop real time, high throughput, performance critical digital signaling processes. In order to connect these languages, GNU Radio uses *SWIG*, an open source software wrapper, merging a variety of programming and scripting languages.

While GNU Radio can be built and added to the host system from scratch (for example from Github) and since it depends on various different open source software, it is recommended to instead install it directly using Debians advanced package tool (*apt*). See also the host setup the appendix chapter **Host Packages**, which includes the *gnuradio* package.

GNU Radio is licensed under the GNU GPL (**G**eneral **P**ublic **L**icense) version 3 or later. All of the code is copyright of the Free Software Foundation.

There are several hardware platforms that are supported. Platform developers supply their own implementations freely to be added to the GNU Radio environment. For example, GNU Radio contains the mentioned UHD driver in chapter 3, which makes development with the USRP hardware platform available, while Lime Microsystems supplies its own implementation on Github for their various boards[28].

Another feature of GNU Radio are so-called *tags*. These are basically markers that get added to specific points of a data stream to simplify searching for these location. It makes later blocks in a processing chain more efficient, instead of searching for a specific pattern, blocks only need the location of specific tags.

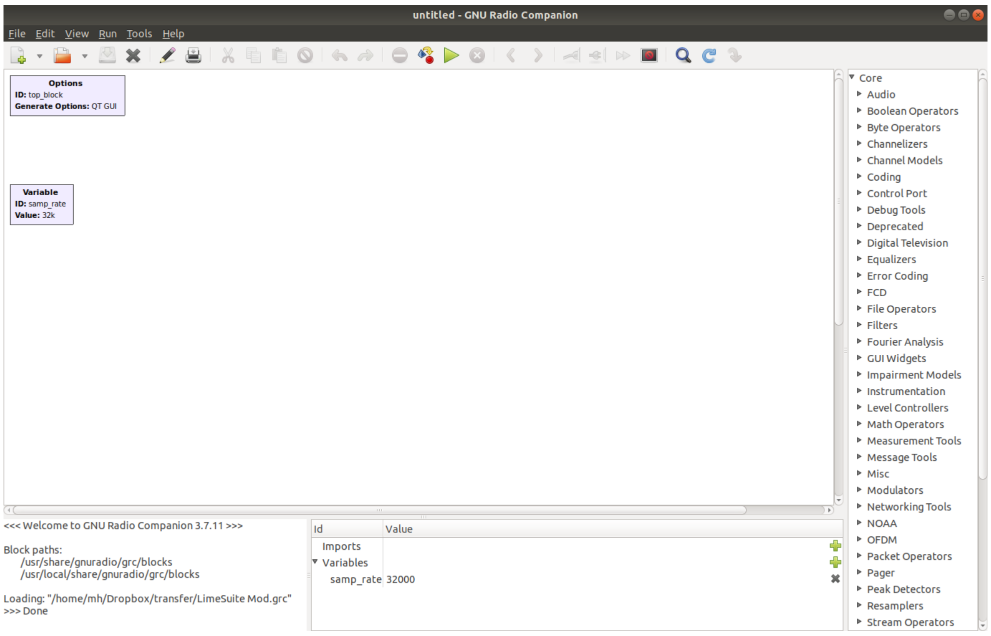
While GNU Radio comes with a variety of tools that can be used in console, there is also a graphical tool available to create signal processing flow graphs. This tool is called GNU Radio Companion.

5.2 GNU Radio Companion

GNU Radio Companion (abbreviated as GRC) is a graphical tool for easy and fast development of GNU Radio flow graphs. Figure 5.1 shows an empty GRC project. Blocks can be selected on the right and dragged into the flow graph. When executing a flow graph (assuming it has no errors), GRC creates a *Python* script that gets executed. This script runs in an own process on the host. The program can be executed in console with the command

```
gnuradio-companion
```

There are different types of available blocks: While variables are rectangular blocks without connections (i.e. ports), most other blocks actually either generate data streams (source block) through ports, consume data (sink block) through ports or get data, process it and then output them further (e.g. sync blocks). The color of the port symbolizes which kind of data format gets passed. Figure 5.2 shows these different data types from the help section of GRC, ranging from bit or byte streams to integer or float streams and also complex, i.e. constellation data, streams. When connecting blocks together, both ports must have the same data type. However, every block can have multiple ports with different data types.



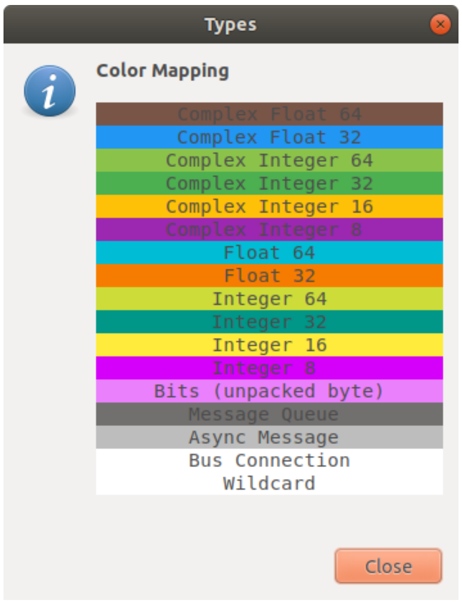
Options
ID: top_block
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

```
<<< Welcome to GNU Radio Companion 3.7.11 >>>

Block paths:
    /usr/share/gnuradio/grc/blocks
    /usr/local/share/gnuradio/grc/blocks

Loading: "/home/mh/Dropbox/transfer/LimeSuite Mod.grc"
>>> Done
```

Id	Value
Imports	
Variables	
samp_rate	32000

Figure 5.1: GNU Radio Companion GUI with empty flow graph.



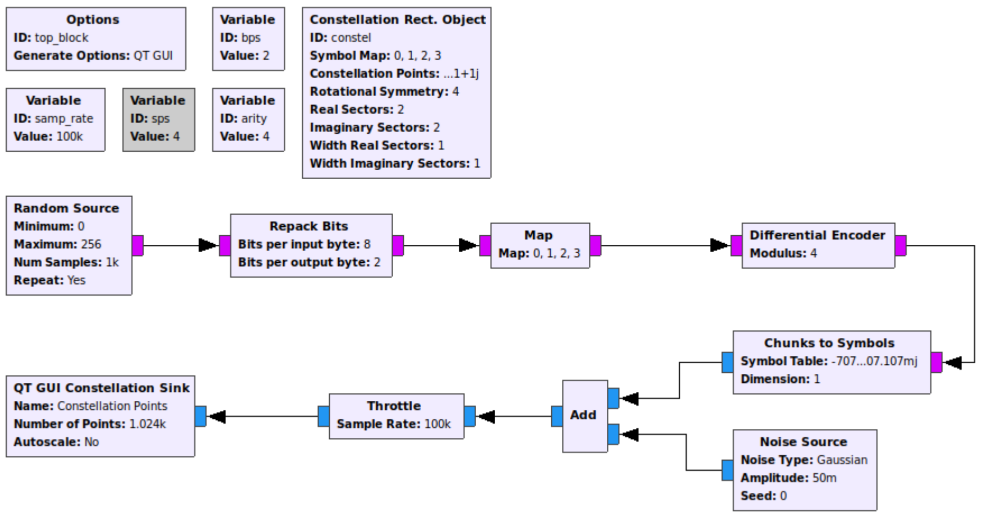
Color Mapping

Complex Float 64
Complex Float 32
Complex Integer 64
Complex Integer 32
Complex Integer 16
Complex Integer 8
Float 64
Float 32
Integer 64
Integer 32
Integer 16
Integer 8
Bits (unpacked byte)
Message Queue
Async Message
Bus Connection
Wildcard

Figure 5.2: The different data types in GRC, as illustrated in the help section.

When using a GRC flow graph without any hardware blocks, it is imperative to use a *throttle* block. This block reduces the amount of data flowing through. If this is not done, the host might use all its computational power to execute the flow graph, which could deteriorate or even damage the hardware. If an SDR is connected and part of the flow graph, the interface to the hardware limits the data speed, neglecting the need of a throttle block.

Figure 5.3 shows an example flow graph without hardware, modulating QPSK with additive noise. In the top left corner are blocks to define variables, where the *sbs* variable is deactivated, therefore colored in a darker gray. There are also *object* blocks like the *Constellation Rect. Object* which defines an object with multiple member variables. All other blocks have ports, first consisting of byte streams (from the random source until the input of the symbols mapper, i.e. the modulator) and finally complex (i.e. constellation) data. The random source repacks its stream into 2 bit pairs and modulates it, then AWGN gets added and finally visualized in a QT GUI sink. QT is a open source widget tool kit for creating GUIs.


Options
ID: top_block
Generate Options: QT GUI

Variable
ID: bps
Value: 2

Constellation Rect. Object
ID: constel
Symbol Map: 0, 1, 2, 3
Constellation Points: ...1+1j
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

Variable
ID: samp_rate
Value: 100k

Variable
ID: sps
Value: 4

Variable
ID: arity
Value: 4

Random Source
Minimum: 0
Maximum: 256
Num Samples: 1k
Repeat: Yes

Repack Bits
Bits per input byte: 8
Bits per output byte: 2

Map
Map: 0, 1, 2, 3

Differential Encoder
Modulus: 4

Chunks to Symbols
Symbol Table: -707...07.107mj
Dimension: 1

QT GUI Constellation Sink
Name: Constellation Points
Number of Points: 1.024k
Autoscale: No

Throttle
Sample Rate: 100k

Add

Noise Source
Noise Type: Gaussian
Amplitude: 50m
Seed: 0

Figure 5.3: A very basic flow graph which modulates a random source with additive noise.

The result of this flow graph is illustrated in Figure 5.4, or rather one snapshot of the data. There is clearly a QPSK modulation scheme visible, also the added Gaussian noise.

The next section will discuss how to add additional blocks to GRC by programming an example block.

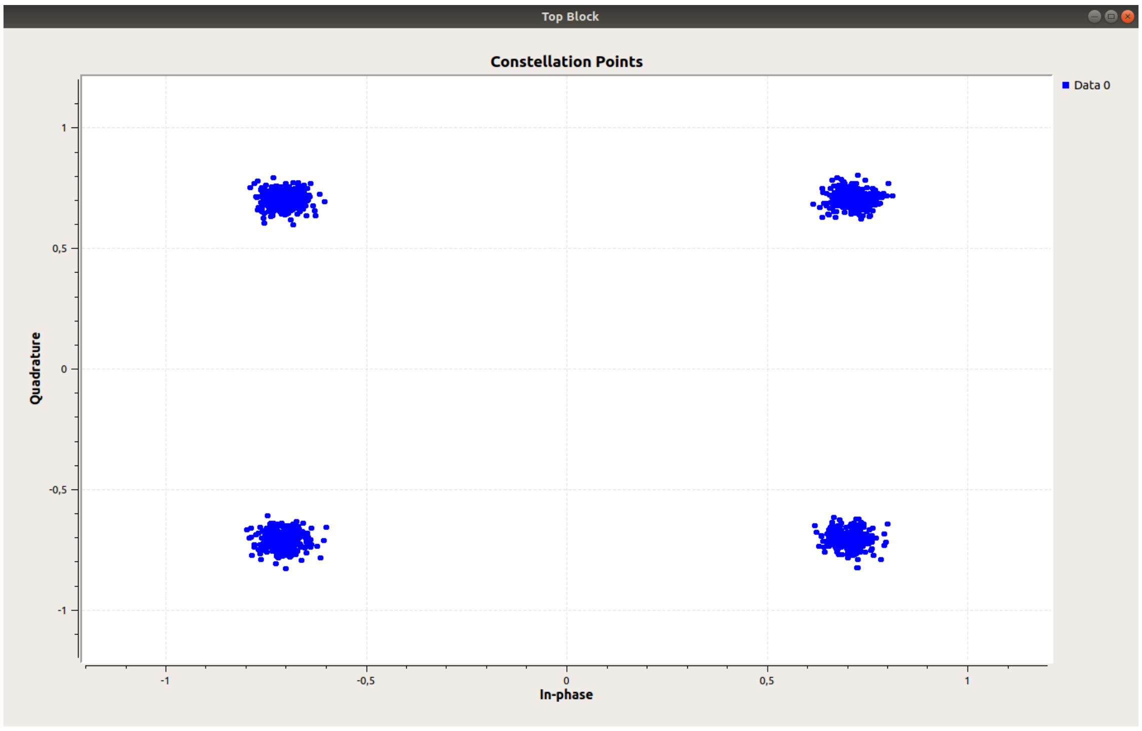


Figure 5.4: Graphical result of the noisy constellation points.

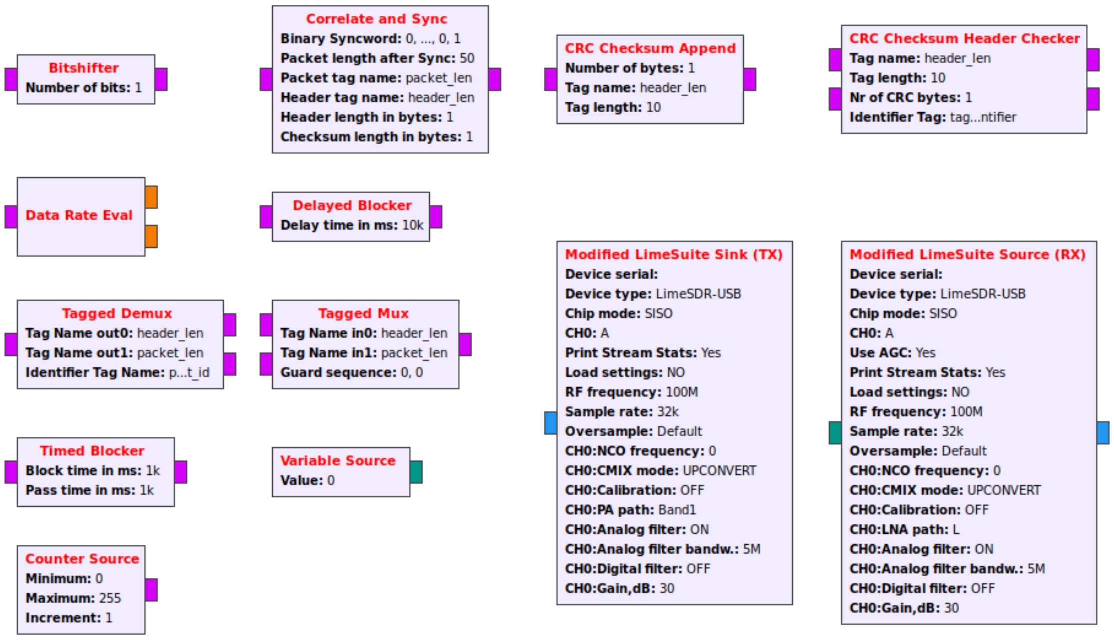
5.3 Additional Custom Blocks

In the course of this thesis, multiple GNU Radio blocks were programmed. Most of them were newly created, while two were adaptation of existing ones. All custom blocks are grouped in the *gr_limesdrmod* module and are illustrated in Figure 5.5. Note that even though they were developed, not all of them actually find use in the coming flow graphs. Some were just for testing or became deprecated during the process of the thesis.

Following is a list of the tasks and data processing that each of these blocks provide.

Bitshifter: This block takes a bit stream and shifts the whole stream by the number of bits specified. This is useful in text files: if the whole bit sequence is shifted, the displayed text from the ASCII Table is no longer correct. A bit shift can solve this issue.

CRC Checksum Append: This calculates a CRC for a defined length of bytes. First, the binary data for the checksum gets passed to the output (the length of this data is defined by the *Tag length* variable). The same data is then used to calculate the CRC checksum (length of this checksum can be between 1 and 4 bytes), which gets passed to the output as well with a *user defined tag*.



Bitshifter
Number of bits: 1

Correlate and Sync
Binary Syncword: 0, ..., 0, 1
Packet length after Sync: 50
Packet tag name: packet_len
Header tag name: header_len
Header length in bytes: 1
Checksum length in bytes: 1

CRC Checksum Append
Number of bytes: 1
Tag name: header_len
Tag length: 10

CRC Checksum Header Checker
Tag name: header_len
Tag length: 10
Nr of CRC bytes: 1
Identifier Tag: tag...ntifier

Data Rate Eval

Delayed Blocker
Delay time in ms: 10k

Tagged Demux
Tag Name out0: header_len
Tag Name out1: packet_len
Identifier Tag Name: p...t_id

Tagged Mux
Tag Name in0: header_len
Tag Name in1: packet_len
Guard sequence: 0, 0

Timed Blocker
Block time in ms: 1k
Pass time in ms: 1k

Variable Source
Value: 0

Counter Source
Minimum: 0
Maximum: 255
Increment: 1

Modified LimeSuite Sink (TX)
Device serial:
Device type: LimeSDR-USB
Chip mode: SISO
CH0: A
Print Stream Stats: Yes
Load settings: NO
RF frequency: 100M
Sample rate: 32k
Oversample: Default
CH0:NCO frequency: 0
CH0:CMIX mode: UPCONVERT
CH0:Calibration: OFF
CH0:PA path: Band1
CH0:Analog filter: ON
CH0:Analog filter bandw.: 5M
CH0:Digital filter: OFF
CH0:Gain,dB: 30

Modified LimeSuite Source (RX)
Device serial:
Device type: LimeSDR-USB
Chip mode: SISO
CH0: A
Use AGC: Yes
Print Stream Stats: Yes
Load settings: NO
RF frequency: 100M
Sample rate: 32k
Oversample: Default
CH0:NCO frequency: 0
CH0:CMIX mode: UPCONVERT
CH0:Calibration: OFF
CH0:LNA path: L
CH0:Analog filter: ON
CH0:Analog filter bandw.: 5M
CH0:Digital filter: OFF
CH0:Gain,dB: 30

Figure 5.5: Additionally programmed GRC blocks.

Tagged Mux: Even though there are native multiplexer in the GNU Radio environment, there are none that multiplex depending on tags. This multiplexer searches for the *in0 tag* on the input of *in0* and passes the amount of data (stored as the tags' value) to the output. Then it switches to *in1* input and the procedure repeats with the *in1 tag*. However, after the data from *in1* got passed to the output, an additional *guard sequence* gets added, which can be defined by the user. This guard sequence is optional. The block alternates this way between both input ports.

Correlate and Sync: In order to filter the incoming byte streams between useable and neglectable data, this block correlates the data with a defined bit sequence. The length of this correlation sequence should be a multiple of 8. After an initial correlation with the first data on the input, if the *sync word* was not found, the block shifts the input data by one bit and correlates again. If no sync word was found, the entire input data set will be discarded. Once a correlation was successful, the work function first adds a *header tag* to the data to symbolize the start of the frame. Then the *sync word*, and an optional *CRC data length*, get passed to the output. Additionally, the block adds a *packet tag* afterwards and passes the amount of specified *packet length* in bytes to the output, therefore completing the frame data.

Tagged Demux: This block is the inverted version of the previously discussed *Tagged Mux*, to give later blocks the opportunity to process header and packet independently. It imports all tags on the input port and checks if the first one matches the defined *header tag*. If it matches, the header data, the length of which is stored as the tags value, gets passed to output *out0*, together with an *identifier tag* to numerate the frame. If the tag name does not correlate, the next tag is checked and the data up to this point gets discarded. Once the first header tag was found, the block searches for the *packet tag*, correlates the same as with the header tag, outputs the data length to *out1* and adds the same numerated *identifier tag* to this output as with the previous one. Following blocks just need to check the identifier tag ID to find out which header belongs to which data packet.

CRC Checksum Header Checker: There are two inputs to this block. The first one, *in0*, expects enumerated header data, while the second *in1* expects the associated packet data, tagged with the same enumerated ID. Therefore it is the optimal block to come after the previously described *Tagged Demux*. It reads the header data at *in0* and checks if the provided CRC matches the sync word, i.e. the header data. If the checksum correlates, the packet data is passed to output port *out0*. If it does not match, it is passed to port *out1*, therefore separating correct data from false data packets.

Data Rate Eval: This is a very specific block. It consumes all the input data on *in0* and correlates the data with 4 specific characters: 'c', 'a', 'f' and 'e'. If the first byte matches one of these ASCII characters, it is considered as a correct byte. If it does not correlate with any of this characters, then the MSB of this byte is considered as a bit error, the whole byte is bit shifted by 1 bit and the test repeats by including the MSB of the next byte on the input port. Therefore the block can then calculate how much bits were correct. Every two seconds it outputs its *data rate* from the input onto *out0* port as an floating point variable, while simultaneously writing the calculated BER onto output port *out1*. Additionally, it also outputs both of these values into the console every two seconds. Therefore, both output ports can also be connected to a *null* sink.
The reason for this specific character test was that the given source test data to transmit during development was a repeating sequence of the word "cafe".

Delayed Blocker: Since this is a very simple block, it is one of the two that have been written in *Python*. The user defines a time in milliseconds and the block only passes the input to the output port once this time has passed after executing the flow graph. Before the time is passed, it simply stops the data, not discarding it.

Timed Blocker: The second block to be written in *Python*. First it blocks all data, similar to the *Delayed Blocker*, for the given *block time* in milliseconds. Then it passes its input data to the output port for the defined *pass time* in milliseconds. Afterwards it repeats this procedure.

Modified LimeSuite Sink (TX): This is a clone of the open source *gr-limesdr* GNU Radio block on Gihtub[28] provided by Lime Microsystems. For documentation purpose, refer to the given website. The only change is that the block has an additional parameter *Print Stream Stats*. If this is enabled, the device periodically prints the devices *lms_stream_status_t* information to the console. This struct contains information about the data rate in MB/s, buffer sizes and packet drops. Beside of this changes and incorporating the member variable into the *general_work* function, nothing was changed.

Variable Source: A tutorial on how to program and add this block to the GNU Radio environment is documented in the appendix chapter **Adding Custom Blocks**. Whatever value is set in the variable will be output to port *out0*. If the value gets changed during the execution of the flow graph, the callback function gets executed and the value on the output port is updated.

Counter Source: This block outputs an incremental number between its defined range by the given increment value. This is a very simple block, since it outputs a byte stream, the maximum possible value is 255. Use case for this block is to create and incremental packet ID, which will again start at its minimum once it overflows.

Modified LimeSuite Source (RX): Similar to the TX variant, this is also a clone of the source code from Github[28] and has the same new parameter *Print Stream Stats* to enable printout of stream statistics into console for the RX device. Again refer to the documentation on the website. One additional change though, is that the block has an integer input. The value on *in0* defines the AGC value of the RX. If the value is zero, AGC is deactivated (the implementation has an additional callback function called *set_agc(int agc)*). If the value is not zero the device calls the same custom API function as described in section 4.3.4 with formula 4.1. This AGC input is controlled with the previous described variable source block.

Chapter 6

GNU Radio Flow Graphs

This chapter will give an overview of the developed GNU Radio flow graphs. The first section will illustrate the system as a whole, remaining sections focus on the three parts that make up the the full system.

Once all parts have been discussed and their blocks linked to the theoretical chapter of this thesis, chapter 7 will explain how to compare the different parts with software, and chapter 8 will contain the actual evaluation using a measurement setup.

6.1 Full System Concept

Figure 6.1 shows the full system, with obvious similarities to the DCS in section 2.1. The advantage of a DCS is that the different blocks can work independently of each other. Every block takes the input that gets passed to it, processes it accordingly and sends it to the next block. This constraints possible errors to individual parts and avoids problems with CPU limitation.

In order to ease the workload of the host PC, the whole system will be divided into three independent flow graphs. Each one of these can then be evaluated separately.

1. Data preparation and modulation.
2. Transmit and receive.
3. Data synchronization, demodulation and correlation.

Part 2 will contain hardware blocks that actually send and receive data via a channel respectively, while the other parts perform only data processing on the host. In order to compare different stages of this system with each other, a standardized method for evaluation is required. Since there is an actual wireless transmission in place, comparing in the frequency domain would be preferred. It's possible to save data into a sink at the end of one flow graph and use it as input of the next one. Therefore, a tool to translate data from the complex symbol plane to the frequency domain was programmed using MATLAB. This will be discussed in chapter 7.

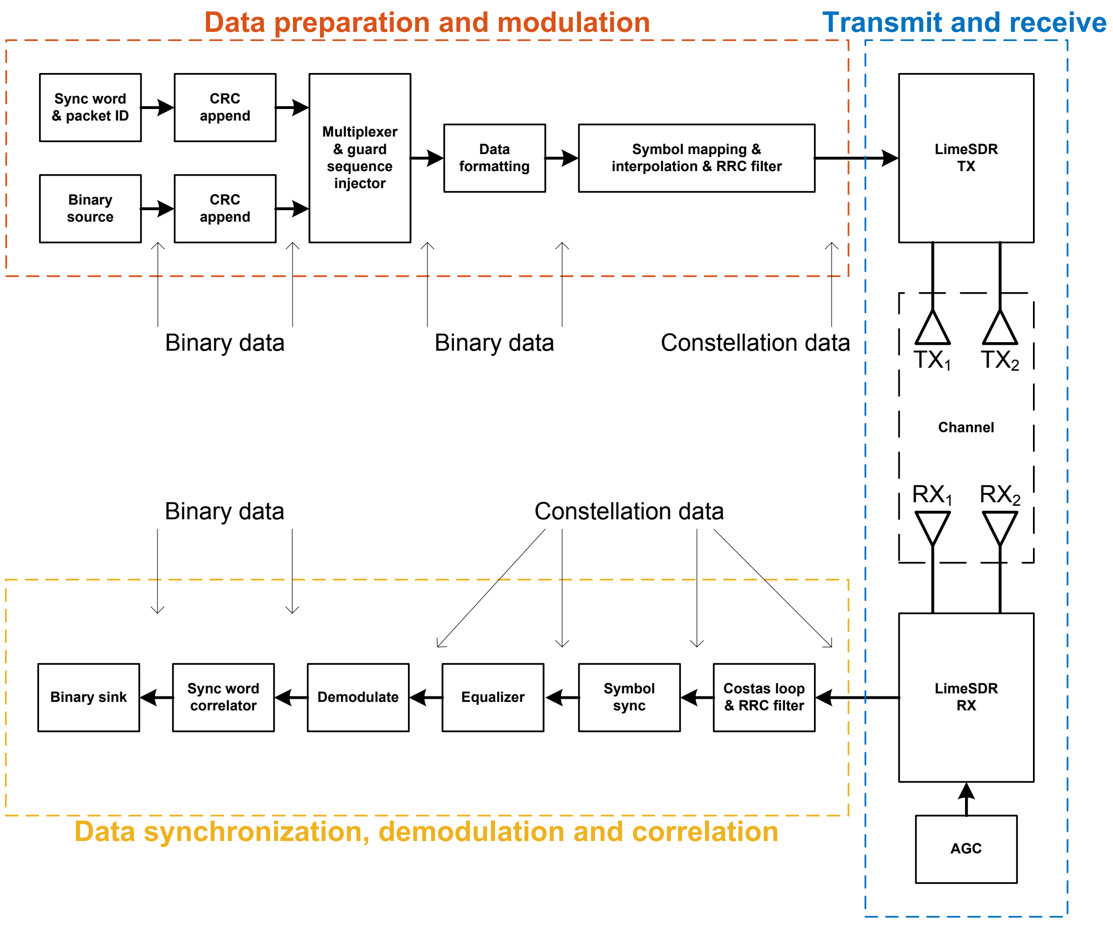


Figure 6.1: Full system concept of GNU Radio flow graphs.

Note that all parts execute consecutively on one single host. If a higher distance between TX and RX were required, flow graph 2 would need to be divided.

6.2 Data Preparation and Modulation

This part focuses on preparing the source data to be sent over the physical channel. Figure 6.2 illustrates the concept of this flow graph.

It starts with a defined synchronization sequence (frame synchronization), including a simple packet ID mechanism and a trailed CRC checksum to recognize bit errors in the header. This gets multiplexed with the binary information source (chopped into defined packet lengths), which also gets a trailed CRC checksum. The multiplexer can add an optional guard sequence. Figure 6.3 depicts the structure of one single frame. Red is the header, yellow the packet and blue the optional guard space.

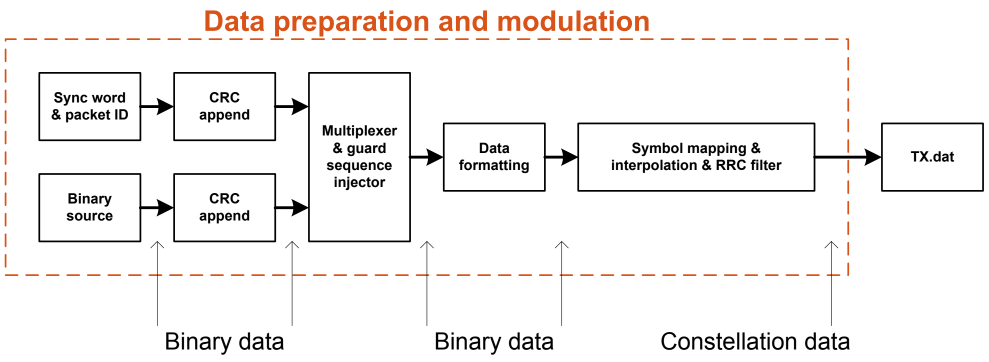


Figure 6.2: Concept for data preparation and modulation.

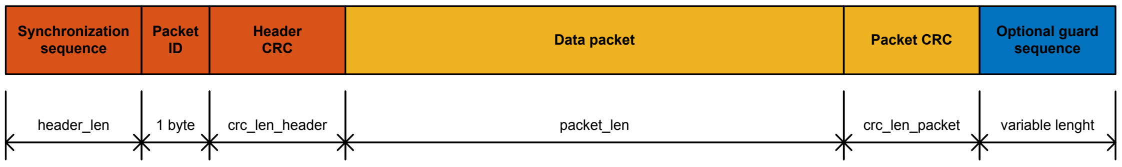


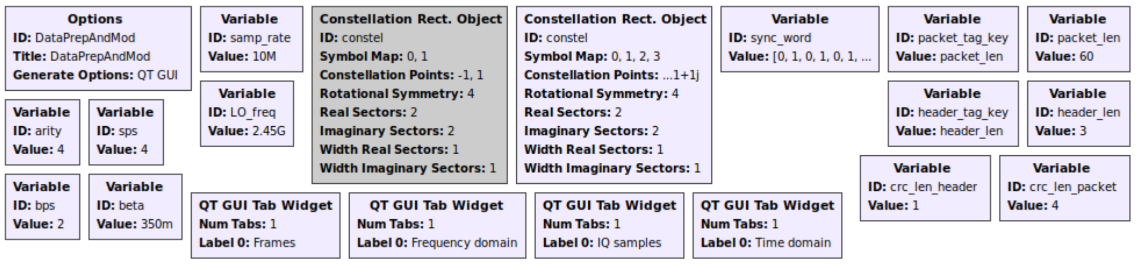
Figure 6.3: Frame structure concept.

Depending on the modulation scheme, the binary data stream is then processed into correct symbol lengths and modulated to complex constellation points. This modulation is performed in one single block: First, the binary symbols are mapped to the constellation. Then the signal gets upsampled. Thereafter, to satisfy the equivalent channel equation, the samples are then processed by a RRC filter.

In the last step of this flow graph, the results are visualized and saved into a *.dat* file. This file can then be used to evaluate the result and as input for the next flow graph.

6.2.1 Variables

Figure 6.4 shows the variable blocks of this flow graph.



Options
ID: DataPrepAndMod
Title: DataPrepAndMod
Generate Options: QT GUI

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: bps
Value: 2

Variable
ID: beta
Value: 350m

Variable
ID: samp_rate
Value: 10M

Variable
ID: LO_freq
Value: 2.45G

Constellation Rect. Object
ID: constel
Symbol Map: 0, 1
Constellation Points: -1, 1
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

Constellation Rect. Object
ID: constel
Symbol Map: 0, 1, 2, 3
Constellation Points: ...1+1j
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

Variable
ID: sync_word
Value: [0, 1, 0, 1, 0, 1, ...

Variable
ID: packet_tag_key
Value: packet_len

Variable
ID: packet_len
Value: 60

Variable
ID: header_tag_key
Value: header_len

Variable
ID: header_len
Value: 3

Variable
ID: crc_len_header
Value: 1

Variable
ID: crc_len_packet
Value: 4

QT GUI Tab Widget
Num Tabs: 1
Label 0: Frames

QT GUI Tab Widget
Num Tabs: 1
Label 0: Frequency domain

QT GUI Tab Widget
Num Tabs: 1
Label 0: IQ samples

QT GUI Tab Widget
Num Tabs: 1
Label 0: Time domain

Figure 6.4: Variables needed for data preparation and modulation flow graph.

The first block that is always present in flow graphs is the **Options** block. It defines the project name and how the *Python* file for execution will be named. Other possible options are the size of the flow graph in the GUI and what kind of software will be used to illustrate the data. In this case, QT was chosen, as described in section 5.2.

Regarding the frequency and bandwidth values for modulation, the variable **LO_freq** defines the carrier frequency to which the baseband signal is translated to and **samp_rate** defines the sampling rate. For now this sampling rate refers to how much symbols will be processed per second by the different blocks in this flow graph. These variables will be more meaningful once hardware blocks are present.

There are two possible modulation schemes in this flow graph: BPSK and QPSK. The **Constellation Rect. Object** defines the parameters of these schemes. One of these objects must be deactivated (i.e. grayed out), the other modulation scheme will then be active. Since it is a *C++* object, other blocks can access its member variables by first giving the object name (in this case *constel*) followed by a dot and the name of the member variable (e.g. *constel.points()* for the location of the symbols in the complex plane). The parameters of the modulation scheme can either be set manually or by using GNU Radios' digital objects and functions. For example, the symbol locations in the complex plane of BPSK can be set with *digital.psk_2()[0]* and the symbol mapping with *digital.psk_2()[1]*, while for QPSK this can be done with the *psk_4()* member function. Other parameters refer to symmetry in the complex plane, these were left as defaults.

Variables that depend on the constellation scheme are: Bit per symbol (**bps**) with the command *constel.bits_per_symbol()*, samples per symbol (**sps**) which defines the upsample value and the order of modulation (**arity**) by *constel.arity()*. Additionally, **beta** stores the excess bandwidth of the RRC filter.

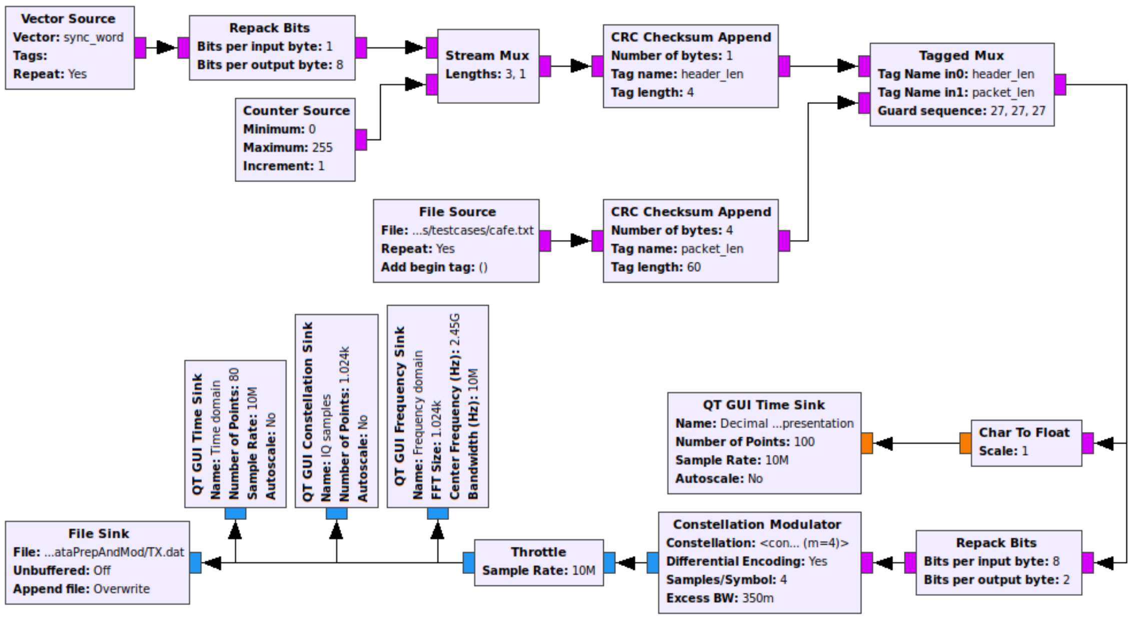
The variable **sync_word** defines the frame synchronization sequence that gets multiplexed with the packed ID and then with the data packet. This synchronization sequence symbolizes the start of a packet frame and is defined by a bit sequence. Because of the structure of the flow graph and the implementation of later blocks, this sequence must have a length of modulo 8.

QT GUI Tab Widgets are for positioning graphs on the results screen that gets presented to the user.

The final variables define data lengths and tag names. As for the frame header, the variable **header_tag_key** defines the name of the tag and **header_len** the length of the header in bytes. Furthermore, **crc_len_header** defines the length of the CRC checksum in bytes that get appended to the synchronization sequence plus packet ID. On the other hand, **packet_tag_key** defines the name of the tag for the packet data, **packet_len** sets the packet length in bytes and **crc_len_packet** defines the number of bytes for the CRC checksum of the data packet.

6.2.2 Data Processing

This subsection will explain the blocks used in the flow graph illustrated in the following Figure 6.5. It will also discuss how they are configured. Note that this is only one possible configuration of the parameters, different simulation configurations and their results are discussed in chapter 8.



Vector Source
Vector: sync_word
Tags:
Repeat: Yes

Repack Bits
Bits per input byte: 1
Bits per output byte: 8

Counter Source
Minimum: 0
Maximum: 255
Increment: 1

Stream Mux
Lengths: 3, 1

CRC Checksum Append
Number of bytes: 1
Tag name: header_len
Tag length: 4

Tagged Mux
Tag Name in0: header_len
Tag Name in1: packet_len
Guard sequence: 27, 27, 27

File Source
File: ...s/testcases/cafe.txt
Repeat: Yes
Add begin tag: ()

CRC Checksum Append
Number of bytes: 4
Tag name: packet_len
Tag length: 60

QT GUI Time Sink
Name: Time domain
Number of Points: 80
Sample Rate: 10M
Autoscale: No

QT GUI Constellation Sink
Name: IQ samples
Number of Points: 1.024k
Autoscale: No

QT GUI Frequency Sink
Name: Frequency domain
FFT Size: 1.024k
Center Frequency (Hz): 2.45G
Bandwidth (Hz): 10M

QT GUI Time Sink
Name: Decimal ...presentation
Number of Points: 100
Sample Rate: 10M
Autoscale: No

Char To Float
Scale: 1

File Sink
File: ...ataPrepAndMod/TX.dat
Unbuffered: Off
Append file: Overwrite

Throttle
Sample Rate: 10M

Constellation Modulator
Constellation: <con... (m=4)>
Differential Encoding: Yes
Samples/Symbol: 4
Excess BW: 350m

Repack Bits
Bits per input byte: 8
Bits per output byte: 2

Figure 6.5: Data Processing for data preparation and modulation flow graph.

The **Vector Source** in the top left corner of the flow graph outputs the frame synchronization sequence. As for the parameters, it only needs the previously discussed **sync_word**, which is already defined in vector form. Furthermore, the *Repeat* option is set to true to output the vector endlessly. Since the vector is defined as a bit sequence, the following **Repack Bits** block takes all bits from the source and stuffs them together into byte lengths. Otherwise, only the LSB would contain one bit of the synchronization word in the bit stream and all other bits would be 0. A simple packet ID gets created by the **Counter Source** block and multiplexed with this sequence. Finally, this combined header passes through a **CRC Checksum Append** block. This processes the header by adding one byte of CRC checksum.

For representation purpose, this flow graph has a **File Source** as data source. There will be multiple test sources in chapter 8 to emulate different test cases, in which case the source block might be replaced. This block opens the file on the hard drive and outputs the data. Same as with the vector source, the output is set to repeat. A **CRC Checksum Append** bundles the incoming data into fixed lengths, adds the packet tag and appends a CRC checksum.

Both the header data and the data package will then be multiplexed using the **Tagged Mux** block. As described in section 5.3, this block uses GNU Radios' tags to multiplex data alternately, with the addition of an optional guard sequence. In the depicted Figure, the guard sequence consists of a repeating sequence of the number *27*. This decimal number gets added as a byte sequence after the data packet.

The **Char To Float** block converts the binary data stream to float in order to display the frame data in decimal format. This representation will be further discussed in subsection 8.1.2.

Following is the modulation process. First of all, the binary data stream must be chopped into the correct symbol length, depending on the modulation scheme, i.e. the **bps** variable. In the illustrated case, QPSK is active, which has 4 possible symbols, which according to equation 2.4 results in 2 bits. This operation is performed by the **Repack Bits** block.

The **Constellation Modulator** block performs three actions to modulate the data:
First it maps the incoming binary symbols to the constellation points in the complex plane. For this, it needs to know the constellation scheme, which is stored in the *Constellation* parameter. A user can also decide if the scheme should have *Differential Encoding* or not. With differential coding, the symbol depends not only on the current data input, but also on the location of the previous symbol.
Second, it upsamples the data, which shrinks the bandwidth. The resample value is defined by the parameter *Samples/Symbol*.
And finally third, to complete the interpolation process and satisfy the Nyquist criteria, the signal gets RRC filtered. The parameter *Excess BW* defines the roll-off factor of the filter. For details to Nyquist filters, see section 2.3.3. Since the RX side of the system deploys a RRC filter as well, the equivalent channel equation 2.22 is satisfied and there should be a ISI free transmission.
While previous blocks worked with binary data streams, this is the first block that outputs constellation data.

Since there is no hardware block present in the flow graph, a **Throttle** block is mandatory. This block confines the rate at which data passes through the system. If none such blocks were used, the flow graph would utilize the whole computational power of the host PC, potentially damaging its hardware by overloading the CPU or the interfaces to the hard drives.

Finally, the symbols pass into different GUIs to illustrate the data for the user. Furthermore, the data gets saved into a *.dat* file for the next flow graph.

Figure 6.6 shows the result of this flow graph with the illustrated parameter configuration. Clearly visible are the effects of the RRC filter. The interpolation process adds additional symbols in the complex plane. These are the transitions between one symbol and the next.

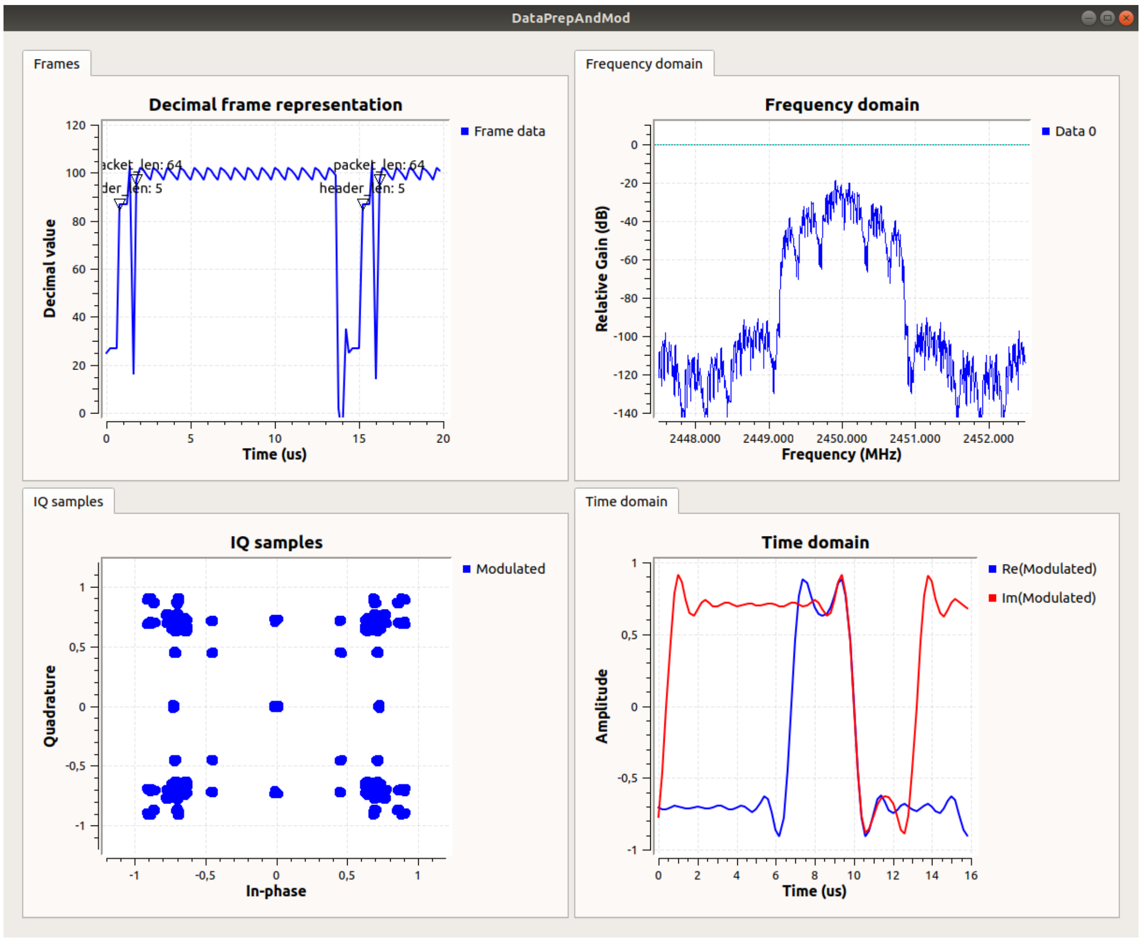


Figure 6.6: Results of the data processing in frequency, complex and time domain.

6.3 Transmit and Receive

The transmitting and receiving flow graph incorporates actual hardware blocks, Figure 6.7 illustrates this concept.

While the *Transmit* part reads a *.dat* file and sends the symbols into the channel using an antenna, the *Receive* part outputs the received signal constellation data into a different *.dat* file. The input of the *Receive* block is used to control the AGC of the device.

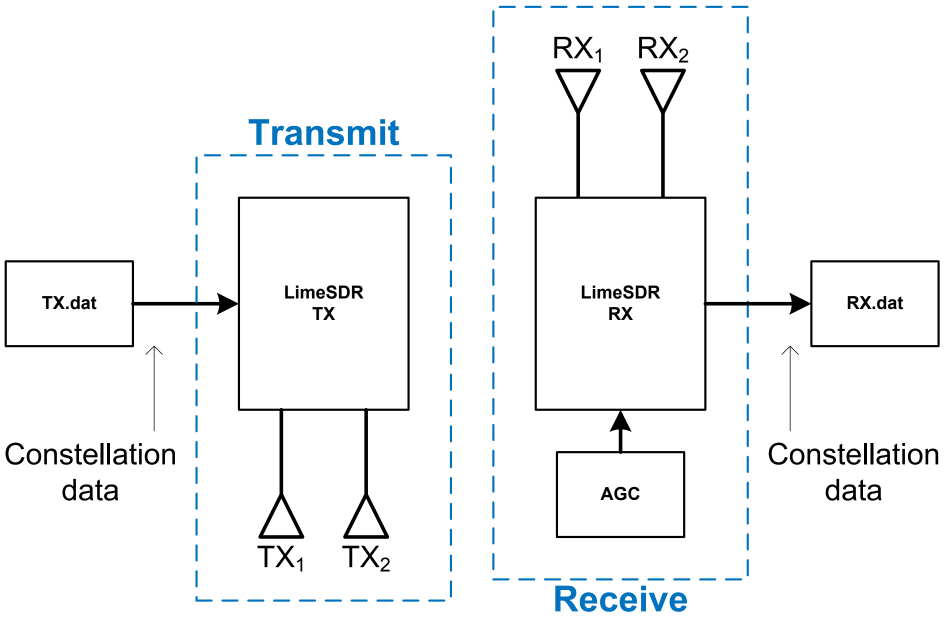
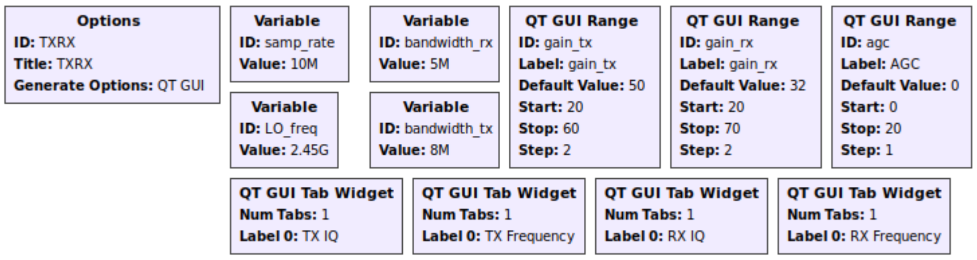


Figure 6.7: Concept for transmit and receive.

6.3.1 Variables

The first variable is again the **Options** block. Previously discussed **samp_rate** and **LO_freq** actually determine physical properties in this case: One defines the sample rate of the ADCs and DACs while the other characterizes the carrier frequency. Both devices need a gain and a frequency value to be set for the amplifier and filter chains, discussed in subsections 3.4.1 and 3.4.2. These gain values are stored in a graphical slider block to change values on the fly during flow graph execution. Finally, the RX hardware block also needs a variable input to determine the AGC value. This is again realized as a graphical slider block.



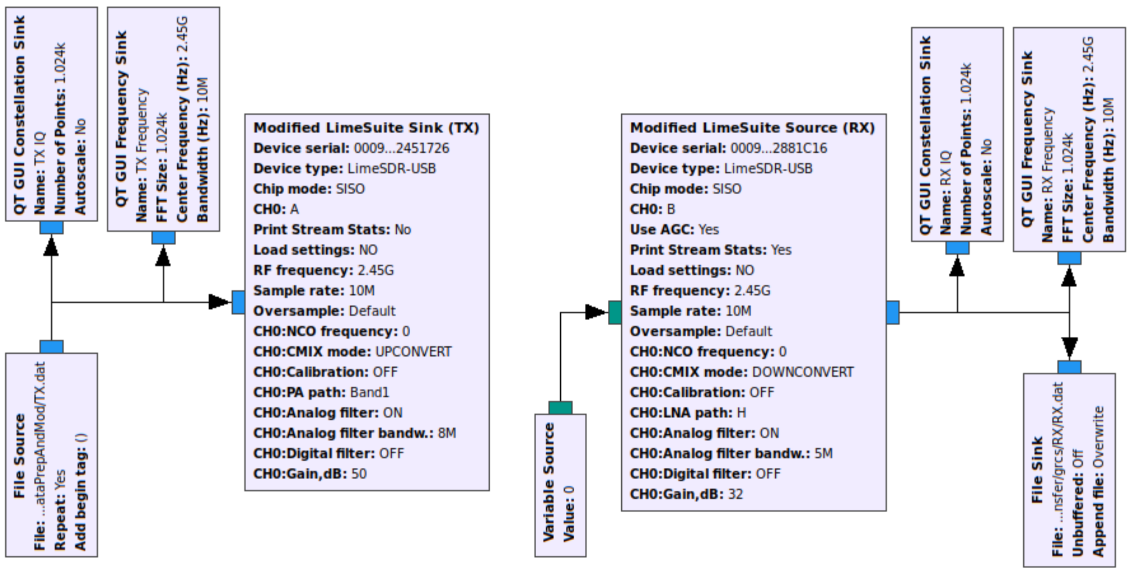
```
Options                    Variable          Variable             QT GUI Range         QT GUI Range         QT GUI Range
ID: TXRX                   ID: samp_rate     ID: bandwidth_rx     ID: gain_tx          ID: gain_rx          ID: agc
Title: TXRX                Value: 10M        Value: 5M            Label: gain_tx       Label: gain_rx       Label: AGC
Generate Options: QT GUI                                          Default Value: 50    Default Value: 32    Default Value: 0
                           Variable          Variable             Start: 20            Start: 20            Start: 0
                           ID: LO_freq       ID: bandwidth_tx     Stop: 60             Stop: 70             Stop: 20
                           Value: 2.45G      Value: 8M            Step: 2              Step: 2              Step: 1

                           QT GUI Tab Widget     QT GUI Tab Widget          QT GUI Tab Widget     QT GUI Tab Widget
                           Num Tabs: 1           Num Tabs: 1                Num Tabs: 1           Num Tabs: 1
                           Label 0: TX IQ        Label 0: TX Frequency      Label 0: RX IQ        Label 0: RX Frequency
```

Figure 6.8: Variables needed for transmit and receive flow graph.

6.3.2 Data Processing

As mentioned in section 5.3, the used hardware blocks are modifications from the open source implementations on Github[28]. The modifications include an additional parameter to activate stream statistics to be printed out to console, and also an input on the RX block to activate and control AGC.

The data processing flow graph is shown in Figure 6.9. The TX block gets the complex symbols as input, which are also sent to GUIs to illustrate them for the user. Furthermore, the RX block stores its received symbols into a file.



QT GUI Constellation Sink
Name: TX IQ
Number of Points: 1.024k
Autoscale: No

QT GUI Frequency Sink
Name: TX Frequency
FFT Size: 1.024k
Center Frequency (Hz): 2.45G
Bandwidth (Hz): 10M

File Source
File: ...ataPrepAndMod/TX.dat
Repeat: Yes
Add begin tag: ()

Modified LimeSuite Sink (TX)
Device serial: 0009...2451726
Device type: LimeSDR-USB
Chip mode: SISO
CH0: A
Print Stream Stats: No
Load settings: NO
RF frequency: 2.45G
Sample rate: 10M
Oversample: Default
CH0:NCO frequency: 0
CH0:CMIX mode: UPCONVERT
CH0:Calibration: OFF
CH0:PA path: Band1
CH0:Analog filter: ON
CH0:Analog filter bandw.: 8M
CH0:Digital filter: OFF
CH0:Gain,dB: 50

Variable Source
Value: 0

Modified LimeSuite Source (RX)
Device serial: 0009...2881C16
Device type: LimeSDR-USB
Chip mode: SISO
CH0: B
Use AGC: Yes
Print Stream Stats: Yes
Load settings: NO
RF frequency: 2.45G
Sample rate: 10M
Oversample: Default
CH0:NCO frequency: 0
CH0:CMIX mode: DOWNCONVERT
CH0:Calibration: OFF
CH0:LNA path: H
CH0:Analog filter: ON
CH0:Analog filter bandw.: 5M
CH0:Digital filter: OFF
CH0:Gain,dB: 32

QT GUI Constellation Sink
Name: RX IQ
Number of Points: 1.024k
Autoscale: No

QT GUI Frequency Sink
Name: RX Frequency
FFT Size: 1.024k
Center Frequency (Hz): 2.45G
Bandwidth (Hz): 10M

File Sink
File: ...nsfer/grcs/RX/RX.dat
Unbuffered: Off
Append file: Overwrite

Figure 6.9: Data Processing for transmit and receive flow graph

The first option is the devices' serial number to identify it on the host interface. A list of serial numbers of the used devices is documented in section 3.3. Next is the *Device type* option that defines which kind of model of LimeSDR is to be used. If the *Chip mode* is set to MIMO, the device uses antenna diversity and activates its second input port. In the illustrated flow graph the option is set to SISO, therefore only enabling one input port. If *Print Stream Stats* is enabled, the previously mentioned console printouts will be activated.

LimeSDR can be configured by a *.ini* file. This is activated by setting the *Load settings* option. If chosen, all other parameters get deactivated, since they are already stored in the configuration settings of the *.ini* file. The *RF frequency* is the mentioned synthesizer frequency in section 3.4.3. Additionally, *Sample rate* is the speed of the ADCs and DACs with associated oversampling, see also section 3.4.4.

Next is a configuration of the analog channel *CH0*. Some of the options originate from the digital part of the TRX, as briefly described in section 3.4.5. The list of configuration variables include:

- *NCO frequency* is the frequency of the optional numerically controlled oscillator.
- *CMIX mode* is the mixing mode of the complex mixer[26].

- *Calibration* is a boolean variable to determine if the device should calibrate before any sort of transmission takes place.
- *Calibration bandw.* (not active since calibration is deactivated) is, as the name implies, the bandwidth for what the analog and digital paths should be calibrated for.
- *PA path* defines on which of the two possible TX paths the antenna is actually connected. The choice is between Band1 and Band2.
- *Analog filter* activates the filters in the analog paths.
- *Analog filter bandw.* is only visible if the previous *Analog filter* option is enabled and it defines the bandwidth to be set for the filters.
- *Digital filter* is similar to the previously mentioned *Analog filter* option and also activates the bandwidth for which the filter taps should be optimized.
- *Gain,dB* finally defines for what values the amplification stages should be set and tuned.

If the mentioned chip mode is set to MIMO, a second register card is active and the second channel *CH1* can be configured with the same parameters as mentioned.

The RX hardware block configuration is very similar. It has an additional parameter called *Use AGC* to toggle the input port. If it is enabled, the input port expects an integer value. Is the value zero, then the block deactivates AGC. If it is a non zero value, AGC gets activated using formula 4.1. The input is connected with the **Variable Source** block documented in section 5.3. This block uses the graphical slider value to change AGC dynamically.

Finally, the received data is illustrated in GUI blocks and also saved into a *.dat* file for the next flow graph.

6.4 Data Synchronization, Demodulation and Correlation

The data that the LimeSDR receives is noisy and not synchronized. During times when the transmitter does not send data, the receiver still processes the background noise. For these reasons the next flow graph visualized in Figure 6.10 performs data synchronization (see section 2.6), demodulation and data correlation (i.e. frame synchronization).

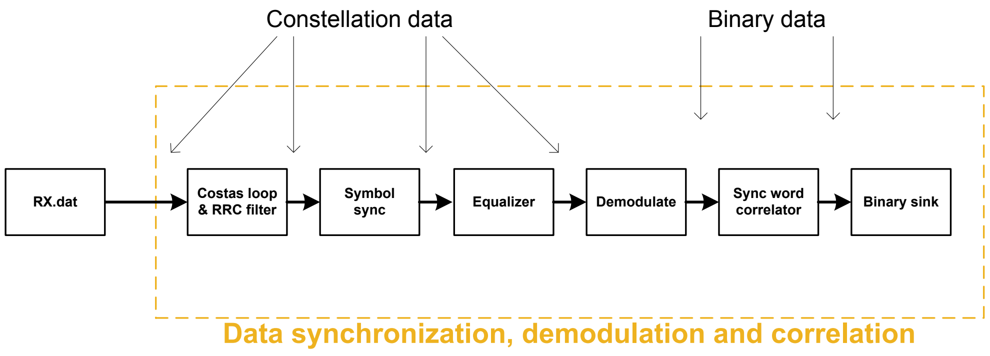


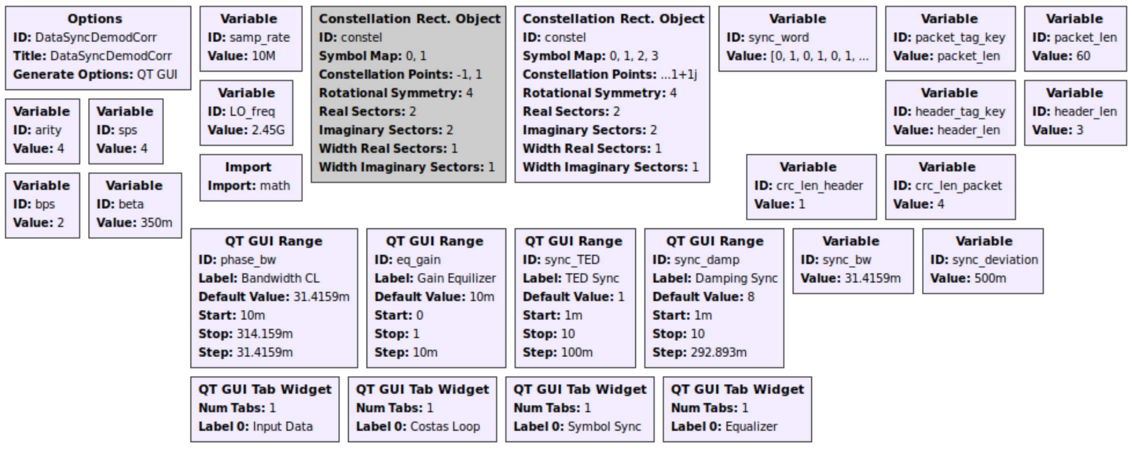
Figure 6.10: Concept for data synchronization, demodulation and correlation.

As described in section 2.6, the synchronization chain starts with a **Costas Loop** to performs carrier locking. Next is a decimating RRC filter to satisfy the equivalent channel equation. Following this filter is the timing recovery realized by a **Symbol sync** block and finally an **Equalizer** to reduce smearing effects.

Afterwards, the symbols will get demodulated, resulting in a byte stream. The data from this byte stream will in most cases not be perfectly aligned regarding its MSB and LSB positions. Therefore the mentioned custom programmed **Sync word correlator** shifts the whole bit sequence to the correct positions every time it detects the frame synchronization sequence. The resulting data gets saved into a binary file for evaluation.

6.4.1 Variables

The previous sections already mentioned most of the variables required for this flow graph.



Options
ID: DataSyncDemodCorr
Title: DataSyncDemodCorr
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 10M

Constellation Rect. Object
ID: constel
Symbol Map: 0, 1
Constellation Points: -1, 1
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

Constellation Rect. Object
ID: constel
Symbol Map: 0, 1, 2, 3
Constellation Points: ...1+1j
Rotational Symmetry: 4
Real Sectors: 2
Imaginary Sectors: 2
Width Real Sectors: 1
Width Imaginary Sectors: 1

Variable
ID: sync_word
Value: [0, 1, 0, 1, 0, 1, ...

Variable
ID: packet_tag_key
Value: packet_len

Variable
ID: packet_len
Value: 60

Variable
ID: arity
Value: 4

Variable
ID: sps
Value: 4

Variable
ID: LO_freq
Value: 2.45G

Variable
ID: header_tag_key
Value: header_len

Variable
ID: header_len
Value: 3

Import
Import: math

Variable
ID: crc_len_header
Value: 1

Variable
ID: crc_len_packet
Value: 4

Variable
ID: bps
Value: 2

Variable
ID: beta
Value: 350m

QT GUI Range
ID: phase_bw
Label: Bandwidth CL
Default Value: 31.4159m
Start: 10m
Stop: 314.159m
Step: 31.4159m

QT GUI Range
ID: eq_gain
Label: Gain Equilizer
Default Value: 10m
Start: 0
Stop: 1
Step: 10m

QT GUI Range
ID: sync_TED
Label: TED Sync
Default Value: 1
Start: 1m
Stop: 10
Step: 100m

QT GUI Range
ID: sync_damp
Label: Damping Sync
Default Value: 8
Start: 1m
Stop: 10
Step: 292.893m

Variable
ID: sync_bw
Value: 31.4159m

Variable
ID: sync_deviation
Value: 500m

QT GUI Tab Widget
Num Tabs: 1
Label 0: Input Data

QT GUI Tab Widget
Num Tabs: 1
Label 0: Costas Loop

QT GUI Tab Widget
Num Tabs: 1
Label 0: Symbol Sync

QT GUI Tab Widget
Num Tabs: 1
Label 0: Equalizer

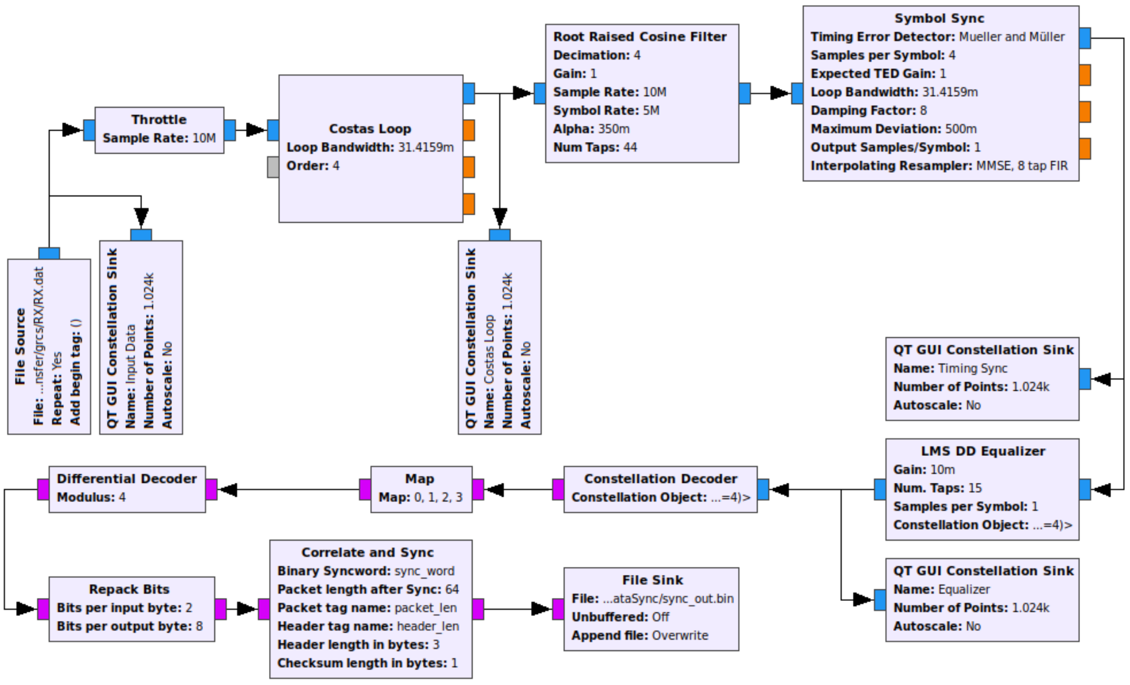
Figure 6.11: Variables needed for data synchronization, demodulation and correlation flow graph.

A few new **QT GUI Range** variables are in use that can be changed during execution of the flow graph. From left to right these variables are: The loop bandwidth of the Costas loop, the gain of the equalizer, the mentioned TED value in section 2.6 and the damping value of the loop filter. The loop bandwidth of the symbol timing algorithm is fixed, as well as its deviation tolerance. The depicted values are optimal parameter values for most configurations of this flow graph.

This flow graph also imports a *Python* module named *math*. This gives access to mathematical constants, especially π. There are also **QT GUI Tab Widgets**, which arrange the depicted GUIs and sliders on the resulting window.

6.4.2 Data Processing

As expected, the flow graph depicted in Figure 6.12 starts with a source block on repeat mode that loads the data file of the LimeSDR RX output. Next is again a **Throttle** block to constrain hardware use for the host.


Throttle
Sample Rate: 10M

Costas Loop
Loop Bandwidth: 31.4159m
Order: 4

Root Raised Cosine Filter
Decimation: 4
Gain: 1
Sample Rate: 10M
Symbol Rate: 5M
Alpha: 350m
Num Taps: 44

Symbol Sync
Timing Error Detector: Mueller and Müller
Samples per Symbol: 4
Expected TED Gain: 1
Loop Bandwidth: 31.4159m
Damping Factor: 8
Maximum Deviation: 500m
Output Samples/Symbol: 1
Interpolating Resampler: MMSE, 8 tap FIR

File Source
File: ...nsfer/grcs/RX/RX.dat
Repeat: Yes
Add begin tag: ()

QT GUI Constellation Sink
Name: Input Data
Number of Points: 1.024k
Autoscale: No

QT GUI Constellation Sink
Name: Costas Loop
Number of Points: 1.024k
Autoscale: No

QT GUI Constellation Sink
Name: Timing Sync
Number of Points: 1.024k
Autoscale: No

Differential Decoder
Modulus: 4

Map
Map: 0, 1, 2, 3

Constellation Decoder
Constellation Object: ...=4)>

LMS DD Equalizer
Gain: 10m
Num. Taps: 15
Samples per Symbol: 1
Constellation Object: ...=4)>

Correlate and Sync
Binary Syncword: sync_word
Packet length after Sync: 64
Packet tag name: packet_len
Header tag name: header_len
Header length in bytes: 3
Checksum length in bytes: 1

Repack Bits
Bits per input byte: 2
Bits per output byte: 8

File Sink
File: ...ataSync/sync_out.bin
Unbuffered: Off
Append file: Overwrite

QT GUI Constellation Sink
Name: Equalizer
Number of Points: 1.024k
Autoscale: No

Figure 6.12: Data Processing for data synchronization, demodulation and correlation flow graph.

Next is a **Costas Loop** block that uses a PLL circuit with a defined *Loop Bandwidth* to recover the carrier frequency by locking to the center frequency. This locks the symbols in the complex constellation diagram, see also subsection 2.6.2. An additional optional input port called *noise* is a floor estimate to calculate the SNR of symbols. Besides of the *out* port there is an optional output port called *frequency* that passes the normalized frequency of the loop.

The following **Root Raised Cosine Filter** performs the decimation process by first downsampling the signal and then using a RRC matched filter. This block also needs to know the excess bandwidth, defined by the parameter *Alpha*.

Next is a very crucial block called **Symbol Sync**. This is responsible for correct symbol timing recovery. It implements a few different timing error detector algorithms to be chosen by the user. The most relevant ones for later are:

- Mueller and Müller, decision-directed.
- Zero Crossing, decision-directed.
- Gardner, non-data-aided.
- Early-Late, non-data-aided.
- sgn(y[n]y'[n]) Maximum Likelihood, non-data-aided.

All of these were described in subsection 2.6.1. Figure 6.13 shows the concept diagram of this block. It implements an *Interpolating Resampler* that can decimate the signal. The remaining parameters control the expected gain of the TED (the slope of the S-curve), the *Loop Bandwidth* of the symbol clock tracking loop, and the *Maximum Deviation* of the average clock period estimate. Additionally, if the chosen timing error detector algorithm is decision-directed, the block also needs access to a constellation object. Finally, the *Damping Factor* defines the stability of the loop: Values below 1 are under-damped, $1/\sqrt{2}$ is maximally flat response, a value of 1 is critically damped and a value greater 1 means over-damped.

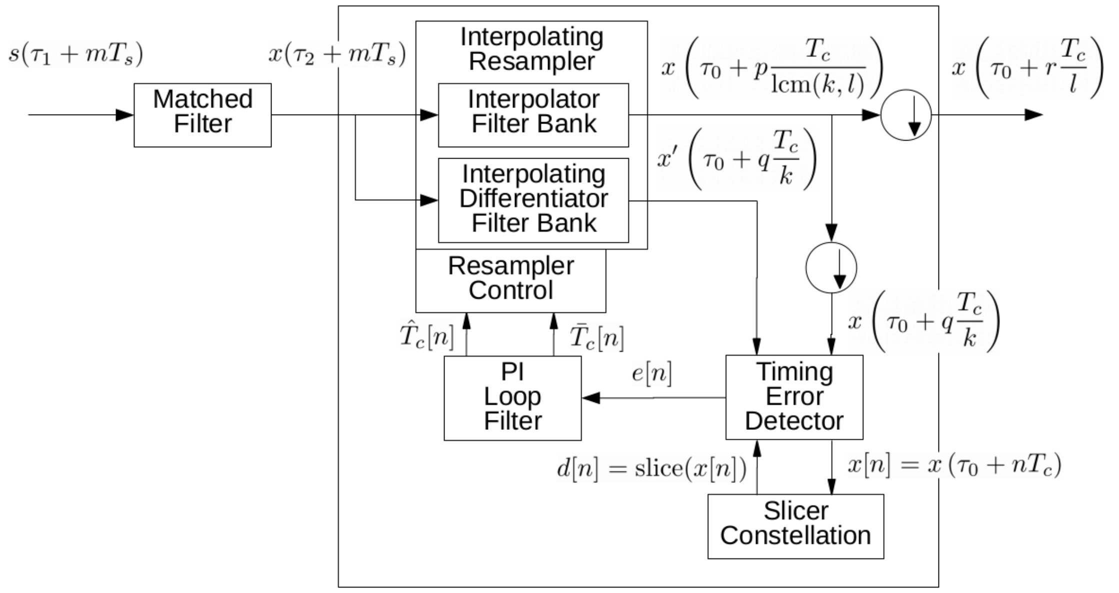

$s(\tau_1 + mT_s)$
$x(\tau_2 + mT_s)$
Matched Filter
Interpolating Resampler
Interpolator Filter Bank
Interpolating Differentiator Filter Bank
Resampler Control
$x\left(\tau_0 + p\frac{T_c}{\mathrm{lcm}(k,l)}\right)$
$x'\left(\tau_0 + q\frac{T_c}{k}\right)$
$x\left(\tau_0 + r\frac{T_c}{l}\right)$
$x\left(\tau_0 + q\frac{T_c}{k}\right)$
$\hat{T}_c[n]$
$\bar{T}_c[n]$
PI Loop Filter
$e[n]$
Timing Error Detector
$d[n] = \mathrm{slice}(x[n])$
$x[n] = x(\tau_0 + nT_c)$
Slicer Constellation

Figure 6.13: Concept diagram of symbol sync block[29].

The last of the synchronization blocks is a **Least Mean Square Decision-Directed Equalizer**. As the name suggests, this block uses a LMS algorithm to calculate the error between the incoming sample and the defined sampling locations of the given modulation scheme. Therefore, the constellation object used must be passed into the parameter *Constellation Object*. The block does not reduce the sampling rate since the parameter *Samples per Symbol* is set to 1. The decision for the equalizer is defined by a number of weights specified by *Num. Taps* and a *Gain* factor.

> "It uses a set of weights, w, to correlate against the inputs, u, and a decisions is then made from this output. The error in the decision is used to update the weight vector.
> y[n] = conj(w[n]) u[n] d[n] = decision(y[n]) e[n] = d[n] - y[n] w[n+1] = w[n] + mu u[n] conj(e[n])
> Where mu is a gain value (between 0 and 1 and usually small, around 0.001 - 0.01." From the GNU Radio API reference[30].

Next is the demodulation process. The **Constellation Decoder** converts the signals from the complex plane into binary data. Similar to previous blocks, knowledge about the modulation scheme is required. Therefore, this block gets passed the *Constellation Object*. The **Map** block maps incoming symbols to their specific value in the *Map* variable and the **Differential Decoder** uses knowledge of the incoming binary symbol and their previous ones to perform differential decoding. It's variable *Modulus* defines how many symbols the modulation scheme used.

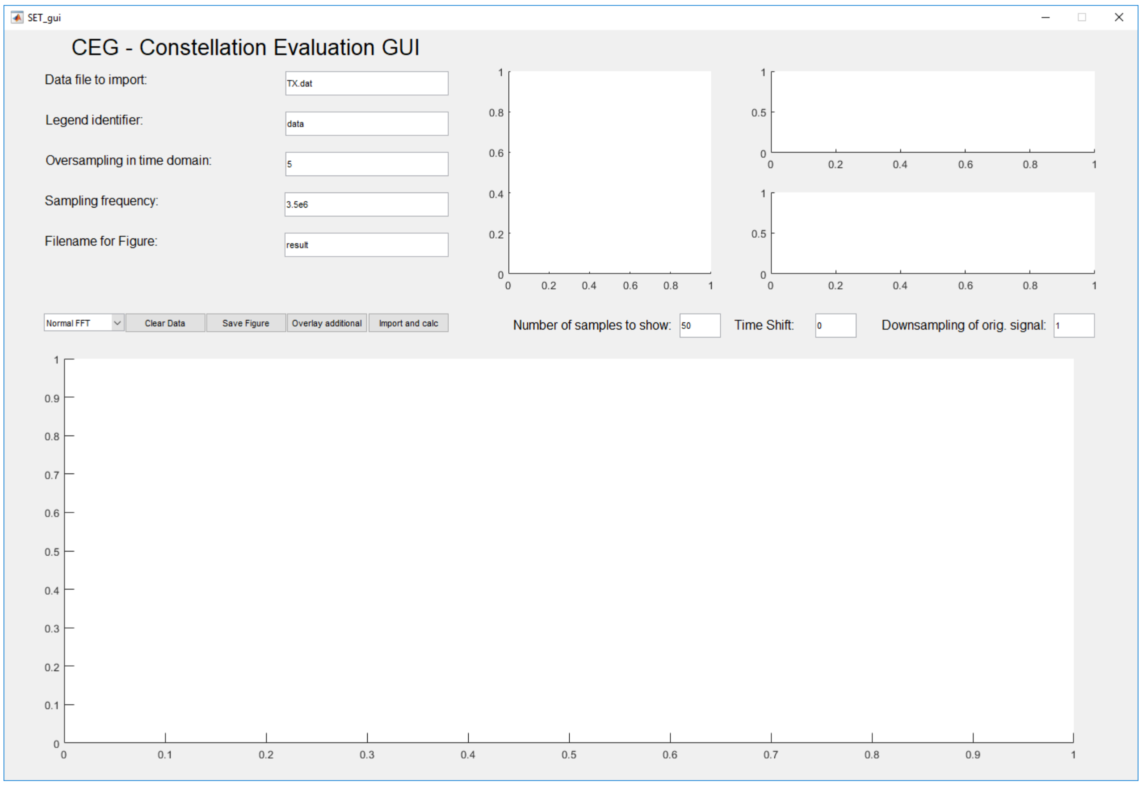
The final crucial block before the information sink is the **Correlate and Sync** block. This is a custom block and its function was documented in section 5.3. In short, it correlates the incoming data with the expected synchronization sequence. If the sequence is found, the whole frame (header plus data packet) gets bitwise shifted for correct representation on the host and passed to the output. Data which does not contain the synchronization sequence will be discarded.

The final data will be saved in a binary **File Sink** for later evaluation.

Chapter 7

Constellation Evaluation GUI

As described in section 6.1, to evaluate the GNU Radio flow graphs, a method was developed to compare different data files containing complex constellation data. This tool is called Constellation Evaluation GUI, abbreviated as CEG, and was programmed in MATLAB. Figure 7.1 shows the blank tool, right after startup.



Data file to import: TX.dat
Legend identifier: data
Oversampling in time domain: 5
Sampling frequency: 3.5e6
Filename for Figure: result

Number of samples to show: 50
Time Shift: 0
Downsampling of orig. signal: 1

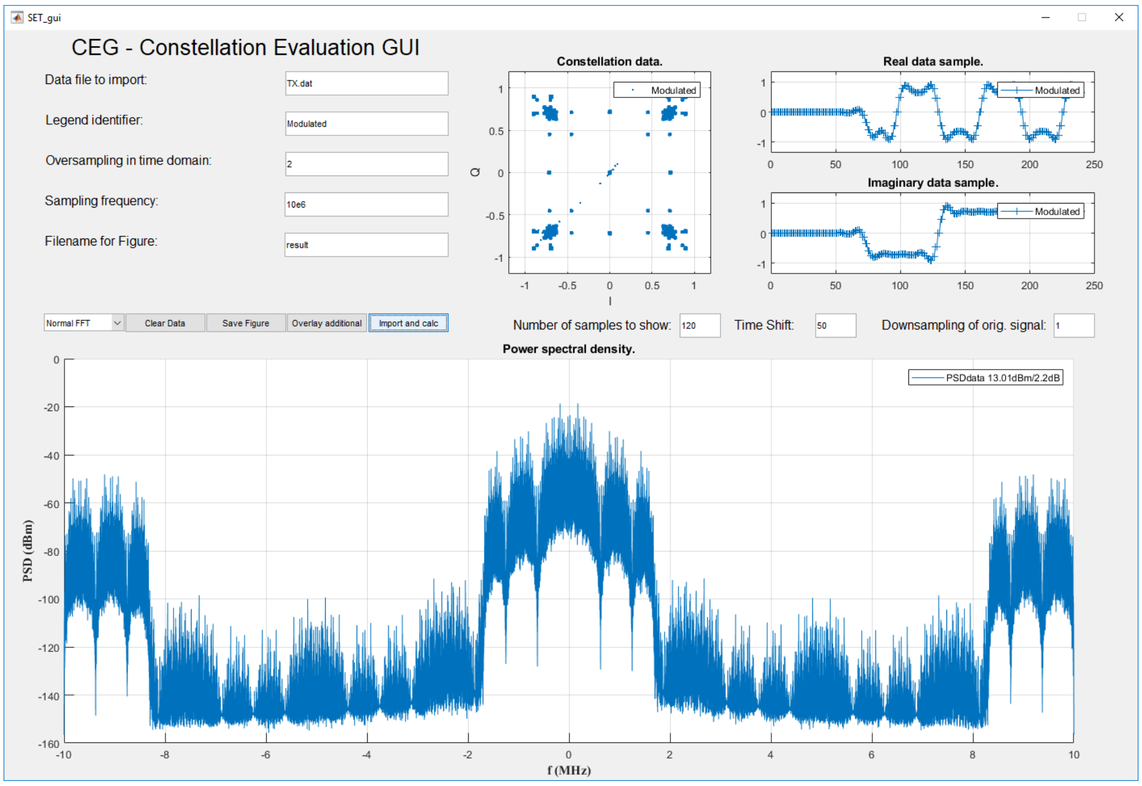
Figure 7.1: CEG right after startup.

7.1 Implementation

The GUI was created using the *guide* tool in MATLAB. First of all, the data file containing the constellation data must be chosen. It's important that this file is in the same directory as CEG. Next is the identifier by which the data will be labeled in plots.

After clicking *Import and calc* the program opens the file and reads the data. This GNU Radio data alternates between real values and complex values, both of which are saved as floating point numbers. Together they form a single symbol. The constellation data is displayed on the first Figure. Next, the data is transformed to time domain using a combination of *repmat* and *reshape*. In order to better convert the data to frequency domain later, the time domain values are oversampled, given by the *Oversampling in time domain* parameter. The time domain data then gets displayed on the next two Figures, divided by real and imaginary part. Note that the user can also decide how many samples to display, how much to time shift them to the left and by how much to downsample it again. This downsampling is advantages for oversampled data.

Finally, the data gets transformed into frequency domain. For this purpose the Institute of Microwave and Photonic Engineering provided functions to pass upsampled complex data to. These functions then calculate the frequency values and displays them in a Figure. For this, the user also needs to provide the *Sampling frequency* parameter. The mentioned MATLAB functions either calculate the PSD (**P**ower **S**pectral **D**ensity) via normal FFT or using Welch's method. The user can chose which form to use.



SET_gui

CEG - Constellation Evaluation GUI

Data file to import: TX.dat
Legend identifier: Modulated
Oversampling in time domain: 2
Sampling frequency: 10e6
Filename for Figure: result

Normal FFT | Clear Data | Save Figure | Overlay additional | Import and calc

Number of samples to show: 120 | Time Shift: 50 | Downsampling of orig. signal: 1

Figure 7.2: Constellation, time and frequency data from the modulated TX data.

As an example, Figure 7.2 shows CEG after importing the example data from Figure 6.6.

When comparing the data, it is obvious that the constellation data is the same, as well as the time domain data. However, in time domain, it is visible that the data needs a few samples before it ”starts”. This is due to the design of the RRC filter being a differential equation. The PSD shows the frequency representation. Next chapter will show that this is also the same when transmitting it using a SDR.

Additionally, the user can *Overlay additional* data sets, with a maximum of 3 data sets to import. Pressing *Clear Data* deletes all internal variables and clears all Figures. *Save Figure* creates a copy of the frequency illustration and saves it.

CEG will be used to compare different data sets in the next chapter. It will also complement measurements on a signal analyzer and compare the different synchronization steps.

Chapter 8

Design Evaluation

In order to prove the functionality of the system and its parameters, a step by step evaluation will be performed in this chapter. To complement this evaluation, measurement instruments will be used to show the practicability and repeatability of this design.

Section 8.1 will showcase test files and frame structures. A theoretical prove of concept is discussed in section 8.2. In other words, this means not using the SDR hardware, instead using the modulated data output from the first flow graph as input for the synchronization, demodulation and correlation flow graph.

Following steps expand the system further. Section 8.3 proves that the system works as expected when one SDR is used to both transmit and receive the data simultaneously. This is achieved by connecting the analog paths of the device via a cable. Section 8.4 takes this one step further by connecting two LimeSDR via a cable. Finally, section 8.5 implements an actual free-space transmission between two SDR.

To expand this evaluation, section 8.6 will compare the frequency domain representation of the different steps in the RX synchronization chain. The section will also contain a comparison of different symbol recovery algorithms at different RRC roll-off factors.

8.1 Data Format

The first step is to define the data that will be processed, how it is structured and also how a data frame looks like. Since this design starts at a host PC, the information source has a binary format. In order to ease evaluation for a human developer, some of the test cases are in form of text files with known sequences of ASCII characters, which translate to binary data on the hard drive.

8.1.1 Test Cases

There are two types of test files in use. While in both cases this data is in binary format, one type uses random binary sequences while the other uses defined sequences to better recognize bit errors.

Random Test File

The random binary sequence is generated with GNU Radio, using the simple flow graph illustrated in Figure 8.1. A **Random Source** outputs the samples, these are then stored in data files. The *Num Samples* attribute of the source determines how large the data file will be.

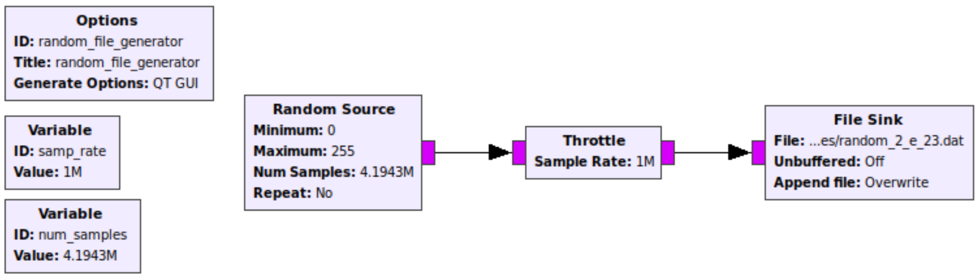


Figure 8.1: Example flow graph to create random binary sequence test cases.

cafe Test File

As for the test cases with known binary sequences, any ASCII form text file on the host is sufficient. In order to standardize this test case, a repeating sequence of *cafe* will be used. This sequence can be repeated multiple times in a test file. In order to examine these files on a binary level, one can look up the binary sequence of the characters in an ASCII Table and use the command **xxd -b** in a Linux terminal. An example of the first few rows of this command is shown in Listing 8.1. The left most column shows the address of data in the file, followed by a binary representation of the data and finally of the ASCII character representation on the far right.

```
mh@mh:~$xxd -b cafe.txt
00000000: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
00000006: 01100110 01100101 01100011 01100001 01100110 01100101  fecafe
0000000c: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
00000012: 01100110 01100101 01100011 01100001 01100110 01100101  fecafe
00000018: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
0000001e: 01100110 01100101 01100011 01100001 01100110 01100101  fecafe
...
```

Listing 8.1: Usage of the xxd command to view cafe test case data.

l Test File

An additional test case is a repeating sequence of the letter *l*. Looking this up in a ASCII Table, the value is *108* in decimal and *01101100* in binary. This includes all 4 possible symbols for QPSK: *01*, *10*, *11* and *00*.

```
mh@mh:~$xxd -b l.txt
00000000: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
00000006: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
...
```

Listing 8.2: Usage of the xxd command to view l test case data.

8.1.2 Frame Structure

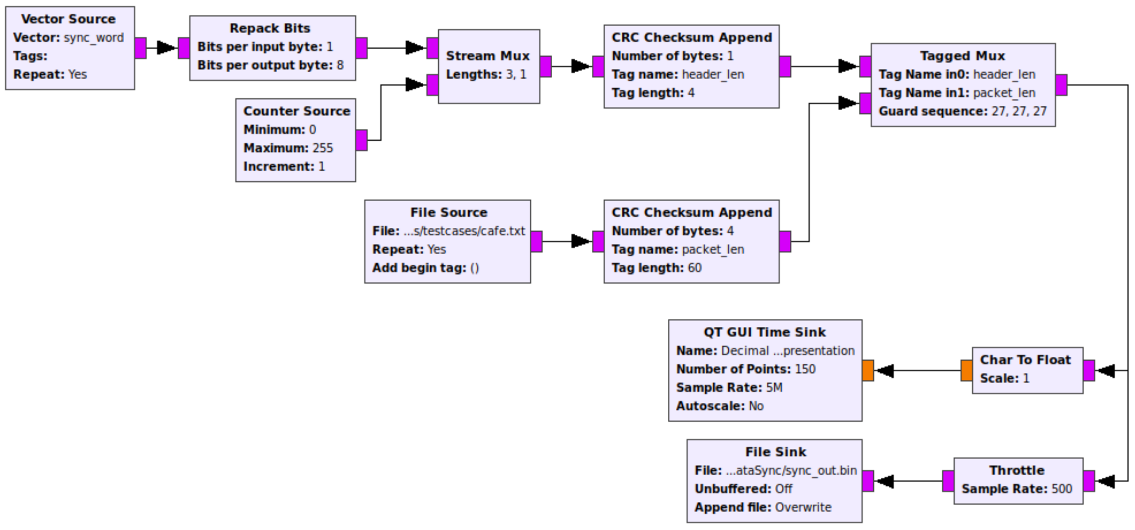
Section 6.2 discussed the flow graph to prepare and modulate the test cases. A simple frame structure is set up in the first half of this flow graph, and Figure 6.3 illustrated how a frame is structured.

The synchronization sequence is repacked from a binary sequence to a byte structure. In order to distinguish different frames, this synchronization sequence gets multiplexed with an identification byte using the custom programmed block **Counter Source**. This ID number is between 0 and 255, i.e. has a length of 1 byte. Together these two steps produce a simple header with the length *header_len + 1*. This header is then sent into a **CRC Checksum Append** custom block to add a header tag and to append a checksum for later error detection.

As for the data packet, a **File Source** loads the previously described test files and appends a checksum in the same fashion as with the header. The only difference being the length of the data packet, set by a variable (*packet_len*). Both header and packet data are then multiplexed to get a frame. Finally, the **Tagged Mux** also adds an optional *Guard sequence*. The reason behind this guard sequence is to separate packages and give following blocks the ability to better lock on the carrier frequency.

Figure 8.2 shows an example flow graph to illustrate the frame structure. This is basically the same flow graph as the data preparation and modulation flow graph, but without the modulation step.

This frame structure in decimal format is visible in Figure 8.3. The synchronization sequence is a repetition of three *W* characters, which according to the ASCII Table translates to a decimal value of *87*. Clearly visible is the header tag at the start of the frame. After the synchronization sequence follows the identification byte (0) and the checksum byte. This marks the end of the header, which is followed by the data package. The data package is clearly tagged with its packet length, and identifiable by the repeating *cafe* sequence (*99-97-102-101*). The packets' length is defined as 60 bytes, together with a checksum length of 4 bytes. Finally, the frame is completed by a guard sequence, in this case 3 bytes with the value *27*.



Vector Source
Vector: sync_word
Tags:
Repeat: Yes

Repack Bits
Bits per input byte: 1
Bits per output byte: 8

Counter Source
Minimum: 0
Maximum: 255
Increment: 1

Stream Mux
Lengths: 3, 1

CRC Checksum Append
Number of bytes: 1
Tag name: header_len
Tag length: 4

File Source
File: ...s/testcases/cafe.txt
Repeat: Yes
Add begin tag: ()

CRC Checksum Append
Number of bytes: 4
Tag name: packet_len
Tag length: 60

Tagged Mux
Tag Name in0: header_len
Tag Name in1: packet_len
Guard sequence: 27, 27, 27

QT GUI Time Sink
Name: Decimal ...presentation
Number of Points: 150
Sample Rate: 5M
Autoscale: No

Char To Float
Scale: 1

File Sink
File: ...ataSync/sync_out.bin
Unbuffered: Off
Append file: Overwrite

Throttle
Sample Rate: 500

Figure 8.2: Flow graph to illustrate frame structure.

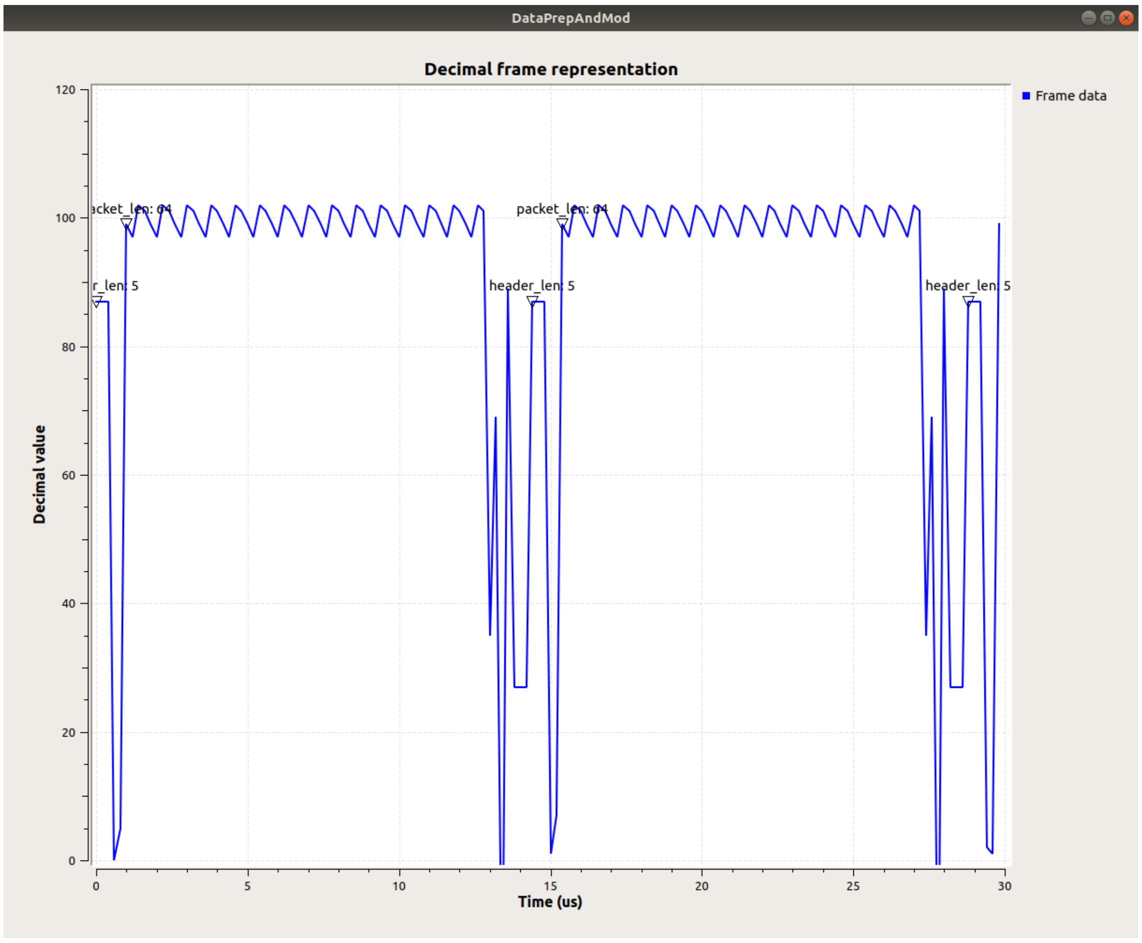


Figure 8.3: Representation of complete frames as decimal values. After the header data, the packet contains a reoccurring "cafe" sequence.

Using the *xxd* command in console, a user can confirm this structure. Listing 8.3 shows the first rows of the output of this command. **Clearly the data corresponds with the described frame structure:** Synchronization sequence *WWW*, packet ID beginning with 0, CRC byte, 60 bytes of packet data, CRC with length 4, and 3 bytes of guard sequence.

```
mh@mh:~$xxd -b frame_structure.dat
00000000: 01010111 01010111 01010111 00000000 00000101 01100011  WWW..c
00000006: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
0000000c: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
00000012: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000018: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
0000001e: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000024: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
0000002a: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000030: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
00000036: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
0000003c: 01100101 01100011 01100001 01100110 01100101 00100011  ecafe#
00000042: 01000101 11100100 01011001 00011011 00011011 00011011  E.Y...
00000048: 01010111 01010111 01010111 00000001 00000111 01100011  WWW..c
0000004e: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000054: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
...
```

Listing 8.3: Showcase of the frame structure, the first frame and beginning of the second one (address 48 in hexadecimal).

Of course the frame structure is the same when using random sequence test cases, in which case the data packets would contain the random data information. The test case with l sequences contains this data accordingly.

8.2 Theoretical Prove of Concept

The previous section illustrated the test cases and how the frames are structured. The following section will show that the developed system actually works by only running the flow graphs that do not use hardware blocks. These are the data preparation and modulation flow graph (section 6.2) and the data synchronization, demodulation and correlation flow graph (section 6.4). Throughout the evaluation the configuration stays the same as described in these two sections.

If the data at the end of the second flow graph resembles the test case data, this will prove the theoretical concept of this system. The structure of this simulation is illustrated in Figure 8.4.

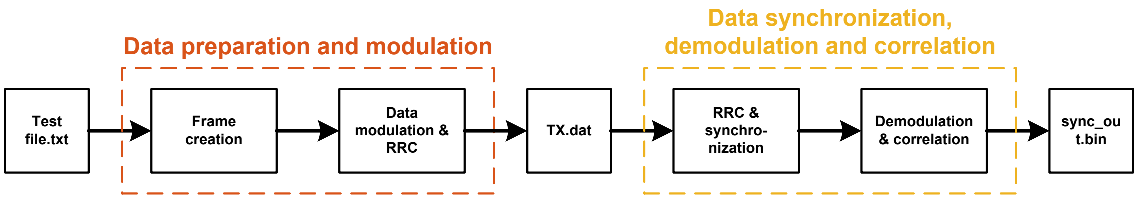
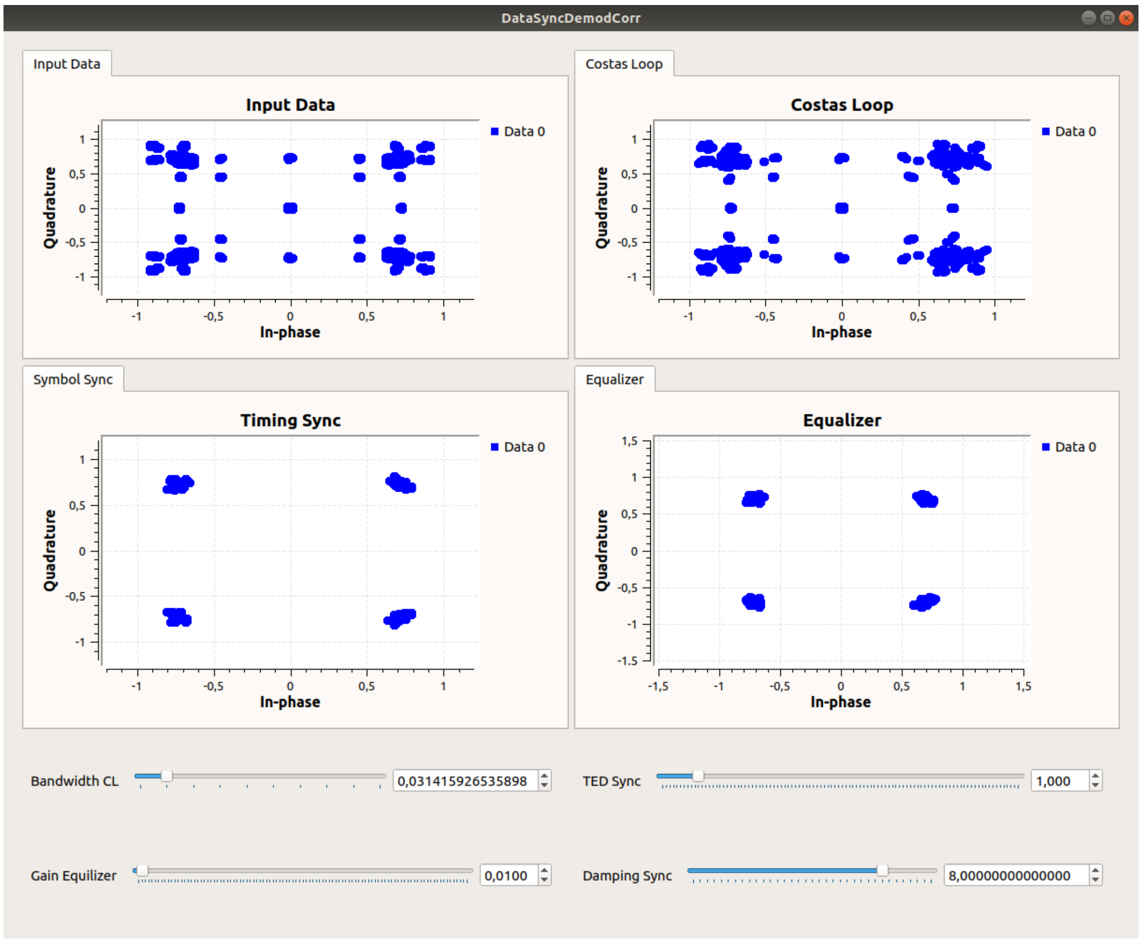

Data preparation and modulation
Data synchronization, demodulation and correlation
Test file.txt
Frame creation
Data modulation & RRC
TX.dat
RRC & synchro-nization
Demodulation & correlation
sync_out.bin

Figure 8.4: Simulation setup for prove of concept.

The result of the modulation and resampling process was already discussed. Figure 6.6 depicted the data in GNU Radio and Figure 7.2 illustrated it using CEG. When using this data as input for the flow graph in section 6.4, the result is obvious: The data is ideal (i.e. no additive noise and no synchronization problems), therefore the constellation after the carrier locking is the same. Furthermore, the symbol timing recovery only performs small corrections, same as the equalizer. This results in an ideal QPSK constellation. Figure 8.5 illustrates exactly these results.



Bandwidth CL 0,031415926535898
TED Sync 1,000
Gain Equilizer 0,0100
Damping Sync 8,00000000000000

Figure 8.5: Concept prove of data processing only flow graphs: Synchronized data.

The resulting binary information can again be observed when examining the resulting file using the *xxd -b* command. Listing 8.4 shows the resulting information when running the flow graph. The data packets are perfect, and are the same as the illustrated frame structure in Listing 8.3.

```
mh@mh:~$xxd -b sync_out.bin
00000000: 01010111 01010111 01010111 00000000 00000101 01100011  WWW..c
00000006: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
0000000c: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
00000012: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000018: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
0000001e: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000024: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
0000002a: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000030: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
00000036: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
0000003c: 01100101 01100011 01100001 01100110 01100101 00100011  ecafe#
00000042: 01000101 11100100 01011001 00011011 00011011 00011011  E.Y...
00000048: 01010111 01010111 01010111 00000001 00000111 01100011  WWW..c
0000004e: 01100001 01100110 01100101 01100011 01100001 01100110  afecaf
00000054: 01100101 01100011 01100001 01100110 01100101 01100011  ecafec
...
```

Listing 8.4: Resulting information sink when using data processing only flow graphs.

This proves that the system works perfectly in theory, i.e. when not including actual SDR hardware.

8.2.1 Concept Prove with l Test File

The concept can also be proven with the previously mentioned *l Test File* for QPSK modulation. After modulating the new text data and sending it through the synchronization flow graph, the information sink shows the result in Listing 8.5. As expected, the data now consists of repeating *l*. Everything else stays the same.

```
mh@mh:~$xxd -b sync_out.bin
00000000: 01010111 01010111 01010111 00000000 00000101 01101100  WWW..l
00000006: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
0000000c: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
00000012: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
00000018: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
0000001e: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
00000024: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
0000002a: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
00000030: 01101100 01101100 01101100 01101100 01101100 01101100  llllll
...
```

Listing 8.5: Resulting information sink when using data processing only flow graphs and l test file.

8.3 Using one Device as TX and RX

The previous section proved that the system works in theory when not using SDR hardware. To continue the evaluation, this section will take one step further and use one of the LimeSDRs. Since these are TRX boards, it's possible to use them to both transmit and receive data simultaneously. However, to keep this concept simple, both analog ports will be connected via a cable instead of a free-space transmission. One advantage of this is that the LimeSDR uses the same LO value for both analog paths. Therefore, the receiving signal will not experience carrier drift, instead the constellation will be locked in place. This concept is visualized in Figure 8.6.

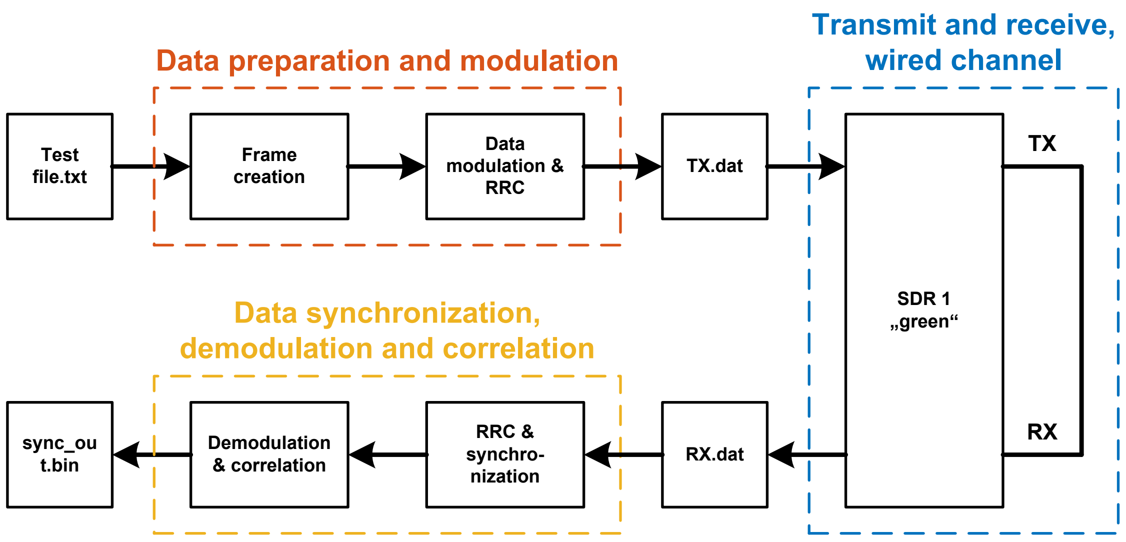

Data preparation and modulation
Transmit and receive, wired channel
Test file.txt
Frame creation
Data modulation & RRC
TX.dat
TX
SDR 1 „green“
Data synchronization, demodulation and correlation
RX
sync_out.bin
Demodulation & correlation
RRC & synchro-nization
RX.dat

Figure 8.6: Simulation setup for using only one device.

8.3.1 TX/RX Flow Graph

The flow graph to modulate will stay the same for now, which means that the constellation data will still be QPSK and resemble Figure 6.6. It will also stay the same configuration for the flow graph to synchronize, demodulate and correlate the data.

However, the transmit and receive flow graph discussed in section 6.3 will be executed between these two flow graphs and actually send and receive data. The flow graph is configured the same as depicted in Figure 6.9. However, care has to be taken to set the variable slider values correctly. If one would maximize both the TX gain and the RX sensitivity, this could potentially damage the hardware. In this case, a *gain_tx* of *50* and a *gain_rx* of *32* was chosen, while AGC was left turned off with *0*. Used device for this transmission was "green" (see list of devices in section 3.3 for the serial number).

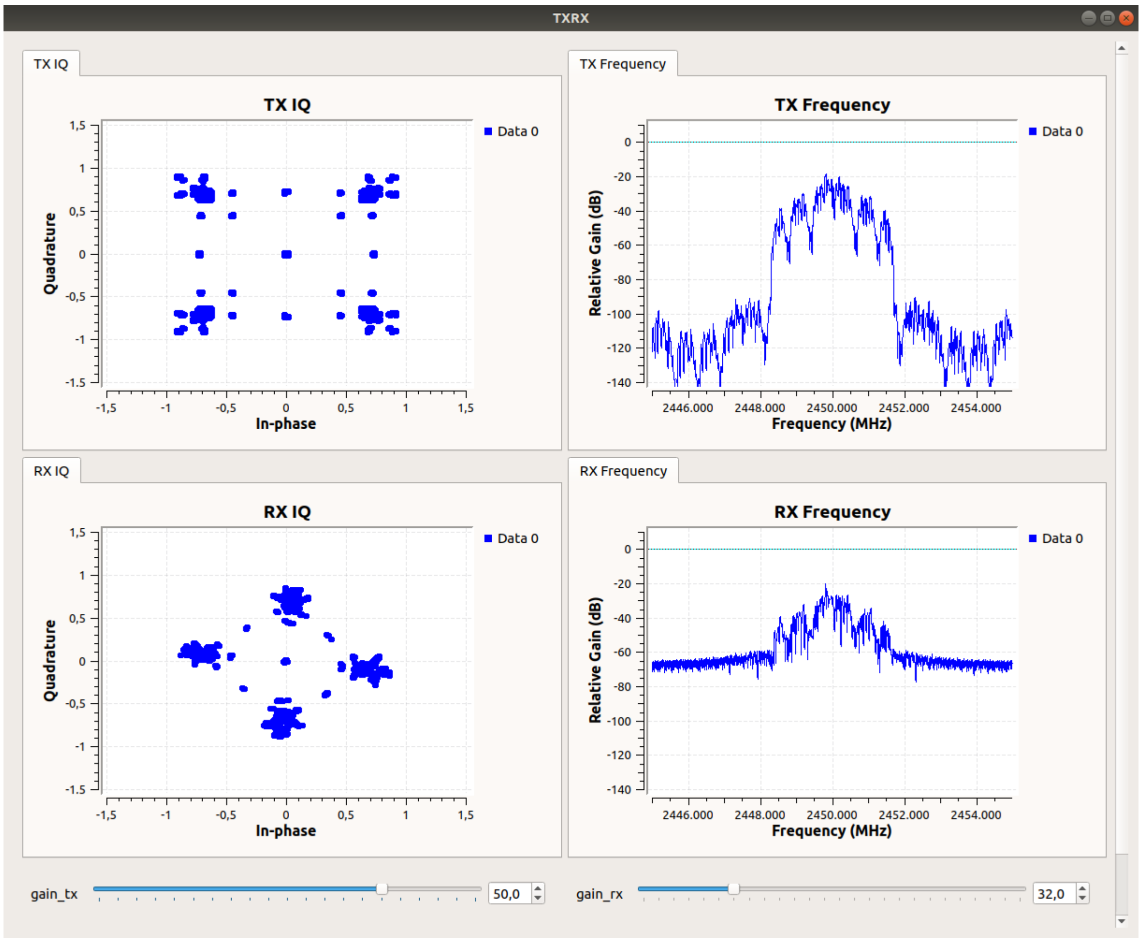


Figure 8.7: Results of the wired connection between TX and RX path.

The result of this is visible in Figure 8.7. Clearly visible is the modulated TX data in symbol plane and frequency domain. The received constellation still resembles the transmitted one, and although it's phase shifted in the diagram, this shift is in fact constant during execution of this flow graph.

8.3.2 Spectrum and Constellation

There are a few possibilities to compare the transmitted spectrum of the SDR. First of all, the modulated data was already illustrated using CEG, see Figure 7.2. Next, it's possible to evaluate the actual transmitted signal using appropriate measurement equipment. This measurement setup is illustrated in Figure 8.8. The host reads the modulated data and sends it via USB to the LimeSDR. From the analog transmission path, the signal gets wired into a **FSQ 20Hz-26.5GHz Spectrum Analyzer**, manufactured by **Rohde & Schwarz**.

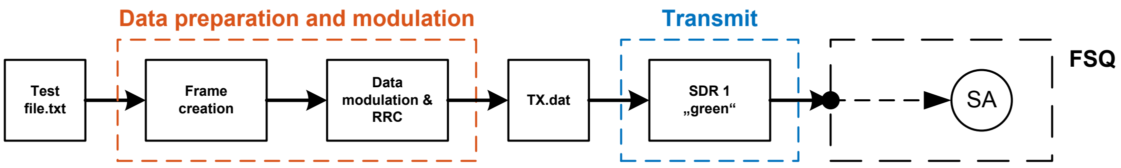


Figure 8.8: Measurement setup using an FSQ Spectrum Analyzer.

The *FSQ* is configured for a center frequency of *2.45GHz* using a span of *10MHz*, a resolution bandwidth of *100kHz* and a video bandwidth of *10kHz*. Resulting frequency domain signal, when setting the TX gain to the maximum value of *60*, is depicted in Figure 8.9. Clearly, the signal is the same as in Figure 8.7, and it also corresponds to the result using CEG in Figure 7.2. The only difference is in the bandwidth, which is in this case at around *3.5MHz*, while the maximum power is at *-20dB*.

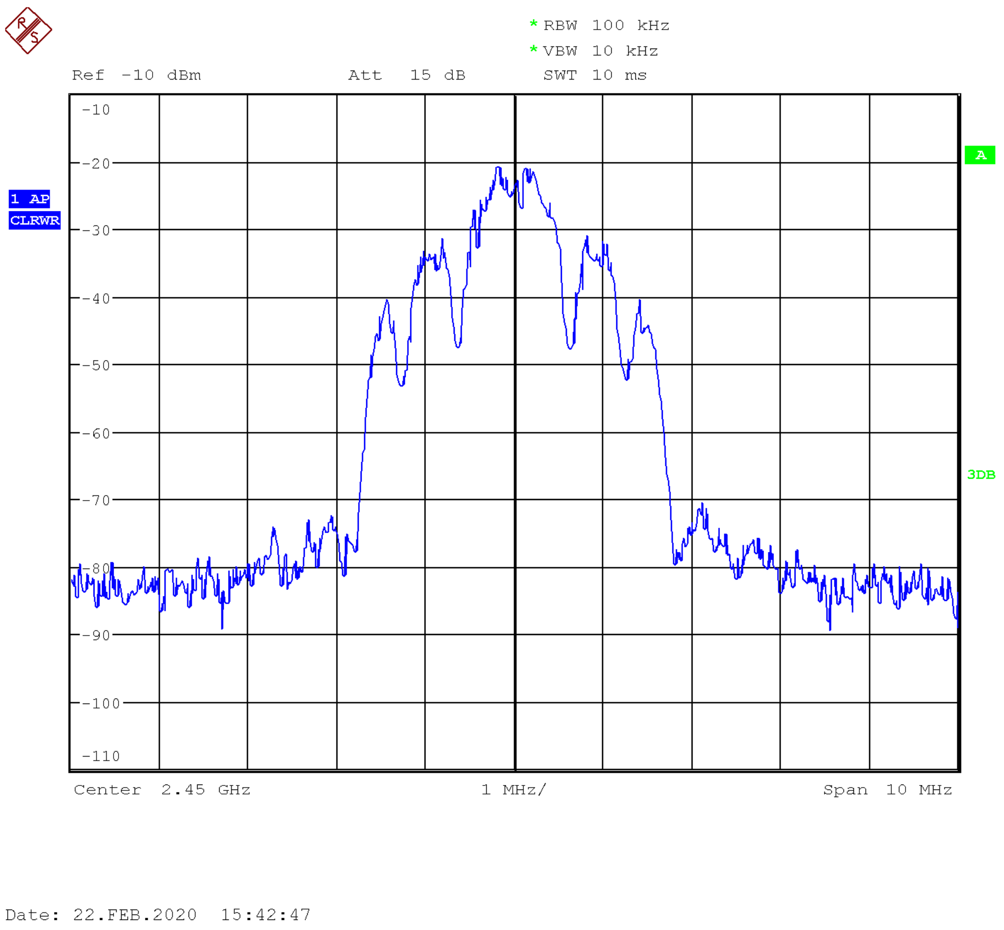
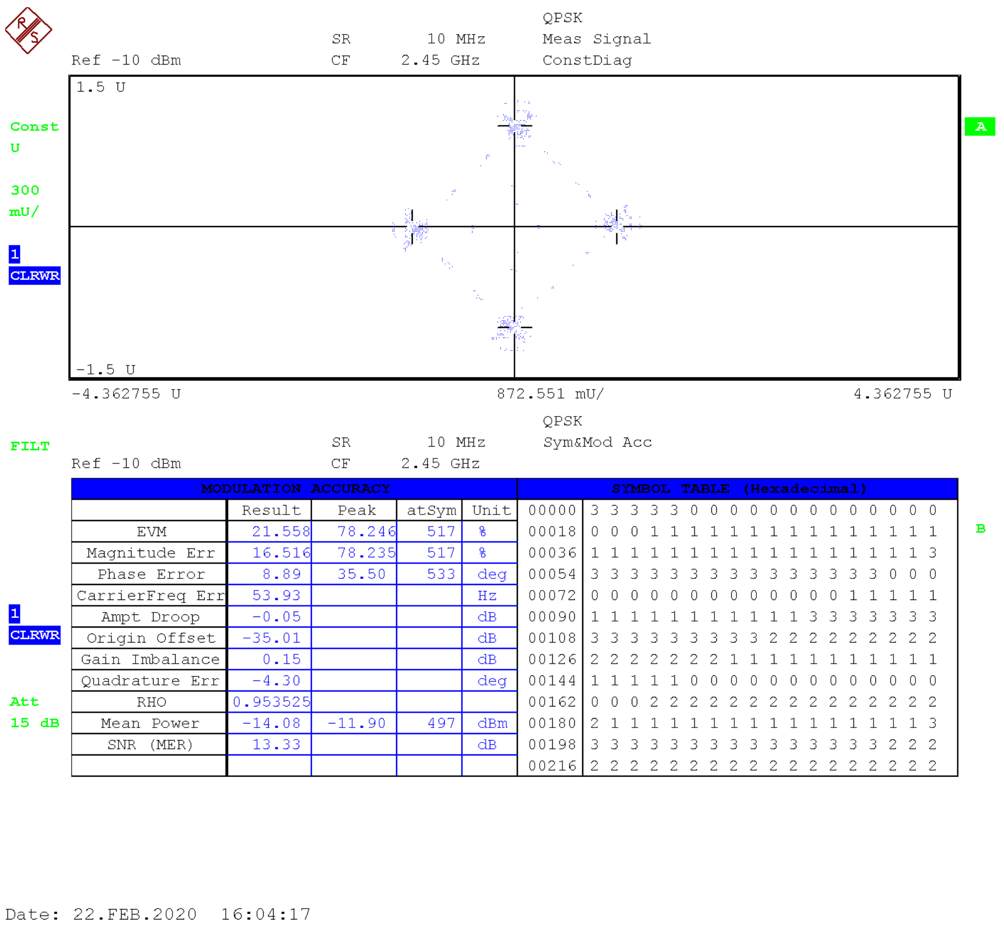


Figure 8.9: Transmitted spectrum from LimeSDR using upsampled QPSK.

Another possibility with the *FSQ* is to visualize the signal in the complex constellation domain. Figure 8.10 shows the constellation emitting from the transmitter at a *sample rate* of *10MHz* (the sample rate of the DACs). It clearly shows the correct constellation diagram, including the additional samples between the symbols. Since the *FSQ* does consider those samples for its measurement, the EVM (**E**rror **V**ector **M**agnitude, which is the summarized error magnitude between the measured samples and the optimal sample positions) value is very high at *21.5%*.



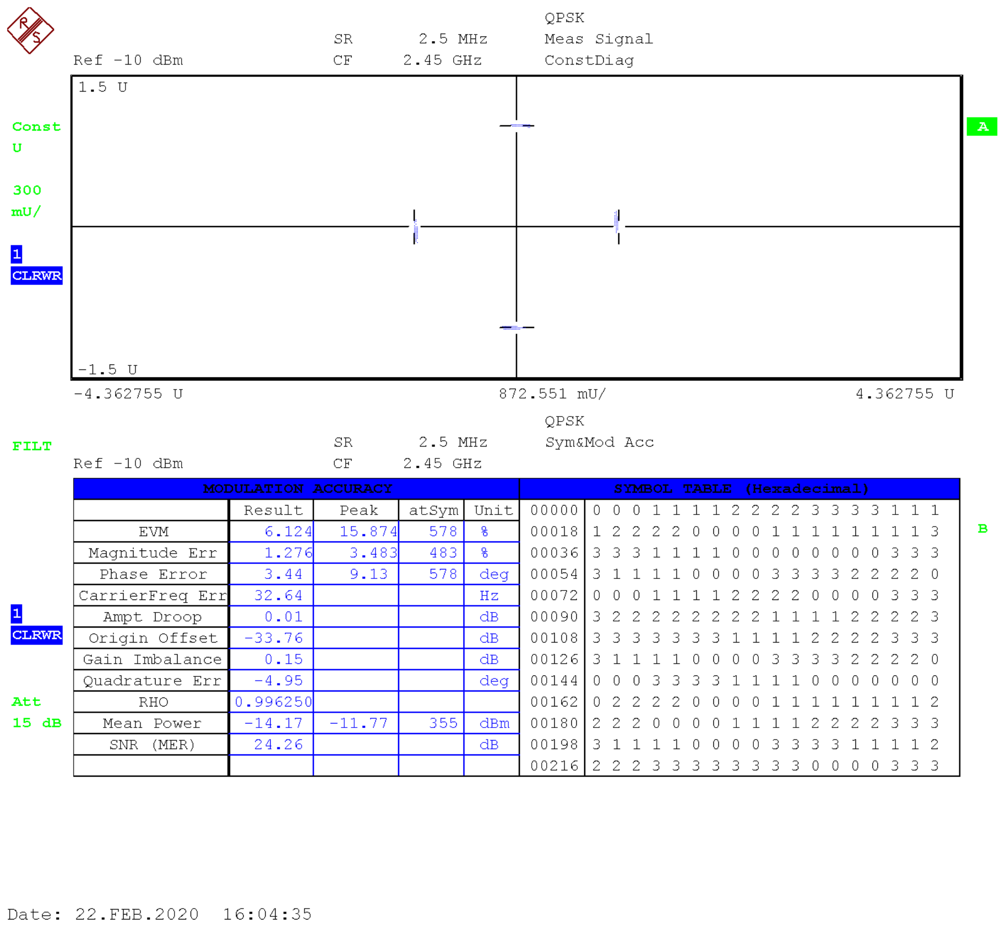
```
                          QPSK
          SR    10 MHz    Meas Signal
Ref -10 dBm   CF  2.45 GHz    ConstDiag

                          QPSK
          SR    10 MHz    Sym&Mod Acc
Ref -10 dBm   CF  2.45 GHz
```

MODULATION ACCURACY					SYMBOL TABLE (Hexadecimal)																		
	Result	Peak	atSym	Unit	00000	3	3	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0
EVM	21.558	78.246	517	%	00018	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Magnitude Err	16.516	78.235	517	%	00036	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3
Phase Error	8.89	35.50	533	deg	00054	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	0	0	0
CarrierFreq Err	53.93			Hz	00072	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
Ampt Droop	-0.05			dB	00090	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3
Origin Offset	-35.01			dB	00108	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2
Gain Imbalance	0.15			dB	00126	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1
Quadrature Err	-4.30			deg	00144	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
RHO	0.953525				00162	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Mean Power	-14.08	-11.90	497	dBm	00180	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3
SNR (MER)	13.33			dB	00198	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2
					00216	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Date: 22.FEB.2020 16:04:17

Figure 8.10: QPSK constellation data with the additional samples.

However, this can be configured to show the real symbols. The signal was upsampled by a factor of 4, so by reducing the *sample rate* to *2.5MHz* on the *FSQ*, the additional samples get removed and the raw QPSK data is left. Figure 8.11 shows the new constellation. Since the received samples now better match with the QPSK modulation scheme, the EVM reduces to a value of *6.1%*.



MODULATION ACCURACY				
	Result	Peak	atSym	Unit
EVM	6.124	15.874	578	%
Magnitude Err	1.276	3.483	483	%
Phase Error	3.44	9.13	578	deg
CarrierFreq Err	32.64			Hz
Ampt Droop	0.01			dB
Origin Offset	-33.76			dB
Gain Imbalance	0.15			dB
Quadrature Err	-4.95			deg
RHO	0.996250			
Mean Power	-14.17	-11.77	355	dBm
SNR (MER)	24.26			dB

SYMBOL TABLE (Hexadecimal)																		
00000	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	1	1	1
00018	1	2	2	2	2	0	0	0	0	1	1	1	1	1	1	1	1	3
00036	3	3	3	1	1	1	1	0	0	0	0	0	0	0	0	3	3	3
00054	3	1	1	1	1	0	0	0	0	3	3	3	3	2	2	2	2	0
00072	0	0	0	1	1	1	1	2	2	2	2	0	0	0	0	3	3	3
00090	3	2	2	2	2	2	2	2	2	1	1	1	1	2	2	2	2	3
00108	3	3	3	3	3	3	3	1	1	1	1	2	2	2	2	3	3	3
00126	3	1	1	1	1	0	0	0	0	3	3	3	3	2	2	2	2	0
00144	0	0	0	3	3	3	3	1	1	1	1	0	0	0	0	0	0	0
00162	0	2	2	2	2	0	0	0	0	1	1	1	1	1	1	1	1	2
00180	2	2	2	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3
00198	3	1	1	1	1	0	0	0	0	3	3	3	3	1	1	1	1	2
00216	2	2	2	3	3	3	3	3	3	3	3	0	0	0	0	3	3	3

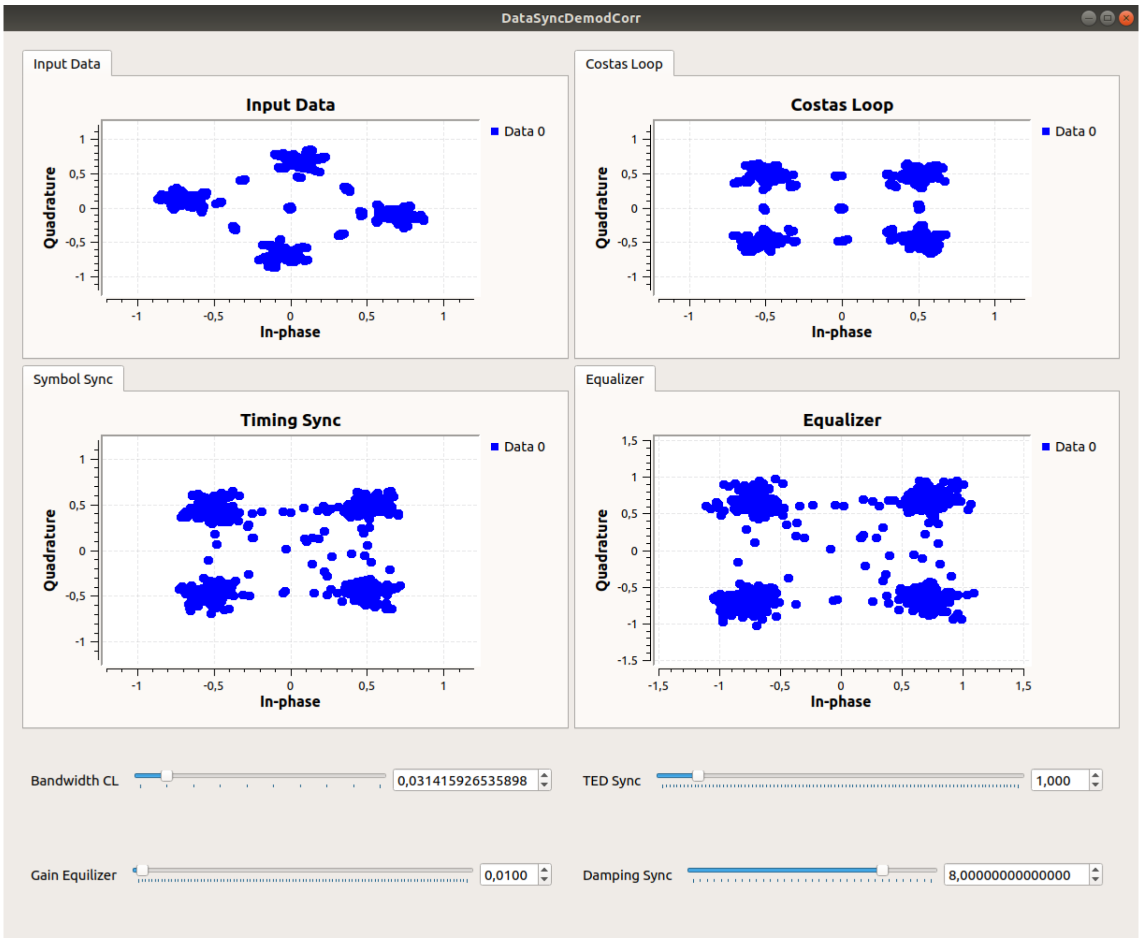
Date: 22.FEB.2020 16:04:35

Figure 8.11: QPSK constellation data without the additional samples.

8.3.3 Data Results

The received constellation data from the SDR is of course then saved into a separate file. This file then gets imported by the synchronization flow graph and further processed. Figure 8.12 shows the result of this operation. Clearly visible is the constant shift in the input data, and the operations of following blocks. Primarily this proves that the **Costas loop** successfully locks on the carrier mismatch, especially since this mismatch is constant, and that rotates the constellation. The following blocks don't have to perform a big amount of synchronization, yet the equalizer clearly can't match all samples correctly.

Therefore, other then in the previous section with the ideal settings, this time the data will not be perfectly synchronized. This becomes clear when expecting the resulting binary data information sink in Listing 8.6.



Bandwidth CL 0,031415926535898
TED Sync 1,000
Gain Equilizer 0,0100
Damping Sync 8,00000000000000

Figure 8.12: Results of the received samples when using only one LimeSDR as TX and RX simultaneously.

Some frames are synchronized very good, in this example the frame starting at address *cc*. Others start off correctly, but then the symbol recovery shifts, as can be seen with the frame at address *288*. There is still a repeating sequence of *X.Y.*, and when expecting it further, it's clear that this is bit shifted by 2 bit to the *cafe* sequence. However, some of the frames were synchronized partially wrong on a bit level, see address *a80* and the following lines of data.

```
mh@mh:~$xxd -b sync_out.bin
000000cc: 01101101 00000100 11100110 01010111 01010111 01010111  m..WWW
000000d2: 11001110 00101111 01100001 01100110 01100101 01100011  ./afec
000000d8: 01100001 01100110 01100101 01100011 01100001 00000000  afeca.
000000de: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
000000e4: 01100110 01100101 01100011 01100001 01100110 01100101  fecafe
000000ea: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
000000f0: 01100110 01100101 01100011 01100001 01100110 01100101  fecafe
000000f6: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
...
00000288: 01010111 01010111 01010111 01111110 00100010 01100101  WWW~"e
0000028e: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
00000294: 01100110 01100101 01100011 01100001 01100110 01100101  fecafe
```

```
0000029a: 01100011 01100001 01100110 01100101 01100011 01100001  cafeca
000002a0: 01100110 01100101 01100011 01100001 01100110 01100101  fecafe
000002a6: 01100011 01100001 01100110 01101100 11010101 11010101  cafl..
000002ac: 11010101 11100000 01001000 11011001 01011000 11011000  ..H.X.
000002b2: 01011001 10011001 01011000 11011000 01011001 10011001  Y.X.Y.
000002b8: 01011000 11011000 01011001 10011001 01011000 11011000  X.Y.X.
...
00000a80: 01010110 00110110 00000000 01010111 01010111 01010111  V6.WWW
00000a86: 01100000 10001101 10000101 10011001 10010101 10001101  `.....
00000a8c: 10000101 10011001 10010101 10001011 11001101 10000101  ......
00000a92: 10011001 10010101 00000000 00010110 01100110 01010110  ....fV
00000a98: 00000101 10001101 10000101 10011001 10010101 10001101  ......
...
```

Listing 8.6: Resulting information sink. First frame is synchronized perfectly, later ones only partially.

8.3.4 Other Test Cases

There are two additional test cases to evaluate: The *l* test file and the *random data* test file. Both of them had a very similar frequency representation to the *cafe* test file. However, since the random data has more possible symbol transitions, more frequency components are present. When comparing the following Figure 8.13 with the spectrum from Figure 8.9, a very small difference can be observed.

In detail, the change in the test files have no major impact on the communication, and the remaining results are the same. This is also observable when using the *l* test file. After the correlation, the binary data observed matches the results in the previous Listing 8.6. Some of the frames have no BER, others are only partially correct and show either bit shifts or that the data was synchronized wrong.

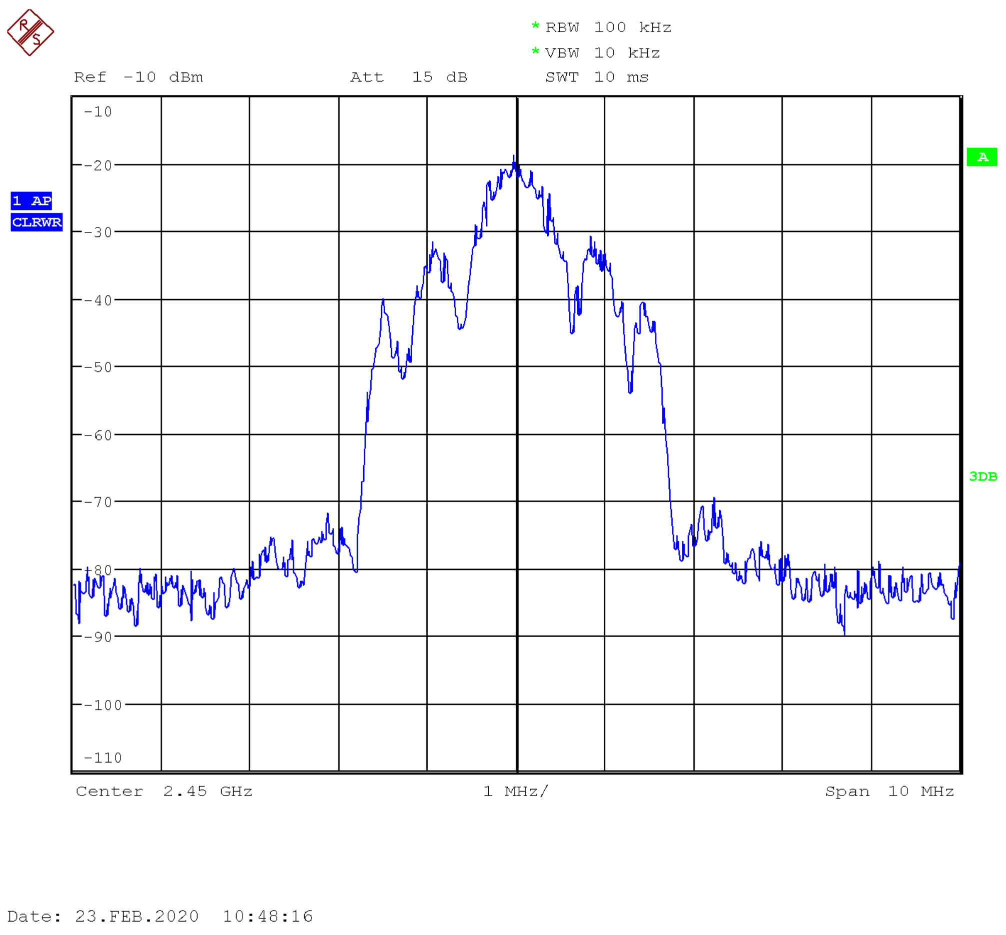
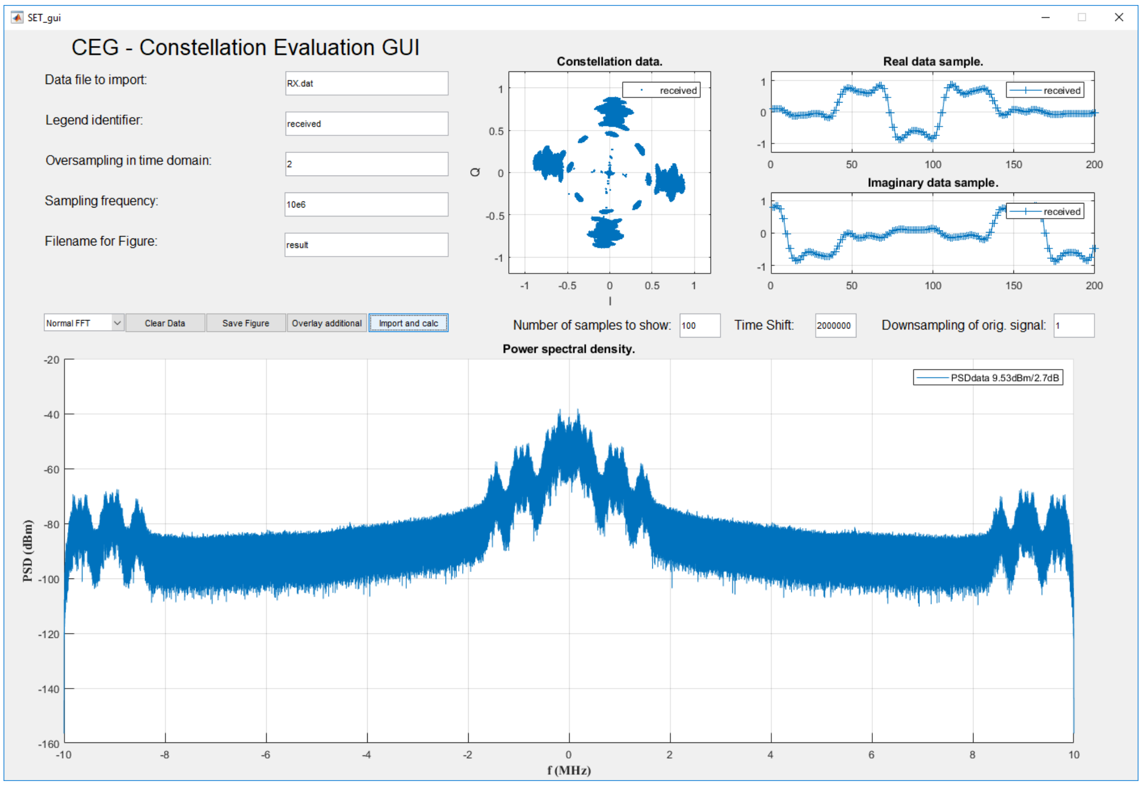


Figure 8.13: Transmitted spectrum from LimeSDR using QPSK and random data.

8.3.5 Comparison using CEG

By importing the received data into CEG, the constellation shows the same constant phase rotation as in the flow graph. Furthermore, the time domain data is very similar to what was seen before, for example by comparing it to Figures 6.6 and 7.2. The same applies to the frequency domain in the referenced Figures.


SET_gui

CEG - Constellation Evaluation GUI

Data file to import: RX.dat

Legend identifier: received

Oversampling in time domain: 2

Sampling frequency: 10e6

Filename for Figure: result

Normal FFT | Clear Data | Save Figure | Overlay additional | Import and calc

Number of samples to show: 100 Time Shift: 2000000 Downsampling of orig. signal: 1

Figure 8.14: Using CEG to illustrate the received data.

8.3.6 Comparison using Reference Vector Signal Generator

It's useful to to compare the constellation to that of a defined source. For this purpose, a **SMBV100A 9kHz-3.2GHz Vector Signal Generator** from **Rohde & Schwarz** was used to generate an optimal signal. The measurement setup is illustrated in Figure 8.15.

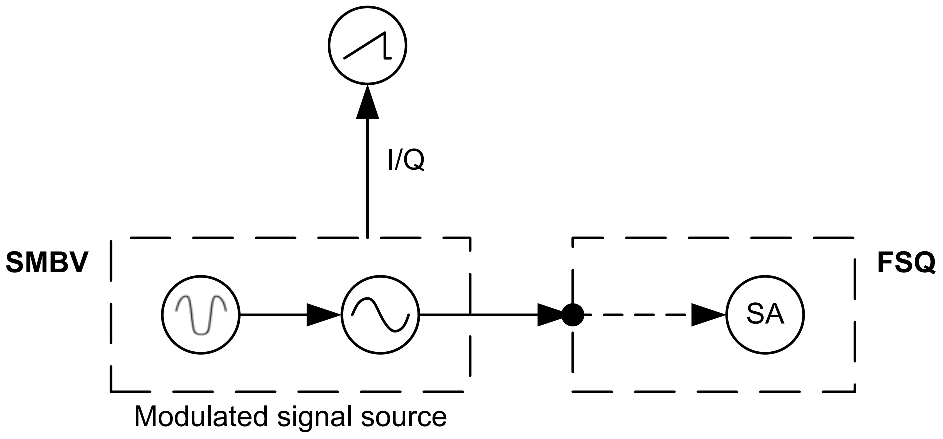
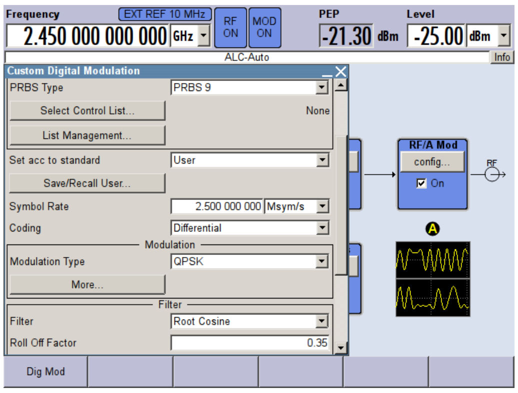


Figure 8.15: Measurement setup using a SMBV Vector Signal Generator, an FSQ Signal Analyzer and an oscilloscope.

The **SMBV** is configured to a frequency of *2.45GHz* and a power level of *-25dBm*.

Configuration of the *Custom Digital Modulation* is set to produce a *PRBS* (**P**seudo-**R**andom **B**inary **S**equence) with a symbol rate of *2.5Msym/s*. The option PRBS 9 generates a sequence of length $2^9 - 1$[31]. Additionally, to match the modulation scheme in the GNU Radio flow graphs, coding was set to *Differential* and modulation type to *QPSK*. This will generate optimal symbols. In order to further match the developed flow graph, the signal is filtered using a *Root Cosine* filter with a roll-off factor of 0.35. This specific configuration of the measurement instrument is also depicted in Figure 8.16.

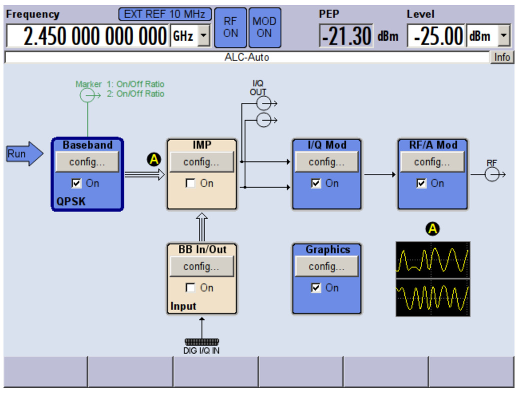


Frequency 2.450 000 000 000 GHz | EXT REF 10 MHz | RF ON | MOD ON | PEP -21.30 dBm | Level -25.00 dBm | ALC-Auto

Custom Digital Modulation

PRBS Type	PRBS 9
Select Control List...	None
List Management...	
Set acc to standard	User
Save/Recall User...	
Symbol Rate	2.500 000 000 Msym/s
Coding	Differential
Modulation Type	QPSK
More...	
Filter	Root Cosine
Roll Off Factor	0.35

(a) Modulation configuration.



Frequency 2.450 000 000 000 GHz | EXT REF 10 MHz | RF ON | MOD ON | PEP -21.30 dBm | Level -25.00 dBm | ALC-Auto

(b) General configuration.

Figure 8.16: SMBV configuration for PRBS generation.

The signal in frequency domain is measured using the *FSQ*, see Figure 8.17. This is obviously not the same signal as previously used in the measurement using a LimeSDR. The difference is clear: This signal has no upsampling in contrast to the modulation process using GNU Radio.

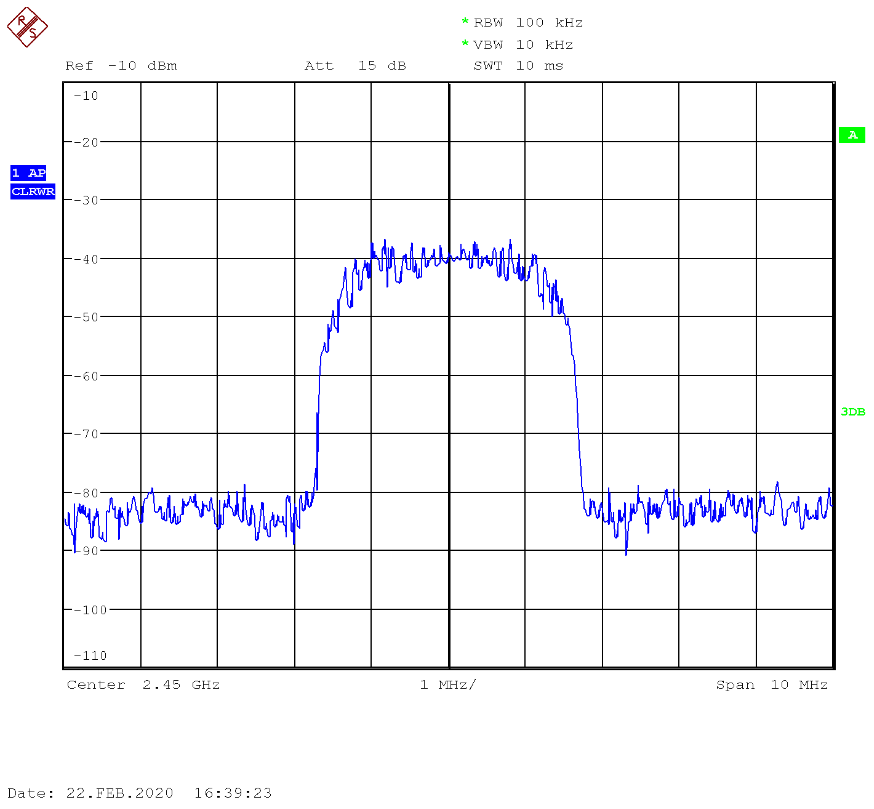
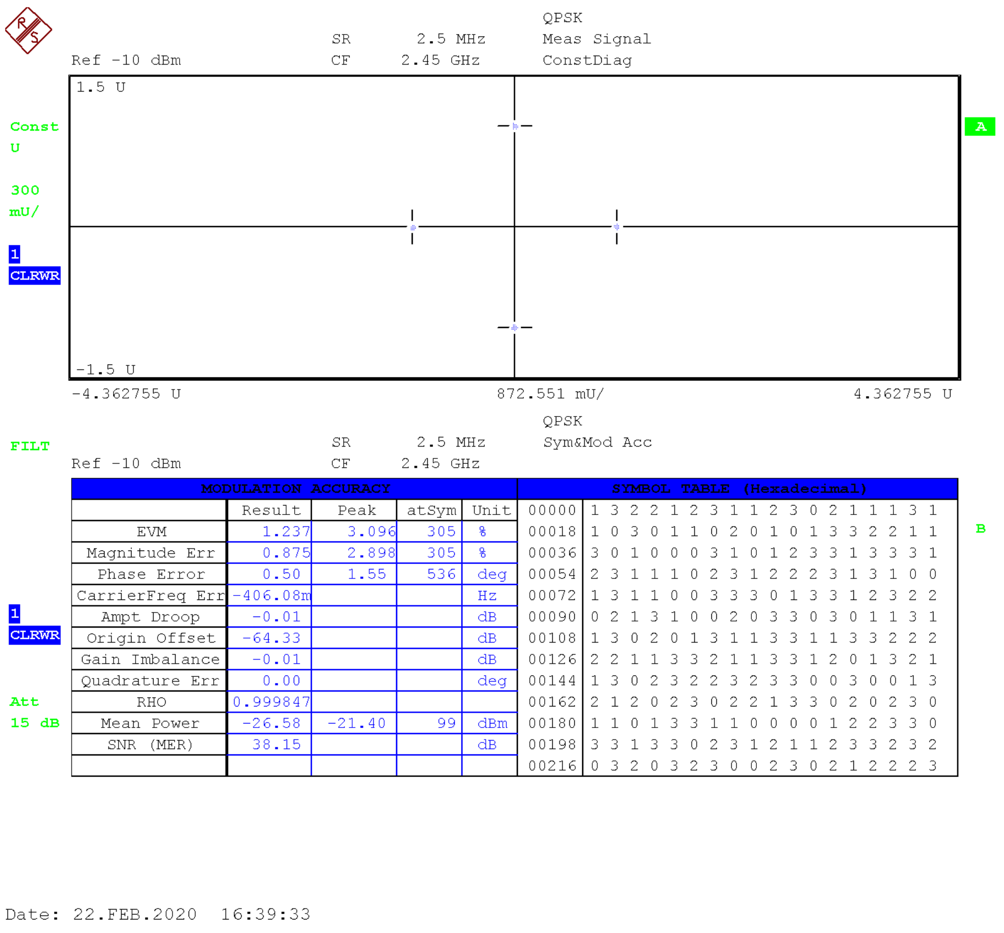


Figure 8.17: PRBS signal from SMBV measured with FSQ.

Additionally, the constellation data can be compared. Figure 8.18 illustrates the same PRBS data in complex symbol domain. By comparing it to the transmitted constellation of the SDR in Figure 8.11, the main difference is the better EVM value of around *1.2%*.

To examine the generated signal in time domain, the *SMBV* can output real and complex signals respectively. These ports are connected to a **HMO3002 4GSa/s Oscilloscope** by **Rohde & Schwarz**. As an example, triggered at a random point, Figure 8.19 illustrates the signal sequence on the oscilloscope. Channel 1 is the In-phase signal and Channel 2 shows the Quadrature sequence. The values are clearly filtered by a RC filter. They also resemble the time domain representation of the modulated data using GNU Radio in Figure 6.6 and using CEG in Figure 7.2.



```
                              SR       2.5 MHz      QPSK
                                                    Meas Signal
Ref -10 dBm                   CF      2.45 GHz      ConstDiag

                              SR       2.5 MHz      QPSK
Ref -10 dBm                   CF      2.45 GHz      Sym&Mod Acc
```

MODULATION ACCURACY				
	Result	Peak	atSym	Unit
EVM	1.237	3.096	305	%
Magnitude Err	0.875	2.898	305	%
Phase Error	0.50	1.55	536	deg
CarrierFreq Err	-406.08m			Hz
Ampt Droop	-0.01			dB
Origin Offset	-64.33			dB
Gain Imbalance	-0.01			dB
Quadrature Err	0.00			deg
RHO	0.999847			
Mean Power	-26.58	-21.40	99	dBm
SNR (MER)	38.15			dB

SYMBOL TABLE (Hexadecimal)																		
00000	1	3	2	2	1	2	3	1	1	2	3	0	2	1	1	1	3	1
00018	1	0	3	0	1	1	0	2	0	1	0	1	3	3	2	2	1	1
00036	3	0	1	0	0	0	3	1	0	1	2	3	3	1	3	3	3	1
00054	2	3	1	1	1	0	2	3	1	2	2	2	3	1	3	1	0	0
00072	1	3	1	1	0	0	3	3	3	0	1	3	3	1	2	3	2	2
00090	0	2	1	3	1	0	0	2	0	3	3	0	3	0	1	1	3	1
00108	1	3	0	2	0	1	3	1	1	3	3	1	1	3	3	2	2	2
00126	2	2	1	1	3	3	2	1	1	3	3	1	2	0	1	3	2	1
00144	1	3	0	2	3	2	2	3	2	3	3	0	0	3	0	0	1	3
00162	2	1	2	0	2	3	0	2	2	1	3	3	0	2	0	2	3	0
00180	1	1	0	1	3	3	1	1	0	0	0	0	1	2	2	3	3	0
00198	3	3	1	3	3	0	2	3	1	2	1	1	2	3	3	2	3	2
00216	0	3	2	0	3	2	3	0	0	2	3	0	2	1	2	2	2	3

```
Date: 22.FEB.2020  16:39:33
```

Figure 8.18: PRBS constellation from SMBV measured with FSQ.

8.4 Two Devices with Wired Connection

The previous section showed that the system functions correctly when using only one LimeSDR. This simplified the constellation data, since there may have been a phase offset, but the carrier frequency was already locked. The next logical step is to use two LimeSDRs, and to connect them via a wired line. Figure 8.20 illustrates this concept. Advantage of this is that there are not many losses in the channel.

The flow graph configurations stay the same. Only the AGC value in the transmit and receive flow graph needs to be adjusted. In this case, it was changed to a value of *8*.

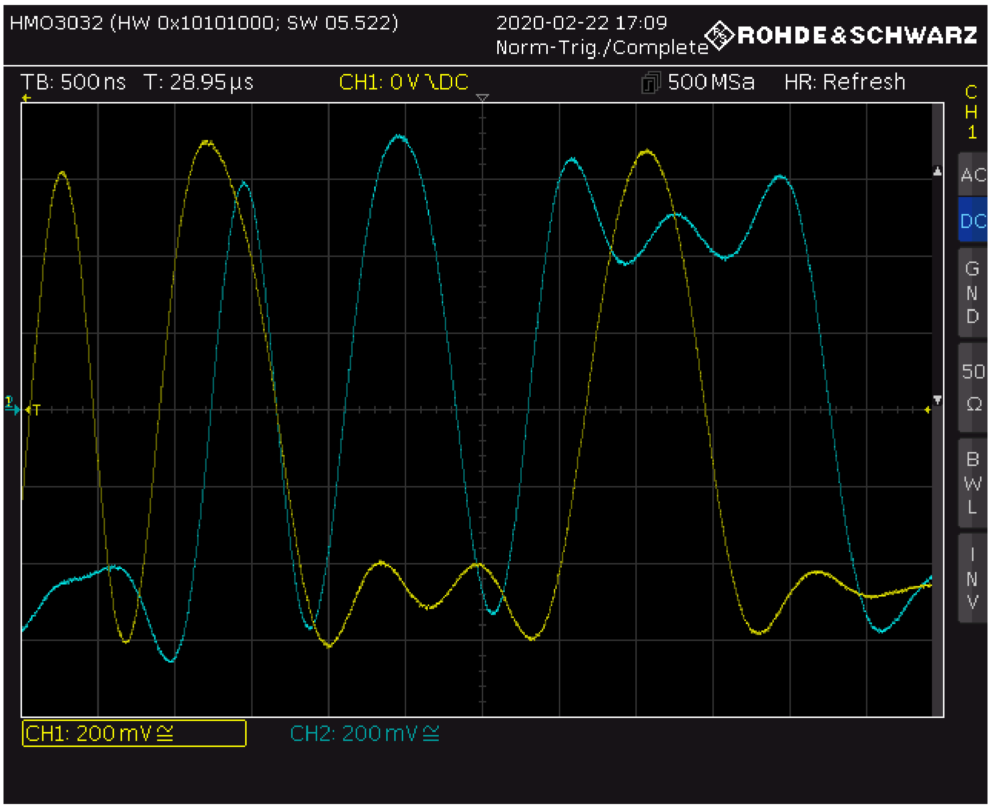


Figure 8.19: Example data from oscilloscope measurement.

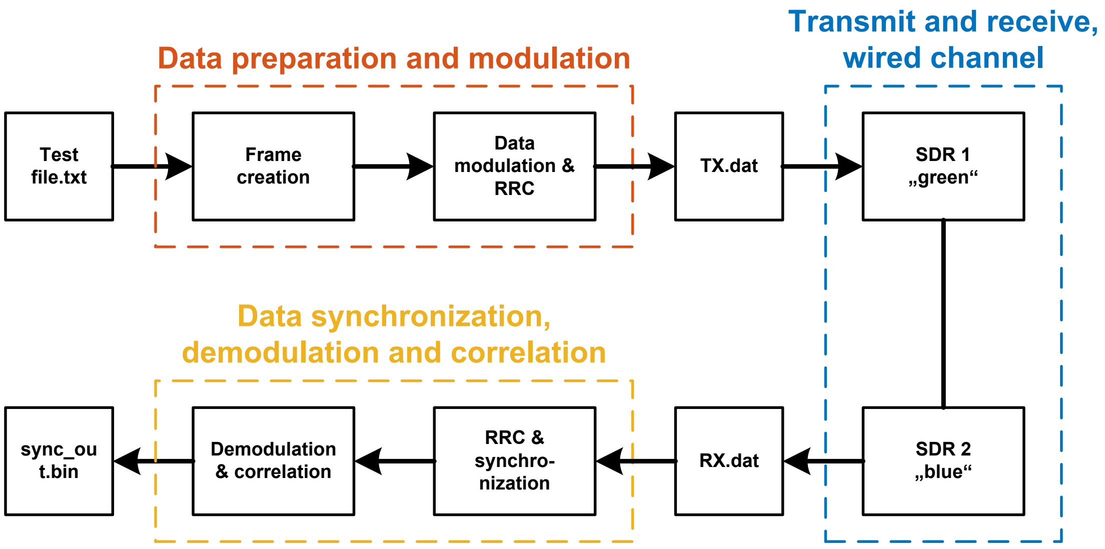

Data preparation and modulation
Transmit and receive, wired channel
Test file.txt
Frame creation
Data modulation & RRC
TX.dat
SDR 1 „green“
Data synchronization, demodulation and correlation
sync_out.bin
Demodulation & correlation
RRC & synchronization
RX.dat
SDR 2 „blue“

Figure 8.20: System setup with two devices connected via cable.

The result from executing this flow graph is illustrated in Figure 8.21. Clearly visible while the measurement was running were three effects:
First, there was a visible carrier frequency drift. This means that the constellation was rotating around the center point.

Second, there were symbol timing problems, resulting in a ”pulsating” constellation. This meant that since the symbol timing was incorrect, the symbols were no longer sampled in their perfect timing, but rather the position of the symbol timing was constantly changing. And finally third, there were visible dropouts during the communication. This meant that the received constellation either was fully zero, or it maximized to the outer possible value. In this case, the maximum value was at 1, either in the real axis or the imaginary axis.

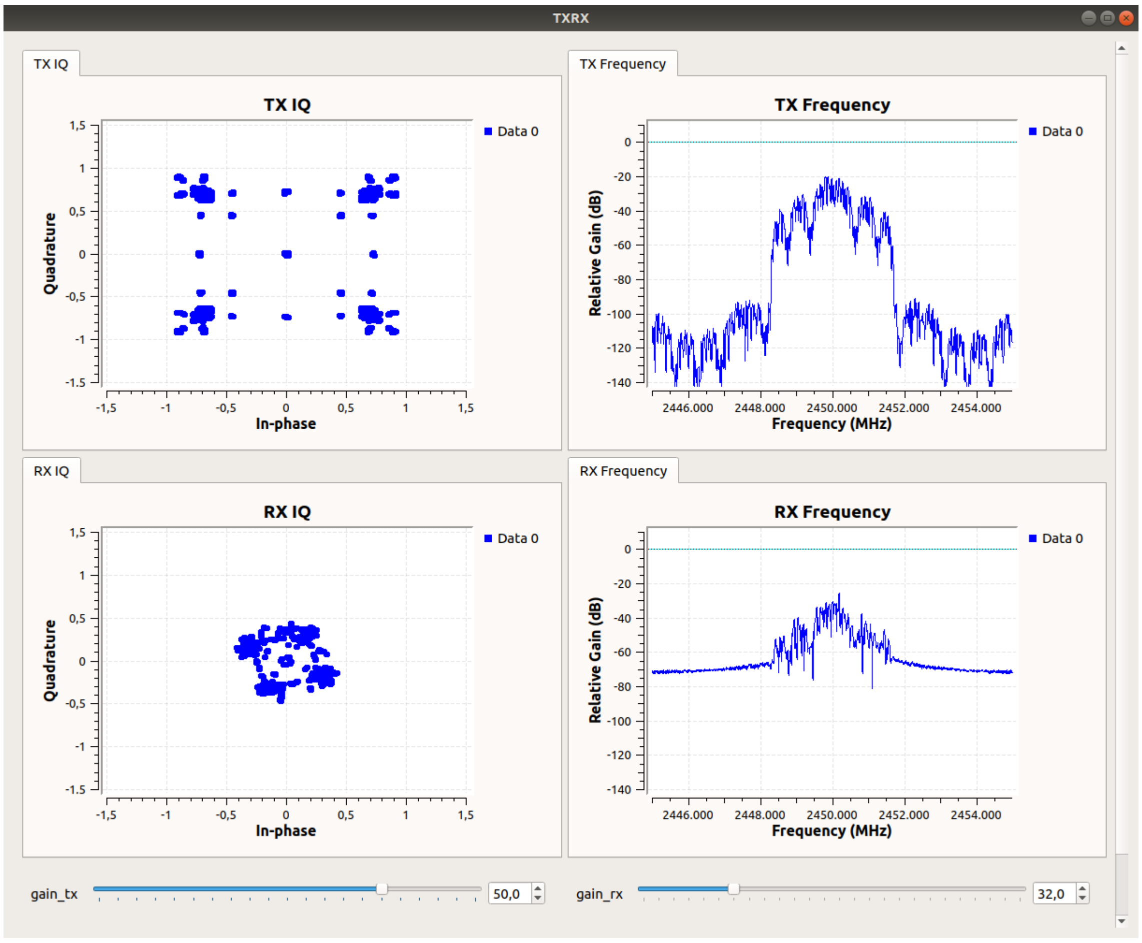
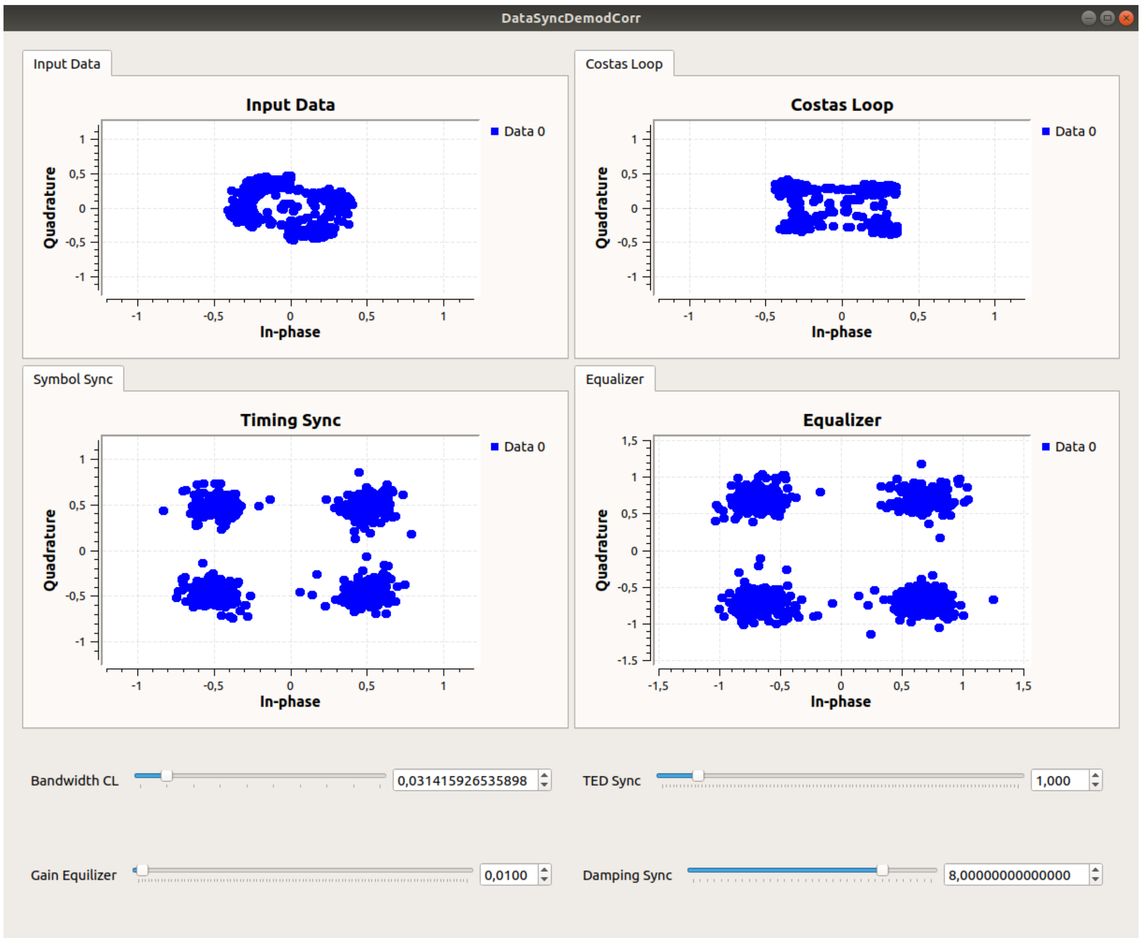


Figure 8.21: Results of the flow graph when transmitting data between two LimeSDR via a wired connection.

Figure 8.22 shows how this data was synchronized with the following flow graph. The **Costas Loop** successfully locked to the carrier drift. Afterwards, the signal was downsampled and the symbol timing recovery algorithm managed timing synchronization. Finally, the equalizer matched the data to the ideal symbol points. After evaluation of the correlated binary data on the host, it was clear that the BER did not worsen compared to the previous section.

However, there was still the problem with the mentioned dropouts. Whenever one of these occurred, the whole synchronization chain went out of sync, and had to first lock again once the constellation was correct again. This meant a lot of time, were only noise was processed. Reasons for this dropout were possible overloads on the analog receiving path and short time termination of the signal from the transmitter.



Bandwidth CL 0,031415926535898

TED Sync 1,000

Gain Equilizer 0,0100

Damping Sync 8,00000000000000

Figure 8.22: Results of the synchronization when transmitting data between two LimeSDR via a wired connection.

When using one of the other test cases, the results were again very similar.

8.5 Actual Free-Space Transmission

Now that the system proved to function correctly when using a wired connections, the final step is to evaluate an actual free-space transmission. For this purpose, the only change to the previous section is that two of the antennas characterized in section 3.5 are used instead of a cable. Therefore, the system setup modifies to illustration 8.23.

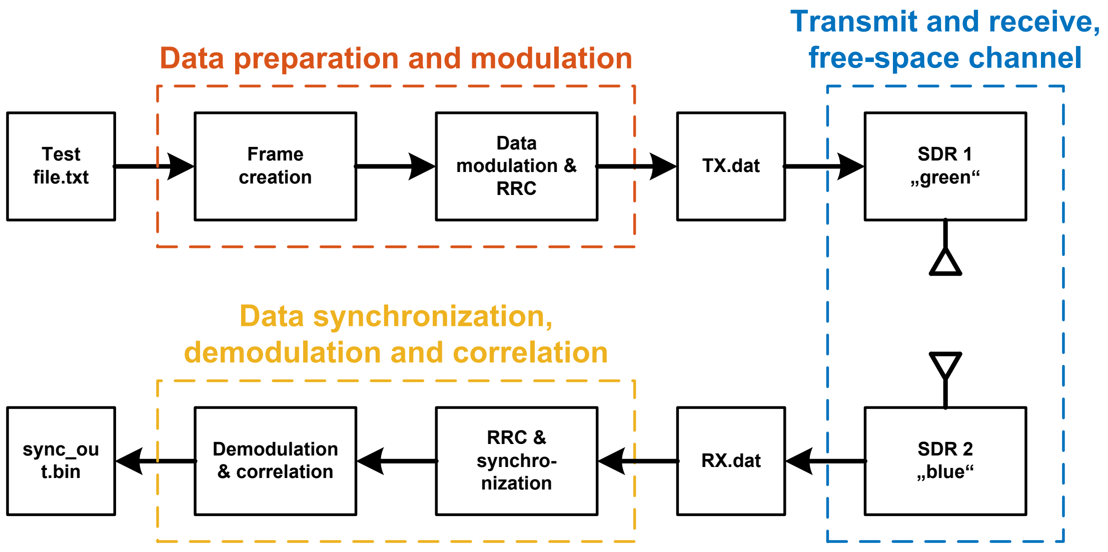

Data preparation and modulation
Transmit and receive, free-space channel
Test file.txt
Frame creation
Data modulation & RRC
TX.dat
SDR 1 „green“
Data synchronization, demodulation and correlation
sync_out.bin
Demodulation & correlation
RRC & synchro-nization
RX.dat
SDR 2 „blue“

Figure 8.23: System setup with two devices connected via a free-space transmission.

Since the antennas are not optimally matched (between *-7dB* and *-9dB* in this frequency range) and the LimeSDR's final power amplifier stage can't output large amounts of power, the received signal is assumed to have a very small power level. This was also discussed in the link budget section 2.5.2. All the additional loses, especially over the free-space channel, lower the signal power. Because of these reasons, the gain on the transmitting LimeSDR was set to maximum, while the sensitivity on the receiving SDR was set to maximum as well. Regarding the AGC value, a good value for a wireless distance of around *20cm* is to set it to *8*.

Figure 8.24 illustrates the received symbol data for this configuration. Similar to the wired connection between both SDRs, this free-space transmission showed a carrier drift and less then optimal symbol timing. However, the main difference was an increase in noise power. Note also, that when the distance between the devices was increased, the signal drastically deteriorated. This was mainly due to the previously discussed low output power from the SDR. In order to increase the distance where stable communications is still possible, either the output power must be increased using a power amplifier, or the antenna gain must be optimized in the direction of communication. This antenna gain could be increased by developing directional beam antennas.

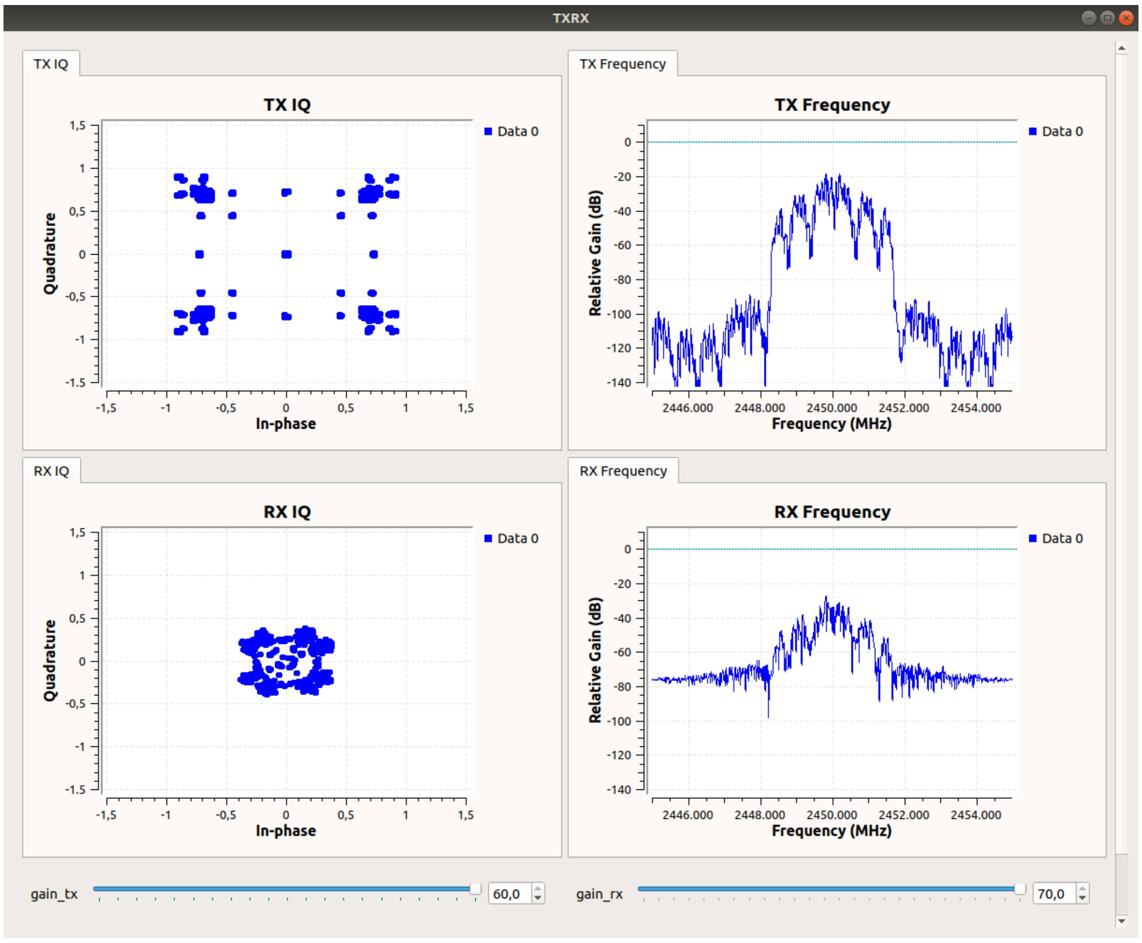
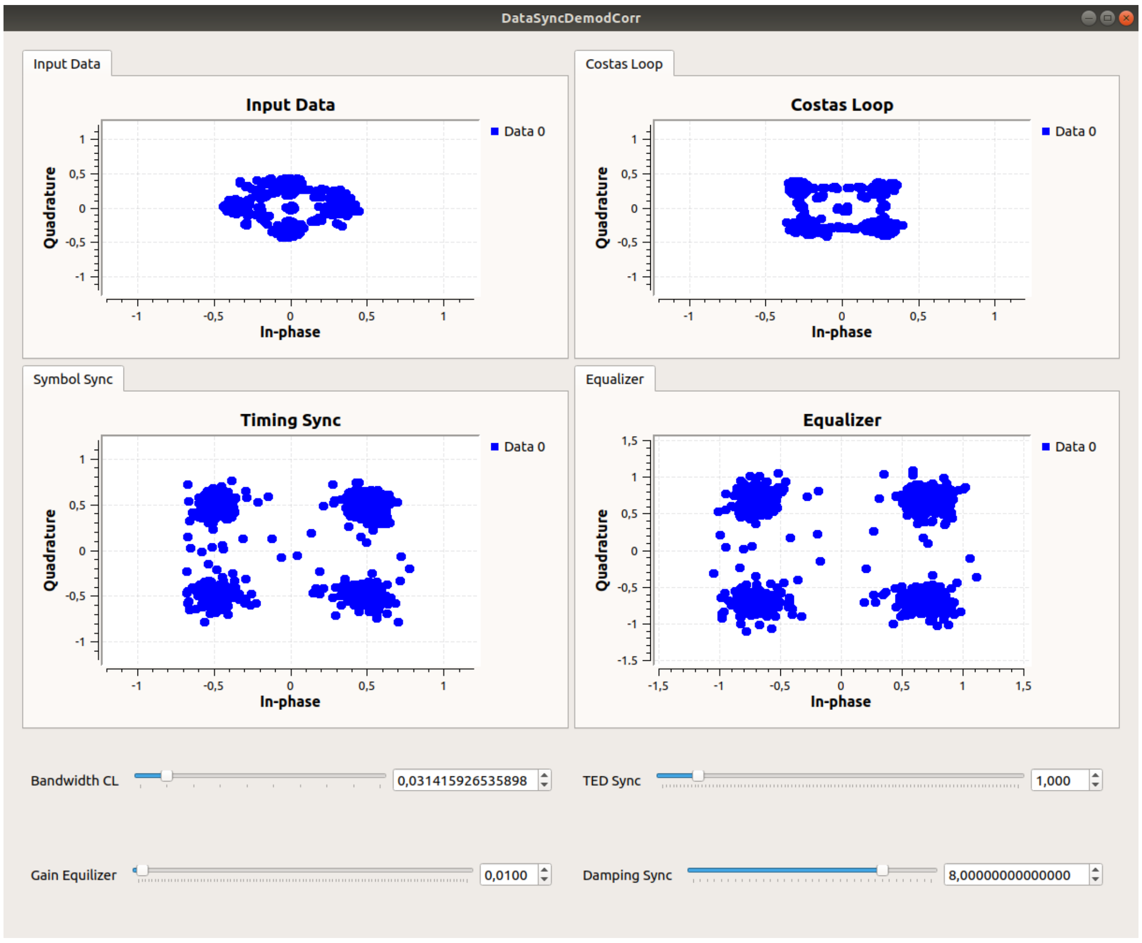


Figure 8.24: Results of the flow graph when transmitting data between two LimeSDR via a free-space connection.

Another visible effect in the received constellation diagram was that the QPSK symbols were divided. It seemed as if the ideal positions were split into two positions. This was either a symbol timing problem, or the representation in GNU Radio overlapped with two different instances of symbol timing.

Clearly the signal propagating through the channel experienced additional noise power. Even though the free-space distance was kept rather short, fading due to multipath propagation is still a problem. Additionally, the mentioned dropouts in received signal power were occurring as well. All these factors together resulted in a higher BER in the correlated binary sink data file, but it still contained frames which were perfectly synchronized.

Figure 8.25 illustrates the synchronization operation. Again it successfully displays QPSK constellation after equalization.



Bandwidth CL 0,031415926535898

TED Sync 1,000

Gain Equilizer 0,0100

Damping Sync 8,00000000000000

Figure 8.25: Results of the synchronization when transmitting data between two LimeSDR via a free-space connection.

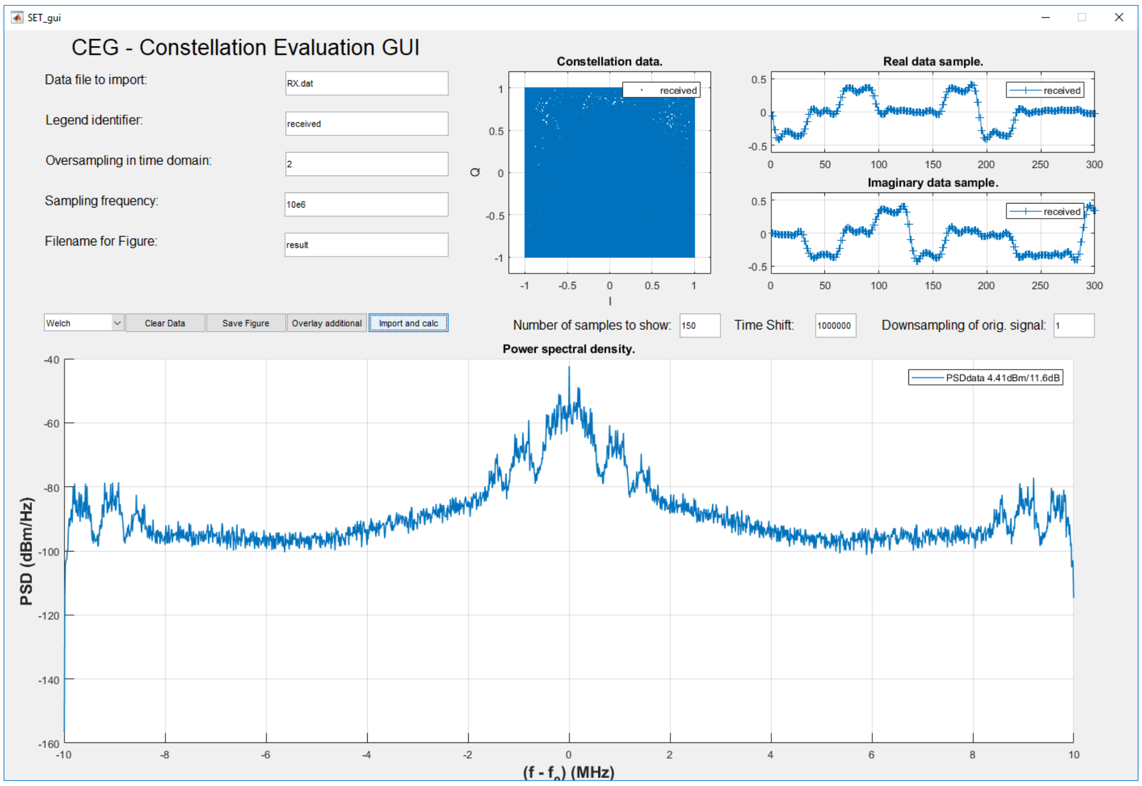
Same as with the previous sections, varying the test data file did neither improve nor worsen the system.

8.6 Synchronization Chain

After developing this system, it was very clear that the most crucial part in the flow graphs is the RX synchronization chain. Section 2.6 documented all possible synchronization techniques in the **Symbol Sync** block. However, it is interesting to see how the frequency domain representation changes between the different steps, as well as comparing different symbol recovery algorithms. Until now, only Mueller and Müller symbol timing recovery was used, with a roll-off factor of *0.35* for the RRC filter.

8.6.1 Evaluation using CEG

For now the system is configured the same as in the free-space transmission. Figure 8.26 illustrates the received data using CEG. The constellation diagram is filled with samples, since the constellation data both spins around the center point and "pulses" because of wrong symbol timing. Time domain data is as expected and can be compared to previous simulations, and to the measurement result from the oscilloscope, see Figure 8.19. There are again visible symbol timing problems. As for the frequency domain, the Welch's method was used to represent it, and it's also very similar to the spectrum in the results of the synchronization flow graph.


SET_gui

CEG - Constellation Evaluation GUI

Data file to import: RX.dat

Legend identifier: received

Oversampling in time domain: 2

Sampling frequency: 10e6

Filename for Figure: result

Number of samples to show: 150

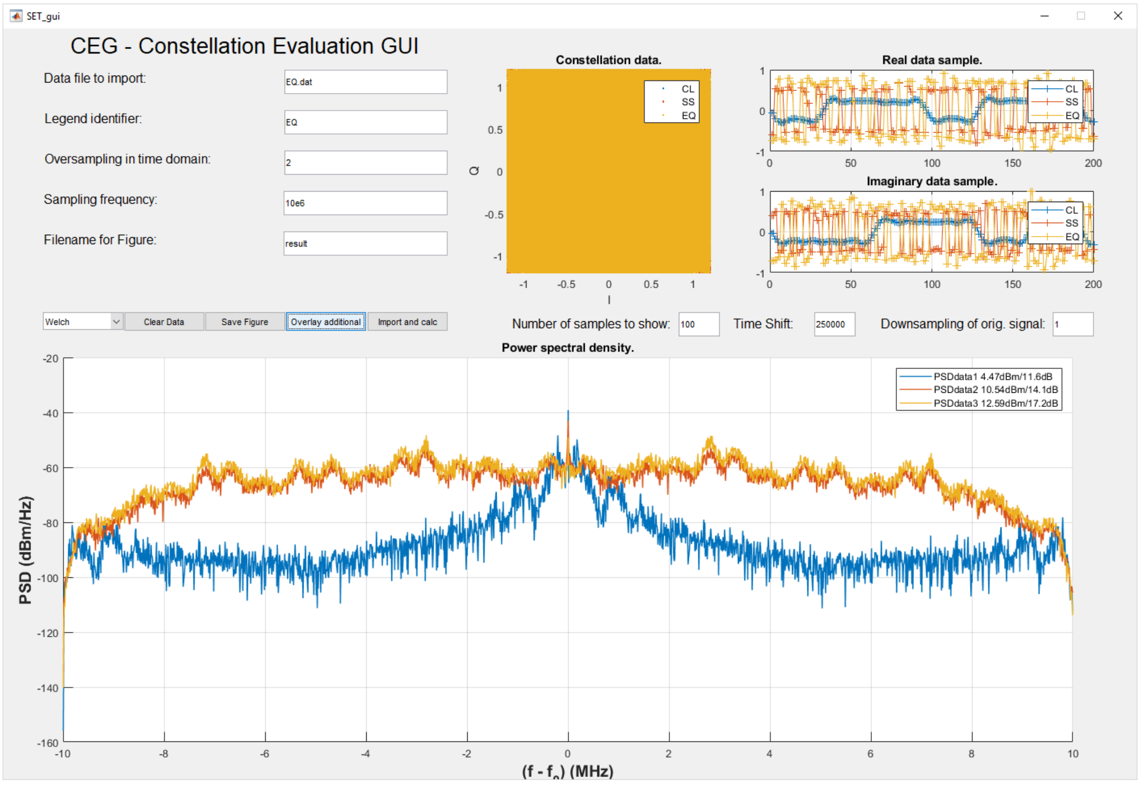
Time Shift: 1000000

Downsampling of orig. signal: 1

Figure 8.26: Using CEG to visualize the received symbol data.

In contrast, Figure 8.27 illustrates the different synchronization stages. Unfortunately, the constellation diagram contains to many data to separately visualize the symbols. The time domain representation illustrates that the processed data from the **Costas loop** is as expected a typical upsampled and RRC filtered signal. Before the **Symbol Sync** block, the signal gets downsampled and filtered. Therefore, its time domain representation contains 4 times more samples then the one of the Costas loop. The **Equalizer** matches the symbols to the expected constellation points.

Regarding the frequency domain, the spectrum of the Costas loop does not change significantly compared to Figure 8.26. However, the downsampling operation of the decimating RRC filter is clearly visible in the spectrum after the symbol timing recovery. The signal gets multiplied over the spectrum and appears 4 times and the overall PSD is higher. Finally, the spectrum after equalization does not change significantly. It only increases the PSD by a value of around *2dB*.



SET_gui

CEG - Constellation Evaluation GUI

Data file to import: EQ.dat

Legend identifier: EQ

Oversampling in time domain: 2

Sampling frequency: 10e6

Filename for Figure: result

Number of samples to show: 100

Time Shift: 250000

Downsampling of orig. signal: 1

Figure 8.27: Using CEG to visualize the synchronization chain.

8.6.2 Varying Symbol Timing Recovery Algorithm

Following is a comparison between different symbol timing recovery algorithms at different roll-off factors. As mentioned in section 6.4.2, the symbol timing recovery block implements different symbol timing algorithms. This includes the two decision-directed ones (Mueller and Müller, Zero Crossing) and the non-data-aided (Gardner, Early-Late, sgn(y[n]y'[n]) Maximum Likelihood). All of these were described in subsection 2.6.1.

Previous sections used Mueller and Müller symbol timing recovery. The RRC filter had a roll-off factor of *0.35*. In order to compare the symbol recovery algorithms, the **Data Rate Eval** block described in section 5.3 was used. This block prints the effective BER of the system into console, which makes it possible to draw a conclusion between the different algorithms.

Roll-off factor 0.35

The comparison was conducted via the free-space transmission over a short distance with the previous configuration. Mueller and Müller symbol timing was used as a reference, and the relative BER deviation to this algorithm was measured for the other symbol timing recovery algorithms. Table 8.1 contains the result of these simulations.

Symbol timing algortihm	**Relative BER deviation to M&M**
	$BER{-}BER_{MM}$
Zero Crossing	0.76%
Gardner	0.62%
Early-Late	0.10%
sgn(y[n]y'[n]) Maximum Likelihood	-1,32%

Table 8.1: Relative BER deviation between symbol timing algorithms, $\beta = 0.35$.

As expected, the maximum likelihood algorithm performs best when compared to the other symbol timing recoveries, even though it is not decision-directed. The roll-off factor is rather low, and therefore, the Mueller and Müller algorithm performs better then the other algorithms[12].

Roll-off factor 1.00

This measurement can be repeated with a roll-off factor of 1, which means that the modulation block doubles the bandwidth, see also subsection 2.3.3. Of course this also changes the transmitted signal from the SDR. Figure 8.28 illustrates the measured frequency spectrum when using a roll-off factor of 1. When comparing this to the previous measurement in Figure 8.9, the spectrum is clearly wider.

Regarding the EVM at a sample rate of *2.5MHz*, the value rises to about *6.9%* compared to the *6.1%* when using a roll-off factor of 0.35. The fact that this value gets worse matches with ideal measurements: By connecting the *SMBV* to the *FSQ*, the EVM changes from *1.2%* (*0.35* roll-off factor) to *5.9%* (*1.00* roll-off factor). The reason why this change is so minor using the LimeSDR is that this signal is upsampled, which means it cannot be compared directly, but the negative trend is still visible.

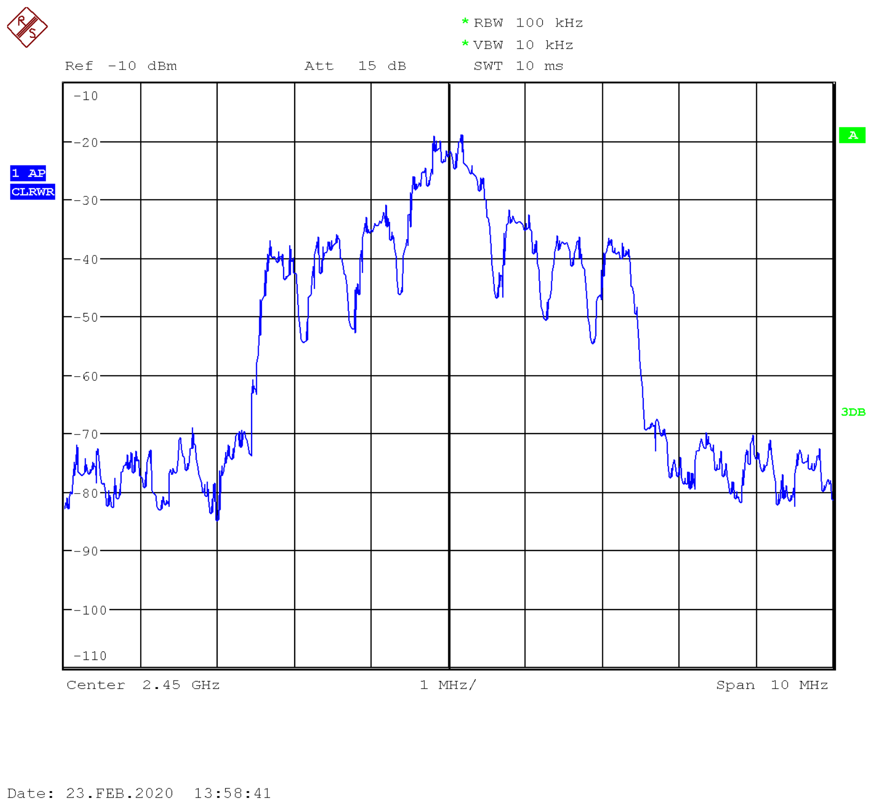


Figure 8.28: Upsampled and filtered QPSK frequency spectrum when using a roll-off factor of 1.

It is critical to increase the bandwidth of the receiving device to account for the additional transmitted bandwidth. Comparing the relative deviation of BER in Table 8.2, the result matches with literature references[12]. Since the roll-off factor is higher, the Mueller and Müller symbol recovery algorithm performs not as good as previously. Again, the best algorithm is the maximum likelihood.

Symbol timing algortihm	**Relative BER deviation to M&M**
	$BER - BER_{MM}$
Zero Crossing	-1.32%
Gardner	-0.90%
Early-Late	-1.60%
sgn(y[n]y'[n]) Maximum Likelihood	-1.83%

Table 8.2: Relative BER deviation between symbol timing algorithms, $\beta = 1.00$.

Chapter 9

Conclusion and Outlook

In this thesis, the applicability of a digital communication system was proven using software-defined radios. The theoretical chapter covered all necessary topics, supported by literature references. An overview of the system was discussed next. The ISM frequency band of *2.4GHz* was chosen, and all transmissions in the thesis were conducted at a carrier frequency of *2.45GHz*. A console based user program was developed, which was able to configure LimeSDR devices and start a simple transmission. Although this free-space transmission was successful, the received signal experienced both carrier frequency drift and symbol timing errors. To counter this problem, focus was switched to GNU Radio, an open source environment specializing on digital signal processing. The developed flow graph for the communication system was then divided into three parts. This gave the opportunity to evaluate every part for itself, and it also eased the workload on the host PC. To support the development and to visualize the results in different stages of the flow graphs, a MATLAB tool was created. And finally, the system was evaluated step by step for practicability.

The final evaluation was supported with high quality measurement equipment. Different stages of the system were evaluated and compared. The received information was inspected on a binary level, to make a statement on the quality of the transmission in place. However, it was observed that the BER was higher then expected, especially when transmitting over a free-space channel. Additionally, the transmission was stable only over short distances.

In summary, the communication was a success. Because the span of this thesis was already reached once a stable, short range transmission was possible, no further improvements were implemented into the system. However, there are a list of possibilities to expand this system even further.

9.1 Outlook

In order to expand the range of the free-space transmission, more radiating output power would be desired. In the evaluation chapter, the output power of the LimeSDR never reached the maximum allowed radiation power specified by the EN norm. This can be improved by using a power amplifier. Connecting this component between the SDR and the transmit antenna, the maximum distance for stable communication would be drastically increased.

However, even when radiating the maximum allowed power, this system would still have limits on the possible distance between two devices. To further improve link stability, other topics need to be considered. One of these improvements would be to use spatial antenna diversity. When transmitting the same information over multiple channels, a system could decide which of these channels is the best one in the current case. On the other hand, the system could also be configured to combine incoming signals and to implement an algorithm to process this data. There are three possibilities for an algorithm to decide which connection to use: Maximum ratio combining, equal gain combining and selective gain combining. Additionally, since the used antennas in this thesis were not optimal, another option would be to design new directional gain antennas, like inverted-F antennas.

Further improvement would be achieved by implementing an adaptive system. The hardware could observe the bandwidth in which it plans to send data before transmitting. If this frequency bandwidth were crowded with foreign signals, the device could choose a different channel in the ISM band and probe this channel for availability.

Another possibility to improve the quality of the communication is to use error correction algorithms, instead of a simple error detection CRC checksum. There are different methods available, mainly convolutional codes (Viterbi) and block codes (Hamming, Reed-Solomon, Turbo codes).

To increase the data rate over the transmission channel, the bandwidth of the signal could be increased. During measurements, the utilized bandwidth was always lower than the permitted bandwidth of the EN norm. A developer could also implement spatial antenna multiplexing. This would mean transmitting several separate information streams over multiple antennas. Another possibility to increase data rates would be to increase the order of the modulation scheme. However, this would in turn effect the maximum possible communication distance. In addition to these possibilities, changing the modulation scheme would also increase the data rate.

A final step to further improve this system would be to implement the discussed multirate digital signal processing into dedicated digital signal processing hardware. Even though this would be a huge development effort, the final product would be a truly configurable embedded wireless device.

Appendix A

Source Code

There were two software projects documented in this thesis, and both of them include software packages licensed under *GNU General Public License.* The first one is the console based *wireless analysis tool* in chapter 4. Although the user interactions were programmed independently, the LimeSuite driver API is a old version copied into the project. And since there were some minor changes committed into the code, the whole project needs to be made public.

The second project was an adaptation of the *gr-limesdr* project on Github[28]. Changes to the code includes the ability to print stream stats into console and an optional AGC input on the RX sink block. Obviously, these changes must be made public as well. And since all additional blocks in section 5.3 were created using the *gr_modtool*, they also fall under the GNU General Public License. Therefore, all of the blocks are summarized into one GNU Radio module.

These two projects are publicly accessible for every one to use on the GitHub page of the author:
`https://github.com/MartinHinteregger`

The source code for the *Constellation Evaluation GUI* in chapter 7 does not use any open source packages, therefore it was not made public.

Appendix B

Host Packages

After a fresh installation of the OS, the following packages need to be installed. This can be achieved using the *apt* command in the console. However, be sure to run the Ubuntu software updater at least once before installing these packages.

First of all, add the Myriad repository for the required SDR tools, and the GNU Radio PPA (**P**ersonal **P**ackage **A**rchive) for release 3.7:

```
sudo add-apt-repository -y ppa:myriadrf/drivers
sudo add-apt-repository ppa:gnuradio/gnuradio-releases-3.7
sudo apt-get update
```

Then install the following packages. To install them, use the following commands in this order. Note that some packages might already be up to date, for the sake of completeness however they are listed here as well.

```
sudo apt-get install cmake cpp libboost-all-dev libcppunit-dev
sudo apt-get install swig python-numpy python-swift libusb-1.0-0-dev
sudo apt-get install limesuite liblimesuite-dev
sudo apt-get install limesuite-udev limesuite-images
sudo apt-get install soapysdr-tools soapysdr-module-lms7
sudo apt-get install gnuplot default-jre
```

The following list is a description of all packages, obtained using the console command *apt show [name]*.

- **cmake** Cross-platform, open-source make system.
- **cpp** GNU C preprocessor (cpp).
- **libboost-all-dev** Boost *C++* Libraries development files.
- **libcppunit-dev** Unit Testing Library for *C++*.
- **swig** Generate scripting interfaces to *C/C++* code.

- **python-numpy** Numerical *Python* adds a fast array facility to the *Python* language.
- **python-swift** *Python* libraries for swift support.
- **libusb-1.0-0-dev** Userspace USB programming library development files.
- **limesuite** Lime Suite - Library applications.
- **liblimesuite-dev** Lime Suite - development files.
- **limesuite-udev** Lime Suite - USB rules for udev.
- **limesuite-images** Lime Suite - Install firmware and gateware images.
- **soapysdr-tools** Software-defined radio interface library tools.
- **soapysdr-module-lms7** Lime Suite - SoapySDR bindings metapackage.
- **gnuplot** Command-line driven interactive plotting program.
- **default-jre** Standard Java or Java compatible Runtime.

For good measure, it is also advised to install *git* if the goal is to further develop this project.

Finally, install GNU Radio using the package manager. Active GNU release was 3.7.13.5, later versions will cause issues when adding the custom data processing blocks discussed in this thesis. These newer releases also need additional packages. Note that more recent versions of *limesuite* will also cause problems. Latest stable version to work was 19.0.4, so this package might need to be installed from source by checking out the specific GIT commit.

```
sudo apt-get install gnuradio
```

B.1 Development Environment

Eclipse was chosen as a development environment. It's freely available and can be installed either via the console or manually, although it is recommended to install it manually. Following a console installation, start it in console with the command *eclipse*. After a manual installation start it with the command

```
~/eclipse/cpp-<Version>/eclipse/eclipse &
```

where <*Version*> is the installed version of eclipse, containing year and moth (for example 2019-12). It's suggested to install the *C/C++* IDE (**I**ntegrated **D**evelopment **E**nvironment) to have all the necessary tools available.

Appendix C

Project Setup

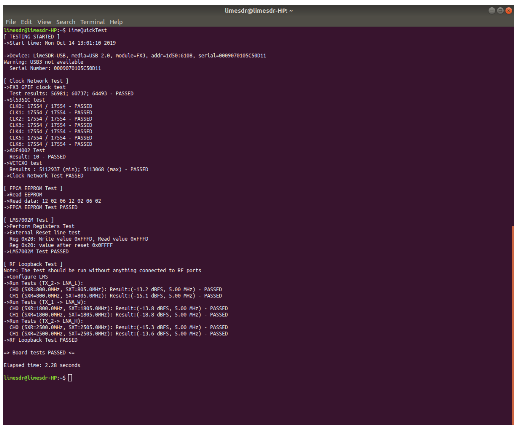
This chapter describes what steps need to be taken on the host to set up a development environment for WAT.

C.1 LimeSuite Driver

After setting up the host as described in section 3.6 and chapter **Host Packages**, the system is ready to test the software driver and the connectivity of the hardware platforms. As mentioned, care has to be taken to install the correct release of the driver. In the console the user can do simple tests after connecting one or more LimeSDR boards with the host via a USB cable.

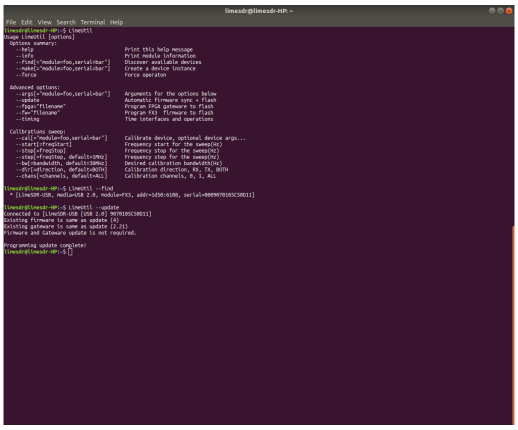
The command *LimeQuickTest* probes all serial interfaces for connected devices and runs advanced hardware tests of the LimeSDR board. This way it can be confirmed that the driver was installed successfully and if the devices still work as intended. All tests should pass. Note that some of the LimeSDR devices might not be able to *cold start*, meaning that they should be connected and therefore powered for a while before communication over the interface is initiated.

The command *LimeUtil* also probes serial interfaces for devices and gives the user the possibility to do automatic firmware and gateware updates of the device. It will automatically flash all FX3 USB controller firmware and FPGA gateware. Additionally this command can also print basic information of the device and do a calibration sweep.



```
limesdr@limesdr-HP:~$ LimeQuickTest
[ TESTING STARTED ]
->Start time: Mon Oct 14 13:01:10 2019

->Device: LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108, serial=0009070105C50D11
Warning: USB3 not available
  Serial Number: 0009070105C50D11

[ Clock Network Test ]
->FX3 GPIF clock test
  Test results: 56981; 60737; 64493 - PASSED
->Si5351C test
  CLK0: 17554 / 17554 - PASSED
  CLK1: 17554 / 17554 - PASSED
  CLK2: 17554 / 17554 - PASSED
  CLK3: 17554 / 17554 - PASSED
  CLK4: 17554 / 17554 - PASSED
  CLK5: 17554 / 17554 - PASSED
  CLK6: 17554 / 17554 - PASSED
->ADF4002 Test
  Result: 10 - PASSED
->VCTCXO test
  Results : 5112937 (min); 5113068 (max) - PASSED
->Clock Network Test PASSED

[ FPGA EEPROM Test ]
->Read EEPROM
->Read data: 12 02 06 12 02 06 02
->FPGA EEPROM Test PASSED

[ LMS7002M Test ]
->Perform Registers Test
->External Reset line test
  Reg 0x20: Write value 0xFFFD, Read value 0xFFFD
  Reg 0x20: value after reset 0x0FFFF
->LMS7002M Test PASSED

[ RF Loopback Test ]
Note: The test should be run without anything connected to RF ports
->Configure LMS
->Run Tests (TX_2-> LNA_L):
  CH0 (SXR=800.0MHz, SXT=805.0MHz): Result:(-13.2 dBFS, 5.00 MHz) - PASSED
  CH1 (SXR=800.0MHz, SXT=805.0MHz): Result:(-15.1 dBFS, 5.00 MHz) - PASSED
->Run Tests (TX_1 -> LNA_W):
  CH0 (SXR=1800.0MHz, SXT=1805.0MHz): Result:(-13.8 dBFS, 5.00 MHz) - PASSED
  CH1 (SXR=1800.0MHz, SXT=1805.0MHz): Result:(-18.8 dBFS, 5.00 MHz) - PASSED
->Run Tests (TX_2-> LNA_H):
  CH0 (SXR=2500.0MHz, SXT=2505.0MHz): Result:(-15.3 dBFS, 5.00 MHz) - PASSED
  CH1 (SXR=2500.0MHz, SXT=2505.0MHz): Result:(-13.6 dBFS, 5.00 MHz) - PASSED
->RF Loopback Test PASSED

=> Board tests PASSED <=

Elapsed time: 2.28 seconds

limesdr@limesdr-HP:~$ 
```

(a) LimeQuickTest.



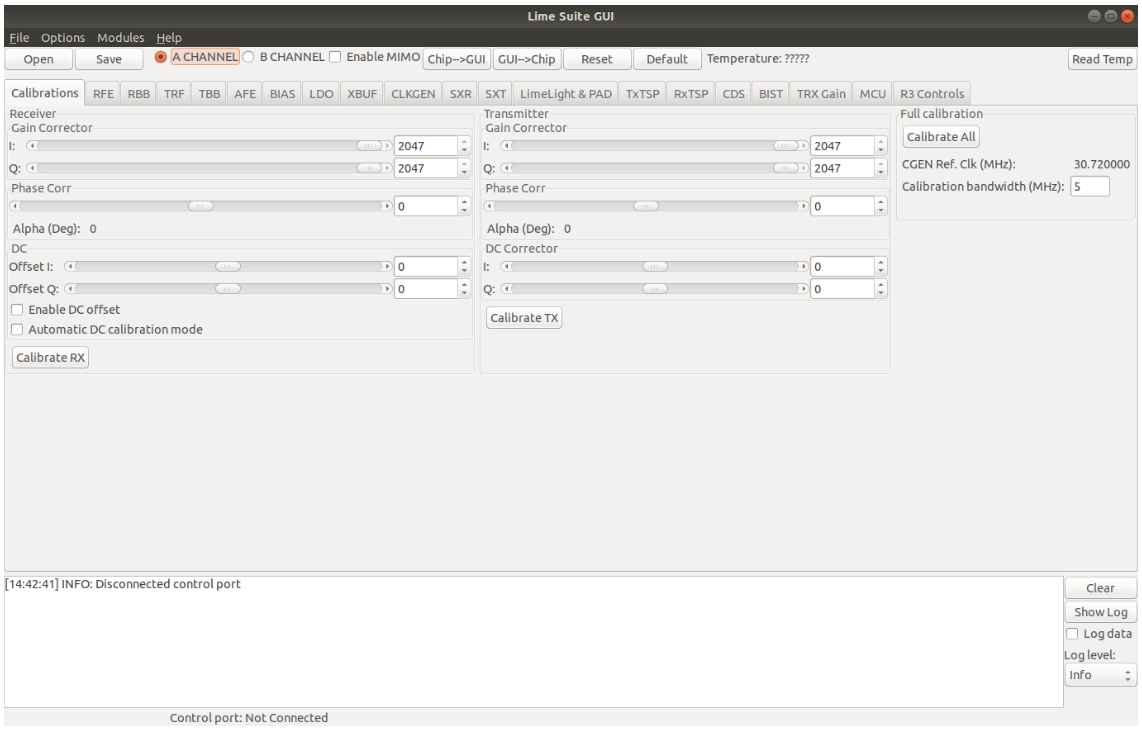
```
limesdr@limesdr-HP:~$ LimeUtil
Usage LimeUtil [options]
  Options summary:
    --help                             Print this help message
    --info                             Print module information
    --find[="module=foo,serial=bar"]   Discover available devices
    --make[="module=foo,serial=bar"]   Create a device instance
    --force                            Force operaton

  Advanced options:
    --args[="module=foo,serial=bar"]   Arguments for the options below
    --update                           Automatic firmware sync + flash
    --fpga="filename"                  Program FPGA gateware to flash
    --fw="filename"                    Program FX3  firmware to flash
    --timing                           Time interfaces and operations

  Calibrations sweep:
    --cal[="module=foo,serial=bar"]    Calibrate device, optional device args...
    --start[=freqStart]                Frequency start for the sweep(Hz)
    --stop[=freqStop]                  Frequency stop for the sweep(Hz)
    --step[=freqStep, default=1MHz]    Frequency step for the sweep(Hz)
    --bw[=bandwidth, default=30MHz]    Desired calibration bandwidth(Hz)
    --dir[=direction, default=BOTH]    Calibration direction, RX, TX, BOTH
    --chans[=channels, default=ALL]    Calibration channels, 0, 1, ALL

limesdr@limesdr-HP:~$ LimeUtil --find
  * [LimeSDR-USB, media=USB 2.0, module=FX3, addr=1d50:6108, serial=0009070105C50D11]

limesdr@limesdr-HP:~$ LimeUtil --update
Connected to [LimeSDR-USB [USB 2.0] 9070105C50D11]
Existing firmware is same as update (4)
Existing gateware is same as update (2.21)
Firmware and Gateware update is not required.

Programming update complete!
limesdr@limesdr-HP:~$ 
```

(b) LimeUtil.

Figure C.1: LimeQuickTest and LimeUtil use cases in console.

Finally the user can also start the GUI of the installed driver by calling the command *LimeSuiteGUI*. Figure C.2 shows a screenshot of the GUI just after starting. Here the user has complete control over all devices once they were connected by the program (i.e. opening a *udev* USB device using *libusb* on Linux) by choosing *Options* → *ConnectionSettings*. The user can also *Save* a configuration file. These configuration files are stored as .ini files and can be uploaded onto any other LimeSDR by using *Open* to get the same configuration.

The menu *Modules* gives access to a few different tools like a FFT viewer, MCU (**M**icro **C**ontroller **U**nit) programming, SPI read/write commands and a FPGA waveform player. On the top side of the program the user can choose between different index cards, each one referring to parts or functions of the device. Most of the index cards are abbreviations. For further information on what these mean and what they relay to, refer to the data sheets[20][26]. This thesis will not go into detail about the program, since this work is mainly interested in the driver API. The GUI version represented in the Figure is version 19.04, build date 2019-05-13. However, the driver version during development of this console based user program was from 2018-08-16. The GIT commit number to this release v18.10.0 is 99731b677a250aef7fd27469c5dac23b87cd80d7.



Temperature: ?????

Receiver — Gain Corrector: I: 2047, Q: 2047; Phase Corr: 0; Alpha (Deg): 0; DC Offset I: 0, Offset Q: 0

Transmitter — Gain Corrector: I: 2047, Q: 2047; Phase Corr: 0; Alpha (Deg): 0; DC Corrector I: 0, Q: 0

CGEN Ref. Clk (MHz): 30.720000

Calibration bandwidth (MHz): 5

Log level: Info

```
[14:42:41] INFO: Disconnected control port
```

Control port: Not Connected

Figure C.2: LimeSuite driver GUI, right after start-up.

C.1.1 Driver API

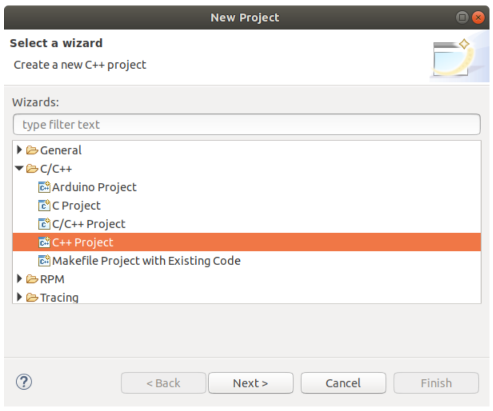
The host also installed the LimeSuite driver API libraries. These can be found in the directory */usr/include/lime/*. However, the user can also choose to get a clone of the open source code from Github[32].

C.2 Development Environment Setup

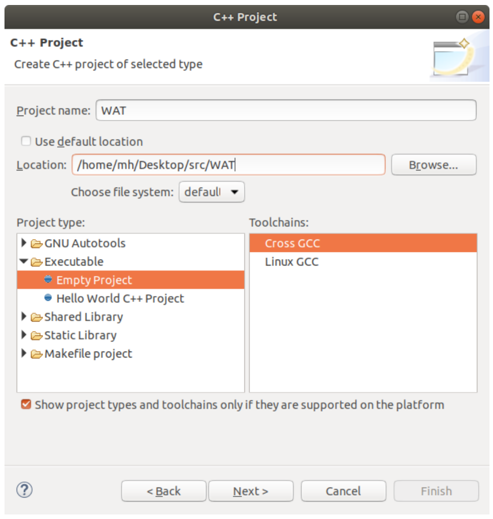
As already mentioned in the previous chapter, Eclipse was used as the main development platform. This section will describe how to set up the project using Eclipse, both by manual setup and by importing the project settings. It is of vital importance to have all packages listed in chapter **Host Packages** installed for a successful build of the project.

C.2.1 Manual Setup

The project was set up as a Automatic Makefile project. This is easily achieved using Eclipse development platform: Create a new *C/C++* Project and choose a *C++* managed build. In other words, the project should use a Cross GCC (**G**NU **C**ompiler **C**ollection) compiler toolchain and a GNU Make Builder. Figure C.3 shows how to set up the project.



(a) Choose a wizard.



Project name: WAT

Use default location

Location: /home/mh/Desktop/src/WAT

Choose file system: default

Show project types and toolchains only if they are supported on the platform

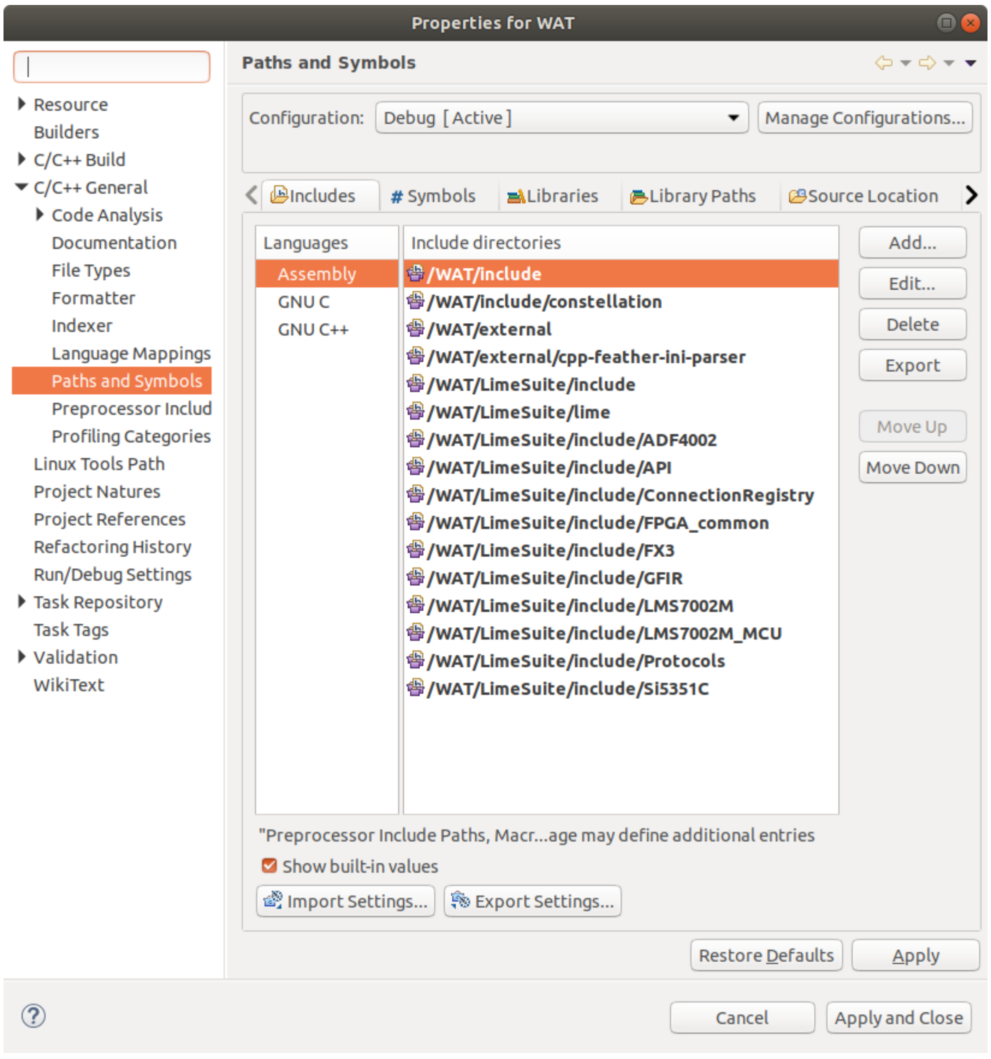
(b) Choose project options.

Figure C.3: Which options to choose when setting up the project in eclipse.

After defining a project name and folder location, the initial project is empty. If one would want to start a new project from scratch (for example a hello world project) the first step would be to create a new *.cpp* file with the same name as the project, add a main function and some lines of code. A simple *Build* command in Eclipse will create a folder called *Debug* (this is usually the initial configuration of the build) where the Makefile will be auto generated and the project will be build. After successful completion, the object file is named as the project name and can be found in the *Debug* folder. This object can then be called in the console to start the program.

However, to get the WAT project going the user first needs to import the source code into the setup and then set some additional options. To import source files, right click the project in the project explorer and choose *Import*, then select *File System* and browse to the directory containing the source files. Select all folders and files to import the whole project. This will copy all files into the folder of the eclipse workspace.

Next make sure the necessary libraries, include paths and symbols are all set. Open the project options and choose the register card *C/C++ General* → *Paths and Symbols*. Add all folders containing header files to all languages and to all configurations. Adding them as workspace paths makes them independent of the file system location. See Figure C.4 for a list of all include paths.



Paths and Symbols

Configuration: Debug [Active]

Languages	Include directories
Assembly	/WAT/include
GNU C	/WAT/include/constellation
GNU C++	/WAT/external
	/WAT/external/cpp-feather-ini-parser
	/WAT/LimeSuite/include
	/WAT/LimeSuite/lime
	/WAT/LimeSuite/include/ADF4002
	/WAT/LimeSuite/include/API
	/WAT/LimeSuite/include/ConnectionRegistry
	/WAT/LimeSuite/include/FPGA_common
	/WAT/LimeSuite/include/FX3
	/WAT/LimeSuite/include/GFIR
	/WAT/LimeSuite/include/LMS7002M
	/WAT/LimeSuite/include/LMS7002M_MCU
	/WAT/LimeSuite/include/Protocols
	/WAT/LimeSuite/include/Si5351C

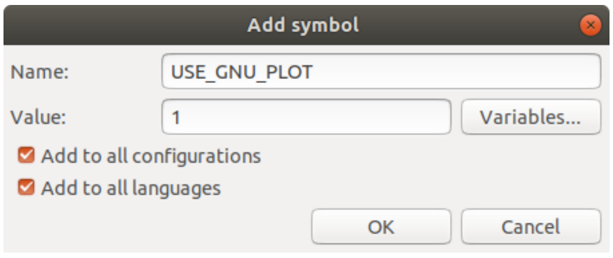
"Preprocessor Include Paths, Macr...age may define additional entries

✓ Show built-in values

Figure C.4: Project options, register card Paths and Symbols, included build paths.

Next switch to the register card *Symbols*, add the symbol USE_GNU_PLOT to all configurations and languages and give it the value 1. Setting this pre-processor symbol to zero is possible, but no visualization of the data using GNUPlot will be available. Lastly add the necessary libraries that should be installed in the */usr/include/* folder, see Figure C.6 for a list of needed libraries.

After this configuration steps the project will compile and create the binary executable in the *Debug* folder.



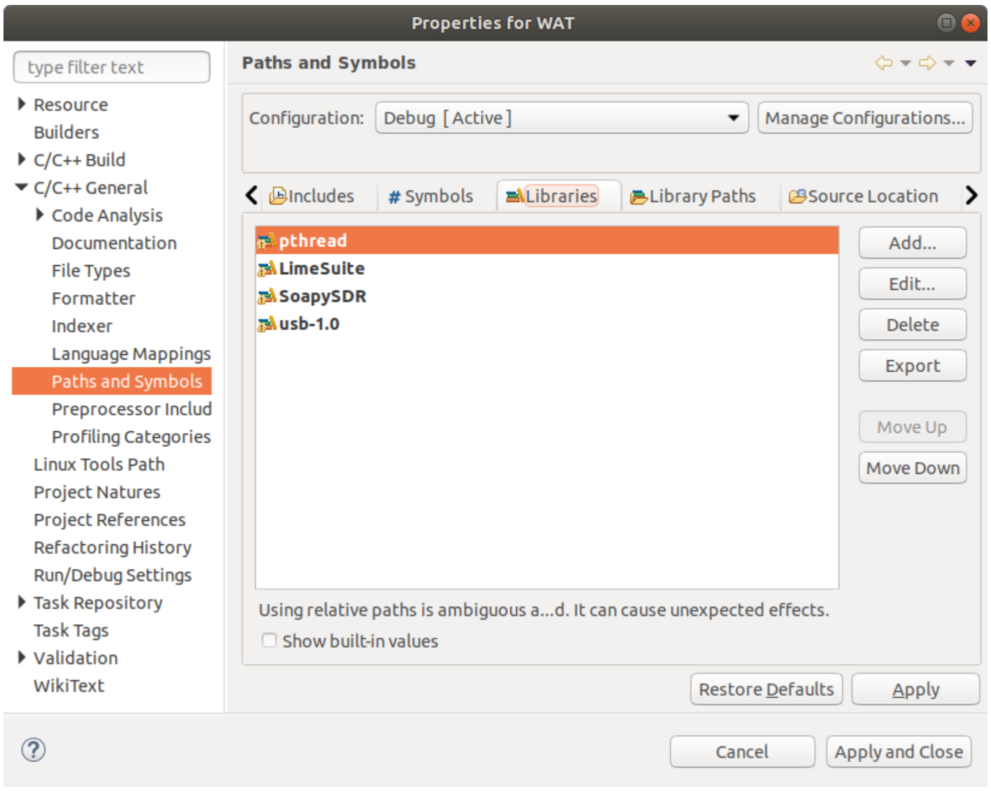
Name: USE_GNU_PLOT

Value: 1

[✓] Add to all configurations

[✓] Add to all languages

Figure C.5: Which options to set when adding a symbol.



Configuration: Debug [Active]

pthread
LimeSuite
SoapySDR
usb-1.0

Using relative paths is ambiguous a...d. It can cause unexpected effects.

[ ] Show built-in values

Figure C.6: Libraries that need to be included into the build.

C.2.2 Import Setup

By default the folder containing the source code already has the necessary project files. In this case simply open the import wizard, choose *General* and select *Projects from Folder or Archive* and browse to the directory where the source code of the project resides. The import wizard automatically detects the Eclipse project. After successful import, the project should be able to compile using the debug configuration. If problems arise, make sure that all necessary packages are installed on the host (see chapter **Host Packages**), and check the needed libraries, include paths and symbols described in the manual setup.

Appendix D

Adding Custom Blocks

This chapter will describe how to set up a new block for GNU Radio by giving an example implementation. It will also cover how to add custom blocks to the OS such that the user can implement them in their GRC flow graph.

D.1 Modtool

When installing GNU Radio, the user also adds a tool to enable setting up development of custom GNU Radio blocks. This tool is called *gr_modtool* and it can be used in console. It simplifies development of own blocks by setting up the basic file structure needed, including bare-bone code.

```
mh@mh:~$ gr_modtool help --
Usage:
gr_modtool <command> [options] -- Run <command> with the given options.
gr_modtool help -- Show a list of commands.
gr_modtool help <command> -- Shows the help for a given command.

List of possible commands:

Name      Aliases          Description
=====================================================================
disable   dis              Disable block (comments out CMake entries
    for files)
info      getinfo,inf      Return information about a given module
remove    rm,del           Remove block (delete files and remove
    Makefile entries)
makexml   mx               Make XML file for GRC block bindings
add       insert           Add block to the out-of-tree module.
newmod    nm,create        Create a new out-of-tree module
rename    mv               Rename a block in the out-of-tree module.
```

Listing D.1: Usage information to gr_modtool and list of possible commands.

Before adding a block, the user must create a new *module*. This is a generic term to group blocks together, much likely how GNU Radio combines its digital blocks (e.g. digital modulation blocks, linear-feedback shift registers and synchronization blocks) into the *gr-*

digital module.

This tutorial will explain how to develop a source block that will have no input, but instead will output an integer value defined by the user. This block will be called *var_source_i*, to signal every potential developer, that the source block outputs an integer value. The module will be called *gr-tut_source* in accordance to GNU Radio naming policies.

To add the module, simply type *gr_modtool newmod* in the console and specify the name (without *gr-*, as this will be added automatically). Finally, switch to the newly created directory and *add* a block, select *source* as block type, *cpp* to create *C++* files, the name of the block (*var_source_i*) and *int value* as argument list. Finally, chose not to add QA (**Q**uality **A**ssurance) code. The tool then adds all the necessary files to start development. Before the block can be built and added to the environment, some minor changes need to be made.

```
mh@mh:~$ gr_modtool newmod
Name of the new module: tut_source
Creating out-of-tree module in ./gr-tut_source... Done.
Use 'gr_modtool add' to add a new block to this currently empty module.
```

Listing D.2: Create a new module with gr_modtool.

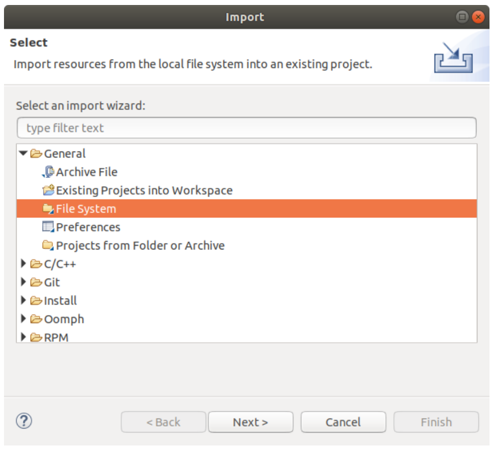
```
mh@mh:~$ cd gr-tut_source/
mh@mh:~/gr-tut_source$ gr_modtool add
GNU Radio module name identified: tut_source
('sink', 'source', 'sync', 'decimator', 'interpolator', 'general',
    'tagged_stream', 'hier', 'noblock')
Enter block type: source
Language (python/cpp): cpp
Language: C++
Enter name of block/code (without module name prefix): var_sources_i
Block/code identifier: var_sources_i
Enter valid argument list, including default arguments: int value
Add Python QA code? [Y/n] n
Add C++ QA code? [Y/n] n
Adding file 'lib/var_sources_i_impl.h'...
Adding file 'lib/var_sources_i_impl.cc'...
Adding file 'include/tut_source/var_sources_i.h'...
Editing swig/tut_source_swig.i...
Adding file 'grc/tut_source_var_sources_i.xml'...
Editing grc/CMakeLists.txt...
```

Listing D.3: Add a new C++ block to the module.

D.2 Setup Development Environment

Unfortunately, there is no special project option available in Eclipse that works with this kind of structure, a project incorporating three different languages (*C++*, *XML* and *Python*). But it is still possible create a *C++* project to get all the *C++* development features like indexer and refraction. This is of course assuming that the instructions in the appendix chapter **Host Packages** were followed and the *C/C++* IDE was added.

Therefore, first create a new and empty *C++* project as described in the manual setup of chapter **Host Packages**, call the project *gr-tut_source* and import the recently created file structure as illustrated in Figure D.1. Be sure to create *virtual links*, otherwise Eclipse will copy the files into Eclipses' work space directory. The project doesn't need to compile in Eclipse, it will be built in console. Eclipse is just used for development.


Import

Select

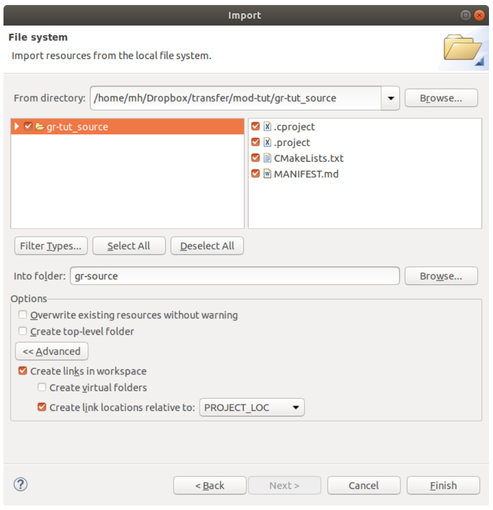
Import resources from the local file system into an existing project.

Select an import wizard:

type filter text

- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Preferences
  - Projects from Folder or Archive
- C/C++
- Git
- Install
- Oomph
- RPM

(a) Choose to import a file system.


Import

File system

Import resources from the local file system.

From directory: /home/mh/Dropbox/transfer/mod-tut/gr-tut_source

- ✓ gr-tut_source

- ✓ .cproject
- ✓ .project
- ✓ CMakeLists.txt
- ✓ MANIFEST.md

Into folder: gr-source

Options
- ☐ Overwrite existing resources without warning
- ☐ Create top-level folder
- ✓ Create links in workspace
  - ☐ Create virtual folders
  - ✓ Create link locations relative to: PROJECT_LOC

(b) Select the directory and check all files and folders.

Figure D.1: How to import the newly created module to the eclipse project.

D.3 Modify Block

The previous steps helped setting up the file structure and the required functions. However, some parts still need modification before the module can be built and added to the environment.

D.3.1 XML File

The first change is to the *XML* file that defines how the block will be visualized in the GUI, and also which parameters and ports the block contains. This file is located in the *grc* directory and is called *tut_source_var_sources_i.xml*. The following tags should be changed:

- Add a reasonable **name** that will be displayed in the GUI.
- The tag **category** specifies the general tab in the GUI under which all blocks of the module will be listed.
- Furthermore, **make** defines the call to a *C++* function that will be used later. It also contains which parameters are passed, the previously discussed *value* parameter

should already be included.

- Add a function callback to a *set_value($value)* function that will be written later with the **callback** tag. This function will be called whenever the value that the block should output is changed, even during execution of the flow graph.
- Next will be the declaration what kind of data format the previously mentioned *value* parameter should be and its default value. This is defined with the **param** tag.
- Finally, since this block will only have one output, define one single output port with integer type using the **source** tag.

The file should then look similar to Listing D.4. As with other *XML* files, the sequence of tags is important.

```xml
<?xml version="1.0"?>
<block>
  <name>Variable Source</name>
  <key>tut_source_var_sources_i</key>
  <category>[tut_source]</category>
  <import>import tut_source</import>
  <make>tut_source.var_sources_i($value)</make>

  <callback>set_value($value)</callback>

  <param>
    <name>Value</name>
    <key>value</key>
    <value>0</value>
    <type>int</type>
  </param>

  <source>
    <name>out</name>
    <type>int</type>
    <nports>1</nports>
  </source>
</block>
```

Listing D.4: The final XML file structure.

D.3.2 var_sources_i.h

It is located in the *include/tut_source* folder and serves as the header implementing the block into GNU Radio and adding API functionality. The previously mentioned callback function must be added as a *virtual* function into the *var_sources_i* class.

```
virtual void set_value(int value) = 0;
```

D.3.3 var_sources_i_impl.h

This header, together with its *.cc* file described next, handle the actual implementation of the block. It can be found in the directory *lib*. Only add the function callback to this header, with the exception that it is not *virtual* this time, but an actual implementation.

```
void set_value(int value);
```

Also add a private integer variable called *val_*:

```
int val_;
```

D.3.4 var_sources_i_impl.cc

The constructor function defines how the inputs and outputs of the block are defined. This needs to be adjusted accordingly, in this case no input connection and a single output port with integer type. Also, save the passed *value* into the private integer *val_*.

```
var_sources_i_impl::var_sources_i_impl(int value)
  : gr::sync_block("var_sources_i",
          gr::io_signature::make(0, 0, 0)
        gr::io_signature::make(1, 1, sizeof(int)))
  {
    val_ = value;
  }
```

Listing D.5: The constructor of the variable source block.

This file contains the actual *work* function of the block, which handles what the block does when the GNU Radio scheduler executes the block to process data. In the work function, just cast the output variable, store the private integer variable into the first element of the array and tell the GNU Radio scheduler to only read the first value.

```
int
var_sources_i_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  gr_int32 *out = (gr_int32 *) output_items[0];

  out[0] = val_;

  return 1;
}
```

Listing D.6: The work function of the variable source block.

Finally, implement the callback function by storing the passed value into the private variable. This function will be called if the user chooses to change the output value during execution of a flow graph.

```
void var_sources_i_impl::set_value(int value)
{
  val_ = value;
}
```

Listing D.7: The callback function of the variable source block.

After these steps, the module is ready to be built and added to the GNU Radio environment.

D.4 Build Modules

In order to build blocks, first navigate in the console to the directory of the module and issue the following commands:

```
mkdir build
cd build
cmake ..
make all
sudo make install
sudo ldconfig
```

This will first create a folder called build, switch into this folder, prepare the build process and check necessary libraries and tools by using *cmake*, then build the project using *make* as well as installing the block. Finally, *ldconfig* updates links and cache to the most recent shared libraries in various directories. When starting GRC, the newly created block can now be selected and used in flow graphs.

To uninstall the module, navigate to the *build* directory in the console and issue the following commands:

```
sudo make uninstall
make clean
cd ..
rm -rf build
```

This will first uninstall the module, then clean and remove the *build* folder. Note that GRC might need a restart for the changes to take effect.

Bibliography

[1] Ops-sat. , European Space Agency. [Online]. Available: https://www.esa.int/Enabling_Support/Operations/OPS-SAT

[2] B. Sklar, *Digital Communications: Fundamentals and Applications*, 2nd ed. Bernard Goodwin, 2005.

[3] K. Finkenzeller, *RFID Handbook*, 2nd ed. Carl Hanser Verlag, 2003.

[4] J. G. Proakis, *Digital Communications*, 4th ed. The McGraw-Hill Companies, Inc., 2000.

[5] T. S. Rappaport, *Wireless Communications: Principles and Practice*, 2nd ed. Bernard Goodwin.

[6] T. F. C. Di, *Software-Defined Radio for Engineers.* Artech House, 2018.

[7] T. Hentschel, *Sample rate conversion in software configurable radios.* Artech House Publishers, 2002.

[8] S. Hara and R. Prasad, *Multicarrier Techniques for 4G Mobile Communications.* Artech House Publishers, 2003.

[9] A. Lozano and N. Jindal, "Transmit diversity vs. spatial multiplexing in modern mimo systems," *IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, VOL. 9, NO. 1, JANUARY 2010*, 2010.

[10] E. Biglieri, *Coding for Wireless Channels (Information Technology: Transmission, Processing and Storage).* Springer, 2005.

[11] H. Meyr, M. Moeneclaey, and S. A. Fechtel, *Digital Communication Receivers, Synchronization, Channel Estimation, and Signal Processing.* Wiley-Interscience, 1997.

[12] U. Mengali, *Synchronization Techniques for Digital Receivers (Applications of Communications Theory).* Springer, 1997.

[13] J. Mitola, "The software radio architecture," *IEEE Communications Magazine*, 1995.

[14] B. Hall and W. Taylor, "X- and ku-band small form factor radio design." [Online]. Available: https://www.analog.com/en/technical-articles/x-and-ku-band-small-form-factor-radio-design.html

[15] *Wideband transmission systems; Data transmission equipment operating in the 2,4 GHz band; Harmonised Standard for access to radio spectrum*, ETSI Std. ETSI EN 300 328, Rev. V2.2.1, Nov. 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_en/300300_300399/300328/02.02.01_30/en_300328v020201v.pdf

[16] Usrp b210. , Ettus Research. [Online]. Available: https://www.ettus.com/all-products/ub210-kit/

[17] Usrp-2901. , National Instruments. [Online]. Available: https://www.ni.com/en-us/support/model.usrp-2901.html

[18] Usrp communications mimo teaching bundle. , National Instruments. [Online]. Available: http://sine.ni.com/nips/cds/view/p/lang/en/nid/213007

[19] Usrp software-defined radio device. , National Instruments. [Online]. Available: http://www.ni.com/en-us/shop/select/usrp-software-defined-radio-device

[20] Limesdr. , Lime Microsystems. [Online]. Available: https://limemicro.com/products/boards/limesdr/

[21] Lms7002m product brief. , Lime Microsystems. [Online]. Available: https://github.com/myriadrf/LMS7002M-docs/blob/master/LMS7002M_Product_Brief.pdf

[22] Limesdr crowdsupply funding page. , Lime Microsystems. [Online]. Available: https://www.crowdsupply.com/lime-micro/limesdr

[23] Xtrx. , Fairwaves. [Online]. Available: https://xtrx.io/

[24] Xtrx - the first ever truly embedded sdr. , Fairwaves. [Online]. Available: https://www.crowdsupply.com/fairwaves/xtrx

[25] Xtrx source code on github. , Fairwaves. [Online]. Available: https://github.com/xtrx-sdr

[26] Lms7002m data sheet. , Lime Microsystems. [Online]. Available: https://github.com/myriadrf/LMS7002M-docs/blob/master/LMS7002M_Data_Sheet_v3.2r00.pdf

[27] Lms7002m - multi-band, multi-standard mimo rf transceiver ic - programming and calibration guide -. , Lime Microsystems. [Online]. Available: https://github.com/myriadrf/LMS7002M-docs/blob/master/LMS7002M_Programming_and_Calibration_Guide_v31r05.pdf

[28] gr-limesdr source code on github. , Lime Microsystems. [Online]. Available: https://github.com/myriadrf/gr-limesdr

[29] Gnu radio wiki for symbol sync block. [Online]. Available: https://wiki.gnuradio.org/index.php/Symbol_Sync

[30] Gnu radio api reference about lms dd equalizer. , GNU Radio. [Online]. Available: https://www.gnuradio.org/doc/doxygen/classgr_1_1digital_1_1lms__dd__equalizer__cc.html

[31] (2017) Smbv100a operation manual web help. , Rohde & Schwarz. [Online]. Available: https://www.rohde-schwarz.com/webhelp/smbv_html_usermanuals_online_en_1/SMBV_HTML_UserManuals_Online_en.htm

[32] Limesuite source code on github. , Lime Microsystems. [Online]. Available: https://github.com/MyriadRF/LimeSuite