NPS-SP-16-003CR

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DEVELOPMENT OF A VERSATILE GROUNDSTATION UTILIZING SOFTWARE DEFINED RADIO

by

Jan Malte Roehrig

August 2016

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 2016	Research Report 05-10-2016 to 08-10-2016

4. TITLE AND SUBTITLE

DEVELOPMENT OF A VERSATILE GROUNDSTATION UTILIZING SOFTWARE DEFINED RADIO

5. FUNDING NUMBERS

6. AUTHOR(S)

Jan Malte Roehrig

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

NPS-SP-16-003 CR

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

As software defined systems establish modern technology, the Space Systems Academic Group (SSAG) of the Naval Postgraduate School (NPS) is compelled to replace legacy hardware systems with inexpensive and flexible software defined solutions. One key aspect in ensuring stable communication with the multiple *small satellites* and *CubeSats* launched by the SSAG, is to develop and maintain a versatile satellite groundstation utilizing inexpensive off-the-shelf *Software Defined Radio* (SDR). This thesis presents the demodulation and decoding of a 1200 baud AFSK signal from a BigRedBee, as well as a 9600 baud GFSK signal from a PropCube CubeSat. The demodulating signal processing steps are designed in GNU Radio Companion. The thesis also deals with parts of the AX.25 data link layer protocol decoding. Bit-per-bit comparison of the outputs in performed tests show that the SDR can replace the currently used software. A *bit error rate* (BER) calculation proves the excellent performance of the GFSK demodulation scheme. The performed BER test results in a $BER < 1 \times 10^{-4}$ for a SNR of $E_b/N_0 = 6dB$, whereas the estimated BER for a coherent MSK signal is $BER_{MSK} \approx 2 \times 10^{-4}$ for the same SNR. Additionally, this thesis presents one option of a real-time implementation of the PropCube Receiver. This thesis also provides test results of performed function tests, subsystem tests and integrated system tests, accompanied by in depth explanations on the settings and preferences of each used GNU Radio block.

14. SUBJECT TERMS

15. NUMBER OF PAGES 127

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Ronald A. Route President	James Newman Interim Provost

The report entitled "*Development of a Versatile Groundstation Utilizing Software Defined Radio"* was prepared for the Naval Postgraduate School.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Jan Malte Roehrig
2nd Lieutenant, German Navy

Reviewed by:	**Released by:**
Rudolf Panholzer, Chairman Space Systems Academic Group	Jeffrey D. Paduan Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As software defined systems establish modern technology, the Space Systems Academic Group (SSAG) of the Naval Postgraduate School (NPS) is compelled to replace legacy hardware systems with inexpensive and flexible software defined solutions. One key aspect in ensuring stable communication with the multiple *small satellites* and *CubeSats* launched by the SSAG, is to develop and maintain a versatile satellite groundstation utilizing inexpensive off-the-shelf *Software Defined Radio* (SDR). This thesis presents the demodulation and decoding of a 1200 baud AFSK signal from a BigRedBee, as well as a 9600 baud GFSK signal from a PropCube CubeSat. The demodulating signal processing steps are designed in GNU Radio Companion. The thesis also deals with parts of the AX.25 data link layer protocol decoding. Bit-per-bit comparison of the outputs in performed tests show that the SDR can replace the currently used software. A *bit error rate* (BER) calculation proves the excellent performance of the GFSK demodulation scheme. The performed BER test results in a $BER < 1 \times 10^{-4}$ for a SNR of $E_b/N_0 = 6dB$, whereas the estimated BER for a coherent MSK signal is $BER_{MSK} \approx 2 \times 10^{-4}$ for the same SNR. Additionally, this thesis presents one option of a real-time implementation of the PropCube Receiver. This thesis also provides test results of performed function tests, subsystem tests and integrated system tests, accompanied by in depth explanations on the settings and preferences of each used GNU Radio block.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**
1.1 Background 1
1.2 Organization of Study 3

2 Literature Review and Theoretical Framework **5**
2.1 Software Defined Radio 5
2.2 Digital Signal Processing Theory 6
2.3 Related Work on Software Defined Radio Groundstations 9

3 Hardware and Software Environment **11**
3.1 Hardware used 11
3.2 Used Software - GNU Radio 13
3.3 Communication System Specifications 13

4 Groundstation Development **19**
4.1 Development 19
4.2 Sample Rate Determination 19
4.3 Test Setups 20
4.4 How to Measure and Scope Results 21

5 Presentation of Demodulation Schemes **25**
5.1 AFSK Demodulation and Decoding Flowgraph 25
5.2 GFSK Demodulation Flowgraph 30
5.3 AX.25 Decoding 33

6 How to use the Groundstation **35**
6.1 How to set up the Groundstation 35
6.2 How to run the Groundstation 35

7 Real-Time Implementation **39**
7.1 Demodulation Flowgraph 39
7.2 Decoding Software 40

8 Data Analysis and Interpretation **43**
8.1 BRB - AFSK1200 43
8.2 PropCube - GFSK9600 43

9 Conclusion and Recommendations **47**
9.1 Conclusion 47
9.2 Outlook 48

Appendix: Appendix **49**

A Appendix A **51**
A.1 Used GNU Radio Blocks 51
A.2 Block Function Test 63

B Appendix B **71**
A.1 Flowgraphs 71

C Appendix C **77**
A.1 Python Scripts 77

List of References **107**

Initial Distribution List **111**

List of Figures

Figure 1.1 Versatile SDR ground station 2

Figure 2.1 Hardware Radio architecture 6

Figure 2.2 Software-Programmable Radio architecture 7

Figure 3.1 AFSK1200 receiving unit. 14

Figure 3.2 BigRedBee APRS Beacon. 15

Figure 3.3 GFSK9600 receiving unit. 15

Figure 3.4 Satellite Groundstation Setup. 16

Figure 3.5 PropCube Packet and Protocol Structure. 16

Figure 4.1 PropCube - Test Setup. 21

Figure 5.1 AFSK1200 Demodulation Flowgraph. 26

Figure 5.2 GFSK9600 Demodulation Flowgraph. 30

Figure 6.1 BRB Groundstation GUI. 36

Figure 6.2 PropCube Groundstation GUI. 38

Figure 8.1 Bit Error Rates. 44

Figure A.1 USRP Source Block. 52

Figure A.2 BPF Block. 53

Figure A.3 FIR Filter Block. 54

Figure A.4 RRC Filter Block. 55

Figure A.5 Quad Demod. 56

Figure A.6 GFSK Demod Block. 57
Figure A.7 Detectmarkspace Block. 58
Figure A.8 Detectmarkspace Flowgraph. 58
Figure A.9 Mark Frequency Detector Output. 59
Figure A.10 AX25decode Block. 59
Figure A.11 AX25decode Block Diagram. 60
Figure A.12 AX25decode Block Output. 61
Figure A.13 Mark and Space Distinguisher Test. 64
Figure A.14 Mark and Space Distinguisher Test - Mark Input. 64
Figure A.15 Mark and Space Distinguisher Test - Space Input. 65
Figure A.16 FIR Filter Tap Test. 65
Figure A.17 AX25decode with 9 Samples per Symbol. 68
Figure A.18 AX25decode with 10 Samples per Symbol. 69
Figure A.19 AX25decode with 11 Samples per Symbol. 70
Figure A.20 Clock Recovery Error. 70

Figure A.1 GRC Flowgraph for BER and SNR Determination. 73
Figure A.2 GRC Flowgraph BigRedBee Receiver. 74
Figure A.3 GRC Flowgraph PropCube Receiver. 75

List of Tables

Table 8.1 BER PropCube - SDR. 45

Table A.1 USRP settings. 53

Table A.2 LPF Cutoff Frequency Test. 67

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

[dB]	[decibel]
δ**(t-nT**$_S$)	delta function at $t = nT_S$
ADC	analog-to-digital converter
AGC	Automatic Gain Control
APRS	Automatic Packet Reporting System
ASCII	American Standard Code for Information Interchange
AX.25	a Data Link Layer packet format based on. HDLC
BER	bit error rate
bps	bits-per-second
CCITT	Consultative Committee for International Telephony and Telegraphy
CRC	cyclic redundancy check (either 16 bit or 32 bit)
DAC	digital-to-analog converter
DFE	Digital Front End
exp	expotential
FCC	Federal Communictaions Commission
FDM	frequency-division multiplexing
FFT	fast fourier transform
FM	frequency modulation
FPGA	Field Programmable Gate Array
FSK	frequency shift keying
GFSK	gaussian frequency shift keying
GMSK	gaussian minimum shift keying

GPS	Global Positioning System
GRC	GNU Radio Companion
HAB	high altitude balloon
HDLC	High Level Data Link Control
Hz	Hertz
IFI	inter symbol interference
IP	internet protocol
ISO	International Organization for Standardization
LNA	low noise amplifier
modem	modulation/demodulation
NASA	National Aeronautics and Space Administration
NPS	Naval Postgraduate School
NRL	Naval Research Laboratory
NRZ	nonreturn-to-zero
NRZI	nonreturn-to-zero inverted
OSI	open systems interconnection
RF	radio frequency
RRC	root raised cosine filter
SDR	software defined radio
SNR	signal-to-noise ratio
SSAG	Space Systems Academic Group
TDM	time-division multiplexing
USA	United States of America
USRP	Universal Software Radio Periphal

CHAPTER 1:
Introduction

1.1 Background

The borders of flexibility and capacity of modern technology systems are significantly expanded by the implementation of software. Replacing hardware parts with software makes systems more reliable, cheaper, lighter and smaller. Also, it enables a system to handle multiple and greater inputs at the same time and even to be time-independent. Especially in communication technologies, research and development has moved forward rapidly since *software defined radios* (SDR) have become state of the art.

The main objective of the Space Systems Academic Group (SSAG) is the education of military officers at the graduate level. One way to accomplish this objective is to provide hands-on projects and practical experience, in addition to imparting theoretical knowledge. The SSAG of the Naval Postgraduate School (NPS) has launched a *small satellite* as well as two *CubeSats* between 1998 thru today. Also, the SSAG has participated in the development of other satellites. As part of a summer intern program the SSAG launched several *high altitude balloons* (HAB) in the last few years, carrying payloads created by high school and college students.

The SSAG runs a groundstation to communicate with launched satellites and HABs. In order to improve this communication the SSAG seeks to replace hardware radio components with a software defined radio (SDR). The objective is to use off-the-shelf components and run them using software generated in-house. One SDR can replace many thousands of dollars' worth of equipment at the groundstation. Also, with its flexibility to handle many frequencies and various modulations an SDR allows the SSAG to communicate with more satellites while using fewer hardware components. It also enables the SSAG to change parts of the communication systems on-the-fly.

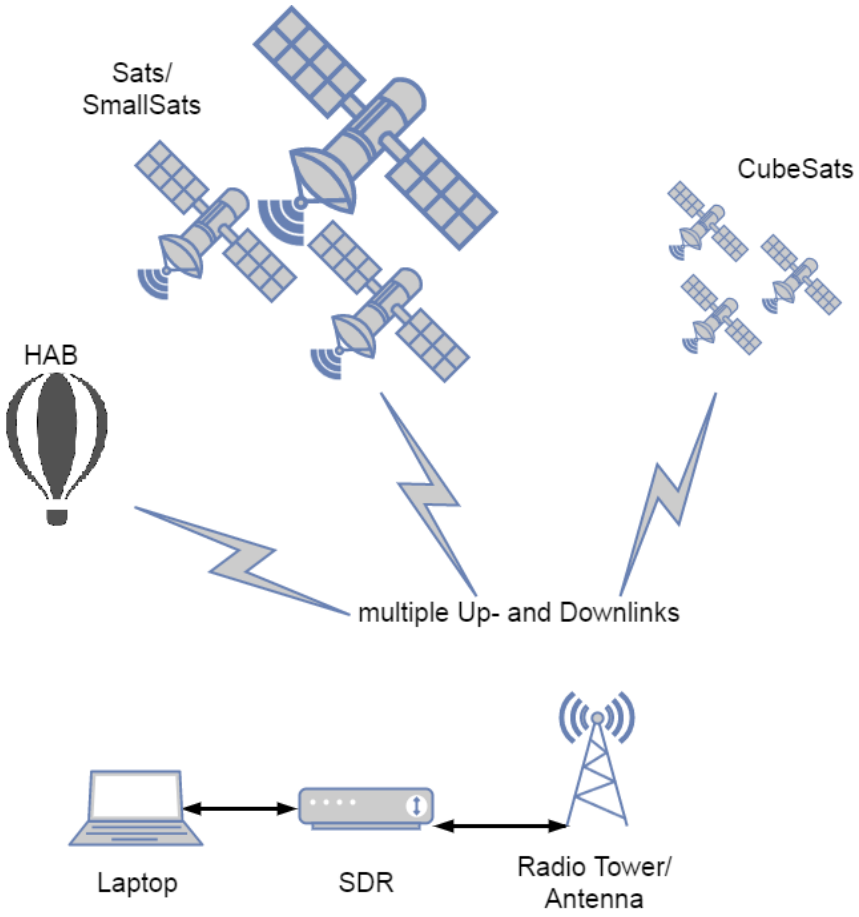

Figure 1.1. Schematic of a versatile ground station utilizing Software Defined Radio.

The system in Figure 1.1 is an overall objective to keep the SSAG's procedures up to date and secure their ability to impart contemporary knowledge sustainably. The concept shown in Figure 1.1 was created at the outset of this thesis. This thesis deals with an investigation on whether inexpensive off-the-shelf SDR technology can replace legacy radio hardware for satellite groundstations. While using one computing device and one SDR, the SSAG is able to control and operate small satellites, CubeSats, as well as HAB payloads with only one common groundstation. Even multiple satellites or HAB payloads may be operated at the same time. Using SDRs, systems can be updated quickly. By using a laptop as a computing device, the groundstation gets highly mobile.

This thesis covers the development and presentation of a 1200 baud AFSK signal demodulation, as well as a 9600 baud GFSK signal demodulation scheme. It explains the implemented software in-depth and also discusses some evaluation of the quality of the developed systems. The satellite groundstation consists of a National Instruments (NI) *radio frequency* (RF) Front End - NI USRP-2922 and signal processing software designed in GNU Radio Companion (GRC) in a Linux environment. The signal processing software is organized in a GRC *flowgraph* and is able to interface with the RF Front End in real-time.

1.2 Organization of Study

Chapter 2 presents a theoretical background and an introduction to SDRs and *digital signal processing* (DSP). It also provides literature references covering basic theoretical background. Chapter 3 introduces the used hardware and software, and presents and specifies the communication system environment. Chapter 4 explains the development process, and boundary conditions for the determination of the SDR sample rate. Furthermore it explains how to measure and evaluate the quality of the developed systems. Chapter 5 discusses in depth the sequence of signal processing steps the signal takes during demodulation and decoding. It explains the GNU Radio blocks used and the Python scripts. Chapter 6 is a tutorial on how to set up and operate the ground station. Chapter 7 presents the scheme of the real-time implementation of the PropCube Receiver. Chapter 8 presents the quality evaluations, such as a comparison of the output for the AFSK system and a *bit error rate* (BER) calculation for the GFSK signal. It also includes an interpretation of the results. Chapter 9 concludes the thesis and gives an outlook concerning future work. Appendix A covers detailed explanations for the functions and settings of the most important GNU Radio blocks used. Appendix A.2 presents tests carried out on the function and performance of certain GNU Radio blocks. Appendix B shows various flowgraphs used throughout the development of the ground station. Appendix C shows the main Python Scripts that are either a part of the ground station itself are a part of the evaluation of the ground station.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Literature Review and Theoretical Framework

This chapter covers an introduction to *Software Defined Radio* (SDR) and points out the distinction between hardware and software parts. In order to understand the actual groundstation as a system this chapter gives a theoretical background of SDR in general and implemented functionalities. Further it gives an overview of *digital signal processing* (DSP). In following chapters these steps will be put together into a working system. The section on 2.2.3 introduces fundamental DSP functions basic to this thesis. Also, it provides literature references for in-depth information on each topic. The final section gives an overview of related work on radio ground stations, that are similar to this one.

2.1 Software Defined Radio

Jeffrey H. Reed defines *Software Defined Radio* (SDR) in his textbook *Software Radio: A Modern Approach to Radio Engineering* as follows: "[An SDR is a] radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software" [1, p. 2]. "Substantially" means that the main parts of the signal processing chain, especially modulation, and also encryption and error correction have to be done by software. Usually all these parts are reprogrammable as well [1, p. 2]. Being derived from software, an SDR has greater flexibility than *Hardware Defined Radio* (HDR) [2, p. 1]. An SDR can handle a wide range of carrier frequencies and modulation formats. Also it is very adaptable, as it can be integrated into multiple networks with various interfaces and different protocols. In addition, advanced types of error recovery can be achieved [1, p. 2]. See Reed [1] and Tuttlebee [3] for more detailed information on SDR and modes of implementation.

Figure 2.1 displays the schematic of a traditional hardware radio with all-analog signal processing. The signal processing is done by hardware components. By comparison,

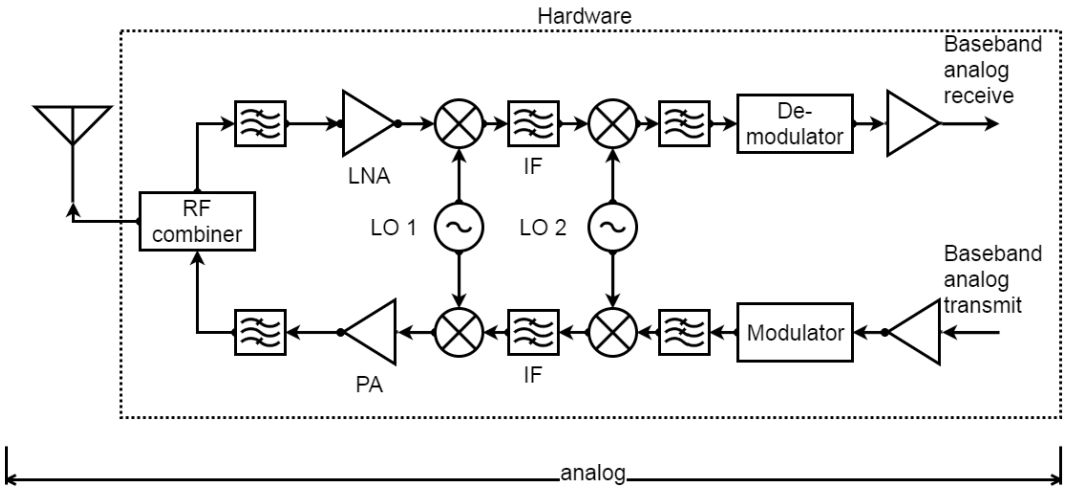

Figure 2.1. Traditional Hardware Radio architecture, see [2, p. 3].

Figure 2.2 shows the architecture of a NI USRP-292x/293x, which is the SDR used by the SSAG. In this SDR the signal gets converted from analog to digital on the receiving path and converted from digital to analog on the transmitting path. All following signal processing steps are digital and defined by software. The software defined modulation and demodulation is done by computer.

2.2 Digital Signal Processing Theory

In order to process signals with software, the signals need to be converted from analog to digital. An *analog-to-digital converter* (ADC) digitizes an analog signal by sampling, quantization, and encoding. For further information refer to Roehrig [5] and Mutagi [6]. In *Digital Signal Processing* (DSP) the samples of the digitized signal are processed using functions derived from analog signal processing [3].

2.2.1 Digital Front End

The *Front End* includes all functionalities enabling processing steps the signal undergoes between the antenna and the baseband processing. The steps within the Front End, which are software programmable, are termed the *Digital Front End* (DFE) [3, p. 151]. In the DFE a digital signal is either converted to analog or an analog signal is

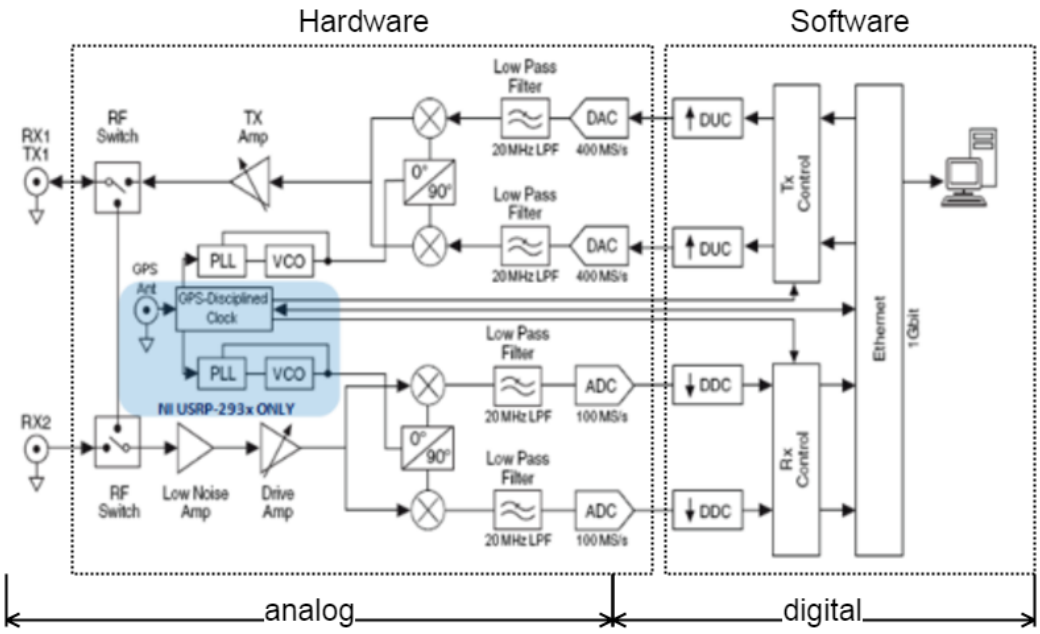

Figure 2.2. RF software-programmable Radio Transceiver architecture. NI USRP-292x/293x, see [4].

converted to digital. A digital signal is converted to analog, when an analog signal is required for transmission. An analog signal may be converted to a digital signal for digital baseband processing. Figure 2.2 shows a Radio Front End. The part marked as "analog" is the *Analog Front End* and the part marked as "digital" is the *Digital Front End*. Tuttlebee explains: "[In a digital receiving sequence a DFE] must provide a *digital* signal of a certain *bandwidth*, at a certain *center frequency*, and with a certain *sample rate*" [3, p. 152].

The functionalities of a DFE include digital-to-analog conversion, analog-to-digital conversion, digital up- and down-conversion, filtering and resampling [3] [1] [4]. All signals need to undergo these steps, no matter which digital baseband processing follows. Thus, there are a number of field programmable DFEs available off-the-shelf, such as National Instruments' Universal Software Radio Peripheral (USRP) [7].

2.2.2 Digital Baseband Processing

All *digital signal processing* (DSP) steps between the in- or output data stream and the Front End are termed *digital baseband processing* [3]. The single steps are specific to the modulation scheme and the data-encoding scheme, hence different for any transceiving unit. Beasley and Miller present all parts of digital baseband processing in their book "Modern Electronic Communication" [8]. For an in-depth theoretical background, see Xiong [9].

2.2.3 DSP Literature References

For detailed information on *bandpass filter* design and working schemes, see [10] [11, p. 109] [3, p. 167] or [12]. *Multirate Filtering* is used in order to reduce multiplications per second, as well as reducing hardware effort [3, p. 171]. The advantages and function of Finite Impulse Response Filter are found in [13] and [14], as well as [3], which provides an example of a C and Assembly Code implementation. The procedure of *sample rate conversion* or *resampling* is explained in [3, p. 181] and [15]. A *matched filter*, such as a *root raised cosine filter* (RRC), extracts a single frequency. The RRC is briefly explained in [16], whereas Haykin broadly explains *matched filters* [17, p. 248], while Joost explains RRC in more detail [18] . The data is encoded in *nonreturn-to-zero inverted* (NRZI) code. The Article on [21] briefly explains NRZI. In GFSK9600 the bitstream gets randomized before modulation. The process of *randomizing* or *scrambling* is explained by Carlson [11]. In both AFSK1200 and GFSK9600, the data is packetized following AX.25 standard, which is defined by the Tucson Amateur Packet Radio Corporation (TAPR) [22]. A summary of the AX.25 standard is given in [23].

Tuttlebee [3] gives a good overview about SDR basics, as well as DSP implementation. For *signal processing* in general, see Haykin [17]. Beasley and Miller [8] provide a detailed explanation of *electronic communication*.

2.3 Related Work on Software Defined Radio Ground-stations

Similar to the AFSK part of this ground station, there is an SDR AFSK demodulation project. The project was created in in June 2013 by Volker Schroer (HAM radio call sign: dl1ksv) [24]. The AFSK demodulation scheme developed in this thesis is based on the project by Schroer. Related to the GFSK demodulation scheme, there are two projects: First, there is the BisonSat ground station [25]. Second, there is the ESTCube ground station [26], [27].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Hardware and Software Environment

This chapter introduces the hardware and software used, including technical data, and why this hardware and software was selected for use by the SSAG. Kopitzki evaluated different hardware and software at NPS leading to implementing the USRP in cooperation with GNU Radio. His investigation is the basis for this thesis and can be reviewed in [28] and [29]. Further, this chapter specifies the communication systems depending on the *radio frequency* (RF) payload, as well as a working scheme of each system.

3.1 Hardware used

The presented hardware comes in each case with needed infrastructure and other hardware, such as computers, antennas, preamplifiers or local oscillators. The additional devices are not specifically discussed in this thesis. The following sections briefly present the main hardware devices used in the development and use of the ground station.

3.1.1 RF Front End - NI USRP-2922

The *RF Front End* selected for use is the *USRP-2922* by National Instruments (NI). Working at a frequency range of $f_{min} - f_{max} = 400MHz - 4.4GHz$ and up to a bandwidth of $B = 20MHz$, it operates in the UHF spectrum, which is used by the CubeSats launched by the SSAG [4] [30]. It is an affordable RF transceiver and can be used with various modulation and encoding schemes [4]. For fast data transmission, the USRP is connected to the computer via an ethernet cable. Depending on individual tasks, different types of antennas may be connected to the device.

3.1.2 APRS Transmitter - BigRedBee

BigRedBee is a GPS APRS transmitter. It transmits its GPS positions via the *Automatic Packet Reporting System* (APRS) using AX.25 protocol. The BigRedBee may operate at a frequency range of $f_{min} - f_{max} = 420 - 450MHz$ at a data rate of $r = 1200baud$ [31]. In the SSAG the BigRedBee is used as a RF payload in a summer internship project in 2016 on *high altitude balloons* (HAB).

3.1.3 TNC - Kantronics WirelessModem-9612

To display correctly, received packets, as well as to transmit known packets, a *Kantronics* device is used as a *Terminal Node Control* (TNC). The *Kantronics WirelessModem-9612* enables the user with different settings among other things to send and receive files at different, defined levels of data protocol layers [32]. It works together with a computer and an amateur radio ground station.

3.1.4 Ground-Based PropCube Simulator

PropCube is a series of three CubeSats used by the Naval Research Laboratory (NRL). The developers state:

> "The PROPCUBE mission is of interest to NRL basic research to demonstrate (1) the capability of beacon CubeSats for determining potentially harmful effects of the ionosphere on radio systems, (2) techniques for tracking multiple CubeSats immediately after deployment from a launch vehicle, and (3) showing the added value of radio beacons for complementing ground based soundings of the ionosphere" [33].

Two of the PropCube CubeSats (Flora and Merryweather) have been launched into *low earth orbit* (LEO) in October, 2015 as part of the GRACE mission. The launch of the third one, Fauna, is scheduled for October, 2016 [34]. The command and control frequencies for PropCube are $f_{up} = 449.775MHz$ uplink and $f_{down} = 914.00MHz$ downlink. The signal is GFSK modulated at a data rate of $r = 9600baud$ [30]. A ground-based simulator is used for testing purposes in this thesis. For command and

control purposes, the ground-based simulator operates identically to the PropCube's on orbit.

3.2 Used Software - GNU Radio

GNU Radio is an open source software development toolkit that enables the user to design and implement software radios with processing blocks and processing runtime [35]. The users may use pre-configured blocks or write and implement their own blocks which can be written in C++ or Python [36]. The GNU Radio web page says: "[GNU Radio] is widely used in hobbyist, academic and commercial environments to support wireless communications research as well as to implement real-world radio systems" [35]. The SSAG uses GNU Radio, because it is an inexpensive solution, which enables them to flexible develop, improve and modify their communication systems, utilizing off-the-shelf Software Defined Radios.

GNU Radio Companion (GRC) is a graphical programming tool to design signal processing flowgraphs. It automatically generates the dependent source code [37]. For further information on how to launch and use GRC see Kopitzki [29] or the GNU Radio web site [37]. Some of the used GRC blocks are excerpted from other projects and are not pre installed with GNU Radio. All necessary files can be found on the CD that supplements this thesis. Section 6.1 explains how to build and install all needed dependencies.

3.3 Communication System Specifications

This sections covers an brief explanation on the properties, design and architecture of both communication systems implemented in this ground station. Also it describes the physical layer protocol, which is used in each system.

3.3.1 BigRedBee - AFSK1200

With the used settings the BigRedBee is transmitting at a carrier frequency of $f_c = 433.92MHz$ at a data rate of $r = 1200baud$. The baseband is *Audio Frequency Shift*

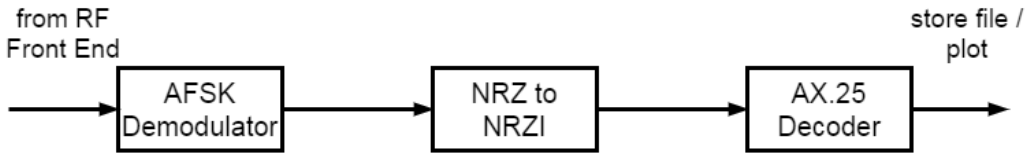

Figure 3.1. Scheme of a 1200 baud AFSK receiving unit.

Keying modulated. Basically, AFSK is an FSK signal which is then FM modulated. The mark is $f_{mark} = 1200Hz$, whilst the space is $f_{space} = 2200Hz$ [38]. The data stream is NRZI encoded [39]. The data packet format is defined by AX.25 data link layer protocol, which is derived from the *High Level Data Link Control* (HDLC) protocol [39], [22], [40]. Figure 3.1 shows the schematic of a 1200 baud AFSK modem on the receiving side.

Figure 3.2 shows the communication system. The BigRedBee receives a GPS signal. It transmits its position via APRS at the downlink frequency. The USRP SDR receives the signal and passes it to the computer, which runs GNU Radio. The signal is demodulated and decoded in GNU Radio and the data is stored to a file after that. Replacing the Computer with a Laptop turns the communication system into a mobile system, equipping the track vehicle.

3.3.2 PropCube - GFSK9600

The PropCube transmits at a carrier frequency of $f_c = 914.00MHz$ with a data rate of $r = 9600baud$. The signal is *gaussian frequency shift keying* (GFSK) modulated with a bandwidth of about $B = 10kHz$. The bit stream is scrambled and NRZI encoded. The data packet format is defined by a AX.25 data link layer protocol [41], [42], [40]. Figure 3.3 shows the schematic of a 9600 baud GFSK modem on the receiving side.

Figure 3.4 shows the communication system. The satellite receives an uplink, transmitted by the ground station and replies. The USRP SDR receives the replied downlink and passes it to the computer, which runs GNU Radio. The signal is demodulated in GNU Radio and passes the data to a Python script, which decodes the data. The

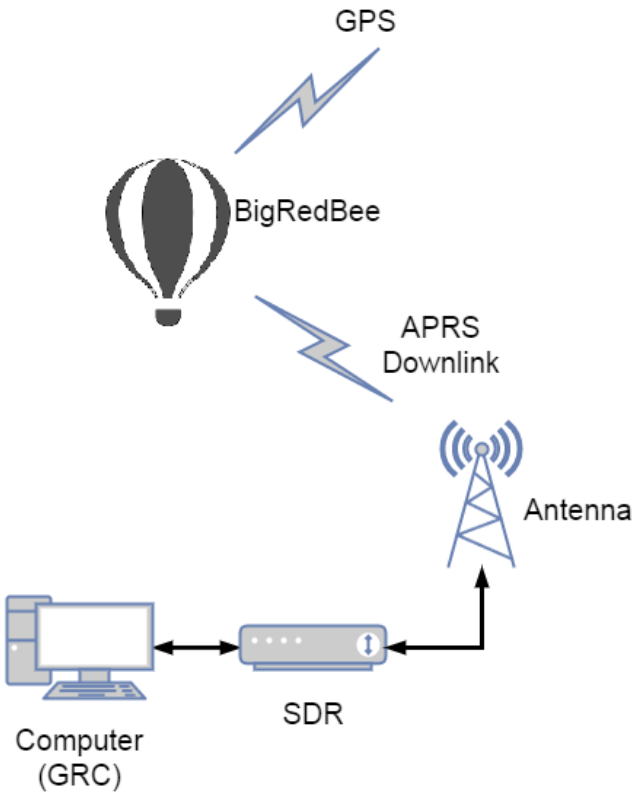

Figure 3.2. BigRedBee APRS Beacon Setup.

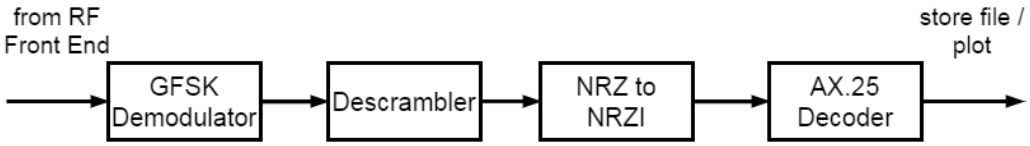

Figure 3.3. Scheme of a 9600 baud GFSK receiving unit.

decoded data is sent to a software provided by the satellite manufacturer. In this thesis, the developed ground station includes just the receiving part. The transmitting part is done by the currently operating ground station via hardware radio.

3.3.3 PropCube - Packet Encoding

Figure 3.5 shows the packet structure of the PropCube signal on the transmission side. it also shows the sequence of physical layer operations, such as *bit stuffing*, *NRZI to*

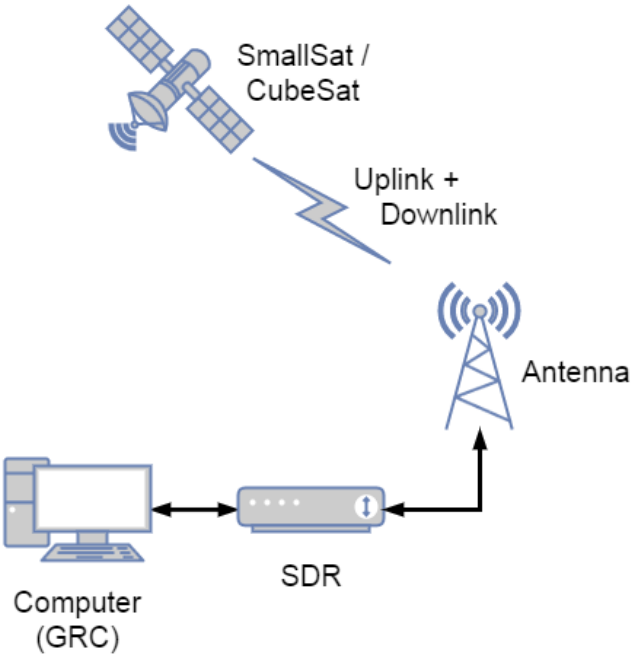

Figure 3.4. Satellite Groundstation Setup.

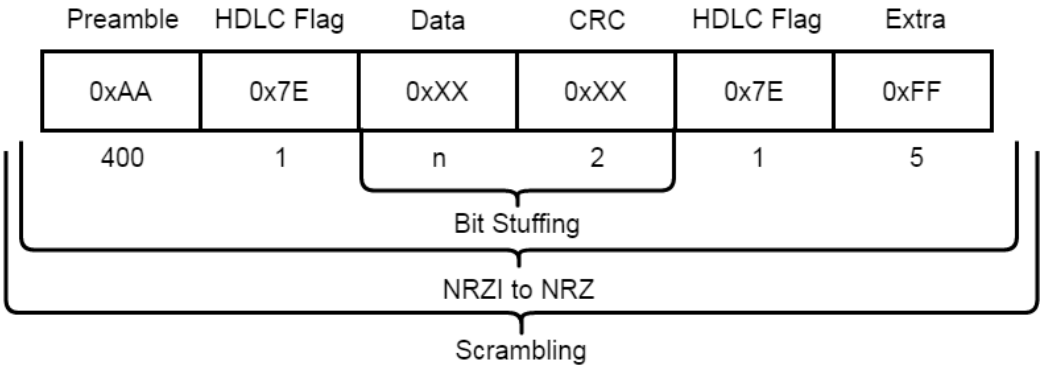

Figure 3.5. PropCube Packet and Protocol Structure.

NRZ conversion and *scrambling*. The stated physical layer encoding structure belongs to the *9600Baud G3RUH Packet Radio Modem* [40], [41]. The packet structure itself is special to the PropCube signal, whilst it conforms the AX.25 standard [30], [23]. Figure 3.5 shows, a 16 bit *cyclic redundancy check* (CRC16) sequence is calculated on the data bytes and appended. The bit stream including the data and the CRC16 checksum is bit stuffed. The bit stuffed stream is framed by HDLC flags "0x7E" and a 400 byte long preamble "0xAA", respectively an extra of 5 bytes "0xFF" at the

end. The whole packet is expressed in NRZI code, which is then converted into NRZ code. The NRZ encoded signal is randomized and passed to the modulator.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Groundstation Development

This chapter presents the process of development of the groundstation. Also, it covers a description on how to determine the right sample rate for the system. The chapter explains the specific test setups for the BRB system, and then for the PropCube system. It concludes with an introduction of how to measure the quality and scope the results.

4.1 Development

The development of the ground station started by stating key requirements. Key requirements are demodulation of a APRS signal, as well as the PropCube downlink. First, analog demodulation schemes were investigated. Second, the GNU Radio library was searched for blocks to rebuild an analog demodulation digitally. Third, every single block was tested to determine or confirm it's functions, since documentation is not provided for every block. Testing methods and results are presented in Appendix A.2. For some functionalities, blocks developed by other needed to be used. Single blocks were then grouped together and tested as subsystems. All subsystems together comprise the whole system which was then tested. Finally, the whole system was integrated into the real-time communication system, and tested again. The tests were based on signals generated by source blocks within the GRC or recorded downlink signal sequences, recorded with the USRP and GRC. A real-time downlink signal was used for the final tests in each case.

4.2 Sample Rate Determination

The USRP is able to deliver a minimum sample rate of $r_s = 195.312sps$. This rate was provided via email by Ettus Research LLC [43]. The ADC inside the USRP uses a sample rate of $r_s = 100Msps$ [4]. The sample rate the device puts out, has to be chosen depending on the frequency band of interest, observing the Nyquist

theorem [44]. The Ettus support team also recommended choosing a device sample rate that is an integer fraction of the basic $r_s = 100Msps$ [43].

4.3 Test Setups

Figures 3.2 and 4.1 show the test setups for the hardware-in-the-loop integration test of the BigRedBee and the PropCube. The same setups are used to record sample signals for software-in-the-loop and subsystem tests, as well.

4.3.1 BigRedBee

Figure 3.2 displays the BigRedBee test setup. The BigRedBee receives a GPS signal and transmits its position via the APRS downlink signal at a frequency of $f_{downlink} = 433.92MHz$. This signal is received by the USRP SDR via an antenna. The USRP converts the analog signal to a digital signal and passes it on to the computer. On the computer, the signal is processed utilizing GNU Radio. The output data is stored to a file. For software-in-the-loop tests, a stored raw signal is reprocessed. For a final hardware-in-the-loop integration test, a tracking vehicle of the *HAB summer intern project* is equipped with a mobile antenna, the USRP SDR and a laptop, to record and process real time flight data. The mobile communication system is described in Section 3.3.1.

4.3.2 PropCube

Figure 4.1 shows the setup for the ground based PropCube simulator. One computer sends pings via the used modem to the satellite simulator. The PropCube simulator replies at a downlink frequency of $f_{downlink} = 914.00MHz$. The downlink signal is received in parallel by an off-the-shelf amateur radio and the USRP SDR. The amateur radio passes the signal to a Kantronics WirelessModem-9612, which acts as a TNC. It demodulates and decodes the signal and passes the data on to the computer. The computer stores the received data to a file.Meanwhile, the USRP SDR receives the same signal as the amateur radio. The USRP passes the received signal on to a

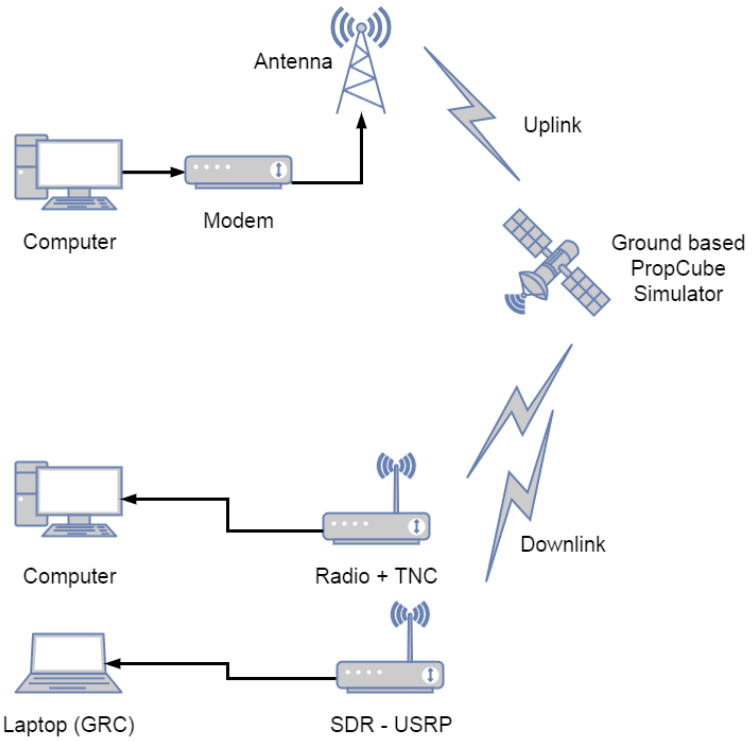

Figure 4.1. Test Setup for the PropCube downlink demodulation and decoding scheme.

laptop, which runs GNU Radio. The signal is demodulated using the GNU Radio. A Python script decodes the signal and stores the data to a file.

4.4 How to Measure and Scope Results

One way to evaluate the quality of the demodulation is to compare the number of detected transmissions with the number of recovered packets. The number of noticeable transmissions is specified by the number bursts at the frequency band of interest, visible in a waterfall plot. The number of correctly recovered packets is the number of packets that pass the CRC check. In case of the AFSK1200 demodulation, one recovered packet is one printed packet by the AX25decode block in debug level <= 4, which only prints packets that pass the CRC check. In case of the GFSK9600 demodulation, the recovered packets have to be compared to the equivalent packets, and printed by the TNC in a plain "KISS" mode, since there is no CRC check

implemented in the HDLC decoding / "HDLC_unframe_and_unstuff.py" script. The script can be found in Appendix A.1.1. In KISS mode, the TNC passes received data at the lowest possible level of decoding. A TNC in KISS mode only un-frames the data and checks the CRC [45]. Due to its operation, the TNC replaces '0xC0' with '0xBD 0xDC', '0xBD' with '0xBD 0xDD', as well as frames decoded packets with '0xC0' at each end [45]. In order to compare the TNC print out and the "HDLC_unframe_and_unstuff.py" print out, these frames and replacements must be undone. If both outputs match 100%, the tested system is assumed to be perfect.

A *bit error rate* (BER) depending on a varying *signal-to-noise-ratio* (SNR) is determined, to evaluate the actual quality of the tested system. Figure A.1 shows the GRC flowgraph that has been used to determine the *Signal-to-Noise-Ratio* (SNR). The following section explains how to determine a BER.

4.4.1 Bit Error Rate

The *bit error rate* (BER) obeys the following equation:

$$BER = \frac{n_{be}}{n_t}. \tag{4.1}$$

Here, n_{be} is the number of bit errors and n_t is the number of bits sent.

The Python script "BER_find.py" reads a set of demodulated data, defined by the user. The script can be found in Appendix A.1.2. "BER_find.py" compares the detected HDLC packets bit per bit. It finds and counts bit errors. However, if there is a bit error in the preamble, close to the HDLC flag, or in the HDLC flag, "BER_find.py" is not able to detect a given packet. In this case, the script skips this entire packet and adds the number of bits of the lost packet to the number of bit errors. All bits following a bit loss or a bit add that cause a shift are detected as false. "BER_find.py" is not able to detect and handle a bit shift. In the case of a bit shift,

the data has to be manipulated manually by adding or deleting a bit at a certain position. This manipulation has to be taken in account for the BER calculation by the user. The script creates one file including the detected packets and one file with the positions of detected errors for each input file, as well as one file with a list of the calculated BER per input file. The calculated values have to be comparable to the values given by the following equations:

$$BER_{coh.BFSK} = \frac{1}{2} erfc(\sqrt{E_b/2N_0}), \tag{4.2}$$

for a coherent binary FSK signal and for a MSK signal:

$$BER_{MSK} = \frac{1}{2} erfc(\sqrt{E_b/N_0}). \tag{4.3}$$

Equation 4.2 and Equation 4.3 define an estimate for the BER for the demodulation, since MSK is the optimum case of FSK with an excellent power efficiency [9, p. 15, p. 288].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Presentation of Demodulation Schemes

This chapter shows the two implemented demodulation schemes. The GNU Radio flowgraph of each is presented, and the settings of each block are discussed. For functionalities of a single block, see Appendix A.1 or go to [46] or [47].

5.1 AFSK Demodulation and Decoding Flowgraph

Figure 5.1 shows the GNU Radio flowgraph of the AFSK demodulation. The overall sample rate is set to $r_s = 400kSps$, as it is an integer fraction of the basic $r_s = 100MSps$ of the USRP. Section 4.2 explains how to calculate the correct sample rate. The signal is delivered from the USRP source, which is connected to the computer via ethernet cable. The raw signal is stored. Parallel, the signal is processed. After shifting and filtering, the signal is FM-demodulated. Mark frequency and space frequency are detected and combined to a NRZ baseband. The NRZ baseband is fed into a block called *AX25decode* and stored as the encoded data after that.

UHD: USRP Source

The field "Device Address" is set to the device's IP address. The device's *Center Frequency* is off-tuned by $f_{offset} = 20kHz$. This off-tune is performed, so that no DC-bias or other artifact coming from the local oscillator inside the USRP affects the frequency band of interest. The *Gain Value* is set to "0" as a default and can be adjusted. A low default value is chosen, so the device is not oversaturated. In the field "Antenna," the connected antenna in-/output port is specified.

Rational Resampler

The device sample rate is set to $r_s = 400ksps$. The *Rational Resampler* block resamples the signal to:

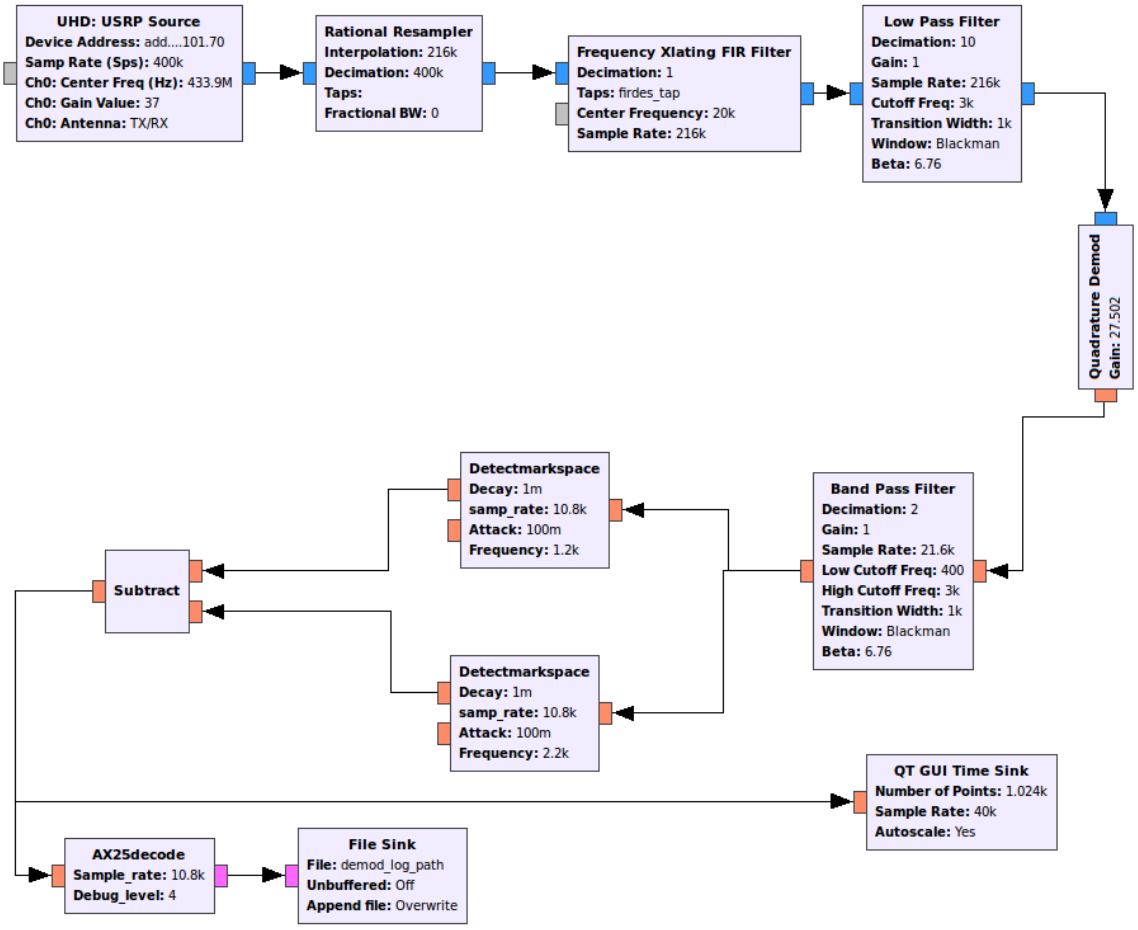

Figure 5.1. GNU Radio Flowgraph of the AFSK Demodulation.

$$r_{s\_new} = r_{s\_old} \times resamp\_quot, \tag{5.1}$$

whereas the "resampling quotient" is:

$$resamp\_quot = \frac{data\_rate \times samples\_per\_symbol \times decimation\_factor}{r_s}. \tag{5.2}$$

The factor "data_rate" is the data rate of the signal, which is $r = 1200baud$ in this case. The "samples_per_symbol" factor is defined by the user and has to conform to the value set in the clock recovery inside the *AX.25decode* block. The "decimation_factor" depends on the sample rate decimation throughout the demodulation. It is the sum of all decimation factors between the *Rational Resampler* block and the clock recovery, which happens inside the *AX.25decode* block. The value r_s is the input sample rate of the *Rational Resampler* block.

Frequency Xlating FIR Filter

The decimation of the block is set to "1," thus, no decimation occurs. The filter taps are defined as:

$$firdes\_taps = firdes.low\_pass(1, samp\_rate, samp\_rate/2, 25000, firdes.WIN\_BLACKMAN, 6.76)$$

This expression specifies that the FIR Filter filters with *low pass* behavior, with the following properties in order: filter gain : 1, sampling frequency : samp_rate, center of transition band conforming to the Nyquist theorem, transition width : 25000 Hz, a Blackman window, followed by the beta parameter, which only applies for a Kaiser window [48].

The center frequency of the FIR filter is set to $f_{center} = 20kHz$. This factor means, that the center frequency of the signal is shifted by this amount. By setting the center frequency to $f_{center} = 20kHz$ the initial tune offset is removed. The sample rate of this block has to conform to the output sample rate of the previous block.

Low Pass Filter

The next two steps are combined within a *multirate filter*. On the one hand the number of samples is decimated to reduce the sample rate, while on the other hand the signal is limited to the frequency band of interest. The *low pass filter* decimates by

10. The filter gain is set as "1" by default. The sample rate has to match the value of the output sample rate of the previous block. The highest occurring frequency of interest is the space frequency at $f_{space} = 2200Hz$. Thus a cutoff frequency of $f_{cutoff} = 3KHz$ is sufficient. For the cutoff frequency, the output sample rate of this block has to conform to the Nyquist theorem. A Blackman window is chosen for its steep rising flanks. The *beta* value is left at it's default, as it is just for the Kaiser window [46].

5.1.1 Quadrature Demodulation

For the *quadrature demodulation gain* the default value has to be adjusted depending on the input sample rate. In this case the gain is:

$$gain = \frac{r_{s\_total}}{dec\_factor \times 2 \times \pi \times deviation \times 8}. \tag{5.3}$$

The decimation factor has to be implemented, since the initial sample rate r_{s_total} is decimated in the low pass filter.

5.1.2 Band Pass Filter

This band pass filter is implemented as a multirate filter. It decimates by two, as it filters the frequency band of interest. Mark frequency of the signal is $f_{mark} = 1200Hz$, whereas space frequency is $f_{space} = 2200Hz$. Thus it is sufficient to set the *low cutoff frequency* to $f_l = 900Hz$ and the *high cutoff frequency* to $f_u = 3000Hz$. The cutoff frequencies are set, depending on the roll offs of the mark frequency peak and the space frequency peak.

5.1.3 DetectMarkSpace

The *Detectmarkspace* block is a hierarchical block, which means it incorporades a number of blocks that work together as the functionality of mark frequency detection

and space frequency detection. This hierarchical block was constructed out of given GNU Radio blocks by Schroer (dl1ksv) [24]. For a detailed explanation on the working scheme and the contained functionalities, see Appendix A. The *Decay* and *Attack* values are set to their default values, which are $Decay = 10^{-3}$ and $Attack = 10^{-1}$. The blocks' sample rate conforms to the input sample rate of $r_s = 10.8ksps$. The *Frequency* field specifies the frequency to detect. In this case there are two parallel *Detectmarkspace* blocks. One detects the mark frequency at $f_{mark} = 1200Hz$, the other one detects the space frequency at $f_{space} = 2200Hz$.

5.1.4 Subtract

The output values of the space detecting block are subtracted from the output of the mark detecting block. This causes a signal, which alternates around zero over time. If the input of the detectors is close or equal to $f_{in} = 1200Hz$ the result of the subtraction is positive; if the input is close or equal to $f_{in} = 2200Hz$ the result of the subtraction is negative. The output signal is basically the mark and space transferred into a NRZ encoded baseband signal.

5.1.5 AX25decode

The *AX25decode* block takes the NRZ encoded signal at a sample rate of $r_s = 48ksps$. The *Debug_level* may be any number between 1 and 5. In this case the *Debug_level* is 4, which defines the block's output to be decoded AX.25 packets with a valid CRC. The output of the block are decoded APRS-Packets. For more information on this block and its properties in particular, see A.1

5.1.6 File Sink

The stream of data sent into this block is stored at a directory on the hard drive, specified by the variable "demod_log_path." This variable contains a log path, as well as a file name. The file name contains a time stamp, as well as dependent information on the recording, such as sample rate and center frequency of the device. This information is important for the user, for reprocessing of the data.

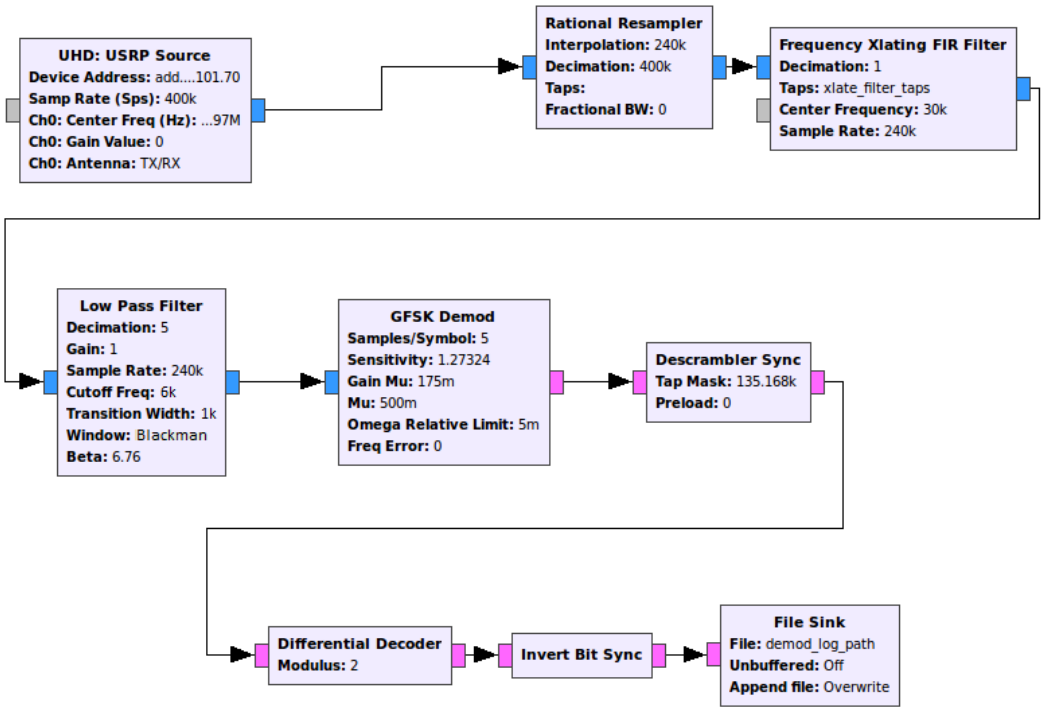

Figure 5.2. GNU Radio Flowgraph of the GFSK Demodulation.

5.2 GFSK Demodulation Flowgraph

Figure 5.2 shows the GRC flowgraph of the GFSK demodulation. The overall sample rate is set to $r_s = 400kSps$. Section 4.2 explains how to calculate the correct sample rate. The signal is delivered from the USRP source, which is connected to the computer via ethernet cable. The signal is stored and processed in parallel. After shifting and filtering, the signal is FSK-demodulated. After descrambling, the NRZ signal is transformed into a NRZI signal. The NRZI baseband is output to a file. The settings of the individual blocks mainly conform to the settings of the blocks in the AFSK demodulation flowgraph. This section discusses only the settings that are different from those in the AFSK demodulation.

5.2.1 UHD: USRP Source

Similar to the AFSK demodulation flowgraph, there is a *UHD: USRP Source* block at the beginning. Again, the field "Device Address" is set to the device's IP address. The

downlink frequency of the PropCube Satellites varies from $f_{downlink} = 913.975MHz$ to $914.025MHz$, because of the Doppler shift. Thus, the device's *Center Frequency* is shifted to $f_{USRP} = 913.97MHz$, meaning it is off-tuned by $f_{offset} = 30kHz$. Other values are configured in a similar way to the *UHD: USRP Source* block in the AFSK demodulation.

5.2.2 Rational Resampler

The same way as in the AFSK demodulation, the device sample rate is set to $r_s = 400ksps$ as it represents an integer fraction of the basic 100Msps (see 4.2). The *Rational Resampler* block resamples the signal to:

$$r_{s\_new} = r_{s\_old} \times resamp\_quot, \tag{5.4}$$

whereas the "resampling quotient" is:

$$resamp\_quot = \frac{data\_rate \times samples\_per\_symbol \times decimation\_factor}{r_s}. \tag{5.5}$$

For a detailed explanation of the factors, refer to Section 5.1.

5.2.3 Frequency Xlating FIR Filter

The value of the center frequency varies according to the doppler shift. As the ground based PropCube simulator has a downlink frequency of $f_{downlink} = 914MHz$, the center frequency is set to $f_{center} = 20kHz$. This center frequency is derived from $f_{downlink} - f_{USRP} = 914MHz - 913.975MHz$. The filter taps are defined as

$xlate_filter\_taps =$
$firdes.low\_pass(1, samp\_rate, samp\_rate/2, 25000, firdes.WIN\_BLACKMAN, 6.76)$

5.2.4 Low Pass Filter

The required bandwidth of a 9600 baud binary data signal is $B = 4800Hz$ [41]. Thus, it also is the frequency band of interest. An empirical study results in the best output of the demodulation, when filtering with a cutoff frequency of $f_{cutoff} = 6kHz$. The test itself is presented in Appendix A.2.

5.2.5 GFSK Demod

The number of samples per symbols is set to "5," as this value works best for the demodulation. This value has been evaluated, as presented in Appendix A.2. For the Sensitivity the following equation is valid:

$$Sensitivity = fracsamp\_rate 2 \times \pi \times fsk_{deviation} \quad (5.6)$$

The other settings of the *GFSK Demod* block are left at the default values.

Descrambler Sync

The Tap Mask of the descrambler is set to 135168_{10}, which is the decimal representation of the binary number 1000010000000000000_2. The binary number specifies the settings of the shift register of the descrambler [42].

5.2.6 Differential Decoder and Invert Bit Sync

The set of these two blocks performs the NRZ to NRZI transformation.

5.2.7 File Sink

The NRZI baseband signal is output to a file. Directory and filename are specified in the field "File."

5.3 AX.25 Decoding

The AX.25 decoding for the AFSK1200 signal is done within the *AX.25decode* block. For the 9600 baud signal, the decoding is done via a Python script. The script "HDLC_unframe_and_unstuff.py" takes an NRZI bit stream packed one significant bit per byte (LSB). The script detects and aligns AX.25 packets and repacks the bits into bytes with eight significant bits. It also bit un-stuffs the stream. The script follows the counter sequence of the packetizing described in Section 3.3.3

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
How to use the Groundstation

This chapter covers an explanation on how to build the developed ground station on a computer, on which GNU Radio is installed. The ground station is designed for a Linux Ubuntu system, however there is a way to install GNU Radio and the ground station on a Windows system, too. For a tutorial, see [49].

6.1 How to set up the Groundstation

To set up the ground station the folder "ground station" must be copied to the desktop and the current directory of the terminal window needs to be the ground station folder. Further, the block libraries of the *AX.25decode* block, as well as some blocks for the GFSK demodulation need to be built. Within the folder "ground station" is a file named "README." The steps described in the "README" file must be taken to build the blocks, as well as other directories.

The SDR needs to be connected to an antenna and a *low noise amplifier* (LNA). On the output side, the SDR must be linked to the computer. The last step is to set the values of the *UHD: USRP source* block properties in the GRC flowgraphs properly. The properties include the IP address of the SDR, as well as the antenna port specification.

6.2 How to run the Groundstation

This section explains how to start and how to operate each of the two demodulation flowgraphs and the dependent Python scripts.

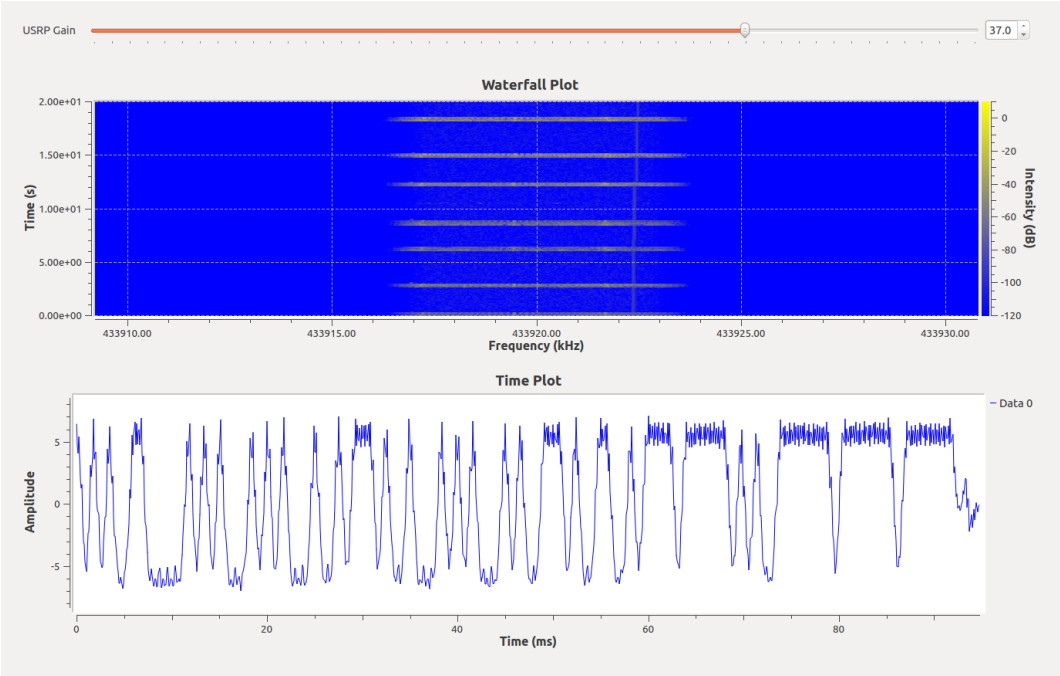

Figure 6.1. Graphical User Interface of the BigRedBee Groundstation.

6.2.1 BigRedBee Groundstation

The *AX25decode* block has been created in a version different from the current GNU Radio version. Thus, the *shared library* path has to be adjusted prior to every start of the ground station. Therefore, the command:

```
\$ export LD\_LIBRARY\_PATH=/usr/local/lib
```

must be entered into the terminal window. Following this command, the Python file including the flowgraph must be opened via the same terminal window, with:

```
$ cd <file directory>
```

```
$ python BRB_receiver.py
```

Figure 6.1 shows the GUI of the BRB ground station. The GUI includes a slider to vary the *USRP Gain*. The default of the USRP gain is zero, in order not to

over saturate the USRP. The waterfall diagram shows the FFT of the signal at a center frequency of $f_{center} = 433.92Hz$. The time plot underneath the frequency is an indicator, whether the flowgraph is demodulating the received signal. In order to adjust specific settings, such as the carrier frequency, the flowgraph must be opened in GRC.

6.2.2 PropCube Groundstation

Figure 6.2 shows the GUI of the PropCube ground station. The user is able to adjust the gain of the USRP. The waterfall plot indicates the occurrence of signal peak, as well as their position in the frequency spectrum. The *Tune Frequency* slider enables the user to manually center the signal. The time plot indicates, whether the received signal is demodulated. If the flowgraph is able to demodulate the signal, the time plot resembles a NRZ signal plot, as seen in Figure 6.2.

To decode the demodulated signal, the Python script "HDLC_unframe_and_unstuff.py" has to be opened via the command terminal. The script can be found in Appendix A.1.1. The stored, demodulated data file must be dragged into the terminal window. The Python script "HDLC_unframe_and_unstuff.py" stores the decoded packets in a separate file.

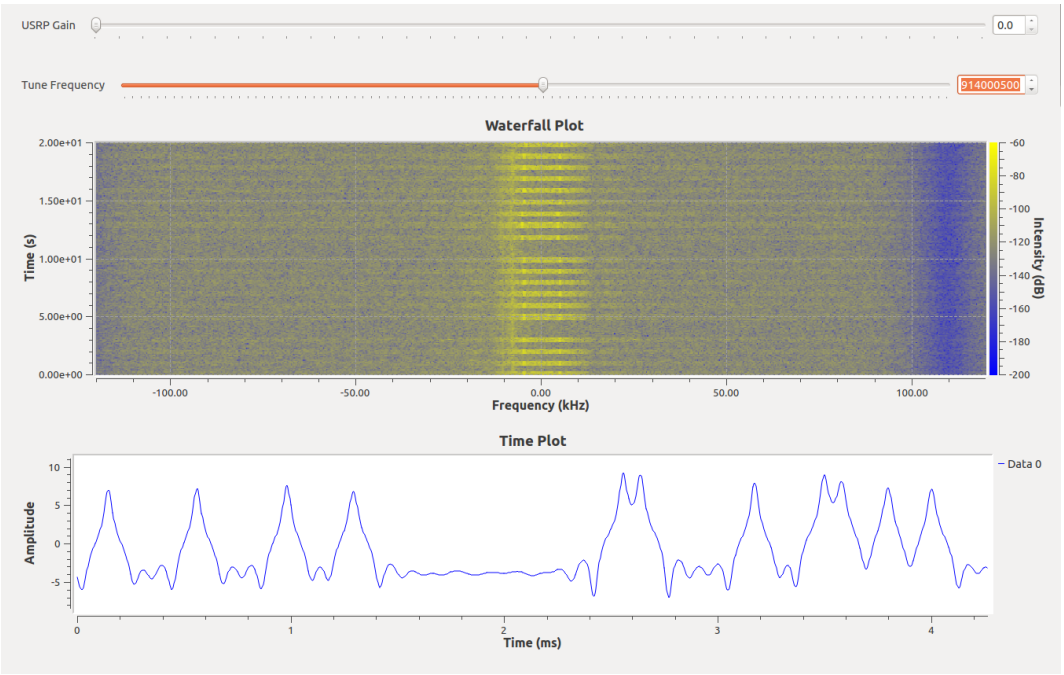

Figure 6.2. Graphical User Interface of the PropCube Groundstation.

CHAPTER 7:
Real-Time Implementation

This chapter is an add-on to the original thesis. The chapter explains the real-time implementation of the PropCube Receiver. First, the adjustments in the GNU Radio flowgraph are presented. Second, the "GRC_to_TYVAK.py" code is explained. This code includes HDLC frame detection, bit un-stuffing, *least significant bit* (LSB) first to *most significant bit* (MSB) first transformation, byte repacking, CRC calculation, KISS framing, and an interface to the command and control software of the manufacturer of the radio payload of the PropCube.

7.1 Demodulation Flowgraph

A number of blocks must be added to GFSK demodulation flowgraph in order to incorporate it into the ground-station. A number sink is implemented to show the current receive frequency. A *Power Squelch* block is implemented to limit the signal to bursts only, where the signal has an amplitude that is higher than a defined level. The power or amplitude level is defined in [dB] by the user. The actual values for the threshold must be empirically evaluated with an actual PropCube downlink signal. The "Alpha" value specifies the averaging filter of the block. "Alpha" requires a value between zero and one, where one means no averaging and zero means averaging overall samples [50], [46]. An appropriate value must be evaluated the same way as an appropriate threshold value. "Ramp" specifies the attack and decay phases [46]. In this flowgraph "Ramp" is set to zero. "Gate:Yes" means that the block has no output at all when the amplitude of the signal is below the threshold.

The *XMLRPC Server* block allows the user to remotely control variables of the flowgraph. By leaving the "Adress" field blank the GRC flowgraph can be reached via the IP address of the computing device. For the "Port" any address, which does not specify any port on the device so far can be chosen. In this case, the port is specified to $Port = 2651$. The following Python code example shows how to access variables of the flowgraph:

```
import xmlrpclib
import time

#replace <...> with device IP
s = xmlrpclib.Server('<device IP>:8082')
#replace <variable name>; <value>
#s.set_<variable name>(<value>)
#example for variable: "downlink_freq"; value:
    "914025000"
s.set_downlink_freq(914025000)

#crash
```

Accessed can be any variable which defines values of preferences that are underlined in the properties window of the GRC blocks. The option of remotely controlling variables is used to vary the downlink frequency of the demodulation scheme according to the real-time Doppler shift. Software written by G. Minelli, calculates and remotely introduces the appropriate frequency. Also, the threshold of the power squelch can be adjusted remotely.

The output of the flowgraph can be stored as a file with a *coordinated universal time* (UTC) time stamp. For the real-time implementation, the output is written to a "named pipe" or first-in-first-out (FIFO) file.

7.2 Decoding Software

The Python script "GRC_to_TYVAK.py" reads data from the named pipe, to which the GRC flowgraph writes. The script reads chunks of bytes. Each of the bytes carries one bit of information in the LSB. Eight of these bytes are one actual data byte with LSB sent first. Part of the decoding happening in this script is HDLC packet detection, bit un-stuffing, CRC check and a "bit flip". The bit flip takes the bytes, and flips them from LSB first to MSB first. After successful decoding, the data gets encoded in KISS packet format. The KISS packet format is explained in Section

4.4. The packet format is specified in [45]. The KISS packet are then written to a network socket. The socket is the connection between the decoding software and the command and control software. The software code can be found in Appendix A.1.3.

7.2.1 Cyclic Redundancy Check

The 16 bit *cyclic redundancy check* (CRC) used in the AX.25 packet encoding format is specified by the Consultative Committee for International Telephony and Telegraphy (CCITT), as well as in ISO 3309 [51]. A 16 bit CRC calculating function is included in the "GRC_to_TYVAK.py", in cooperation with a function that checks the CRC of received data, as well as a function that appends the CRC to a known message. An explanation of the working scheme can be found in the comments on the software code.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 8:
Data Analysis and Interpretation

This chapter covers a presentation of quality of the developed ground station. It presents some valuable output data from both demodulation schemes, as well as the results of a BER calculation on the PropCube communication system. Also, this chapter includes an interpretation of the presented data for the BigRedBee AFSK1200 system and the PropCube GFSK9600 system.

8.1 BRB - AFSK1200

Section 4.4 describes that the *AX25decode* block only outputs packets with valid CRC sequences, when in debug level <= 4. The file in Appendix A.17 shows that the demodulation scheme for the AFSK1200 signal plots packets, containing the GPS position of the BigRedBee. Thus the demodulation scheme presented in Section 5.1 works properly. Comparing the output of the currently used hardware radio in cooperation with the Kantronics TNC shows that the SDR outputs at a minimum the same amount of valid packets, and thus can replace the current system.

Interpretation

A specified objective of the developed communication system was to perform as well as the currently used system. As the plotted packets show, that the SDR system works as well as the hardware radio system. No further evaluation of the quality is needed.

8.2 PropCube - GFSK9600

The decoded data output of the the USRP SDR in cooperation with GNU Radio is attached on the CD supplementing this thesis, as well as the output of the currently used system. Comparing the two outputs shows that both solutions result in the exact same output. Thus, the quality of both demodulations and decoding schemes is equal on this level.

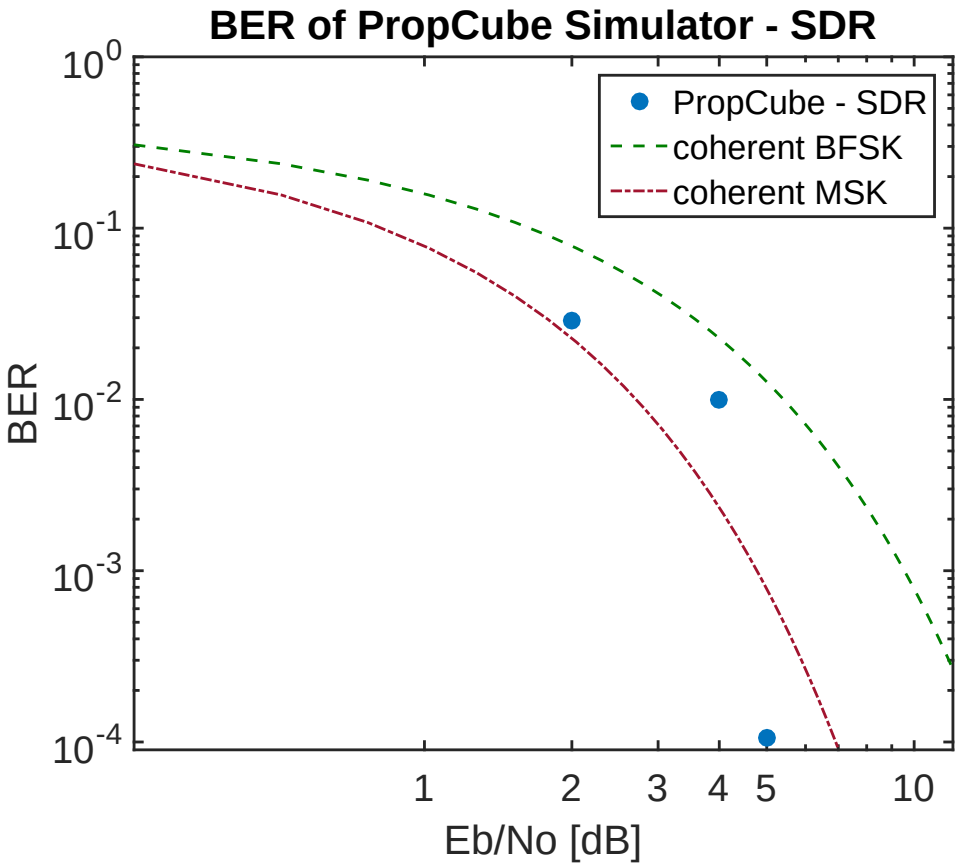

Figure 8.1. Bit Error Rate of 1) PropCube - SDR Setup 2) coherent MSK 3) coherent BFSK.

8.2.1 Bit Error Rate

To give a first order evaluation of the quality of the USRP SDR system, the bit error rate is evaluated. In the performed test 104 packets were transmitted and received. Therefore, the file contains about 10^5 bits. The BER evaluation at different SNRs results in the values plotted in Figure 8.1. The graph shows the theoretical BER over SNR for a coherent binary FSK signal and a coherent MSK signal. The equations for plotting the graphs are excerpted from [17, p. 417]. The results of the BER calculation are plotted as dots. The graph shows, that the BER is above the values of the MSK estimate for a SNR of $E_b/N_0 = 2dB$ and $E_b/N_0 = 4dB$ and even below the estimate for a $E_b/N_0 = 5dB$. For every case, the BER stays below the estimate for the BFSK signal. For SNRs higher than $E_b/N_0 = 5dB$ no bit errors were detected. Table 8.1 shows the exactly calculated values.

SNR [dB]	BER	recovered Packets
2	0.029006	101
4	0.009882	103
5	0.000107	104
6	0.0	104
7	0.0	104
8	0.0	104
12	0.0	104
13	0.0	104
14	0.0	104
15	0.0	104
18	0.0	104
19	0.0	104
20	0.0	104

Table 8.1. BER of the PropCube Simulator and USRP SDR.

Interpretation

The SDR system demodulates and decodes as many packets as the hardware radio system outputs. Therefore, the developed communication scheme definitely works out well. The calculated and plotted BER values show, that the set up communication system works well enough to be compared to a traditional communication system. The proficient performance is especially proven by the fact that the BER for a SNR of $E_b/N_0 = 5dB$ is below the estimated value. If the number of found bit errors for the SNRs of $E_b/N_0 = 2dB$ and $E_b/N_0 = 4dB$ is related to the number of recovered packets, it still proves sufficient performance level of the system, since the calculated BERs are not 100% representative. As mentioned in Section 4.4 the used BER calculating algorithm is not able to handle bit shifts; also it discards entire packet if a bit error occurs ahead of or within an HDLC flag. Thus, the calculated values are not characteristic enough to give an absolute statement about the quality of the communication system. Also, the small number of samples is insufficient for a valid approximation. Taking these factors into consideration, the calculated BER is insufficient; however, it gives a good first order approximation.

8.2.2 Real-Time Implementation

The hardware-in-the-loop test of the real-time implementation showed that the GRC GUI started lagging. During an overnight test the GRC GUI froze. One option to enhance the efficiency of the system, is to adjust the threshold of the implemented power squelch. The power squelch must limit the signal to only bursts of PropCube downlink. Tests on the system with an adjusted threshold showed that the squelch prevents collapsing of the signal. However, no tests with the squelch have been performed on an actual downlink system. In addition, the computational effort of the decoding Python script must be minimized. Also, the data transmission via FIFO file reduces processing speed.

CHAPTER 9:
Conclusion and Recommendations

This thesis deals with an investigation on whether inexpensive off-the-shelf SDR technology can replace legacy radio hardware for satellite groundstations. The thesis covers the development of the BigRedBee receiver, as well as the PropCube receiver. Both of the rceivers have been tested. This chapter concludes the thesis. The chapter points out the major achievements and results, and also gives an outlook on future work.

9.1 Conclusion

The research question of this thesis was whether an inexpensive SDR can replace legacy communication technology hardware. This option is proven to be possible by successfully creating a demodulation and decoding scheme for the BigRedBee and its AFSK downlink signal, utilizing an SDR. Tests showed that the SDR performs at a comparable level as the currently used hardware system. Additionally, this thesis presents the demodulation and decoding scheme of the PropCube CubeSat with it's GFSK downlink signal. Tests have successfully been performed under laboratory conditions with a ground-based satellite simulator. The resulting bit error rate gives a good first order approximation of the actual performance of the system. It shows a $BER < 1 \times 10^{-4}$ for a SNR of $E_b/N_0 = 6dB$, whereas the estimated BER for a coherent MSK signal is $BER_{MSK} \approx 2 \times 10^{-4}$ for the same SNR.

There are still some computational performance problems with the the real-time implementation of the PropCube system. In thenext section, the author presents options to reduce the computational effort of the system, in order to stabilze it.

This development leads the SSAG to a point where it can actually start replacing hardware components with modern SDR technology. The biggest advantage of using the SDR in cooperation with open source software tools is that the defining software

can be programmed in-house and updates and adjustments can be performed without any additional costs.

9.2 Outlook

A major concern will be to reduce the computational effort of the PropCube receiver. Primarily the efficiency of the decoding Python script must be improved.Also, importing the decoding functionality into a custom GRC block will significantly enhance the computational performance of the system, besides it makes the system less cluttered. To prevent "stuttering" of the signal, the author recommends the implementation of a buffer.

To further improve the GFSK demodulation performance, further testing on the implementation of the *Quadrature Demod* block in cooperation with the *Polyphase Clock Sync* block should be made. In order to determine the quality of the system, a packet loss rate is easier to implement than the BER calculation, but the data collection is more time consuming. For quality evaluation, such as BER or packet loss rate, more data is required than used in the "first order calculation" performed in this thesis. To expand the functions of the ground station and to take advantage of the capabilities of the SDR, more modulation and demodulation schemes should be implemented, particularly the GFSK modulation.

APPENDIX: Appendix

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Appendix A

A.1 Used GNU Radio Blocks

This section briefly explains the functionality and possible settings of the most important GNU Radio blocks used in the presented demodulations. The explanations are derived from the blocks' documentation within GRC, the GRC documentation from the University of Victoria [46] and the GNU Radio Manual [47], as well as performed tests. Explanations of more complex blocks are accompanied by a screen capture of the GRC block, as well as the preferences window.

A.1.1 Sources

A source block is always the first block of a GRC flowgraph. It either defines the input interface between GRC and a file or GRC and another device, or defines a waveform, which is an input to the flowgraph. There are many different source blocks. The blocks that work in strict association with sink blocks are briefly presented in Section A.1.7. The development of this ground station mainly uses the *file source* and the *UHD: USRP source*.

File Source

The *file source* block opens and reads a file, which is then fed into the next processing block. The user must define the input data type of the file, as well as the source path. If no hardware is involved, a *throttle* block has to be implemented in any flowgraph to keep the CPU from overdriving.

UHD: USRP Source

The *USRP source* block defines the interface between GRC and the USRP device. Figure A.1 shows a screen capture of the block, along with the preferences window.

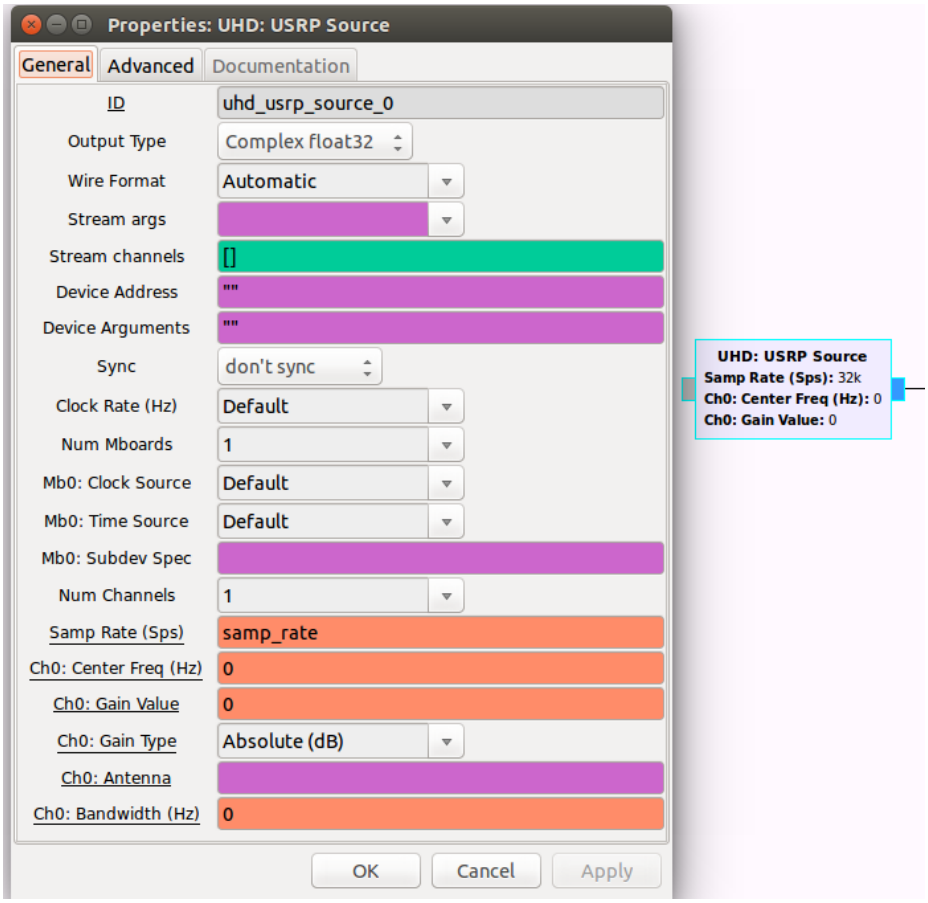

Figure A.1. UHD: USRP Source Block.

Table A.1 shows the property fields of the block, as well as a brief explanation of their functionalities.

For any of the USRP settings, the capabilities of the device have to be kept in mind. Also, for example, processing capabilities of the computer have to be viewed in order to set an appropriate sample rate. For more information on other available fields, see the GNU Radio Manual at "usrp_source Class Reference" [47].

field	function
Device Address	specifies port or IP address of the device
Samp Rate (Sps)	output sample rate for device
Center Freq (Hz)	tune frequency of device
Gain Value	device gain
Antenna	choose used USRP Antenna port

Table A.1. USRP Source block settings.

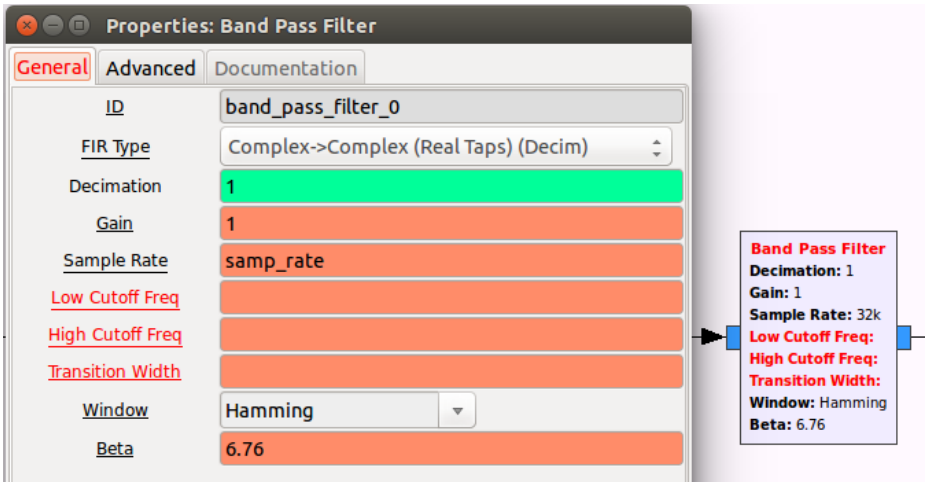

Figure A.2. Band Pass Filter Block.

A.1.2 Filter

Low- and Band Pass Filter

All used filter blocks are designable as multirate filters. They provide the option to set a decimation factor or interpolation factor. "Decimation" means downsampling at:

$$\frac{r_{s\_in}}{decimation\_factor} = r_{s\_out}$$

and "interpolation" means upsampling:

$$\frac{r_{s\_in}}{decimation\_factor} = r_{s\_out}.$$

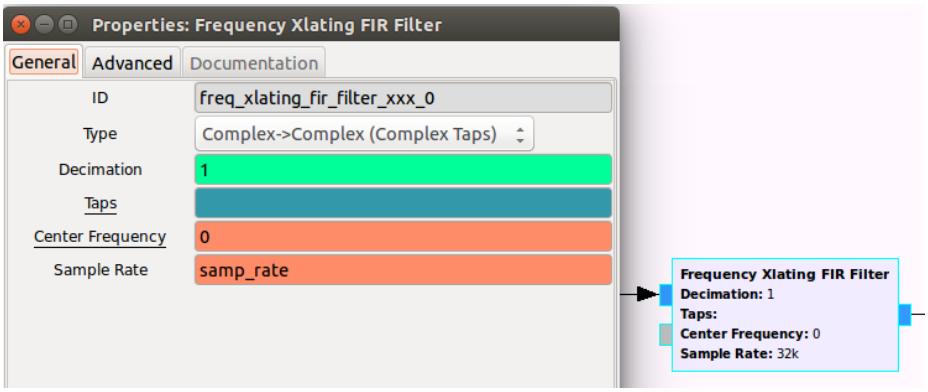

Figure A.3. Frequency Xlating FIR Filter Block.

For more information on *multirate filtering*, see [3, p. 168]. For the process of decimation and an interpolation, see [15] or [3, p. 181]. "Gain," "cutoff frequency" and "Transition Width" are self-explanatory, whereas "Sample Rate" is the block's input sample rate. The window type can be chosen from Hamming, Hann, Blackman, Rectangular or Kaiser. Differences are explained in [52]. The parameters of the *band pass filter* are similar to those of a low pass filter.

Frequency Xlating FIR Filter

For *multirate filtering*, the *Frequency Xlating FIR Filter* block provides the options for decimation or interpolation. "Decimation" means downsampling at:

$$\frac{r_{s\_in}}{decimation\_factor} = r_{s\_out}$$

and "interpolation" means upsampling:

$$\frac{r_{s\_in}}{decimation\_factor} = r_{s\_out}.$$

For more information on *multirate filtering*, see [3, p. 168]. For the process of decimation and an interpolation, see [15] or [3, p. 181]. For the process of decimation and interpolation, see [15] or [3, p. 181]. For the filter taps it is appropriate to use provided functions. The settings of the filter taps specifies the type of filtering the

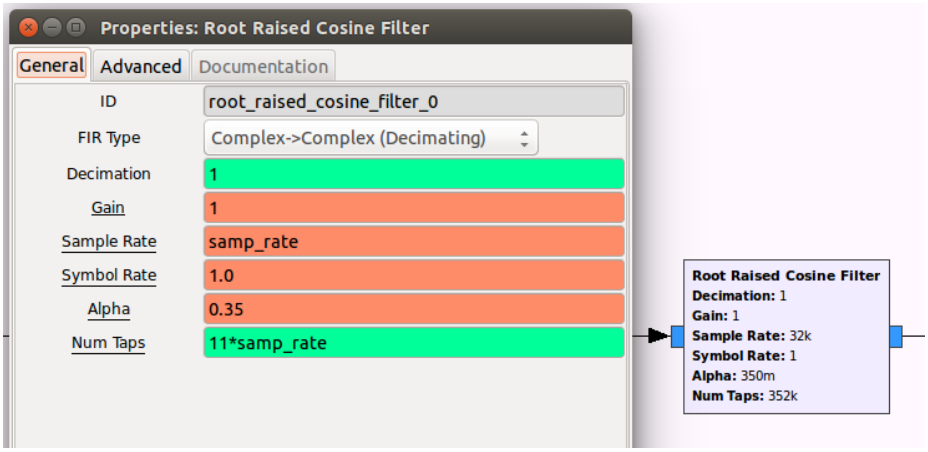

Figure A.4. Root Raised Cosine Filter Block.

FIR filter performs. More information on possible functions is given in [48]. In the developed ground station, all *FIR filters* are implemented as the *low pass filter* type. The filter type itself is specified, as follows.

$$firdes\_taps = firdes.low\_pass(gain, samp\_rate, cutoff\_freq, transition\_width, window = "window\_type", beta$$

6.76)

This expression specifies that the FIR Filter filters with *low pass* behavior with the following properties in order: filter gain, sampling frequency, center of transition band viewing the Nyquist theorem, transition width, window type and the *beta* parameter. Depending on the selected window type, the window_type value would vary. For example a *hamming window* would be "WIN_HAMMING." Whether the beta parameter is implemented or not depends on the window type. The beta parameter is only important for Kaiser window type [48]. For specific and individual evaluation of FIR filter specification, see [53].

Matched Filter - Root Raised Cosine Filter

According to the GNU Radio manual at "firdes Class Reference," the symbol rate of a *root raised cosine* filter has to be a factor of the input sample rate [47]. "Alpha" is

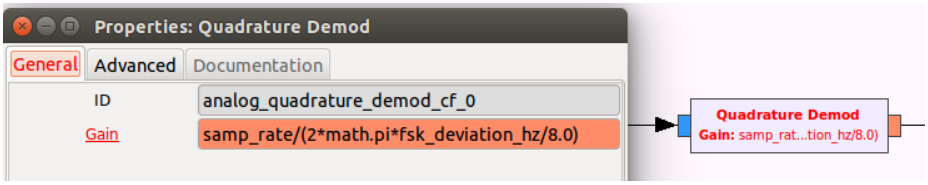

Figure A.5. Quadrature Demodulation Block.

the roll-off factor. Typical values for alpha are between 0.2 and 0.35 [54]. The *Root Raised Cosine Filter* is a common choice for a *matched filter* [55]. A *matched filter* is designed to remove excess noise, while not introducing *inter symbol interference* (IFI) [11, p. 454]

A.1.3 Demodulations

Quadrature Demod

The mathematical function of the *Quadrature Demod* block is explained in "quadrature_demod_cf Class" in [47]. The default of the gain value is set to:

$$gain = \frac{r_s \times 8}{2 \times \pi \times FSK\_deviation}. \tag{A.1}$$

The gain is derived from:

$$gain = \frac{samples\_per\_symbol}{\pi \times modulation\_index}, \tag{A.2}$$

which is the reciprocal of the sensitivity of the FM modulation [56].

GFSK Demod

The *GFSK Demod* block includes the functions of a *Quadrature Demod* block and a *Clock Recovery MM* block. The field "Samples/Symbol" is self explanatory. It

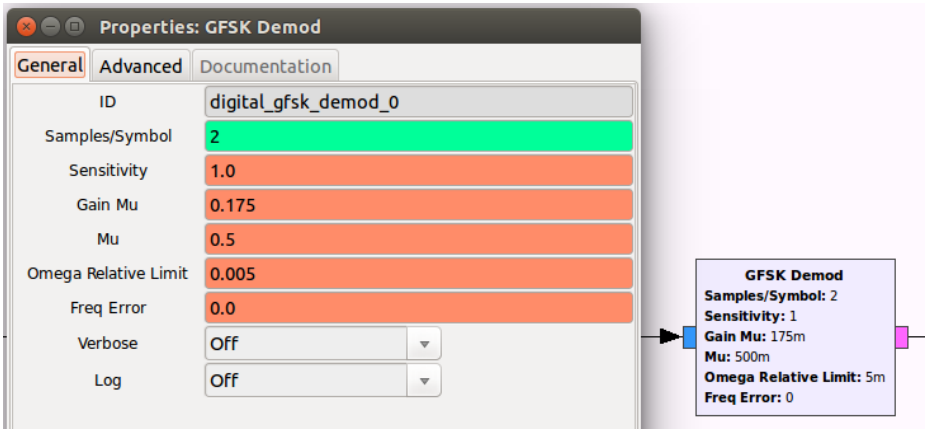

Figure A.6. GFSK Demod Block.

demands an integer value $samp\_per\_symb = \frac{r_s}{r}$, which is the input sample rate of the block divided by the data rate. Section A.2 discusses the optimum value, which is five samples per symbol. The boolean parameter "Log" defines whether or not the block prints modulation data to files. The boolean parameter "Verbose" defines whether or not the block prints information about the modulator [57]. The "Sensitivity" parameter is the "Gain" of the *Quadrature Demod* block. The input for the "Sensitivity" field is the same as stated in Equation A.1.3. "Mu" and "Omega" specifications are specific to the clock recovery. For this ground station, the values of the "Mu" and "Omega" fields were left at their defaults. The parameter "freq_error" demands the bit rate error as a fraction. This field is also set at its default. For more information on the clock recovery block and the parameters in particular, see [58]. The *GFSK Demod* block can be replaced by a *Quadrature Demod* block in cooperation with a *Clock Recovery MM* block. However, it is more convenient to use the *GFSK Demod* block, since it includes further functionalities and the interface is less cluttered. The output of the *GFSK Demod* block is a bit stream, packed one significant bit per byte.

A.1.4 Detectmarkspace

The *Detectmarkspace* block is a hierarchical block that incorporates a multistep frequency detection function. The hierarchical block is constructed by Schroer (dl1ksv)

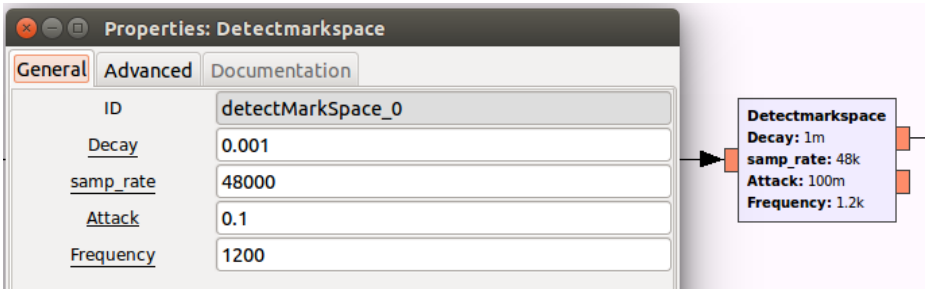

Figure A.7. Detectmarkspace Block.

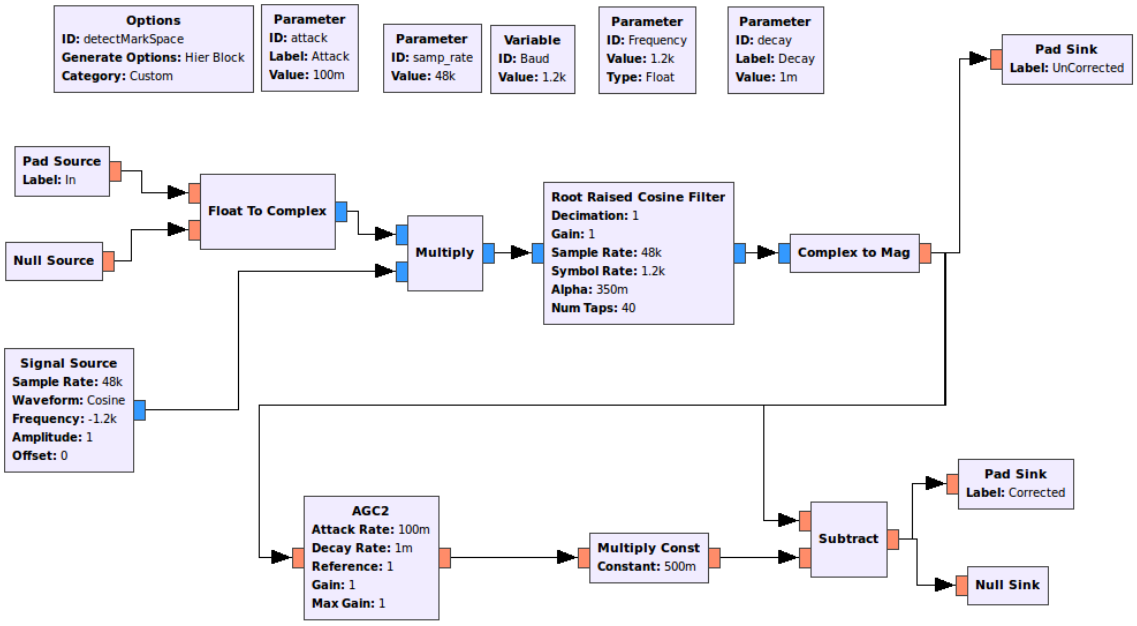

Figure A.8. Flowgraph of the *Mark Frequency* detecting *Detectmarkspace* block.

[24]. The input signal is negatively shifted by a value in Hz, which is specified in the variable "Frequency." A *matched filter*, here a *Root Raised Cosine Filter*, filters the DC signal from excess noise and a *Complex to Mag* converts the complex signal into an absolute floating point value. *Detectmarkspace* has two possible outputs. One is the direct output of the *Root Raised Cosine Filter* block and is called "UnCorrected." The other, "Corrected" one is smoothed by subtracting a gain-controlled version of the filter output from the direct filter output. Figure A.8 shows the flowgraph of the hierarchical *mark frequency* detection block.

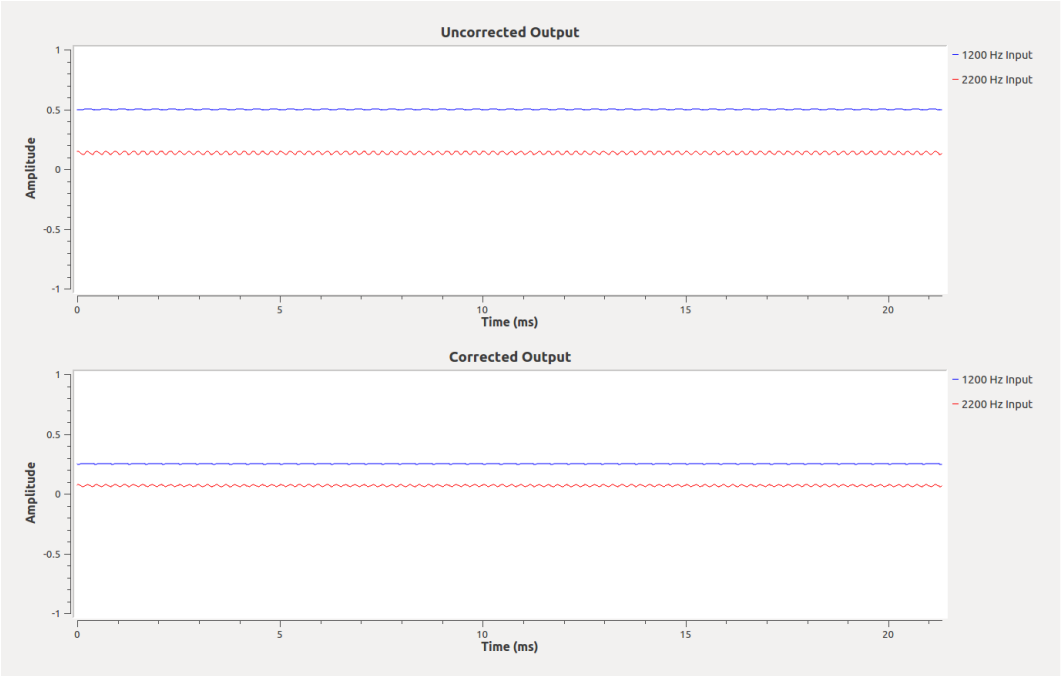

Figure A.9. Corrected and UnCorrected Markdetector Output, with Sinusoidal Input.

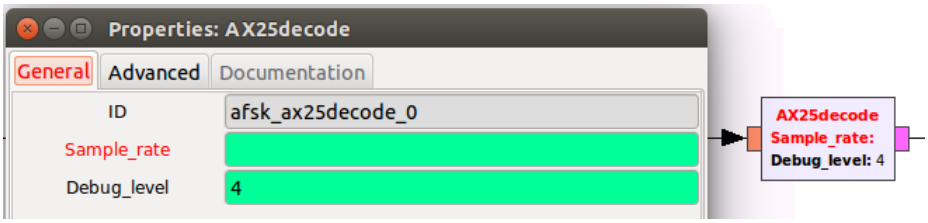

Figure A.10. AX25decode Block.

Figure A.9 shows a plot of the output of two *Detectmarkspace* blocks, that detect *mark frequency*. The input is a pure sinusoidal at either $f = 1200Hz$ or $f = 2200Hz$. Figure A.9 shows the uncorrected output in the upper graph, and the corrected output in the lower graph.

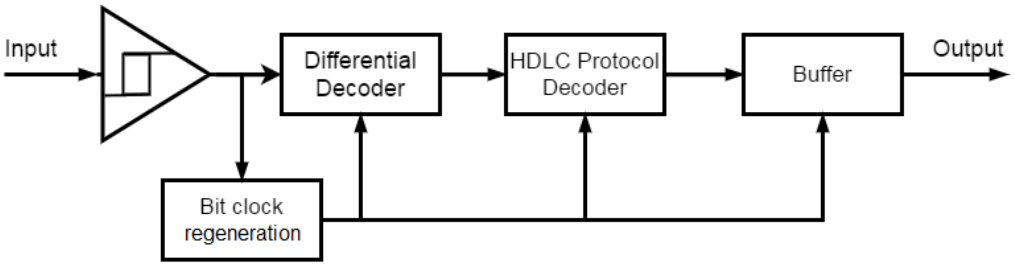

Figure A.11. Scheme of the AX25decode Block by Schroer (dl1ksv).

A.1.5 Decoder

AX.25 decoder

The AX.25 decoding for the AFSK demodulation is done by a block called *AX25decode*, which is programmed by Schroer (dl1ksv). It contains functionalities, based on Sailer (HB9JNX) [40]. The *AX25decode* block takes an NRZ baseband signal. Figure A.11 shows the working scheme of the *AX25decode* block.

The input of the block must be an NRZ signal. Inside the block a bit slicing happens, followed by a bit clock recovery. The bit clock recovery recovers the baseband at the data rate from the input sample rate. The differential decoder converts the NRZ- to an NRZI signal, as described in a following section. From this NRZI signal results a bit stream, which contains APRS packets. The actual information is then pulled out of the packets with a AX.25 decoding functionality.

The output of the block is defined by the value set as "Debug_level." For values 1 thru 4 the block puts out packets that pass the CRC check. Values greater-than-or-equal-to 5 result in an output that plots packets that fail the CRC check. An example for the output can be seen in Figure A.12. Important for the user is 1) the time at which the packet was decoded, 2) the content of the address part of the AX.25 packet and 3) the decoded content of the AX.25 packet, which in this case is the GPS position of the BigRedBee.

One disadvantage of this block is, it can just be used for data rates of $r = 1200baud$, as the clock recovery inside the block tries to recover a signal with $r = 1200baud$.

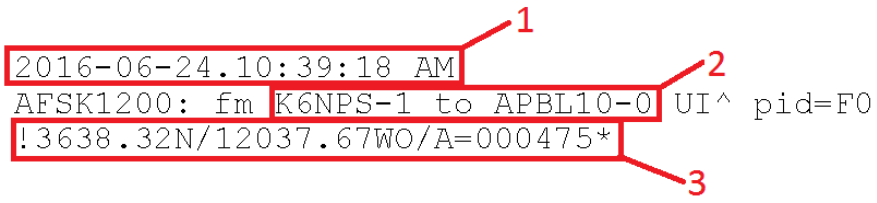

Figure A.12. AX25decode Block Output: 1) Timestamp 2) Address 3) Content.

To handle other data rates, such as $r = 9600baud$, the C++ code of the block would have to be adjusted.

Descrambler Sync

The *Descrambler Sync* block is developed by the team operating the CubeSat Bisonsat and can be found in the provided gr-bs folder. The block works as an derandomizer or descrambler and has to be set dependent on the scrambler on the transmitting side. The analog equivalent of this block is a shift register, set up as described in [40]. The field tap mask specifies the positions of the XOR gates. The field asks for the decimal representation of a binary number, in which the positions of "1"s mark the positions of the XOR gates. For example 72_{10} is 01001000_2 and marks XOR gates at the fifth and seventh position of the register. The value in the "Preload" field specifies the preload of the register.

Differential Decoder and Invert Bit Sync: NRZ To NRZI

NRZ to NRZI conversion is done by a set of one *Differential Decoder* block and a value inverting block or block sequence. The *Differential Decoder* block processes two binary samples, adds the values and outputs the added values modulo two. This mathematical process results in a "1" if there was a transition and a "0" if there was no transition. Since NRZI encodes a "0" for transition and a "1" for no transition, a value inverter converts the output of the differential decoder. A block called *Invert Bit Sync* is implemented in the presented GFSK demodulation. This block is created by the team operating the CubeSat Bison Sat. Another option to invert a binary value was to use an *add* block and a XOR gate with one constant input.

A.1.6 Miscellaneous

Variable and Parameter
The *Variable* block, *Parameter* block and the *GUI Range* block each contain a variable, which can be used in the preferences of processing blocks. The *Parameter* block will come up as a preference field for a hierarchical block, when used in a hierarchical flowgraph. A range of values can be defined in the preferences of the *GUI Range* block, allowing the user to drag a slider in the GUI to adjust the value on-the-fly.

The *import* block loads additional Python modules, such as "math," which allows the usage of mathematical expressions like pi.

A.1.7 Sinks

There are four types of sinks available in GRC: *graphical scope sinks*, *file sinks*, *port sinks* and the *null sink*. The *null sink* stands alone, as it discards every output.

Scope sinks allow the user to graphically scope the signal in different ways on the *graphical user interface* (GUI). A *frequency sink* displays the Fourier transform of the signal, meaning the frequency spectrum. A *time sink* displays the signal in the time domain. A *waterfall sink* shows the frequency domain of the signal over time. A *constellation sink* displays the phase of the signal. A *number sink* outputs a float number value.

File sinks are the *WAV file sink* and the simple *file sink*, that write the output data into files. *Variable sinks* and *vector sinks* are not quite storing data to files, but store it as variables or vectors, allowing to use the values as preferences in other blocks.

Port sinks enable the user to link different flowgraphs, devices or programs. The *TCP sink* and the *UDP sink* send the output data to user specified ports. A *virtual sink* works in association with a *virtual source*. It passes on a signal within one flowgraph and works the same way as a connecting arrow in GRC. It is useful to tidy up a

flowgraph. A *pad sink* works in association with a *pad source* and is the endpoint of a hierarchical flowgraph.

The *XMLRPC Server* block utilizes a network port, too. The block allows the user to remotely control variables of the flowgraph. By leaving the "Adress" field blank the GRC flowgraph can be reached via the IP address of the computing device. For the "Port any adress, which does not specify any port on the device so far can be chosen. In this case, the port is specified to $Port = 2651$. The following Python code example shows how to access variables of the flowgraph:

```
import xmlrpclib
import time

#replace <...> with device IP
s = xmlrpclib.Server('<device IP>:8082')
#replace <variable name>; <value>
#s.set_<variable name>(<value>)
#example for variable: "downlink_freq"; value:
    "914025000"
s.set_downlink_freq(914025000)

#crash
```

Accessed can be any variable which defines values of preferences that are underlined in the properties window of the GRC blocks

A.2 Block Function Test

A.2.1 Mark and Space Distinguisher Test

The *Detectmarkspace* block was tested with the flowgraph, shown in Figure A.13, by inputting either mark or space frequency. The main results are displayed in Figure

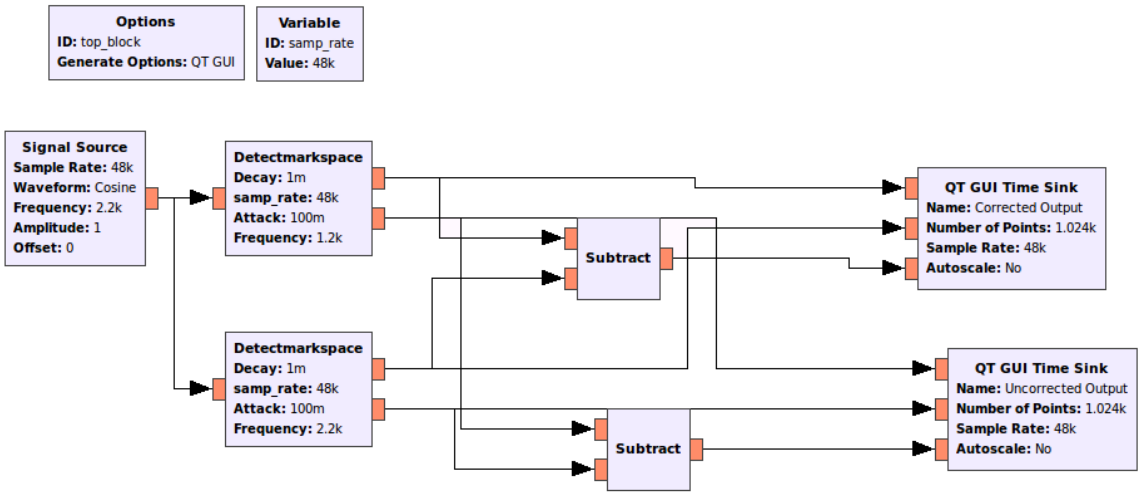

Figure A.13. Flowgraph of Mark and Space Distinguisher Test.

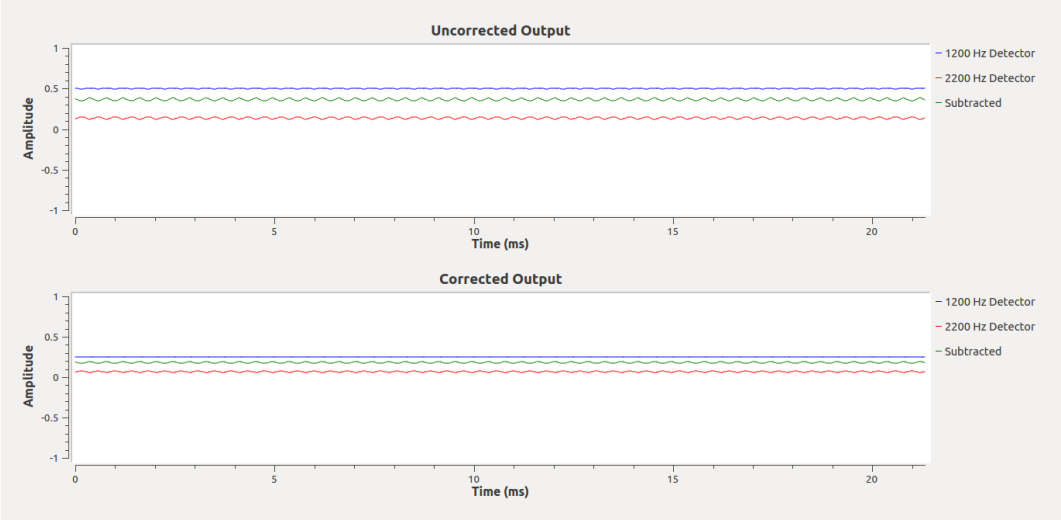

Figure A.14. Mark and Space Distinguisher Test - Mark Input.

A.14 and Figure A.15. Each figure respectively shows the output amplitude of the mark detector and the space detector. Also each figure shows the amplitude of the the space detector output amplitude subtracted from the mark detector output amplitude. Thus, if the mark frequency is input to the system, a positive value is output, and vice versa: if the space frequency is input to the system, a negative value is output.

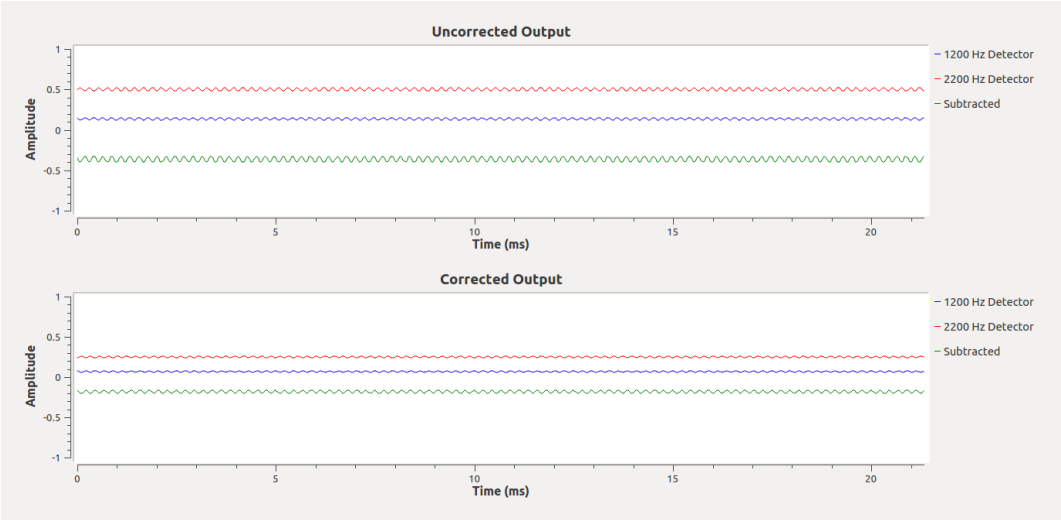

Figure A.15. Mark and Space Distinguisher Test - Space Input.

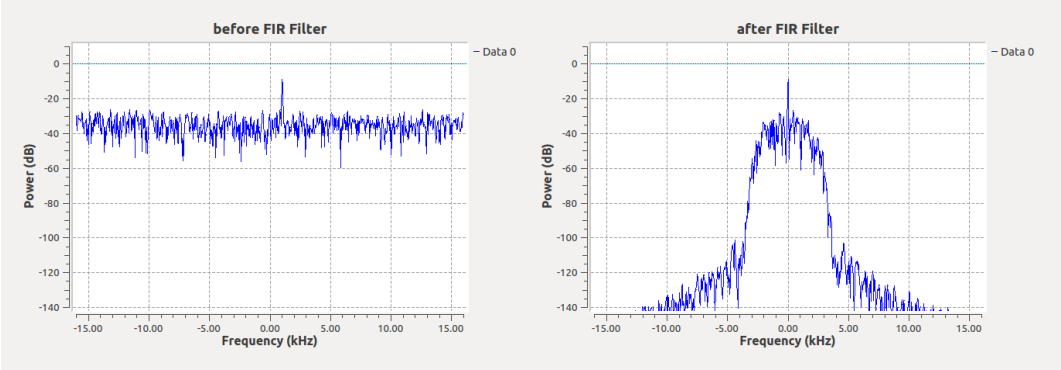

Figure A.16. FIR Filter Tap Test.

A.2.2 FIR Filter Taps Test

Figure A.16 shows a signal before and after FIR filtering with the following tap specifications. The input is *gaussian noise*, added to a sinusoidal signal at $f = 1000Hz$. The filter taps are specified as follows:

$firdes\_taps =$
$firdes.low\_pass(1, samp\_rate, 2000, 2000, firdes.WIN\_BLACKMAN, 6.76)$. The plot shows the Fourier transform of the output signal, filtered as specified in the taps.

A.2.3 AX.25decode: Clock Recovery Test

The clock recovery inside the *AX25decode* block was tested with various inputs. Figure A.17 - A.19 show the output of the *AX25decode* block if the input sample rate is varied from nine samples per symbol to 11 samples per symbol. Rates of smaller-than-or-equal-to eight, as well as greater-than-or-equal-to 15 samples per symbol resulted in no decoded packets. In each case, the same 10 packets were transmitted. The figures show that the most packets can be recovered at a sample rate of nine samples per second. Figure A.17 shows that all 10 transmitted packets are correctly recovered at the rate of nine samples per symbol.

A.2.4 GFSK Demod Block Test

Tests on the *GFSK Demod* block result in an optimum rate of five samples per symbol. For a rate of 25 samples per symbol, the output shows that the symbol rate recovering clock loses track after 5-25 bytes, as Figure A.20 shows. After a few bytes, it locks on again. The sequence of recovered symbols must be bytes of hexadecimal 0xAA in a row. Instead, it shows a sequence of 3-6 badly recovered bytes. The hexadecimal representation 0x55 is hexadecimal represented byte 0xAA shifted by one bit and thus accepted as correct, too. Resulting bit errors are marked yellow. The estimated symbol sequence is displayed in the lower part of Figure A.20. Another test with a sequence of 16 times 0x33 in a row results in similar outputs.

A.2.5 Integrated Low Pass Filter Test

In order to filter the frequency band of interest, as well as to cut out unwanted noise, a *Low Pass* filter is located ahead of the GFSK demodulation. An empirical study on different cutoff frequencies results in $f_{cutoff} = 6kHz$ as the best working cutoff frequency. A frequency range of $f = 5kHz - 10kHz$ is tested in this study. To compare results, the BER of the output is calculated. It is necessary to choose a low SNR in order to force bit errors in the output. Table A.2 shows some of the test results. The demodulation is processed at a SNR of $E_b/N_0 = 2dB$. The BER is determined with the Python script "BER_find.py."

f_{cutoff} **[Hz]**	**BER**	**recovered Packets**
$5k$	1	0
$6k$	110×10^{-6}	101
$7k$	2934×10^{-6}	101
$8k$	353×10^{-6}	2
$9k$	196×10^{-6}	3
$10k$	122×10^{-6}	5

Table A.2. Integrated Low Pass Filter - Cutoff Frequency Test.

```
2016-06-24.10:39:18 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.32N/12037.67WO/A=000475*
2016-06-24.10:39:22 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.32N/12037.68WO/A=000518*
2016-06-24.10:39:27 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.30N/12037.69WO/A=000603*
2016-06-24.10:39:30 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.28N/12037.70WO/A=000705*
2016-06-24.10:39:35 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.27N/12037.71WO/A=000846*
2016-06-24.10:39:38 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.27N/12037.72WO/A=000974*
2016-06-24.10:39:43 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.27N/12037.73WO/A=001089*
2016-06-24.10:39:47 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.26N/12037.75WO/A=001204*
2016-06-24.10:39:50 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.24N/12037.77WO/A=001305*
2016-06-24.10:39:55 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.24N/12037.77WO/A=001404*
```

Figure A.17. AX25decode with 9 Samples per Symbol.

```
2016-06-24.10:41:01 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.32N/12037.67WO/A=000475*
2016-06-24.10:41:06 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.32N/12037.68WO/A=000518*
2016-06-24.10:41:10 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.30N/12037.69WO/A=000603*
2016-06-24.10:41:14 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.28N/12037.70WO/A=000705*
2016-06-24.10:41:18 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.27N/12037.71WO/A=000846*
2016-06-24.10:41:26 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.27N/12037.73WO/A=001089*
2016-06-24.10:41:30 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.26N/12037.75WO/A=001204*
2016-06-24.10:41:33 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.24N/12037.77WO/A=001305*
```

Figure A.18. AX25decode with 10 Samples per Symbol.

```
2016-06-24.10:42:10 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.32N/12037.68WO/A=000518*
2016-06-24.10:42:15 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.30N/12037.69WO/A=000603*
2016-06-24.10:42:38 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.24N/12037.77WO/A=001305*
2016-06-24.10:42:47 AM
AFSK1200: fm K6NPS-1 to APBL10-0 UI^ pid=F0
!3638.23N/12037.79WO/A=001483*
```

Figure A.19. AX25decode with 11 Samples per Symbol.

```
Bad recovered symbol sequence:

7E 41 56 55 55 55 55 55 55 55 55 56 65 53 55 55 55 55 55 55 55 55
55 55 55 55 55 56 E3 82 72 A6 AA AA AA AA AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA AA AA AC D2 A0 D5 7F D5 64 55 55 55 55 55 55
55 55 55 55 35 55 93 55 55 55 55 59 95 42 4F 95 AA AA AA AA AA AA
AA AA AA AA AA AA AA AA AA A6 6A B2 AA AA A6 36 24 65 65 53 55 55
55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55


Estimated symbol sequence:

AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
```

Figure A.20. From bad Clock Recovery resulting Bit Errors .

APPENDIX B:
Appendix B

A.1 Flowgraphs

A.1.1 Determination of SNR and BER

Figure A.1 shows the GRC flowgraph, which is used for SNR and BER determinations. The SNR calculation follows the equation:

$$E_b/N_0 = 10 \times \log \frac{P_{Signal}}{P_{Noise}}, \tag{A.1}$$

which is derived from [17, p. 130]. Different SNRs are achieved by adjusting the amplitude of the *gaussian noise* source. The *QT GUI Number Sink* displays the actual E_b/N_0 value. Transmissions can be determined by viewing the display of the *QT GUI Time Sink*. *Hier:PropCubedemod* is a hierarchical block, that incorporates the GFSK demodulation, seen in Section 5.2

A.1.2 BigRedBee Receiver

Figure A.2 shows the flowgraph of the BigRedBee receiver. The output is written to a file. The filename includes a UTC timestamp of the recording. An option is to store the raw signal in a file, too. Additionally there is an option to load and process a file containing a raw signal. The flowgraph as an image, as well as the GRC file can be found on the CD that supplements this thesis.

A.1.3 BigRedBee Receiver

Figure A.3 shows the flowgraph of the BigRedBee receiver. The output is written to a FIFO file. The Python script "GRC_tot_TYVAK.py" reads from this FIFO file and decodes the data. Options are, to store the demodulated data in a seperate file, as well as to store the raw signal, too. The flowgraph as an image, as well as the GRC file can be found on the CD that supplements this thesis

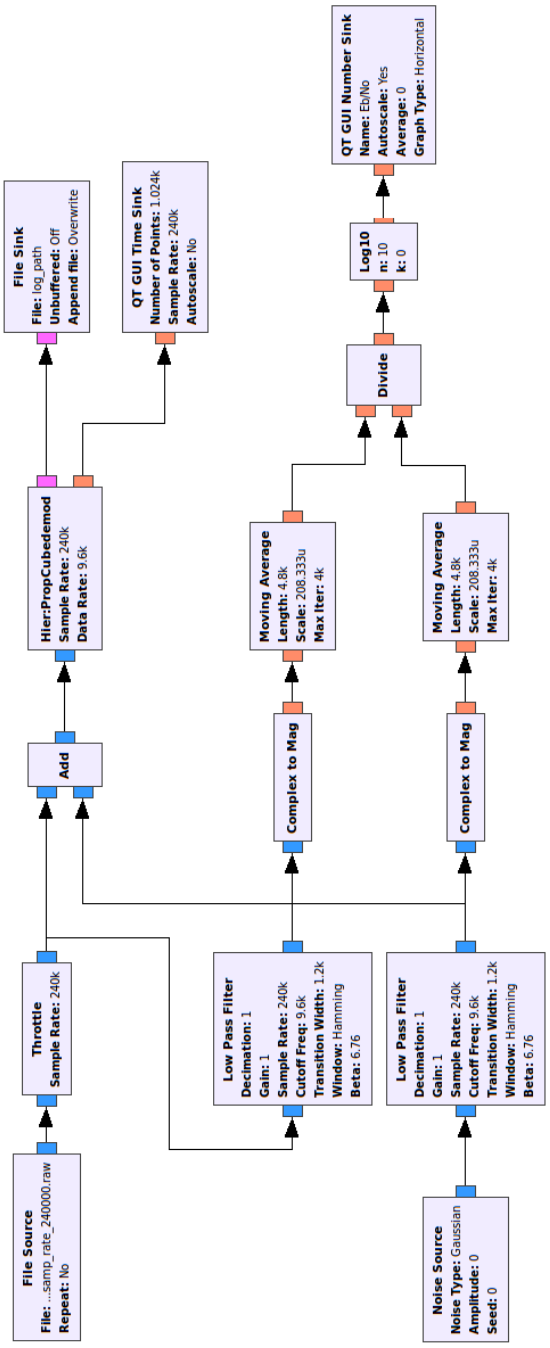

Figure A.1. GNU Radio Flowgraph for BER and SNR Determination.

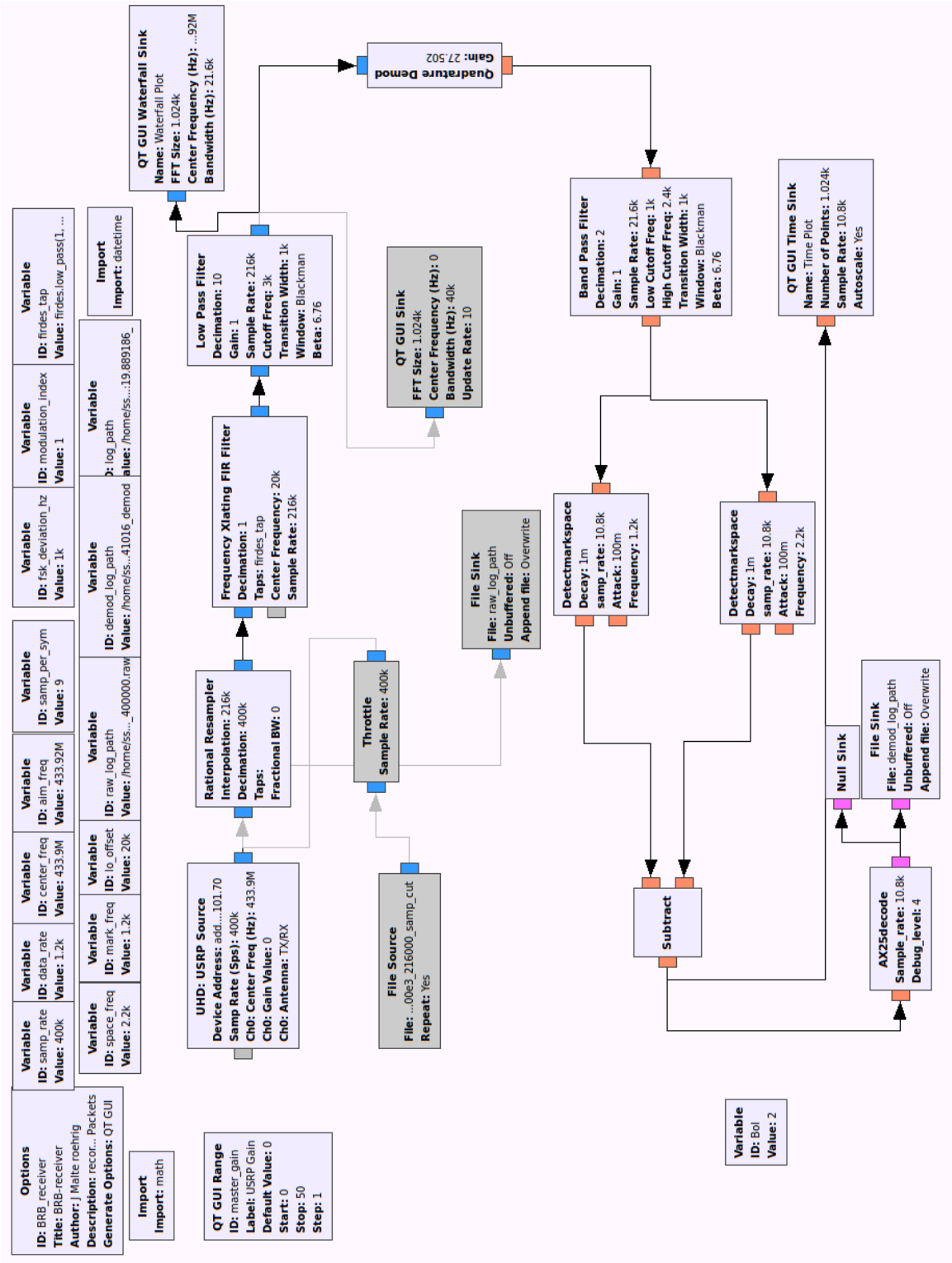

Figure A.2. GNU Radio Flowgraph for full BigRedBee Receiver.

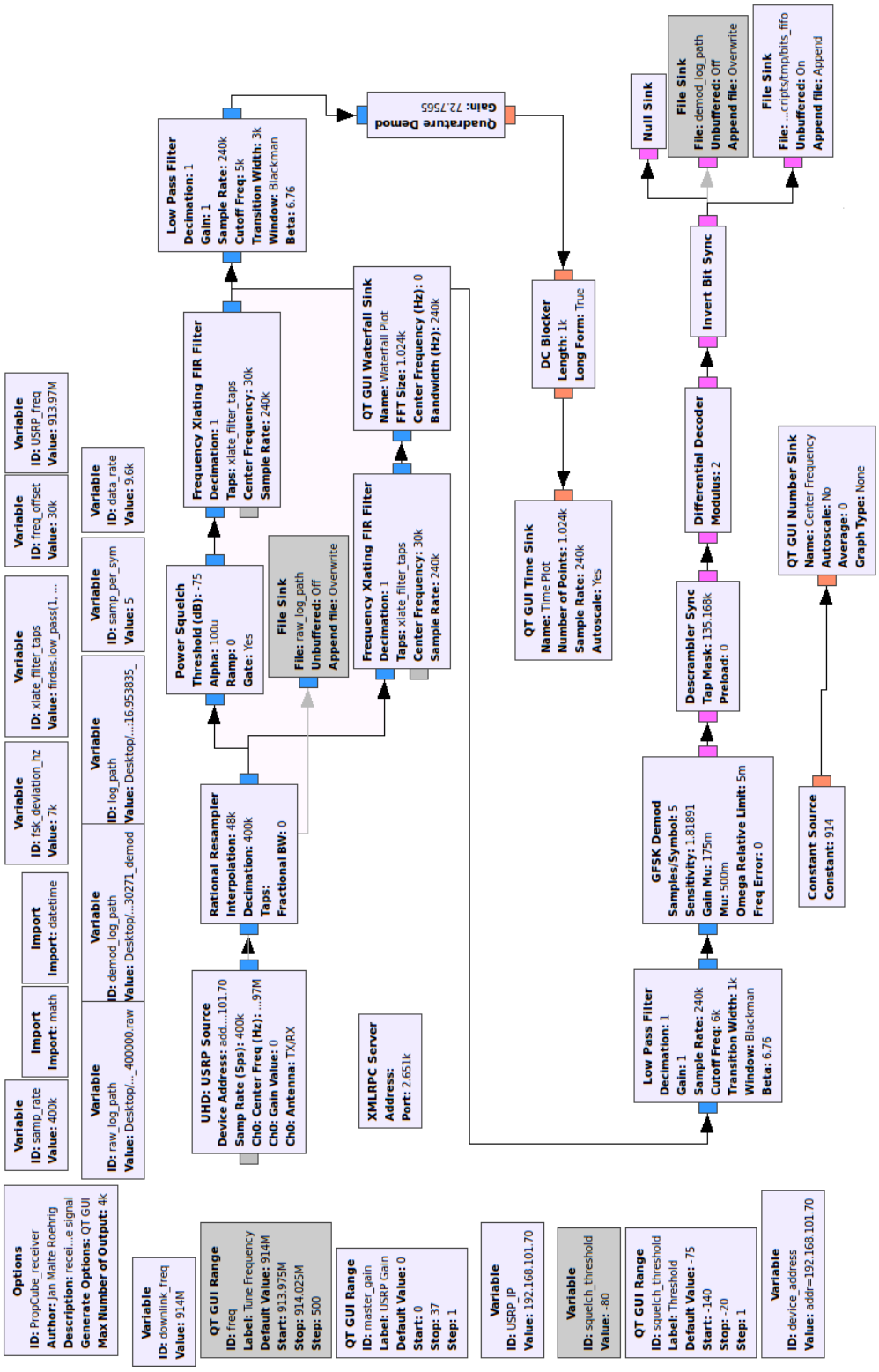

Figure A.3. GNU Radio Flowgraph for full PropCube Receiver.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
Appendix C

A.1 Python Scripts

A.1.1 "HDLC_unframe_and_unstuff.py"

This Python Script takes a stored demodulated signal from the GFSK demodulation Scheme presented in Section 5.2. The script detects and unframes HDLC packets. It does the bitunstuffing but NOT a CRC calculation.

```
"""HDLC_unframe_and_unstuff.py: takes file, stored via GRC,
   and detects packets, does bit unstuffing, prints HEX
   values of the packets and writes them to a file."""

__author__      = "Jan Malte Roehrig"


from __future__ import print_function
import sys



#takes bit string stream and converts it into bit list +
   unstuffs bits
def bits_to_list(data_str_stream):
  out_str = ''

  unstuffed_datalist = []
  reversedbyte_datalist = []
  #counts 1s in a row for bit unstuffing
  one_count = 0
```

```
i = 0
#runs for defined number of bytes
while i < (len(data_str_stream)):

  #appends 1
  if data_str_stream[i] == '\01':
    #add 1 to one_count
    one_count += 1
    unstuffed_datalist.append(1)

  #appends 0 if not 5* 1 before    bitunstuff
  else:
    if one_count != 5:
      unstuffed_datalist.append(0)

    #reset one_count
    one_count = 0

  i+=1

reversedbyte_datalist = flip_bytes(unstuffed_datalist)

byte_list=[]
byte = 0
byte_index = 0
for bit in reversedbyte_datalist:
  if byte_index < 8:
    byte = byte | bit

    byte_index += 1
  if byte_index == 8:
    byte_list.append(byte)
    byte = 0
```

```
      byte_index = 0
    else:
      byte <<= 1

  print('')          #new line

  return byte_list



#flips bytes to MSB first and prints as HEX
def flip_bytes(data_bit_list):
  out_str = ''

  flipped_bit_list = []
  i = 0
  #loop over all data, jumps from byte to byte
  while i < (len(data_bit_list)-8):
    index_in_data= i
    #new byte
    byte = 0
    #runs for 8 bits
    for j in range(8):
      index_in_byte = j
      #flip because stream has LSB first
      flipped_index_in_byte = (7-index_in_byte)

      if data_bit_list[index_in_data+flipped_index_in_byte]
         == 1:
        #fill with 1
        byte = byte | 1
        flipped_bit_list.append(1)
      else:
```

```
            #else fill with 0
            byte = byte | 0
            flipped_bit_list.append(0)

          if index_in_byte < 7:
            #shift 1 left
            byte <<= 1

        #jumps to next 8 bits
        i += 8

        #print byte in Hex
        print('%02X '%byte, end='')
        #write byte into file
        decoded_packets_file.write('0x%02X '%byte)


    return (flipped_bit_list)



#MAIN

#program to detect certain mask sequence in byte stream

#open and read file in binary form
print ('Name of file: ')
#takes raw input or drag file into terminal window!
name = raw_input("> ")[1:-2]
#name = 'demodtest'    #defines filename
#opens
f = open(name,'rb')
#reads file
```

```
data_str_stream = f.read()
decoded_packets_file = open(name + '_decoded_Packets', 'w+')

#set mask: here probcube preamble as specific sequence
number_masks = 100
#1/4 of the PropCube preamble 0xAA
mask_preamb = number_masks*b'\00\01'
#HDLC/AX.25 Flag 0x7E
mask_flag = b'\00\01\01\01\01\01\01\00'
mask3 = b'\01\01\00\00\01\01\00\00'


#counts total number of packets
i = 0
packet = 0
while True:
  #position of found number of preamble followed by flag
  j = data_str_stream[i:].find(mask_preamb+mask_flag)
  if j >= 0:
    #position of HDLC/AX.25 flag byte = position in loop (i+j
       )+(number_masks*length_mask)
    FrontFlag_position=i+j+(number_masks*2)+8
    #packet_length = position of end flag from start_content
    packet_length = 0
    while True:
      packet_length = data_str_stream[(FrontFlag_position):].
         find(mask_flag) + 1   #find end flag

      if packet_length >= 0:
        packet += 1
        print('')
        #print position of start flag
        print("Packet at Byte %d"%(FrontFlag_position/8))
```

```
            #print number of bytes between start and end flag
            print("Packet Length: %d"%(packet_length/8))
            #write position of start flag into file
            decoded_packets_file.write("Packet at Byte %d\t"%(
                FrontFlag_position/8))
#           decoded_packets_file.write("Packet Length: %d\n"%(
    packet_length/8))
            #pack bits between flags into bytes and print as HEX
            output = bits_to_list(data_str_stream[
                FrontFlag_position:FrontFlag_position+
                packet_length])
            print('')
            decoded_packets_file.write('\n \n')
            break

        else:
            break

      i += j+(number_masks*2)+ packet_length + 1

    else:
      break


print('total number of packets: %r'%packet)
print("finish")
sys.exit()
```

A.1.2 "BER_find.py"

This Python script compares a number of files to a file that contains an estimated bit sequence. The script detects bit errors and calculates a BER.

```
""" Reads a file that contains the estimated packets and a
    number of files with different SNR. detects bit errors and
     calculates BER"""
__author__      = "Jan Malte Roehrig"

from __future__ import print_function
from __future__ import division
import sys

def readfile(filename, packet_length):
#opens file "filename"
  f = open(filename,'rb')
#reads file
  data_str_stream = f.read()
#creates dump_file
  dump_file = open('dumps/'+ filename + '_dump', 'w+')

  number_masks = 50
  #1/4 of the probcube preamble 0xAA
  mask = number_masks*b'\00\01'
#HDLC/AX.25 Flag 0x7E
  mask2 = b'\00\01\01\01\01\01\01\00'

  i = 0
  #counts total number of packets
  packet = 0
  data_packed_bit_list = ''

  while True:
  #position of found number of preamble followed by flag
    j = data_str_stream[i:].find(mask+mask2)
    if j >= 0:
      position_packet=i+j+(number_masks*2)+8
```

```
            #repacks following "packet_length"-many bytes
            for k in data_str_stream[position_packet:
                position_packet+ packet_length]:
              if k == '\01':
                data_packed_bit_list += '1'
              else:
                data_packed_bit_list += '0'


            #write packet to dump file
            dump_file.write("\n Packet %d\n"%packet )
            dump_file.write(data_packed_bit_list[packet *
                packet_length:(packet+1)*packet_length])
            dump_file.write("\n")


            packet += 1

            i += j + number_masks * 2 + packet_length

          else:
            break
        print("total number of packets: %d"% (packet))


        return (data_packed_bit_list, packet)

    #compare file to estimated
    def compare(estimate_file, tocompare_file, filename,
        packet_length):
        biterrors_file = open('dumps/'+ filename + '_biterrors', 'w
            +')
```

```
number_packets = 0
#number of errors in this packet
packet_error = 0
#total number of bit errors
bit_errors = 0
#keeps output - is just printed if packet is not discarded!
error_output = ""
BER = 0.0
i = 0
j = 0

while i <= len(tocompare_file):
  index_compare_file = i
  index_estimate_file = j

  #if compared bits dont match
  if estimate_file[index_estimate_file] != tocompare_file[
     index_compare_file]:

    error_output += ("Bit Error in packet %s at bit number
       %s: %s (estimated), %s (actual) \n"% (str(
       index_estimate_file // (packet_length)), str(
       index_estimate_file % (packet_length)),
       estimate_file[index_compare_file], tocompare_file[
       index_compare_file]))
    bit_errors += 1
    packet_error += 1

  #if one packet is completed
  if (index_compare_file%packet_length) == 0:
    biterrors_file.write(error_output)
    error_output = ""
    packet_error = 0
```

```
      number_packets += 1

    #if 1-80%  of the packet (without header and flags ->154
        bit) do not match
    if packet_error >= (.2*(packet_length-154)):
      biterrors_file.write("Packet %d is missing! - discarded
         \n"%(index_estimate_file//packet_length))
      #consider next estimated packet (estimate_file)
      index_estimate_file = (((index_estimate_file//
         packet_length)+1) *packet_length)
      #begin of actual packet (tocompare_file)
      index_compare_file = (((index_compare_file//
         packet_length) *packet_length))
      biterrors_file.write("%d, %d, %d, %d"%(
         index_compare_file//packet_length,
         index_compare_file%packet_length,index_estimate_file
         //packet_length, index_estimate_file%packet_length))
      error_output = ""
      #reduces number of found errors ...
      bit_errors -= packet_error
      #adds total number of bits of the packet to the number
         of bit errors instead
      bit_errors += packet_length
      packet_error = 0
      number_packets -= 1

    else:
      j += 1
      i += 1
    if j >= len(estimate_file):
      break
BER = float(bit_errors / len(estimate_file))
```

```
    print("total number of recovered packets: %d"% (
        number_packets))
    print(BER)
    return (bit_errors, BER, number_packets)

BER_result_file = open('BER_result', 'w+')

#filenames for read in
files = [ '2dB','4dB','5dB','6dB','7dB','8dB']

log_name = 'noisegain'

datas = {}
packet_length = 135*8
datas[0] = readfile(log_name + '0.0', packet_length)

BER = {}
index = 1
BER_result_file.write("File \t\t\t detected Packets \t
    recovered Packets \t Bit Errors \t\t BER \n")
for item in files:
    print('\n for file %s: ' %item)
    #read file
    datas[index]= (readfile(log_name + item, packet_length))

    #compare ar calculate BER
    BER[index] = compare(datas[0][0], datas[index][0], log_name
        + item, packet_length)
    print(BER[index])
    #write BER properties to BER_result file
    BER_result_file.write("%s \t %d \t\t %d \t\t %f \t\t\t %f \
        n" %(log_name + item, datas[index][1], BER[index][2],
        BER[index][0], BER[index][1]))
```

A.1.3 "GRC_to_TYVAK.py"

This Python script decodes the output data of the GRC flowgraph "PropCube Receiver" in real-time and sends it to the command and control software. The script is explained in Section 7.2.

```
"""HDLC_unframe_and_unstuff.py: takes file, stored via GRC,
   and detects packets, does bit unstuffing, prints HEX
   values of the packets and writes them to a file."""

# Jan Malte Roehrig
# malte.roehrig@gmx.de

from __future__ import print_function
from collections import deque
import numpy as np
import sys
import select
import socket
from struct import *
import datetime
import time
import serial
import ssl
import os


__author__ = "Jan Malte Roehrig"

#Setup Tyvak
TYVAK_IP = '192.168.101.64'          # NPS satnet VM
TYVAK_RX_PORT = 10001                   # Agreed-upon port
hostname = socket.gethostname()         # Get local hostname
```

```
TYVAK_RX_ADDR = (TYVAK_IP, TYVAK_RX_PORT)  # Combine IP and
    Port #

bufferSize = 5000                  # Whatever you need
chunk_size = 500*8
cnt = 1
cnt_CRC_fails = 0
CRC_grade = 16
data_rate = 9600

# name of gnuradio binary raw data dump file
input_file_name = 'tmp/bits_fifo'
# path of logfile
#path = '/export_local/home/mc3ops/propcube_files/'


#takes bit string stream and converts it into bit list +
    unstuffs bits
def bits_to_list(data_str_stream):
 out_str = ''
#  init_data_str_stream = data_str_stream
#  data_str_stream = data_str_stream[8:-4*8]
 unstuffed_datalist = []
 reversedbyte_datalist = []
 # counts 1s in a row for bit unstuffing
 one_count = 0
 i = 0
 # runs for defined number of bytes
 while i < (len(data_str_stream)):
  # appends 1
  if data_str_stream[i] == '\01':
   # add 1 to one_count
   one_count += 1
```

```
    unstuffed_datalist.append(1)

   # appends 0 if not 5* 1 before    bitunstuff
   else:
    if one_count != 5:
     unstuffed_datalist.append(0)
    # reset one_count
    one_count = 0

   i+=1
  return unstuffed_datalist


#takes bit string stream and converts it into bit list +
    unstuffs bits
 def bit_unstuff(content_bits):
  out_str = ''
  unstuffed_datalist = []
  reversedbyte_datalist = []
  #counts 1s in a row for bit unstuffing
  one_count = 0
  i = 0
  #runs for defined number of bytes
  while i < (len(content_bits)):
   #appends 1
   if content_bits[i] == 1:
    #add 1 to one_count
    one_count += 1
    unstuffed_datalist.append(1)

   #appends 0 if not 5* 1 before    bitunstuff
   else:
    if one_count != 5:
```

```
      unstuffed_datalist.append(0)
    #reset one_count
    one_count = 0
   i+=1
  return unstuffed_datalist


#packs bits from bit_list into a list with bytes and KISS
    frames the packets
def pack_byte_list (reversedbyte_datalist):
   byte_list=[]
   #Front KISS Frame 0xC0
   byte_list.append(192)
   byte_list.append(16)
   byte = 0
   byte_index = 0
   for bit in reversedbyte_datalist:
     if byte_index < 8:
       byte = byte | bit
       byte_index += 1
     if byte_index == 8:
       #if 0xC0 in content:
       if byte == 192:
         #replace with 0xDB
         byte_list.append(219)
         #and 0xDC
         byte_list.append(220)
       #if 0xDB in content:
       elif byte == 219:
         #replace with 0xDB
         byte_list.append(219)
         #and 0xDD
         byte_list.append(221)
```

```
        else:
          #else: append byte
          byte_list.append(byte)
        byte = 0
        byte_index = 0
      else:
        byte <<= 1
   #End KISS Frame 0xC0
   byte_list.append(192)
   return byte_list



#flips bytes to MSB first and prints as HEX
def flip_bytes(data_bit_list):
 out_str = ''

 flipped_bit_list = []

 i = 0
 #loop over all data, jumps from byte to byte
 while i < (len(data_bit_list)):
  index_in_data= i
  #new byte
  byte = 0
  #runs for 8 bits
  for j in range(8):
   index_in_byte = j
   #flip because stream has LSB first
   flipped_index_in_byte = (7-index_in_byte)

   if data_bit_list[index_in_data+flipped_index_in_byte] ==
      1:
```

```
      #fill with 1
      byte = byte | 1
      flipped_bit_list.append(1)
     else:
      #else fill with 0
      byte = byte | 0
      flipped_bit_list.append(0)

     if index_in_byte < 7:
      #shift 1 left
      byte <<= 1

   #jumps to next 8 bits
   i += 8
   #print byte in Hex
   #print('%02X '%byte, end='')
   #write byte into file
  return (flipped_bit_list)


#CRC calculating

# Shift Register of fixed length 16 is
# prefilled with 0xFFFF
#
# for length of msg shift '0' in
# iterate through msg bit in (bit un-stuffed; LSB first)
#
# IF (the bit that comes out of the shift register XOR
# with the current msg bit == True): compare content of
# shift register with reversed Polynomial 0x1021
# ELSE: compare shift register with 0x0000
#
```

```
# output: calculated FCS

# calculate CCITT-CRC over msg, length of CRC
# (16 bit / 32 bit) is specified by global CRC_grade
def CCITT_CRC(msg):
 global cnt_CRC_fails
 # polynomial is reversed 0x1021
 poly = [1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0]
 # calculated Frame check sequence
 reversed_FCS = []
 FCS = []
 # estimated Frame check sequence if CRC over
 # full content incl CRC on receive side:
 # reversed(flipped(->LSB first)(0x1D0F))
 #good_FCS = '0000111101000111'
 length = len(msg)
 # CRC 16 bit
 global CRC_grade
 # pass or failed
 #check = False
 # initialize shift register # Array with 16*1
 SR = deque(CRC_grade*[1],CRC_grade)
 # loop through all bits; shift into register
 for i in range(length):
  bit_index = i
  # Bit shifted out of shift register
  OutBit = SR.pop()
  # shift SR one right; append a '0' left
  SR.appendleft(0)
  # IF OutBit XOR msg bit == 1 -> SR XOR poly
  if not (OutBit == msg[bit_index] ):
   for j in range(CRC_grade):
    if SR[j]==poly[j]:
```

```
            SR[j] = 0
        else:
            SR[j] = 1
    # ELSE: SR XOR 0

    # after shift "length of message" times:
    # FCS = invered and flipped SR
    for i in range(CRC_grade):
        FCS.append(-SR[CRC_grade-1-i]+1)
    print('')
    return FCS


def check_CRC(content):
    global cnt_CRC_fails
    msg = content[:-CRC_grade]
    CRC_seq = content[-CRC_grade:]
    CRC_calc = CCITT_CRC(msg)
    if not CRC_calc == CRC_seq:
        print("CRC failed (CCITT)")
        for item in CRC_calc:
            print(item, end='')
        print('')
        for item in CRC_seq:
            print(item, end='')
        msg = ''
        cnt_CRC_fails+=1
    return msg


def append_CRC(msg):
    content = msg
    CRC_calc = CCITT_CRC(msg)
```

```
 for item in CRC_calc:
  content.append(item)
 return content



# takes KISS framed data, prints and sends it to TYVAK via
   UDP socket
# Credits go to Giovanni Minelli (gminelli@nps.edu)
def send_to_Tyvak(byte_list):
 try:
  udp_sock_satnet = socket.socket(socket.AF_INET,
                    # UDP definition
   socket.SOCK_DGRAM)
  print ("Connecting to Tyvak on %s port %s" % TYVAK_RX_ADDR)
 except socket.error, msg:
  sys.stderr.write("[ERROR] %s\n" % msg[1])

 except socket.timeout:
  sys.stderr.write(" Tyvak Socket timeout\n")
  pass
 except socket.error, msg:
  print("Error: %s" %msg)
  print ("Closing Tyvak socket")
  udp_sock_satnet.close()

 # Initialize packet counter
 global cnt
 kiss_msg = ''

 try:
  amount_recd = 0
  # Take a complete KISS frame
```

```
 for b in byte_list:
  kiss_msg += chr(b)
 # Send packet to Tyvak
 udp_sock_satnet.sendto(kiss_msg, (TYVAK_IP, TYVAK_RX_PORT))

 #log_file = open(path + 'Log_GDP_Tyvak.txt', 'a')
 log_file = open('Log_TNC_RX_Tyvak.txt', 'a')
 now = time.time()
 milliseconds = '.%03d' % int((now - int(now)) * 1000)
 time_log = time.strftime("%m/%d/%y %H:%M:%S", time.gmtime()
    ) + milliseconds + "\t\t"
 log_file.write(time_log)
 # Print for debug
 print ("\n\nKISS Frame to Server: ", end="")
 for c in kiss_msg:
  print ("0x%02X " %ord(c), end="")
  log_file.write("0x%02X " %ord(c))
 log_file.write('\n\n')
 print ("Counter: %s " %cnt)
 print ("Timestamp: %s "%time_log)
 log_file.close()
 # Increment packet counter
 cnt += 1                        # Increment packet counter

except (RuntimeError, TypeError, NameError):
 print ('exception')
except socket.timeout:
 sys.stderr.write("Socket timeout\n")
except socket.error, msg:
 print ("Error: %s" %msg)
 print ("Closing socket")
 udp_sock_satnet.close()
 #kpc.close()
```

```
 except KeyboardInterrupt:
  udp_sock_satnet.close()
  #kpc.close()
  print ("Goodbye")
   sys.exit()
 udp_sock_satnet.close()


def find_syncword(DR):
 sync_word_flag = [0,1,1,1,1,1,1,0]
 len_DR=len(DR)
 found = True

 for i in range(8):
  if not DR[len_DR-8+i] == sync_word_flag[i]:
   found = False

 return found

def find_syncword_long(DR):
 sync_word_long = []
 sync_word_flag = [0,1,1,1,1,1,1,0]
 sync_word_preamble = [0,1]
 for i in range((number_masks*2)):
  sync_word_long += sync_word_preamble
 sync_word_long += sync_word_flag
 len_sync_word = len(sync_word_long)+len(sync_word_long)
 len_DR=len(DR)
 found = True
 for i in range(len_sync_word):
  if not DR[len_DR-len_sync_word+i] == sync_word_flag[i]:
   found = False
```

```
 return found




#MAIN


# program to detect certain mask sequence in byte stream

# open and read file in binary form
#print ('Name of file: ')
#name = raw_input("> ")[1:-2]
# defines filename
#name = 'demodtestbits'
# opens
#f = open(name,'rb')
# reads file
#data_str_stream = f.read()



#set mask: here probcube preamble as specific sequence
number_masks = 30
#1/4 of the PropCube preamble 0xAA
mask_preamb = number_masks*b'\00\01'
#HDLC/AX.25 Flag 0x7E
mask_flag = b'\00\01\01\01\01\01\01\00'
mask3 = b'\01\01\00\00\01\01\00\00'

sync_word_long = []
```

```
sync_word_flag = ['\00','\01','\01','\01','\01','\01','\01','
    \00']
sync_word_preamble = ['\00','\01']

sync_word_long += number_masks*sync_word_preamble
sync_word_long += sync_word_flag
len_sync_word = len(sync_word_long)

# Data Shift Register
Bit_Deck = (len_sync_word+1)*[1]
# current processed content
content_bitlist = []
# read data
data = ''

start = 0
# counts total number of packets
packet = 0
# bool: currently a packet processed?
packet_in_process = 0
# boolean indicating whether gnuradio
# is streaming data into the fifo file
fifo_is_open = 0
# wait for gnuradio to open its end of the fifo
# before streaming data in for processing
print("waiting for fifo to open")
# prevent python from buffering print statements
# to allow piping program output to a log file
# without a delay, in case of a system crash
sys.stdout.flush()
#if True == True:
with open(input_file_name, 'rb') as fifo:
  # loop forever, reading data from the fifo or
```

```
 # waiting for gnuradio to open its end of the fifo

while 1:

 if True: #try:
  data +=  fifo.read(chunk_size)
  #data += f.read()

  # Append raw data packet to existing buffered string
  #data_str_stream += data
  # if the fifo was closed before this chunk was read
     # gnuradio has started streaming data into the fifo
  if not fifo_is_open:
   print("fifo opened")
   sys.stdout.flush()
   fifo_is_open = 1
   # if gnuradio has stopped streaming data into the fifo
   # try to read from the fifo every half second until
   # gnuradio starts streaming data in again


  # if a packet is currently processed
  # if yes continue reading and processing
  # if not: seek preamble + flag
  if not packet_in_process and data != '' :
   # iterate of items in data and append to shiftregister
   search_index = 0
   while search_index < len(data):

    Bit_Deck.append(data[search_index])
    if len(Bit_Deck) >=10*data_rate*8:
     Bit_Deck = Bit_Deck[-(len_sync_word+1):]
    search_index +=1
```

```
# if find_syncword(DR):
for i in range(8):
 flag_found=1
 if Bit_Deck[-8+i] != sync_word_flag[i]:
  flag_found = 0
  break
if flag_found:
 # if find_syncword_long(DR):
 for i in range(len_sync_word):
  if Bit_Deck[-len_sync_word+i] != sync_word_long[i]:
   flag_found = 0
   break
if flag_found :
 flag_found = 0
 if True:
  # packet in process
  packet_in_process = 1
  # position in content
  content_index = 0
  # empty content
  content_bitlist = []
  # in data from found content position to end
  for content_item in data[search_index:]:
   # append current bit to data shift register (Bit_Deck
      )
   Bit_Deck.append(content_item)
   # append current bit to content
   content_bitlist.append(content_item)
   content_index += 1
   # if end flag found:
   if Bit_Deck[-8:] == sync_word_flag:
    # remove end flag from content
```

```
            content_bitlist = content_bitlist[:-8]
            # remove viewed data from data
            data = data[search_index+content_index:]
            # reset counters
            search_index = 0
            content_index = 0
            # finished processing packet
            packet_in_process = 0
            break
        # found packet seams to be longer than 1000 bytes:
        # end and reset counter + flush old data
        elif content_index > 1000*8:
            print('Halt: timeout for packet!')
            content_bitlist = []
            data = data[search_index+content_index:]
            search_index = 0
            content_index = 0
            packet_in_process = 0
            break
    # if content is available and currently no packet
       processed:
    if content_bitlist != [] and packet_in_process == 0:
    # bit unstuff content
        unstuffed_datalist = bits_to_list(content_bitlist) #
           bit_unstuff(content_bitlist)
        # check CRC
        CRC_valid_msg = check_CRC(unstuffed_datalist)
        # if CRC passed :
        if not CRC_valid_msg == '':
            #flip bytes of message: [:-16] without 16 bit CRC
               sequence
            reversedbyte_datalist = flip_bytes(CRC_valid_msg)
```

```
        # pack bits into aligned bytes and append KISS frame +
            escape
        packed_byte_list = pack_byte_list(
            reversedbyte_datalist)
        # send to Tyvak socket
        send_to_Tyvak(packed_byte_list)
        print('')
       # empty content variable
       content_bitlist = []

# if packet is being processed but end of read chunk
# continue packet here when new chunk of data is read
else:

# process rest of packet as above
 content_index = 0
 search_index = 0
 for content_item in data:
  Bit_Deck.append(content_item)
  content_bitlist.append(content_item)
  content_index += 1
  # if find_syncword(DR):
  flag_found = 0
  for i in range(8):
   flag_found=1
   if Bit_Deck[-8+i] != sync_word_flag[i]:
    flag_found = 0
    break
  if flag_found:
   data = data[search_index+content_index:]
   search_index = 0
   content_index = 0
   packet_in_process = 0
```

```
            break
          elif content_index > 1000*8:
            content_bitlist = []
            data = data[search_index+content_index:]
            search_index = 0
            content_index = 0
            packet_in_process = 0
            break




      else: #except:
        if not fifo_is_open:
          time.sleep(.5)
          continue
        print("fifo closed, waiting for it to reopen")
        sys.stdout.flush()
        fifo_is_open = 0



print("finish")
sys.exit()
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, 2002.

[2] P. Burns, *Software Defined Radio for 3G*. Artech House Inc, Boston, 2003.

[3] W. Tuttlebee, *Software Defined Radio: Enabling Technologies*. Wiley, Chichester, England, 2002.

[4] *NI USRP-292x/293x Datasheet*, National Instruments Corporation, 08 2015.

[5] J. M. Roehrig, "Communication technologies in space applications: Theoretical background and basic methodology," Helmut Schmidt University, Hamburg, Tech. Rep., 2016.

[6] R. Mutagi, "Mixed-signal: Understanding the sampling process," *defense electronics: RF Design*, 09 2004.

[7] "Ni usrp," Online: http://sine.ni.com/nips/cds/view/p/lang/en/nid/212990 5/31/16 09:26.

[8] J. S. Beasley and G. M. Miller, *Modern Electronic Communication*. Pearson Prentice Hall, Upper Saddle River, NJ, 2005.

[9] F. Xiong, *Digital modulation techniques*, 2nd ed. Artech House, Boston, 2002.

[10] B. Wagner and M. Barr, "Introduction to digital filters," *Embedded Systems Programming*, pp. 47–48, December 2002.

[11] A. B. Carlson, *Communication Systems: An Introduction to Signals and Noise in Electrical Communication*, C. P. B. R. J. C., Ed. McGraw-Hill, New York, 2002.

[12] J. M. Roehrig, "Communication technologies in space applications: Theoretical background and basic methodology," Helmut Schmidt Universitaet, Tech. Rep., 2016.

[13] Analog-Devices-Inc, *Linear Circuit Design Handbook*, H. Zumbahlen, Ed. Analog Devices, 2008.

[14] CSBSJU, "Introduction to digital filters," cSBSJU Department of Physics: Online: http://www.physics.csbsju.edu/217/digital_filter.pdf 7/20/16 14:07.

[15] M. Porteous, "Introduction to digital resampling," RF Engines Ltd, Innovation Centere, Newport, Tech. Rep.

[16] E. Cubukcu, Ed., *Root Raised Cosine (RRC) Filters and Pulse Shaping in Communication Systems*. NASA, 2012.

[17] S. Haykin, *Communication systems*, 4th ed. Wiley, New York, 2001.

[18] M. Joost, "Theory of root-raised cosine filter," Tech. Rep., 2010.

[19] "Raised-cosine filter," Online: http://research.omicsgroup.org/index.php/Raised-cosine_filter 6/07/16 09:01.

[20] J. G. Proakis, *Digital Communications*, 5th ed. McGraw-Hill, New York, NY, 2008.

[21] "baseband encoding notes," Online: http://www.radicalabacus.org/post/show/network/baseband_encoding_notes 6/07/16 10:02.

[22] TAPR, "Ax.25 amateur packet-radio link-layer protocol," Tech. Rep. 2.2, 1998.

[23] TAPR, "Ax.25 link-layer protocol specification," Online: http://www.tapr.org/pub_ax25.html 6/07/16 10:18.

[24] V. Schroer, "gr-ax25," Online: https://github.com/dl1ksv/gr-ax25 6/02/16 14:34.

[25] "Bisonsat," Online: http://space.skyrocket.de/doc_sdat/bisonsat.htm 7/20/16 16:06.

[26] "Estcube homepage," Online: http://www.estcube.eu/en/home 7/20/16 16:08.

[27] "Estcube-1receiver," Online: https://moo.cmcl.cs.cmu.edu/trac/cgran/wiki/ESTCube-1Receiver 7/20/16 16:10.

[28] J. Kopitzki, "Software defined radios for space applications-theroretical background and basic implementation," HSU/UniBw H, Tech. Rep., 2014.

[29] J. Kopitzki, "Development and implementation of a communication scheme for software define radios," Master's thesis, HSU/UniBw H, 2014.

[30] P. A. Bernhardt, N. Kassim, M. Sulzer, J. Abel, and G. Minelli, "Propcube science mission objectives," 2016.

[31] *BeeLine GPS Users Guide V0.2*, BigRedBee, LLC, 2006.

[32] *KWM-1200+ KWM-9612+*, Kantronics.

[33] P. A. Bernhardt, N. Kassim, M. Sulzer, J. Abel, and G. Minelli, "Propcube radio beacons satellites for ionospheric and radio astronomical applications," NRL, Arecibo, TYVAK, NPS, Tech. Rep., 2016.

[34] G. Krebs, "Propcube 1, 2, 3 (flora, fauna, merryweather)," Online: http://space.skyrocket.de/doc_sdat/propcube-1.htm 6/13/2016 10:10.

[35] GNU-Project, "About gnu radio," Online: http://gnuradio.squarespace.com/about/ 5/27/16 13:30.

[36] GNU-Project, "Coding guide," Online: http://gnuradio.org/redmine/projects/gnuradio/wiki/Coding_guide_impl 5/27/16 14:00.

[37] GNU-Project, "Gnu radio - guided tutorial," Online: http://gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorials 6/07/16 10:34.

[38] *Bell 202 Modem*, MX-COM, 2000.

[39] N1VG, "1200 baud packet radio details," Online: http://n1vg.net/packet/ 6/08/16 15:43.

[40] T. Sailer-HB9JNX, "Dsp modems," in *11. Internationale Packet-Radio Tagung Darmstadt*, 1995.

[41] J. Miller-G3RUH, "9600 baud packet radio modem design," Online: http://www.amsat.org/amsat/articles/g3ruh/109.html 6/08/16 16:23.

[42] J. A. Magliacane-KD2BD, "The kd2bd 9600 baud modem," Online: http://www.amsat.org/amsat/articles/kd2bd/9k6modem/ 6/08/16 16:26.

[43] M. Leech, "Re: [national instruments] (er-3615) question: Sample rate ettus n210," may 2016, ettus Research LLC, Technical Support Assistance.

[44] C. E. Shannon, "Communication in the presence of noise," *Proc. Institute of Radio Engineers*, vol. 37, no. 1, pp. 10–21, Jan 1949, reprint as classic paper in: Proc. IEEE, Vol. 86, No. 2, (Feb 1998).

[45] M. Chepponis-K3MC and P. Karn-KA9Q, "The kiss tnc: A simple host-to-tnc communications protocol," in *ARRL 6th Computer Networking Conference*, 1987.

[46] "Gnu radio companion," Online: http://www.ece.uvic.ca/ elec350/grc_doc/ 6/13/16 06:27.

[47] "Gnu radio manual and c++ api reference," Online: http://gnuradio.org/doc/doxygen/index.html 6/14/16 08:34.

[48] "firdes class reference," Online: http://gnuradio.org/doc/doxygen/classgr_1_1filter_1_1firdes.html 6/13/16 16:05.

[49] "Gnu radio - windows installation," Online: http://gnuradio.org/redmine/projects/gnuradio/wiki/WindowsInstall 7/20/16 10:19.

[50] B. Seeber, "Gnu radio tutorials," 2014.

[51] "The cyclic redundancy check (crc) for ax.25," Online: http://practicingelectronics.com/articles/article-100003/article.php 8/4/16 13:19.

[52] T. Rondeau, "Hilbert transform and windowing," Online: http://www.trondeau.com/blog/2013/9/26/hilbert-transform-and-windowing.html 6/23/16 08:27.

[53] "Fir digital filter design," Online: https://www.dsprelated.com/freebooks/sasp/FIR_Digital_Filter_Design.html 6/23/16 08:29.

[54] T. Rondeau, "Psk symbol recovery," Online: http://gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorial_PSK_Demodulation/13 6/23/16 15:27.

[55] "Pulse-shape filtering in communications systems," Online: http://www.ni.com/white-paper/3876/en/ 6/24/16 13:29.

[56] D. CaJacob, "Fsk modulation parameters," Online: http://gnuradio.org/redmine/projects/gnuradio/wiki/SignalProcessing 6/23/16 09:40.

[57] "Gfsk.py," Online: http://gnuradio.org/redmine/projects/gnuradio/repository/revisions/71127c2d086f9f881b349922ed8cf24/entry/gr-digital/python/gfsk.py 7/19/16 09:01.

[58] "Notes on mm clock recovery," Online: https://www.tablix.org/ avian/blog/archives/2015/03/notes_on_m_m_clock_recovery/ 7/19/16 09:21.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California