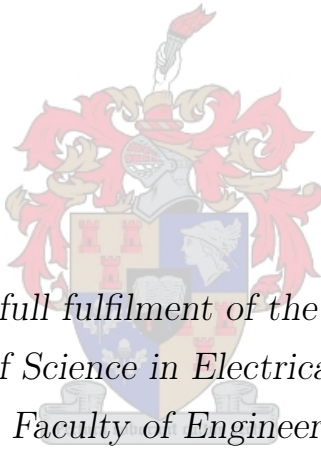


Design and Construction of a Modem for Satellite Use

Hendrik Petrus Daniel van Wyk



*Thesis presented in full fulfilment of the requirements for the
degree Master of Science in Electrical and Electronic
Engineering in the Faculty of Engineering at Stellenbosch
University*

Supervisor: Dr. Riaan Wolhuter
Department of Electrical and Electronic Engineering

April 2014

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: April 2014

Copyright © 2014 Stellenbosch University
All rights reserved.

Abstract

Design and Construction of a Modem for Satellite Use

H.P.D. van Wyk

Department of Electrical and Electronic Engineering,

University of Stellenbosch,

Private Bag X1, Matieland 7602, South Africa.

Thesis: MScEng

April 2014

In this thesis the design and testing of the baseband components of a modem intended for use as a telemetry and control link for a low earth orbit satellite is presented. This includes parts of the physical layer as well as a basic data-link layer. Binary phase-shift keying (BPSK) is used as the modulation scheme and is realised by making use of software defined radio on a standard x86 computer with digital to analogue and analogue to digital converters that use a universal serial bus (USB) connection. The data-link layer makes use of a basic framing scheme and provides bit synchronisation, an automatic repeat request (ARQ) system and Bose Chaudhuri Hocquenghem (BCH) forward error correction (FEC). The ARQ system ensures that data is delivered reliably and the FEC improves the system's performance in noisy conditions. A prototype system was developed to test the performance of the individual layers as well as the system as a whole. For testing purposes the Linux Internet Protocol (IP) stack is used as higher network layers. Radio frequency hardware developed by Verschaeve [1] modulates the signal away from baseband, transmits it over the air and receives it.

Uittreksel

Ontwerp en Bou van 'n Modem vir Satellietgebruik

(“Design and construction of a modem for satellite use”)

H.P.D. van Wyk

*Departement Elektries en Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.*

Tesis: MScIng

April 2014

In hierdie tesis word die ontwerp en toetsing van die basisband komponente van 'n modem, bedoel vir gebruik op 'n satelliet in 'n lae-aarde wentelbaan, bespreek. Die ontwerp sluit dele van die fisiese vlak sowel as 'n verbindingsvlak in. Binêre faseskuiwleuteling word gebruik as die modulasieskema en word verweselik deur gebruik te maak van sagteware gedefinieerde radio. Dit voer uit op 'n standaard x86 rekenaar wat deur middel van USB gekoppel is aan 'n digitaal-na-analoog- en 'n analoog-na-digitaal-omsetter. Die verbindingsvlak het 'n eenvoudige ramingskema. Dit voorsien bis sinkronisasie, die hersending van verlore rame en Bose Chaudhuri Hocquenghem (BCH) voorwaartse foutverbetering. Die hersending van verlore rame verseker dat data betroubaar oorgedra kan word en foutverbetering verbeter die stelsel se vermoëns in ruiserige toestande. 'n Prototipe stelsel is ontwikkel om die vermoëns van die individuele vlakke, sowel as die stelsel as 'n geheel, te toets. Tydens toetsing is die Linux Internet Protokool stapel gebruik vir die hoër netwerk vlakke. Radio komponente wat deur Verschaeve [1] ontwikkel was is gebruik om die sein uit te saai en te ontvang.

Acknowledgements

I would like to express my sincere gratitude to the following people:

- Dr Riaan Wollhuter for his guidance and support.
- Tim Verschaeve for his work on the radio frequency components of this project.
- My parents for their support during my studies.

Contents

Declaration	i
Abstract	ii
Uittreksel	iii
Acknowledgements	iv
Contents	v
List of Figures	x
List of Tables	xiii
Nomenclature	xiv
1 Introduction	1
1.1 Background	1
1.2 Objectives of the Project	1
1.3 Summary of Work Done	2
1.4 Outline and Outcomes	2
1.5 Overview of this Document	3
2 Literature Study	5
2.1 The OSI Layers	5
2.2 Modulation and Demodulation	6
2.2.1 Basics	6
2.2.2 Quadrature Modulation	7
2.2.3 Demodulation	9
2.2.4 Constellation Diagrams	10

<i>CONTENTS</i>	vi
2.3 Conversion Between Digital and Analogue	10
2.3.1 Digital to Analogue Converters	11
2.3.2 Analogue to Digital Converters	13
2.4 Error Correction	13
2.5 Summary	14
3 System Requirements	15
3.1 Requirements of the Complete System	15
3.2 System Layout	16
3.3 Requirements of Components	16
3.3.1 Physical Layer	16
3.3.2 Data-link Layer	18
3.3.3 Higher Layers	19
3.4 Summary	19
4 Overview of Complete System	20
4.1 System Architecture	20
5 Physical Layer	24
5.1 Hardware	24
5.1.1 Hardware Choice	24
5.1.2 Programming	25
5.1.3 USB Interface Firmware	25
5.1.4 Microcontroller Firmware	27
5.1.5 Driver	31
5.2 Software Defined Radio	35
5.2.1 GNU Radio	35
5.2.2 Modulation Flow Graph	35
5.2.3 Demodulation Flow Graph	37
5.3 Integration	40
5.3.1 Format Conversion	40
5.3.2 Connection of Components	41
5.3.3 Physical Layer Operation	41
5.4 Summary	41
6 Testing of Physical Layer	44
6.1 GNU Radio Modulation Tests	44

6.1.1	Test Different Stages of Modulation and Compare to Theory	44
6.1.2	Test Delay of Demodulator Lock	50
6.1.3	Modulate and Demodulate Random Data	52
6.1.4	Modulate and Demodulate Random Data in Presence of Noise	52
6.2	DAC and ADC Tests	53
6.2.1	Output Test Waveforms to DAC	53
6.2.2	Sample Test Waveforms with ADC	57
6.3	Tests of Complete Physical Layer	59
6.3.1	Transmit and Receive Random Data	60
6.3.2	Transmit and Receive Random Data with RF Hardware	61
6.4	Summary	64
7	Data-link layer	66
7.1	Frame Components	66
7.1.1	Attached Synchronisation Marker	66
7.1.2	Cyclic Redundancy Check	67
7.1.3	BCH Forward Error Correction	67
7.2	Randomiser	68
7.3	Automatic Repeat Request	68
7.4	Frame Structure	69
7.5	Integration	69
7.5.1	Transmitting Side	70
7.5.2	Receiving Side	73
7.6	Summary	76
8	Testing of Data-link Layer	77
8.1	Attached Synchronisation Marker	77
8.1.1	Find ASM in Random Bits	77
8.1.2	Find an ASM that is Spread Over Two ASM Finder Library Calls	78
8.1.3	Find Corrupted ASM in Random Bits	78
8.1.4	Determine Number of False Positives	78
8.2	CRC	79
8.2.1	Test the Output of the Library Against a Reference Output	79

<i>CONTENTS</i>	viii
8.2.2 Attempt to Detect Varying Amounts of Corruption . . .	79
8.2.3 Test for Detection of Burst Errors	80
8.3 BCH Forward Error Correction	80
8.3.1 Test the Number of Correctable Errors	80
8.4 Randomiser	81
8.4.1 Test if the Correct Sequence is Produced	81
8.4.2 Test if Sequence Repeats After 255 Bits	81
8.4.3 Test if Sequence Resets Properly	81
8.5 Automatic Retry Request	82
8.5.1 Test Basic Transmission of Frames	82
8.5.2 Test the Retransmission of Frames	82
8.5.3 Test of Whether Synchronisation System Works	83
8.5.4 Test of Whether Synchronisation System Works with Lost Frames	83
8.6 Complete Data-link Layer	83
8.6.1 Test the Transmission of Data	84
8.6.2 Test the Transmission of Data in the Presence of Noise .	84
8.6.3 Test the Performance of the Data-link Layer with FEC .	84
8.6.4 Test the Performance of the Data-link Layer Without FEC	85
8.7 Summary	85
9 System Integration	87
9.1 Higher Network Layers	87
9.2 Overview of Integrated System	87
9.3 Execution Script	90
9.4 Integration with Hardware	90
9.5 Initial Observations	90
9.6 Summary	91
10 Testing of Complete System	92
10.1 Latency and Packet Loss Tests	92
10.2 Throughput Tests	94
10.3 Summary	95
11 Conclusions and Recommendations	96
11.1 Conclusion and Summary	96

<i>CONTENTS</i>	ix
11.2 Work and Development Successfully Completed	97
11.3 Recommendations	97
11.3.1 General	97
11.3.2 Physical Layer	97
11.3.3 Data-link Layer	98
List of References	99
Appendices	102
A Test System Used for Physical Layer	103
A.1 $\frac{E_b}{N_0}$ Estimation	103
A.2 Measuring BER	104
A.3 Test Hardware	104
B Prediction of Frame Error Rate	106
B.1 Transmissions per Frame	107
C Execution Script of Complete System	108

List of Figures

2.1	A quadrature modulator.	7
2.2	Rectangular to polar conversion.	8
2.3	A quadrature demodulator.	10
2.4	BPSK constellation.	11
2.5	QPSK constellation.	11
2.6	The impulse response of a zero order hold.	11
2.7	A signal as a series of impulses.	12
2.8	The signal in Figure 2.7 being output by a DAC.	12
2.9	The Fourier transform of a sampled signal.	12
2.10	The Fourier transform of the rectangular pulse in Figure 2.6.	13
2.11	The Fourier transform of low-pass filtered DAC output.	13
3.1	Basic layout of the entire system.	16
4.1	Complete layout of the transmitter.	22
4.2	Complete layout of the receiver.	23
5.1	An USB-XMEGA module.	25
5.2	DAC output with 32 MHz internal clock.	29
5.3	DAC output with external clock.	29
5.4	Flow diagram of MCU firmware.	32
5.5	Flow diagram of DAC and ADC driver.	34
5.6	GNU Radio modulator flow graph.	36
5.7	The FFT of the software modulator output with random input data.	37
5.8	GNU Radio demodulator flow graph.	38
5.9	Physical layer overview.	40
5.10	I and Q DAC output.	42
5.11	FFT of DAC output as complex signal.	42

<i>LIST OF FIGURES</i>	xi
5.12 Output of ADC.	43
5.13 FFT of ADC output.	43
6.1 GNU Radio modulator flow graph with test sample points marked.	45
6.2 GNU Radio demodulator flow graph with test sample points marked.	45
6.3 Time and frequency domain of differentially encoded symbols. (Sample point 1)	46
6.4 Time and frequency domain of root raised cosine filtered symbols. (Sample point 2)	46
6.5 Time and frequency domain of symbols after mixing away from 0 Hz. (Sample point 3)	48
6.6 Time and frequency domain of symbols after mixing to approximately 0 Hz. (Sample point 4)	48
6.7 Time and frequency domain of symbols after frequency locked loop. (Sample point 5)	49
6.8 Time and frequency domain of symbols after clock synchronisation. (Sample point 6)	50
6.9 Number of samples output versus frequency offset of frequency locked loop for different input frequency offsets.	51
6.10 Bit error rate versus $\frac{E_b}{N_0}$ for demodulator. Theoretical values and ideal coherently demodulated values included for reference.	52
6.11 Output of DAC if given an 75 kHz square wave as input.	53
6.12 FFT of DAC output if given an 75 kHz square wave as input.	54
6.13 Output of DAC if given a 75 kHz sine wave as input.	54
6.14 FFT of DAC output if given an 75 kHz sine wave as input.	55
6.15 Output of I DAC if given modulated random data as input.	55
6.16 Output of Q DAC if given modulated random data as input.	56
6.17 FFT of I DAC with modulated random data input.	56
6.18 FFT of DAC outputs combined into complex baseband.	56
6.19 Output of ADC sampling a square wave.	57
6.20 FFT of output of ADC sampling a square wave.	58
6.21 Output of ADC sampling a sine wave.	58
6.22 FFT of output of ADC sampling a sine wave.	58
6.23 Output of ADC sampling modulated random data.	59
6.24 FFT of output of ADC sampling random modulated data.	60
6.25 Bit error rate of system using DAC connected directly to ADC.	60

<i>LIST OF FIGURES</i>	xii
6.26 Frame error rate of system using DAC connected directly to ADC. . .	62
6.27 Theoretical frame error rates.	62
6.28 Bit error rate of system using RF transmission and reception. . . .	63
6.29 Frame error rate of system using RF transmission and reception. . .	63
7.1 Data-link layer overview.	70
7.2 Diagram of transmitting part of the data-link layer.	72
7.3 Diagram of receiving part of the data-link layer.	74
8.1 Measurement of data-link layer performance in the presence of noise.	85
8.2 Measurement of data-link layer performance in the presence of noise with FEC turned off.	86
9.1 Software overview.	88
9.2 Image of the USB-XMEGA connected to the rest of the hardware. .	90
A.1 Image of the circuit used to test the BER of the DAC directly connected to the ADC.	105

List of Tables

7.1	Frame structure.	70
8.1	Result of testing ASM library's corruption tolerance versus false positives.	79
10.1	Results of latency and packet loss tests.	92
10.2	Results of throughput tests.	94

Nomenclature

Abbreviations

ADC	Analogue-to-digital converter
ARQ	Automatic Repeat reQuest
ASM	Attached synchronisation marker
AWGN	Additive white Gaussian noise
BCH	Bose Chaudhuri Hocquenghem
BER	Bit error rate
BPSK	Binary phase-shift keying
CRC	Cyclic redundancy check
DAC	Digital-to-analogue converter
FEC	Forward error correction
FLL	Frequency locked loop
I/O	Input/output
IP	Internet protocol
LFSR	Linear feedback shift register
LO	Local oscillator
OSI	Open systems interconnect
PC	Personal computer
QPSK	Quadrature phase-shift keying
RF	Radio frequency
SDK	Software development kit
SDR	Software defined radio
TCP	Transmission Control Protocol

NOMENCLATURE

xv

USB Universal serial bus

Variables

$I(t)$	In-phase part of signal	
$Q(t)$	Quadrature part of signal	
θ	Phase	[rad]
f	Frequency	[Hz]
f_c	Carrier frequency	[Hz]
f_s	Sampling frequency	[Hz]
A	Amplitude	
$\frac{E_b}{N_0}$	Energy per bit to noise power spectral density ratio	

Chapter 1

Introduction

1.1 Background

After the experience gained with the communication system of the last South African satellite, SumbandilaSat, it was decided that some research into improving the communication system would be useful. Known problems were, for example, that the old system provided no retransmission or error correction capabilities. Investigation of a possible next generation of software defined modems was identified as one way to approach this research.

This led to the topic of designing a modem for use as a future satellite telemetry and control link.

1.2 Objectives of the Project

The following objectives were identified for this project:

- Design and implementation of a baseband modem for use as the telemetry and telecommand link for a low earth orbit satellite to evaluate a possible next generation of software defined modems.
- Design and implementation of a simple protocol stack for this modem that would guarantee delivery of data.
- Addition of forward error correction to this system for reasons of robustness and throughput under noisy conditions.

- A baseband modem which would need to interface with the physical layer high frequency components required for link implementation. These components are being designed in another master's project [1].

1.3 Summary of Work Done

- Designed and implemented software and analogue to digital hardware for a prototype modem.
- Proved that a low cost DAC/ADC pair can be used for satellite communication with good results.
- Implemented fairly modular data-link layer and physical layer components that could be reused in another project.
- Performed an investigation into using GNU Radio for modulation and demodulation using custom hardware and proved it viable.

1.4 Outline and Outcomes

This document describes the implementation of the baseband components of a software defined radio modem designed with the intention of being used as the telemetry and control link for a satellite. The focus was on the prototyping of ideas, rather than optimising them for practical use. This allowed the project to disregard design factors such as low power usage and computational efficiency.

The physical layer of the modem consists of a relatively inexpensive USB DAC/ADC pair and a GNU Radio based software modulator and demodulator. BPSK was used as the modulation scheme. The radio frequency components of the physical layer were developed in another masters project.

The software modulator and demodulator were found to be within 0.4 dB $\frac{E_b}{N_0}$ of the theoretical bit error rate value. Adding the DAC and ADC pair increased the $\frac{E_b}{N_0}$ by 2 dB over the theoretical and with the addition of the radio frequency components, this increased it 6 dB $\frac{E_b}{N_0}$.

The data-link layer consists of software that interfaces with the physical layer and provides retransmissions through an automatic repeat request (ARQ) mechanism as well as Bose Chaudhuri Hocquenghem (BCH) based forward error correction. The data-link layer was tested and found to operate as expected.

The higher layers of the system were provided by the Linux internet protocol (IP) stack. This allowed the performance of the system to be measured under practical conditions. Under optimal conditions the latency of the system is around 460 ms and the throughput about 19 kbit/s.

1.5 Overview of this Document

The document layout is as follows:

Chapter 2 begins by giving a basic overview of the OSI model and how this project fits into it. It then goes on to examine the basic theory behind quadrature modulation, conversion between the analogue and digital domains and gives a short summary of BCH error correction.

The actual design of the modem starts in Chapter 3, where the requirements of the system are determined. Combining the basic requirements such as data throughput and modulation scheme with a basic layout of the system leads to more in depth requirements for different system components. At this point the distinction between the physical and data-link layers is made.

In Chapter 4 a brief overview of the whole system, including how it interfaces with the radio frequency components that are also developed as part of this project but are not within the scope of this document, is given.

Chapter 5 covers the design and development of the physical layer. It begins by describing the DAC and ADC chosen for the modem as well as the firmware and driver development for these components. The GNU Radio based software modulator and demodulator are then described. It is then shown how the different components in the physical layer communicate and are integrated.

In Chapter 6 there is a description of how the physical layer is tested. Testing starts by sampling the output of the software modulator and demodulator at various points to confirm correct outputs. The bit error rate for the software modulator and demodulator on their own are also calculated. The output of the DACs and ADCs are then tested for different signals. Finally the bit error rates of the physical layer both with and without the radio components are tested.

Chapter 7 presents the development of the data-link layer. It begins by describing the components needed to generate a frame. The frame structure is

then described. It then moves on to a description of the logic of the data-link layer, how frames are buffered, transmitted, received and acknowledged.

The testing of the data-link layer is then shown in Chapter 8. Unit tests were performed on the components of the data-link layer where it was sensible to do so. Tests were also performed on the complete data-link layer to confirm that data can be transmitted reliably.

Chapter 9 shows how the physical layer, data-link layer and higher layers were integrated.

Chapter 10 describes tests performed on the system as a whole. Both the latency of the system and the throughput were measured at different noise levels.

Finally the document is concluded in Chapter 11, which gives some recommendations for possible improvements.

Chapter 2

Literature Study

This chapter gives a short overview of some of the theory used in this project. It gives an overview of the OSI layer, quadrature modulation, conversion between analogue and digital and BCH forward error correction.

2.1 The OSI Layers

The open systems interconnect (OSI) layered network model is a useful reference for designing any telecommunication system. Although practical networks are not always neatly divisible into OSI layers the concept of dividing a network into more easily designable and understandable chunks can be applied to this project and gives some insight into what is required from a communication system.

The OSI model divides a network into the following layers[2]:

1. **Physical**

This layer contains the hardware used for communication as well as the modulators and demodulators used to convert from binary data to analogue signals and back.

2. **Data-link**

This layer provides the means to transfer data between nodes on a network by sending the data as frames. This layer can also include error detection and correction.

3. Network

This layer allows nodes on different networks to communicate with each other. Routing happens on this layer.

4. Transport

This layer provides transparent transmission of data. This is done by providing reliability, flow control and error control.

5. Session

This layer provides the ability to have semi-permanent connections between programs running on nodes on the network.

6. Presentation

This layer translates data between machine dependant formats to network format. This can include encryption or other data representation schemes.

7. Application

Application dependant protocols reside in this layer. Examples of this include HTTP and FTP.

The OSI model is not the only layered network model. The Internet model, for example, has only four layers [3].

The scope of this project covers the design of only the physical and data-link layers. An Internet protocol (IP) based stack is used for the demonstration of higher layers.

2.2 Modulation and Demodulation

2.2.1 Basics

Modulation is the process of using some baseband signal to change some parameter of a high frequency carrier signal. The carrier is usually sinusoidal and the parameters being modified are usually amplitude, frequency or phase, or some combination of these. This leads to the well known amplitude modulation (AM), frequency modulation (FM) and phase modulation (PM) [4, chap. 1]. Equation 2.2.1 gives a generalised sinusoid with A the amplitude, f the frequency in hertz and θ the phase in radians.

$$x(t) = A \cos(2\pi ft + \theta) \quad (2.2.1)$$

Demodulation is the extraction from the carrier of these changes.

The rest of this section will show how quadrature demodulation can be used to simultaneously modulate both amplitude and phase, as well as how this modulation scheme can be demodulated. It is also shown how this leads to the complex baseband representation of a signal and how this is useful for digital communication.

2.2.2 Quadrature Modulation

Quadrature modulation is a method of modulating both amplitude and phase simultaneously by making use of an in-phase ($I(t)$) and quadrature ($Q(t)$) signal and mixing these signals with two carriers of frequency f_c that are 90° out of phase [5]. Figure 2.1 shows this configuration.

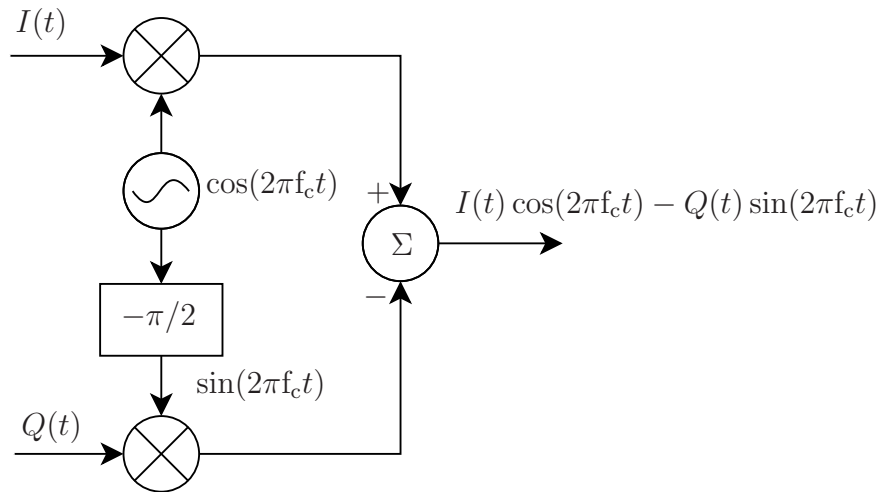


Figure 2.1: A quadrature modulator.

This yields the modulated signal

$$x_{\text{mod}}(t) = I(t) \cos(2\pi f_c t) - Q(t) \sin(2\pi f_c t) \quad (2.2.2)$$

We show that this is indeed modulation of both amplitude and phase by applying

$$\cos(a + b) = \cos a \cos b - \sin a \sin b \quad (2.2.3)$$

to equation 2.2.1. We then set $f = f_c$ where f_c is the carrier frequency. This yields

$$x(t) = A \cos(2\pi f_c t) \cos \theta - A \sin(2\pi f_c t) \sin \theta \quad (2.2.4)$$

From equation 2.2.4 it can easily be seen that $x(t) = x_{\text{mod}}(t)$ with $I(t) = A \cos \theta$ and $Q(t) = A \sin \theta$. Thus changing I and Q will result in changing A and θ , modulating both amplitude and phase.

By using the carrier frequency as a 0° reference the signal $x(t) = A \cos(2\pi ft + \theta)$ can be plotted on a polar diagram as in Figure 2.2. From this diagram it can be seen that the I and Q values can be converted to amplitude and phase by rectangular to polar conversion. The results of equation 2.2.4 show that

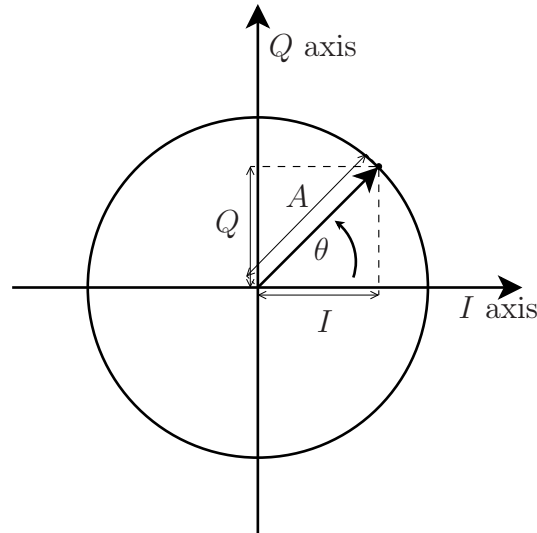


Figure 2.2: Rectangular to polar conversion.

this is exactly what a quadrature modulator does.

Plotting I and Q on a rectangular plane also leads to the realisation that although both I and Q can be seen as separate unrelated signals, they can also be represented as one complex signal:

$$x_{IQ}(t) = I(t) + jQ(t) \quad (2.2.5)$$

This representation is convenient as it produces a baseband signal that is equivalent to its quadrature modulated radio frequency version. For example, the frequency spectra of the complex baseband signal around 0 Hz will match the spectra of the real valued modulated signal around its carrier frequency. This relationship can be proven by considering the following case [6]:

$$\begin{aligned}\Re\{x_{\text{IQ}}(t)e^{j2\pi f_c t}\} &= \Re\{x_{\text{IQ}}(t)[\cos(2\pi f_c t) + j \sin(2\pi f_c t)]\} \\ &= I(t) \cos(2\pi f_c t) - Q(t) \sin(2\pi f_c t)\end{aligned}\quad (2.2.6)$$

This result is the same as that of equation 2.2.2, meaning that $\Re\{x_{\text{IQ}}(t)e^{j2\pi f_c t}\}$ is the same operation as is being done by quadrature modulation.

Another way of representing only the real component of a function is:

$$\Re\{x_{\text{IQ}}(t)e^{j2\pi f_c t}\} = \frac{1}{2}[x_{\text{IQ}}(t)e^{j2\pi f_c t} + x_{\text{IQ}}^*(t)e^{-j2\pi f_c t}]\quad (2.2.7)$$

Combining equation 2.2.2, equation 2.2.6 and equation 2.2.7 yields:

$$x_{\text{mod}}(t) = \frac{1}{2}[x_{\text{IQ}}(t)e^{j2\pi f_c t} + x_{\text{IQ}}^*(t)e^{-j2\pi f_c t}]\quad (2.2.8)$$

Assume the Fourier transform of $x(t)$ is $X(f)$ and remembering that if $F[x(t)] = X(f)$ then $F[x^*(t)] = X^*(-f)$:

$$X_{\text{mod}}(f) = \frac{1}{2}[X_{\text{IQ}}(f - f_c) + X_{\text{IQ}}^*(-f - f_c)]\quad (2.2.9)$$

This means that the modulated signal does indeed have the same spectrum that x_{IQ} has, centred around f_c instead of 0 Hz. It also has a mirror image of the spectra around $-f_c$, which is to be expected of a real signal.

2.2.3 Demodulation

It has been shown how the signal can be modulated for transmission. We now show the demodulation process. Figure 2.3 shows the layout of the demodulator. If we consider only the upper arm of the demodulator and use equation 2.2.2 as input then the output of the upper arm before the low-pass filter is:

$$\begin{aligned}x_{\text{demod}}(t) &= [I(t) \cos(2\pi f_c t) - Q(t) \sin(2\pi f_c t)]2 \cos(2\pi f_c t) \\ &= I(t)[1 + \cos(4\pi f_c t)] - Q(t)[\sin(4\pi f_c t) + \sin(0)] \\ &= I(t) + I(t) \cos(4\pi f_c t) - Q(t) \sin(4\pi f_c t)\end{aligned}\quad (2.2.10)$$

The low-pass filter then removes the images of $I(t)$ and $Q(t)$ around $2f_c$ Hz. This yields $I(t)$, the desired result. The same method can be used on the bottom leg of the demodulator to demodulate $Q(t)$ [4, chap. 4.4].

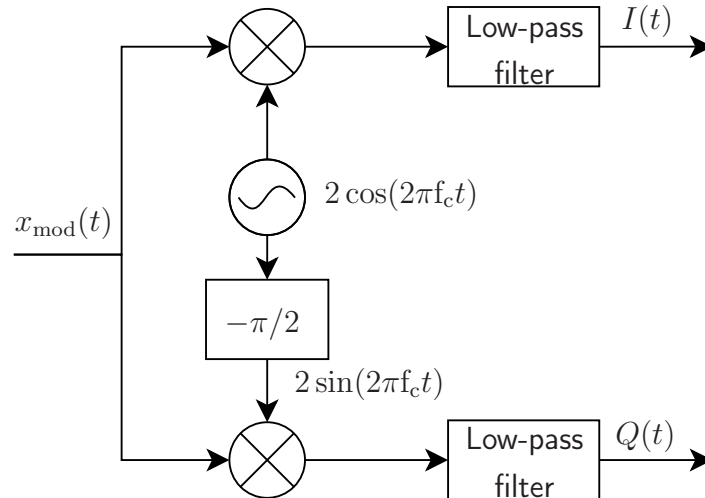


Figure 2.3: A quadrature demodulator.

2.2.4 Constellation Diagrams

So far this chapter has discussed the general case for using quadrature modulation to simultaneously transmit two signals. For use in digital communications it is useful to map data to symbols with discrete I and Q values. When these symbols are plotted, the result is known as a constellation diagram. Constellation diagrams are drawn on the same plane as the rectangular to polar conversion shown in Figure 2.2 and the same method can be used to transform these constellation points to amplitude and phase modulation.

Figure 2.4 shows a binary phase-shift keying (BPSK) constellation diagram and Figure 2.5 shows a quadrature phase-shift keying (QPSK) diagram. Many other common constellations exist. A digital scheme that makes use of only amplitude and phase modulation can be represented as a constellation diagram.

2.3 Conversion Between Digital and Analogue

Digital to analogue conversion is the process of converting binary data into analogue signals and analogue to digital conversion is the process of converting analogue signals to binary data. Both processes are necessary for transmitting and receiving digital data [4, chap. 6.1].

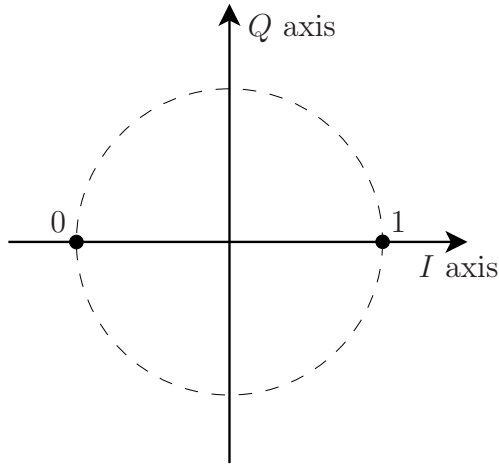


Figure 2.4: BPSK constellation.

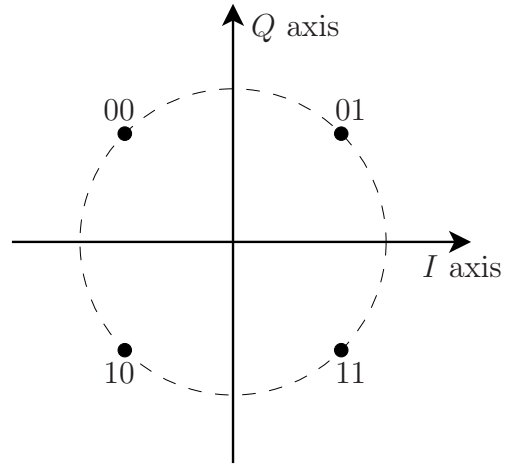


Figure 2.5: QPSK constellation.

2.3.1 Digital to Analogue Converters

A digital to analogue converter (DAC) has a digital connection as input and produces an analogue signal as output. The DAC converts discrete digital values to an analogue signal, usually by converting samples to discrete analogue values and holding these values for the sample period. This is known as a zero-order hold.

This process can be seen as the conversion of the digital values to impulses, each one sampling period apart, with areas related to the digital values (Figure 2.7) and convoluted with a time shifted rectangular function that is one sample period (T_s) long and has an amplitude of 1 (Figure 2.6). This produces the output shown in Figure 2.8.

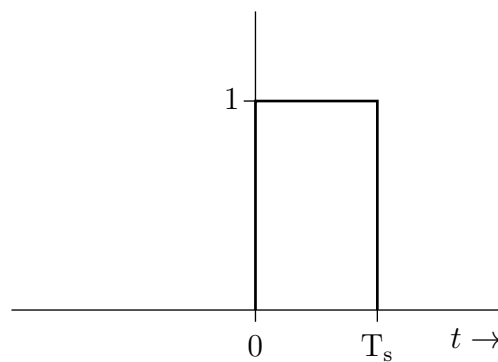


Figure 2.6: The impulse response of a zero order hold.

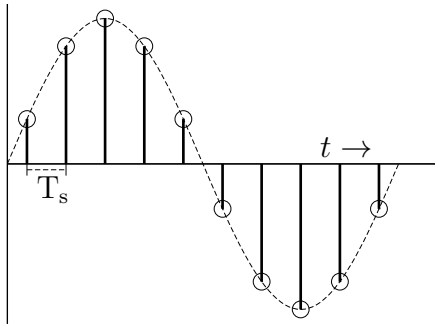


Figure 2.7: A signal as a series of impulses.

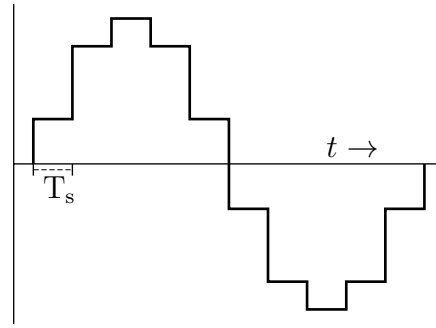


Figure 2.8: The signal in Figure 2.7 being output by a DAC.

The desired signal can then be recovered by using a low pass filter. As will be shown next, the recovered signal is not a perfect reconstruction of the intended signal since the rectangular pulse has a $\text{sinc}()$ frequency response causing the DAC output to have a non linear frequency response.

How this works can better be shown in the frequency domain: Take some signal with a rectangular frequency spectrum with bandwidth B and amplitude A , Figure 2.9 shows the frequency spectrum of this signal after sampling. Figure 2.10 shows the Fourier transform of a rectangular function. Figure 2.11 shows the sampled signal after it has been convoluted with the rectangular function and low-pass filtered to remove all frequency components above $\frac{f_s}{2}$.

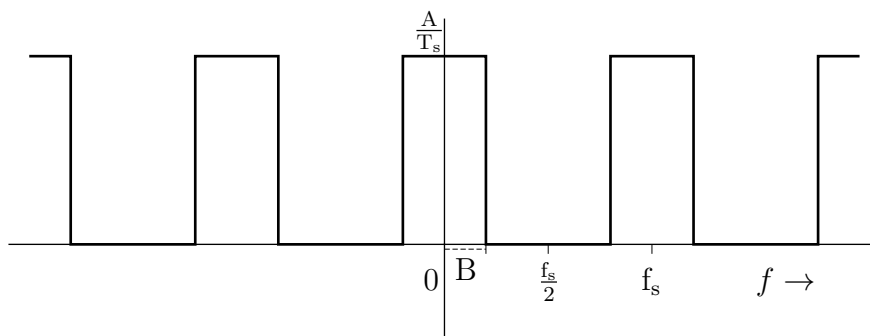


Figure 2.9: The Fourier transform of a sampled signal.

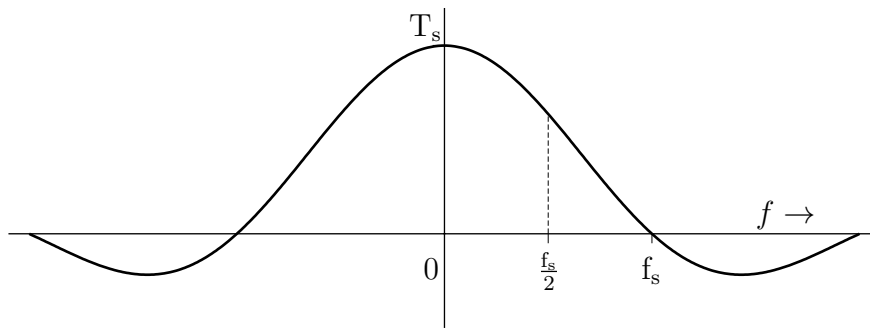


Figure 2.10: The Fourier transform of the rectangular pulse in Figure 2.6.

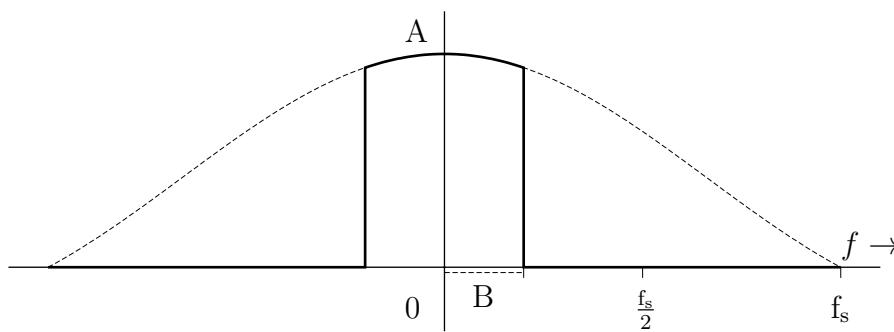


Figure 2.11: The Fourier transform of low-pass filtered DAC output.

2.3.2 Analogue to Digital Converters

An analogue to digital converter (ADC) samples an analogue signal at some interval and converts the analogue value at that time to a discrete digital value. This process can be modelled as multiplying the signal by a pulse train. Figure 2.7 shows this.

2.4 Error Correction

Error correcting codes correct errors by adding redundant information. The time limits of the project did not allow for a deep investigation into error correction nor was it the main focus. It was merely sought to implement a reasonable degree of error correction. BCH codes were chosen for their simplicity to implement. The number of correctable errors can be set during

the creation of the code. Parity bits can also be added to the transmitted data without changing the data.

BCH codes fall into the class of cyclic codes. In simple terms a message is represented by a polynomial and then multiplied by a generator polynomial to produce a code word. This code word is then transmitted.

The modulus of the received code word and the generator polynomial is then taken to produce the syndrome polynomial. From the syndrome polynomial the error-locator polynomial can be computed and the location of errors can be determined. The methods of calculating the error locations are quite involved and will not be covered here.

Once the error locations are known it is a simple matter of correcting the affected bits. [7, chap. 3]

2.5 Summary

The different layers of the OSI model and how this project fits into them have been discussed. The basics of quadrature modulation and demodulation as well as constellation diagrams have also been covered. This was followed by an explanation of the digital to analogue and analogue to digital conversion processes. Finally, a short summary of BCH error correction was given.

In the next chapter the requirements of the system being designed are discussed.

Chapter 3

System Requirements

This chapter details the design requirements of both the system as a whole and of the individual components within the system.

3.1 Requirements of the Complete System

Broadly speaking the aim of this project is to try to improve upon the telemetry and telecommand modem used by SumbandilaSat. The requirements of the system come directly from the consensus reached about reasonable goals for the project, which should result in a realisable system with acceptable performance.

The requirements for the system are a modem capable of the following:

- Baud rate of 32 kBd.
- BPSK modulation.
- Full duplex transmission.
- Use of forward error correction.
- Not suffering from the same performance issues caused by the combination of Transmission Control Protocol (TCP) and packet loss.
- Easily interfacing with higher layers of the network stack.

3.2 System Layout

Figure 3.1 shows the basic layout of the system. This layout as well as the requirements for each component, was created by an iterative process and not necessarily in the order shown here. Some decisions about requirements were made after some of the other components were already partially or fully implemented. The specifics of each component will now be discussed in the next section.

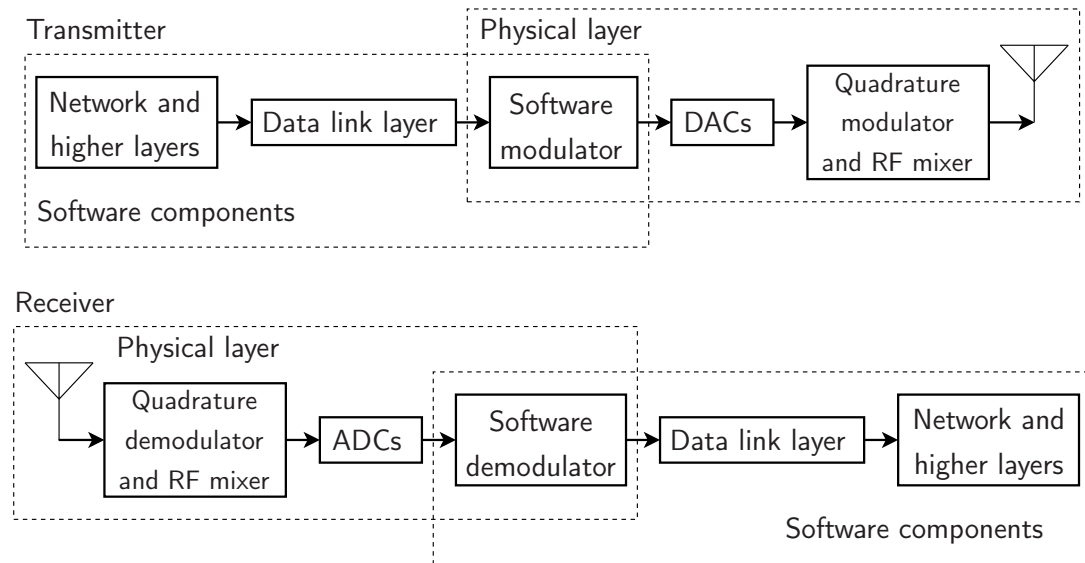


Figure 3.1: Basic layout of the entire system.

3.3 Requirements of Components

This section details the requirements of each individual component in the system. By breaking the system up into smaller components that each have their own requirements it becomes possible to develop and test these components independently.

3.3.1 Physical Layer

The physical layer is required to convert bits into an analogue signal. In this design a large part of the physical layer is done in software.

3.3.1.1 Quadrature Modulator and RF mixers

The design of these components is not within the scope of this document. The only requirements of interest to this document are that these components need to accept an analogue output from the DAC and must produce an analogue signal for the ADC to digitise.

3.3.1.2 DAC

The DAC pair produces the I and Q signals that are fed into the quadrature modulator. Its requirements are as follows:

- Capable of outputting a signal with a bandwidth of 64 kHz to achieve necessary baud rate at worst case bandwidth usage.¹
- Output voltage and impedance should be capable of driving the quadrature modulator.
- Capable of simultaneously outputting a sample on both DACs.
- Should accept a digital stream of data that contains the I and Q signals.

3.3.1.3 ADC

The ADC pair digitises the I and Q signals output by the quadrature demodulator. Its requirements are as follows:

- Capable of digitising a signal with a bandwidth of 64 kHz.
- Input voltage level compatible with output from quadrature demodulator.
- Capable of simultaneously sampling both ADCs.
- Output the sampled data as a digital stream of I and Q values.

3.3.1.4 Software Modulator

This component is responsible for converting from bits to constellation points as well as pulse shaping and filtering. Input should be data frames and output should be a DAC compatible I and Q signal stream.

¹BPSK with a root raised cosine filter and a roll-of factor of 1. [baud rate \times (1+roll-of factor)]

3.3.1.5 Software Demodulator

The software demodulator must lock on to the received signal and decode its constellation. The input is an I and Q data stream from an ADC pair and the output is demodulated bits.

3.3.2 Data-link Layer

This layer is responsible for framing, synchronisation and reliability of communication. On the transmitting side this layer accepts packets from the higher layers and outputs frames to the software modulator in the physical layer. On the receiving side this layer receives bits from the software demodulator that can contain frames, and outputs packets to the higher layers.

3.3.2.1 Framing

Data from higher layers should be broken up into frames. Each frame consists of a header that is used for sequencing, error correction and acknowledgement. It also contains information about the data payload. Some frames contain a data payload and others are only used for acknowledgements.

3.3.2.2 Synchronisation

This component transmits a synchronisation marker and detects this marker on the receiving end. This is needed to detect when a frame is being received.

3.3.2.3 Automatic Retry Request

This component manages the retransmission of lost frames. This is required to ensure reliability.

3.3.2.4 Forward Error Correction

This component further increases reliability of transmissions and is also required by the project goals.

3.3.3 Higher Layers

The aim of this project does not include the design of any of the higher layers. These layers are still useful for testing and demonstration, so a pre-existing implementation of the higher layers should be used for this. These components are required to interface with the data link layer implementation.

3.4 Summary

In this chapter the general requirements of the system as well as more detailed requirements for individual components have been discussed. In the next chapter a detailed overview of the entire system is shown.

Chapter 4

Overview of Complete System

This chapter gives a detailed overview of the complete system. The system requirements in Chapter 3 gave some idea of how the system will look as a whole, but left out large amounts of detail not relevant to a requirement specification. This chapter shows an overview of the final system to serve as a reference for the coming chapters where it will be shown how the system was designed.

Figure 4.1 shows the layout of the transmitter and Figure 4.2 the receiver layout. The individual components and their design will be detailed in the following chapters.

4.1 System Architecture

This section tries to briefly explain the environment the system runs on and how all components interact with each other. These interactions will be covered in depth in Chapter 9.

The parts of the system that are of interest to this document are largely software based, with the only hardware being the DACs and ADCs with their supporting microcontroller. A single microcontroller with all the necessary DAC and ADC hardware was chosen. This microcontroller connects to a standard desktop PC via USB.

This computer runs Ubuntu 12.04, a GNU/Linux based operating system. All the different software components are implemented as separate processes and make use of Linux pipes for inter process communication (IPC). With this scheme the standard input and output (STDIN and STDOUT) can be used for

IPC. This simplifies the design of the components since it removes the need for more advanced IPC libraries. This also allowed the different components to be, for the most part, independently developed and tested.

A Linux TUN device was used as the higher layers in the network stack. This is a software network interface that appears to the system as a hardware interface. Packets going to and from it are passed to a software process instead of being sent over a network.

This is only a prototype system, so many of the design decisions sacrifice performance for ease of implementation. In a production system signal processing will probably be done on a dedicated FPGA or DSP chip. The software components will also be likely to use a more efficient IPC method or be implemented as a single process. These features can always be implemented later when the principles have been proven.

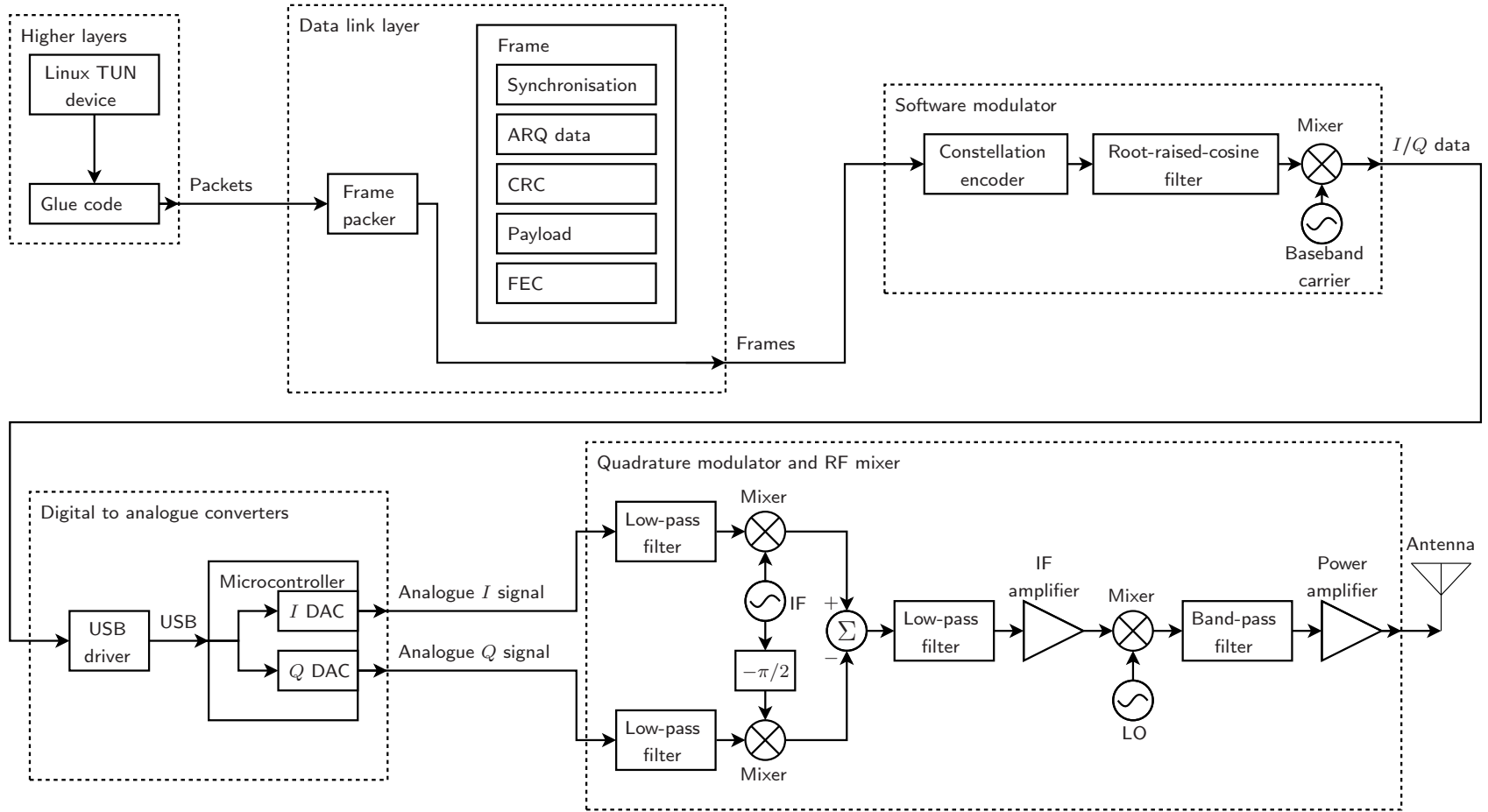


Figure 4.1: Complete layout of the transmitter.

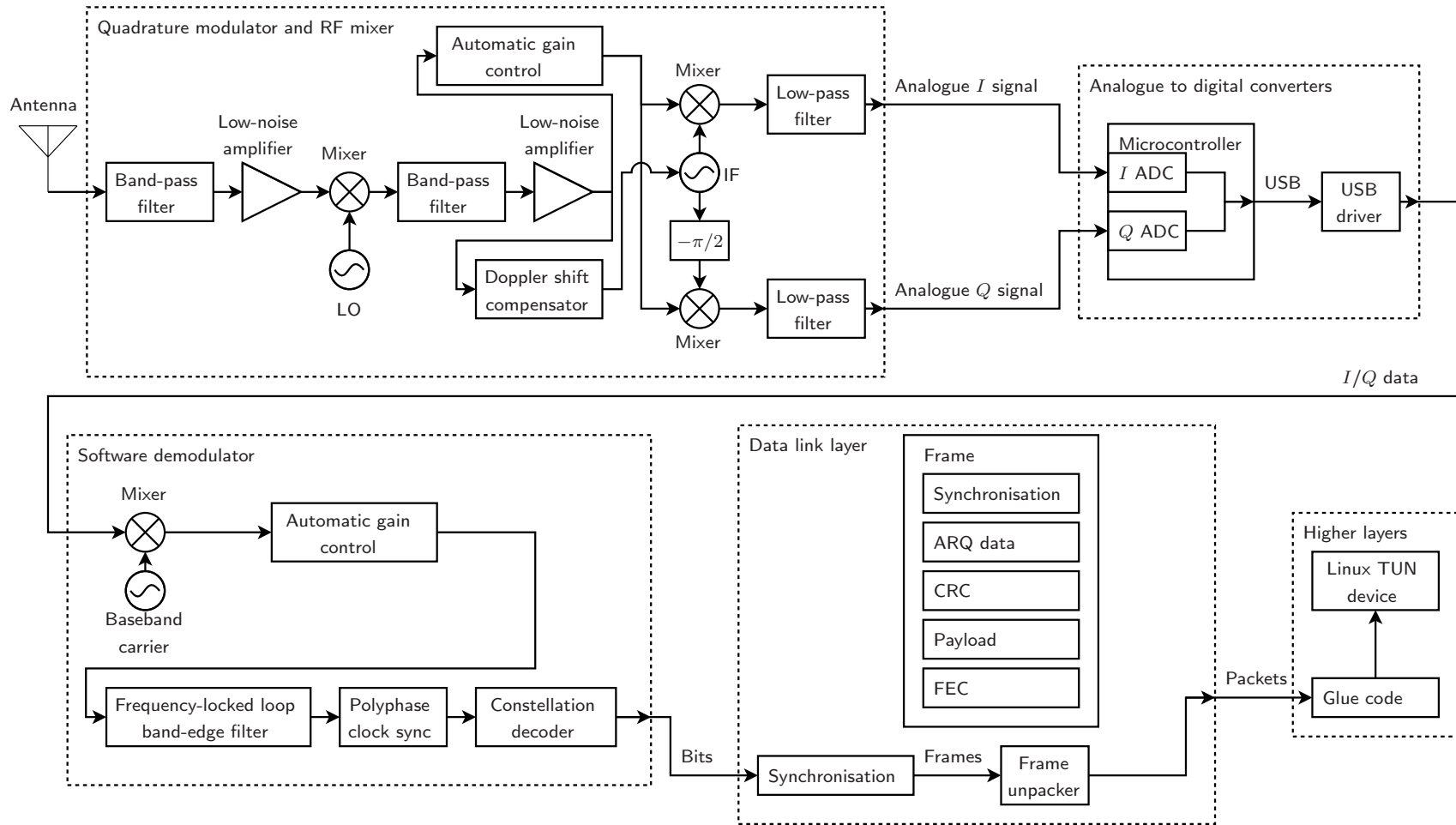


Figure 4.2: Complete layout of the receiver.

Chapter 5

Physical Layer

This chapter shows the details of the design of all the components in the physical layer. This includes the hardware ADC and DAC pairs as well as the software that does the baseband modulation. These designs are based on the requirements shown in Section 3.3.1.

5.1 Hardware

This section covers the design of the DAC and ADC pairs as well as the firmware and drivers required for their operation.

5.1.1 Hardware Choice

The ZTEX USB-XMEGA was chosen as a single module for both the DAC pair and ADC pair. The module consists of an ATxmega128A1 microcontroller (MCU) connected to an EZ-USB FX2LP USB peripheral controller and all supporting circuitry [8]. The EZ-USB FX2LP allows the microcontroller to communicate over USB. The ATxmega128A1 has two 12-bit 1 Msps DACs and two 12-bit 2 Msps ADCs [9]. Figure 5.1 shows the USB-XMEGA module.

According to the requirements in Section 3.3.1 a bandwidth of 64 kHz is required. To output or sample a 64 kHz signal requires a sample rate of only 128 kHz, according to sampling theory. Choosing a DAC and ADC capable of outputting such a high sample rate compared to this requirement could seem extreme. In practice it is useful to use much higher sampling rates in both the DAC and ADC. In both cases this relaxes the design constraints of the

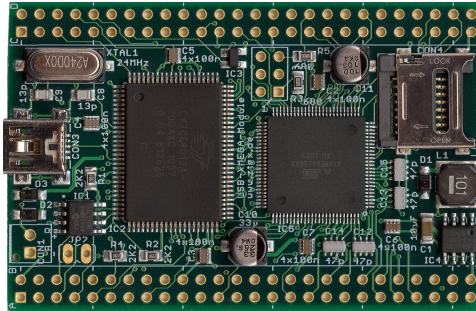


Figure 5.1: An USB-XMEGA module.

analogue filters used for signal reconstruction and anti-aliasing and in the DAC case helps reduce the influence of the sinc response caused by a zero order hold as shown in Section 2.3.1, Figure 2.11. This also gives the advantage of potentially allowing higher data rates. A problem was also encountered with the RF components in that the analogue low-pass filters had a fairly high cut-off frequency forcing the high sample rate to be necessary. If these filters were to be replaced with lower cut-off filters then it would be worth experimenting with decreasing the DAC and ADC sample rate.

5.1.2 Programming

ZTEX provides a software development kit (SDK) that includes the necessary libraries and tools to create firmware for both the microcontroller and the USB peripheral controller. The software allows programming of the USB peripheral controller and the microcontroller over USB so no extra programmer is required.

5.1.3 USB Interface Firmware

The EZ-USB FX2LP requires firmware to configure its internal workings. The controller possesses a completely functional processor and is capable of accommodating a large set of different usage modes. The controller has support for all the USB transfer modes and has a variety of different ways in which the endpoints can be configured. The controller makes use of interface FIFOs (First In First Out) that can be directly connected to USB endpoints or routed through the controller's processor.

The firmware on the controller sets all the registers necessary to set the desired device mode. After the configuration is complete the device processor is

no longer necessary for USB data transfer and can be used for other purposes.

The ZTEX SDK provides macros to help configure the endpoints and headers with register mappings to configure any other options.

The firmware sets the required register values and then loops forever, doing nothing. The only oddity is the use of the `SYNCDELAY` macro to add a delay between writing to different registers. This is needed because the built in processor requires a delay between certain kinds of register accesses.

The following options are set by the firmware:

- Use 48 MHz system clock.
- Use FIFO interfaces in slave mode so that communication can be driven by the ATxmega128A1.
- Use endpoint 2 as an IN endpoint in isochronous mode with 4 512 byte buffers.
- Use endpoint 6 as an OUT endpoint in isochronous mode with 4 512 byte buffers.
- Set endpoint 2 packet size to 512 bytes.
- Automatically send packets to host on endpoint 2.
- Automatically send zero length packets to host if no packets are available.
- Send up to 3 packets per microframe.
- Use 16 bit FIFO outputs. This combines two 8 bit ports into one 16 bit port for communication with the ATxmega128A1.

This configures the system to allow constant time interval streaming to and from the MCU over USB. It also sets the transfer of data to be controlled from the MCU. As will be seen in the development of the MCU firmware, the overhead of handling interrupts on the MCU is high and interrupts would be required if the peripheral controller were to control the transfers.

To clarify the directionality of USB endpoints, IN endpoints send data to the USB host, the computer, and OUT endpoints send data to the USB device.

16 bit integers are used to transfer the 12 bit samples. This wastes a small amount of USB bandwidth, but requires less processing on the MCU to use

the samples. Some of the unused bits were later used for synchronisation, as explained in Section 5.1.4.3.

5.1.4 Microcontroller Firmware

The microcontroller firmware needs to read sample values from the USB peripheral controller and output those values on the two DACs. It also needs to read samples from its ADCs and output them to the USB peripheral controller.

5.1.4.1 Communication Between Microcontroller and USB Peripheral Controller

The MCU and USB peripheral controller are connected by three 8-bit ports. Two of these ports are used to transfer data and the third is used for control. The MCU can request which endpoint FIFO to communicate with, as well as cause data to be written to or from the USB peripheral controller by making use of the control port. Asserting the slave read and slave output enable pins causes data from the FIFO selected by the FIFO address pins to be output to the data ports. Asserting the slave write pin causes the current state of the data ports to be written to the FIFO selected by the FIFO select pins.

5.1.4.2 Design of Program Flow

Fairly early on in the development of this firmware it was seen that the traditional way of writing embedded code did not yield nearly high enough performance. Usually when performing an action at a set interval on a microcontroller a counter will be used that causes an interrupt after a set period. This interrupt is then used to trigger whatever action should be performed. The problem with interrupts in this application is that the overhead for calling an interrupt is around 13 clock cycles for even the simplest interrupts. If the system had a sampling rate of 1 Msps with its default clock of 32 MHz then there would only be time for 32 instructions between samples, of which 13 is a significant amount.

The solution found for this problem was not to use any explicit timing control and let the sample rate be derived from the execution time of the firmware. This completely removes timing overhead.

Since the USB peripheral controller and USB itself can deliver samples at a much higher rate than the 4 MB/s¹ needed for a 1 Msps sample rate, they were not a limiting factor.

5.1.4.3 Dealing with Lost Samples

Using a general purpose computer to do the software modulation does not guarantee that there will always be samples available to send to the DAC or that samples from the ADC will be read quickly enough. Having data become corrupted by this is undesirable but unavoidable in this case. Samples generally get lost once every few minutes and this was deemed to be acceptable. One complication that did arise from this was that if an uneven number of samples were lost then subsequent samples would be interpreted incorrectly by the MCU or host computer. The Q value from the first set of I/Q samples will be interpreted as an I value and the I value of the next set of samples will be interpreted as a Q value. This caused the I and Q channels to be swapped, as well as a small phase distortion between samples.

This was solved by making use of the unused bits from the 16 bit values being sent to the DACs and from the ADCs. The actual samples are only 12 bits in both the DAC and ADC cases, so this leaves 4 unused bits that can carry extra information without needing to transfer more data.

The samples sent to the DAC are unsigned integers, so for 12 bit values the most significant bit is always 0. Setting this bit to 1 on the I sample allows the MCU to check whether it is an I sample or not and skip it if it becomes unsynchronised. The samples coming from the ADC are signed 12 bit values, so the most significant bit can be either 1 or 0. Flipping the most significant bit of the I sample makes it possible for the computer to check if it is an I sample or not and act accordingly.

This approach discards one extra sample if synchronisation is lost. This is deemed acceptable, considering that the alternative is having I and Q periodically reverse. This causes a performance hit on the maximum sample rate that the MCU is capable of, because of the extra code, but again this is better than the alternative.

¹ $\frac{10^6 \text{ samples} \times 16 \text{ bits} \times 2 \text{ channels}}{8} = 4 \text{ MB/s}$

5.1.4.4 Jitter of Microcontroller Clock

The internal 32 MHz clock of the MCU was found to be too jittery for stable communications. The clock output of the EZ-USB FX2LP was used as an external clock source for the MCU. The EZ-USB FX2LP is clocked from an external oscillator and has far less jitter than the ATxmega128A1. Figure 5.2 shows the output of the DAC with the internal 32 MHz clock and Figure 5.3 shows the output with the EZ-USB FX2LP as an external 48 MHz clock source that gives a system clock of 36 MHz.

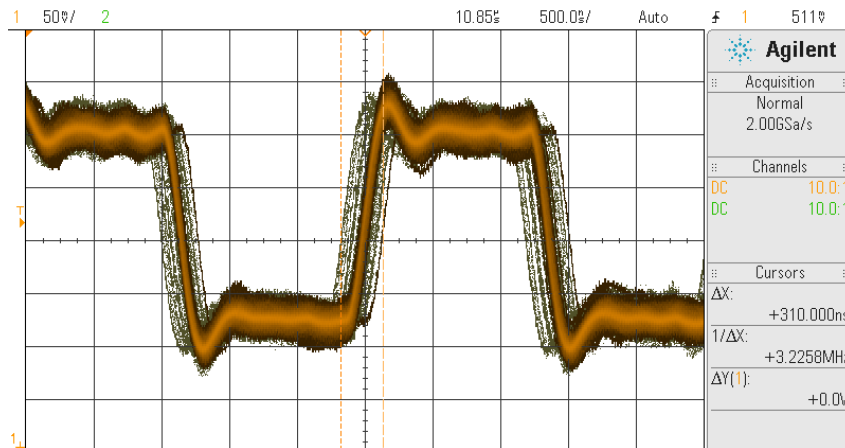


Figure 5.2: DAC output with 32 MHz internal clock.

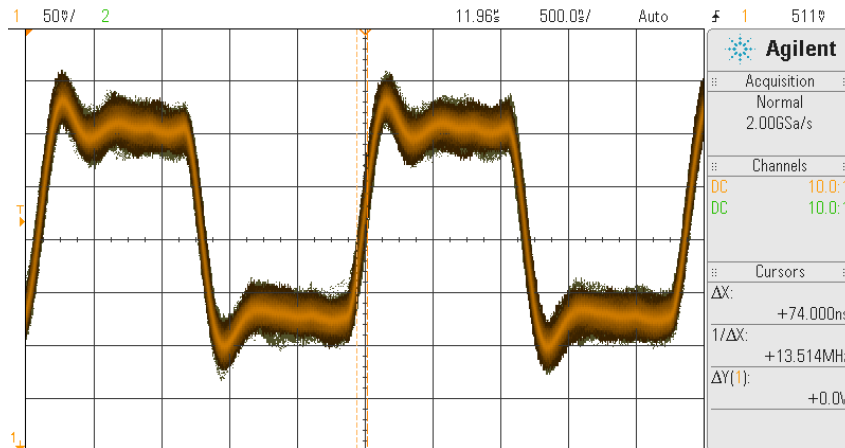


Figure 5.3: DAC output with external clock.

5.1.4.5 Sample Synchronisation

It was found that a one cycle delay between outputting I and Q samples and between reading them was not optimal for quadrature modulation since it causes phase distortion. The ATxmega128A1 has an event system that can be used to trigger multiple actions on a certain event. This system was used to trigger the DAC pair and ADC pair simultaneously. This also requires one instruction instead of the multiple instructions needed to trigger the DACs and ADCs individually, resulting in a performance improvement. This instruction was also optimised by telling the compiler to use one of the indirect address registers to store a pointer to the address that needed to be written to. Instructions making use of the indirect address registers take fewer cycles than direct addressing instructions.

5.1.4.6 ADC Timing Constraints

Since the ADC clock runs at a different rate than the system clock the time between samples will differ if the number of clock cycles in the main loop is not a multiple of the number of clock cycles in one ADC clock cycle. This limits the practical sample rate to $\frac{\text{CLK}_{\text{sys}}}{\text{ADC}_{\text{prescaler}}} \times \frac{1}{n}$ where n is an integer. The difference in cycles between the main loop and ADC clock was corrected by adding no operation (NOP) instructions to the main loop.

5.1.4.7 Further Optimisations

Even using no interrupts it was not possible to achieve a full 1 Msps sample rate. Some other performance improvements were made to increase the sample rate:

The ATxmega128A1 has the ability to map four registers in extended I/O memory space to I/O memory space. This feature allows the `IN` and `OUT` instructions to be used instead of the `LDS` and `STS` instruction for reading and writing to these registers. The `IN` and `OUT` instructions use only 1 clock cycle each while the `LDS` and `STS` use 2 clock cycles each.

Reading from the USB converter requires some small delays between clocking the USB converter and the value being read from USB. These delays were initially implemented with no operation (NOP) instructions. A small gain in

performance was made by replacing the NOP instructions with the instructions that read samples from the ADC registers.

Although not technically code optimisations, further speed improvements were made by upgrading the version of the `avr-gcc` and `avr-libc` that came with the Ubuntu 12.04 repositories. One flaw in the old version causes certain I/O operations to use two or three instructions taking up to 5 cycles instead of a single 2 cycle instruction [10]. The new versions gave a decent improvement in performance.

A final speed improvement was made by slightly overclocking the MCU. The external clock as described in Section 5.1.4.4 provides a 48 MHz clock. This clock can be internally multiplied by 3 and divided by 4 to produce a 36 MHz system clock. This is slightly faster than the 32 MHz rated speed and was not observed to have any negative side effects.

Section 5.1.4.6 limits the possible sample rate to $\frac{\text{CLK}_{\text{sys}}}{\text{ADC prescaler}} \times \frac{1}{n}$ where n is an integer. With the system clock at 36 MHz and the ADC prescaler at 16 the highest sample rate achievable by this system is 750 kHz. Any higher sample rate would require the main loop to be significantly more efficient.

5.1.4.8 Hardware Summary

The general program flow of the firmware is thus to configure the needed registers and to then enter an infinite loop that reads samples from USB and writes them to the DACs and reads samples from the ADCs and writes them to USB. Figure 5.4 shows this.

5.1.5 Driver

The driver is the software program that interfaces with the USB-XMEGA. This driver is based on the `libusb` [11] library. This library gives userspace programs the ability to interface with USB hardware without the need for writing kernel modules.

5.1.5.1 General Design of Driver

The driver uses the standard input, `STDIN`, to read samples from a Linux pipe and uses the standard output, `STDOUT`, to write samples to another Linux pipe. This allows for simplistic inter process communication.

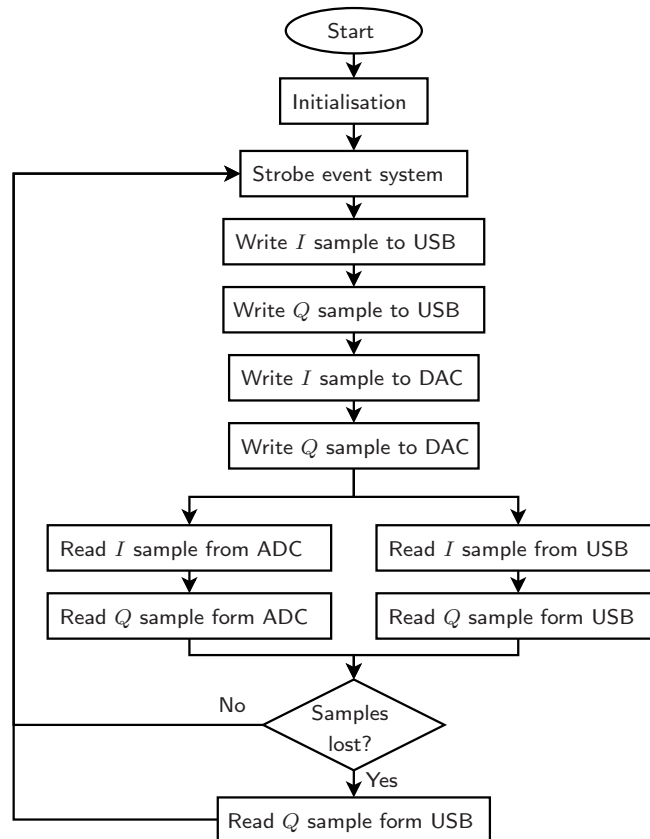


Figure 5.4: Flow diagram of MCU firmware.

The driver uses the `select()` system call to multiplex I/O. This system call monitors several file descriptors and blocks until a file descriptor is ready for some kind of I/O. All I/O performed is non-blocking. This means that the `read()` or `write()` function will return immediately instead of blocking if I/O is not yet possible. This arrangement allows reading and writing to `STDIN` and `STDOUT` as well as reading and writing to the MCU to be done in the same thread with minimal latency.

5.1.5.2 libusb

I/O with `libusb` consists of submitting USB requests to read or write. A USB `IN` request reads data and an `OUT` request writes data. This corresponds to the `IN` and `OUT` endpoints on the USB peripheral controller.

A callback function should also be provided to a USB request. `libusb` then provides a set of file descriptors for use with `select()`. If any of these file descriptors reports that it is ready for I/O then the `libusb` event handling

function should be called to handle any pending I/O operations. Any requests that are completed will call the callback function registered to them.

5.1.5.3 Buffering of Input and Output

A set of circular buffers are used to buffer the input from `STDIN` and the output to `STDOUT`. This is needed for non-blocking I/O operations because the situation can arise where data has been read but the file descriptor to which it should be written is not yet ready to receive more data.

5.1.5.4 Synchronisation to MCU

Samples need to be delivered to the DAC at a specific rate otherwise the buffers on the USB interface chip will underflow or overflow. The only practical way found to output at the correct rate was to match the rate of incoming samples from the ADC. The DAC and ADC have the same sample rate, so for each byte of data received from the ADC a byte of data is sent to the DAC.

5.1.5.5 Feedback Provided to Data-link Layer

It was discovered that using Linux pipes that are written into much faster than they are read from can cause excessive buffering, leading to very high latency. To combat this a feedback mechanism was implemented where the USB driver also informs the Data-link layer of the number of bytes required by the DAC. This prevents the pipe buffers from filling up and increasing latency.

5.1.5.6 Driver Summary

Figure 5.5 shows a flow chart of the driver logic. The driver initially starts two IN transfers, ensuring that a transfer is always queued in the kernel. The main loop is then entered and the file descriptors that should be watched for readiness are chosen. This allows the driver to check only for file descriptors that are currently of use. There is, for example, no point in checking if it is possible to read from `STDIN` if the input buffer is already full. The `select()` call is then made and the driver sleeps until a file descriptor is ready. The driver then wakes up and handles any file descriptors that are ready for I/O. If an IN transfer has completed then feedback is also sent to the Data-link layer requesting more bytes.

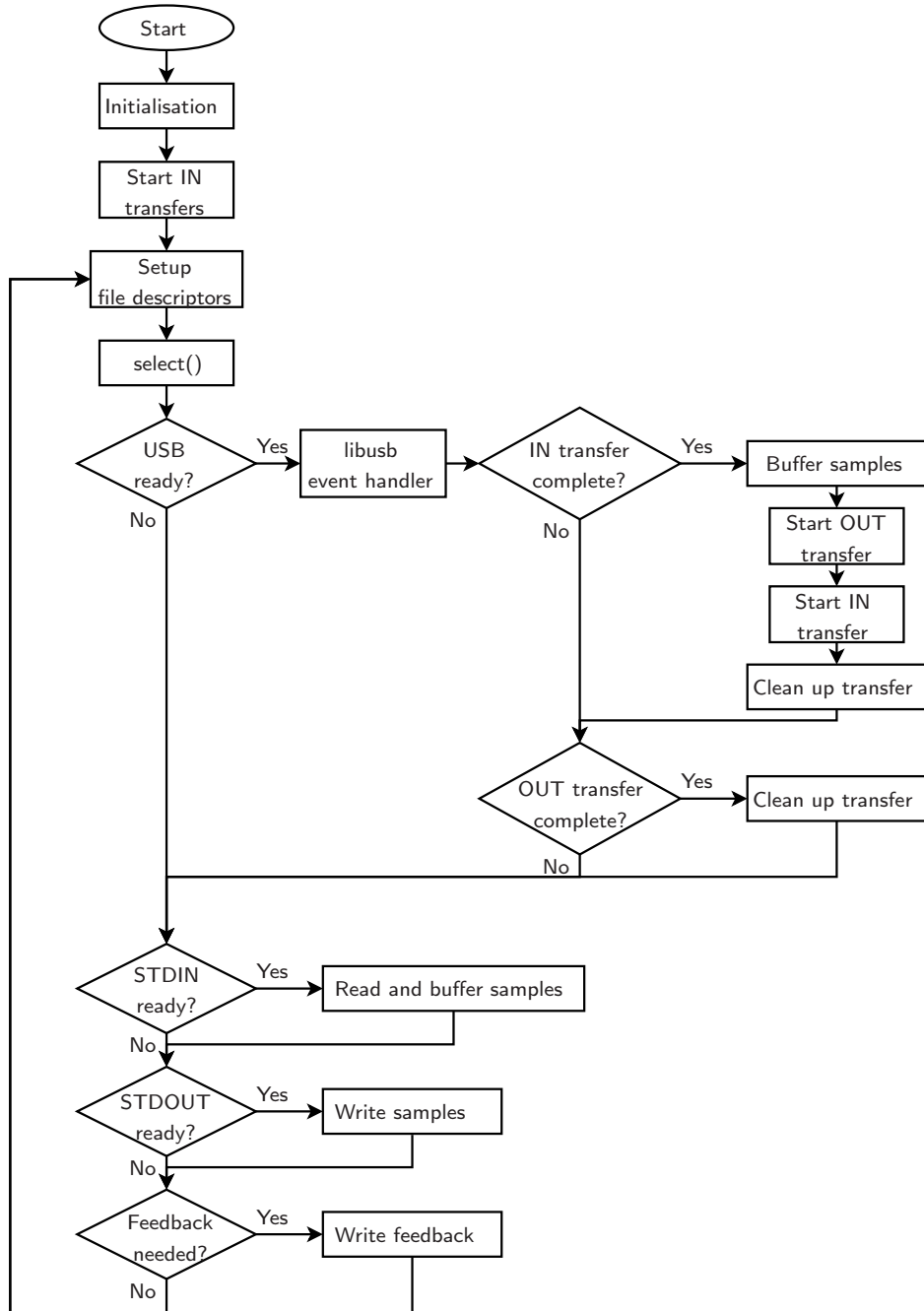


Figure 5.5: Flow diagram of DAC and ADC driver.

5.2 Software Defined Radio

A Software defined radio (SDR) is a radio where some of the components that have traditionally been implemented in hardware have been replaced by software. In the case of this project the modulation from bits to baseband I and Q samples and the demodulation from I and Q samples to bits are done in software.

5.2.1 GNU Radio

GNU Radio is a set of open source libraries and signal processing blocks [12]. It provides a Python interface for connecting blocks together as well as a Simulink-like graphical interface: GNU Radio Companion. In GNU Radio nomenclature a collection of connected blocks is called a flow graph. The flow graphs used for modulation and demodulation will now be shown.

5.2.2 Modulation Flow Graph

The modulator converts bits to I/Q samples. The layout and design of the modulator will now be discussed.

Figure 5.6 shows the flow graph of the modulator and Figure 5.7 shows the FFT of the output of the modulator with random input data.

5.2.2.1 Interface

STDIN is used for input and STDOUT is used for output. This is done by using standard GNU Radio file sources and sinks and making them read from `/dev/stdin` and `/dev/stdout`. The input format is bits stored as bytes with a value of 1 or 0 and the output is a `complex double` stream.

5.2.2.2 Bit Encoding

The bits are first differentially encoded making the next bit dependent on the value of the previous bit. A 1 is now encoded by the same value as the previous bit and a 0 is encoded by the opposite value of the previous bit. This simplifies the demodulator because it can retrieve bits by comparing the received value to the previous one instead of having to compare it to some absolute phase. This

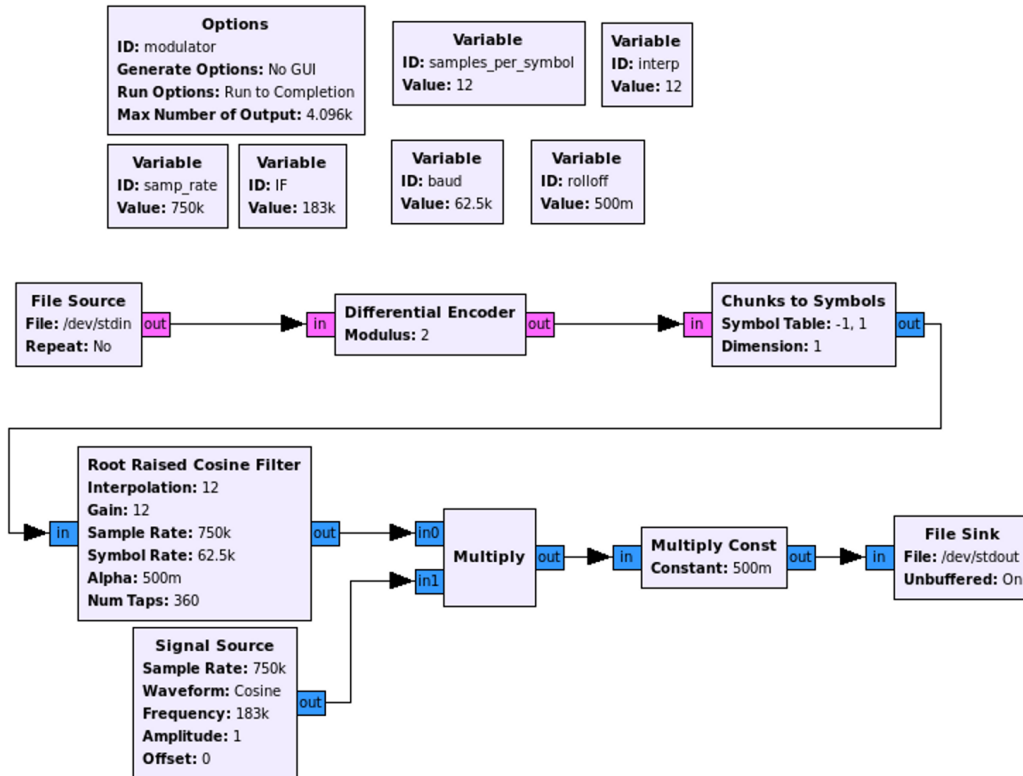


Figure 5.6: GNU Radio modulator flow graph.

simplification comes at the cost of doubling the bit error rate (BER) because every bit error now causes the next bit to also be incorrectly detected.

The bits are then mapped to symbols on the constellation diagram, in this case to $-1 + 0j$ and $1 + 0j$.

5.2.2.3 Pulse Shaping Filter

These symbols are then root raised cosine filtered. This limits the bandwidth of the signal which, when combined with another matching root raised cosine filter on the receiving end, will minimise intersymbol interference. This filter is also used for interpolation. The output of the filter is at the DAC sample rate and the input is at the symbol rate. An integer interpolation factor is used for convenience, since it simplifies the filter design.

5.2.2.4 Mixing

The signal is then mixed slightly away from 0 Hz to avoid being affected by the high-pass filtering required to remove the DC offset from the DAC. This

also helps to avoid phase noise from the oscillators used in the quadrature modulator.

5.2.2.5 Other Considerations

- As a final step the signal is multiplied by a constant to ensure that its amplitude stays within the range -1 to 1.
- GNU Radio possesses the ability to limit the maximum output of every block in a flow graph thus limiting the total flow graph latency. This lowers the latency of the transmitter.

GNU Radio does include a generic modulator block capable of producing differential BPSK. It was not used because it was less optimised for performance and because it makes the system more difficult to debug.

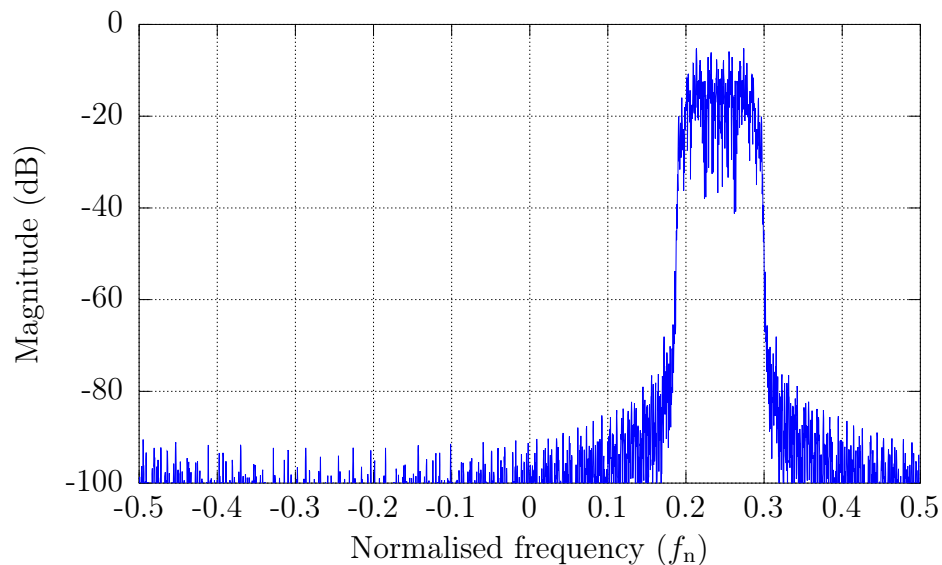


Figure 5.7: The FFT of the software modulator output with random input data.

5.2.3 Demodulation Flow Graph

The demodulator extracts bits from I/Q samples. This is heavily derived from the GNU Radio generic demodulator block.

Figure 5.8 shows the demodulator flow graph.

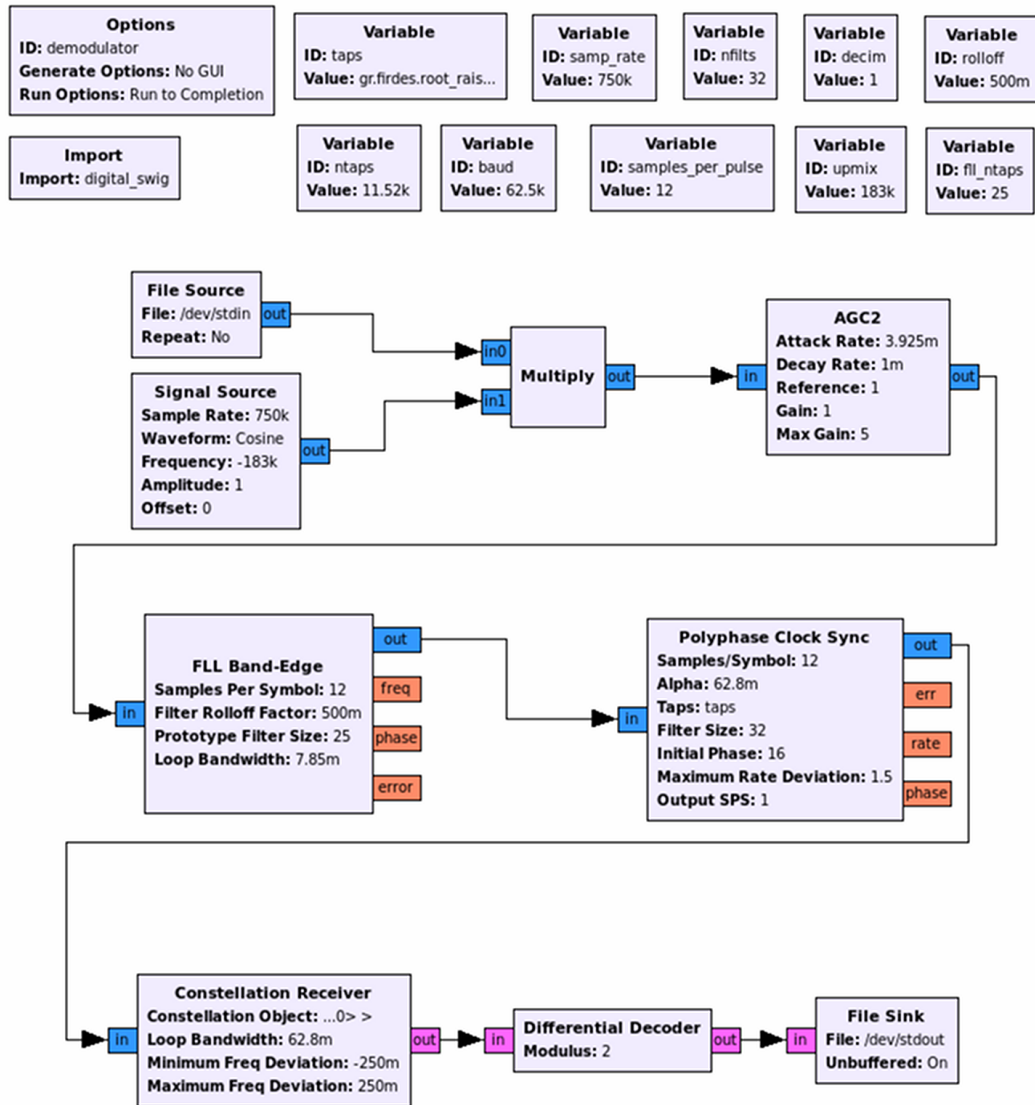


Figure 5.8: GNU Radio demodulator flow graph.

5.2.3.1 Interface

The same STDIO based interface is used as described in subsection 5.2.2.1.

5.2.3.2 Mixing

The mixing away from 0 Hz performed in Section 5.2.2.4 has to be reversed. This is done by mixing with a complex carrier of the same frequency but opposite sign to the carrier that was used in the original mixing.

5.2.3.3 Frequency Compensation

A frequency locked loop (FLL) compensates for local oscillator (LO) mismatches between the transmitter and the receiver. This FLL allows for coarse correction of frequency mismatches. The FLL operates by using a band edge filter on each of the two edges of the signal bandwidth. The outputs of these filters are then compared and used as an error signal in a control loop. This error drives a numerically controlled oscillator that is multiplied by the input signal to mix it to 0 Hz [13; 14].

5.2.3.4 Clock Synchronisation

The phase offset in the signal now needs to be corrected so that the symbol can be sampled at the optimum point for demodulation. This is done by making use of two polyphase filter banks, one containing the matched filter at different phases and the other containing the derivatives of the filters in the first filter bank. The derivative filter bank can be used to estimate the phase error of the incoming signal. A polyphase filterbank can be used as a phase shifter by changing the filter in the bank to which the next sample should go. It can also be used as a rational factor resampler. Combining this with the phase error estimate, a control loop can be created that corrects the phase of the incoming signal while simultaneously filtering with the matched filter and decimating the signal to give a one sample per symbol output. This control loop will also correct small differences between the sampling rate of the transmitting DAC and the receiving ADC. The phase corrected here is the phase difference between the transmitter and the receiver symbol clocks, not the phase represented by the angle of the complex signal [15; 16].

5.2.3.5 Symbol Recovery

The symbol is then extracted from the sample generated by the polyphase filter bank. This is done by making use of a Costas loop to remove any phase error and then checking whether the real component is smaller or greater than 0. The Costas loop determines the phase error by comparing the received symbol with its constellation diagram [17; 18].

5.3 Integration

This section shows how the components of the physical layer are connected. Figure 5.9 gives an overview of this.

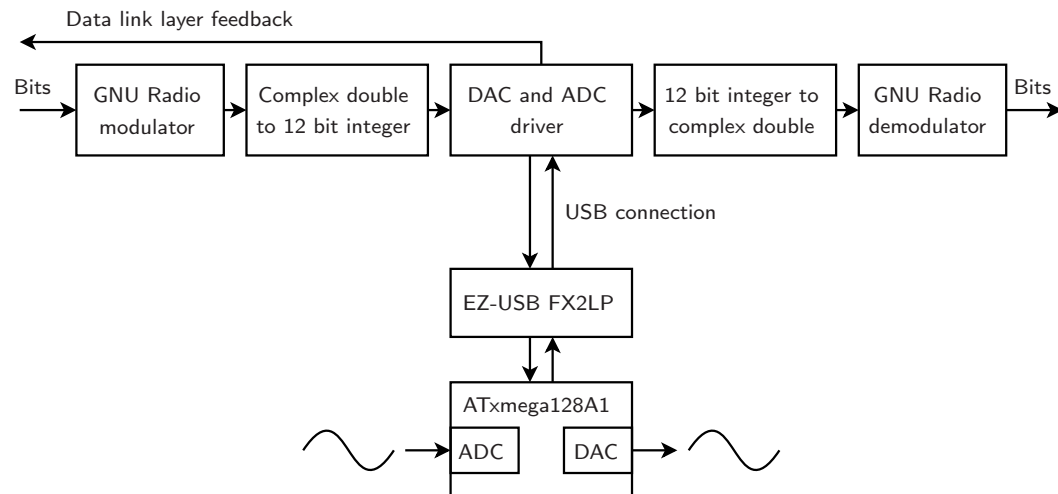


Figure 5.9: Physical layer overview.

5.3.1 Format Conversion

A format conversion was required between the GNU Radio output format, complex floating point, and the format used by the DAC and ADC, a pair of 12 bit integers stored as 16 bit integers. Two short C programs with `SDTIO` inputs and outputs do these conversions. These programs are also responsible for adding and interpreting the synchronisation information described in Section 5.1.4.3.

5.3.2 Connection of Components

The software components all communicate using Linux pipes. A command to execute the physical layer takes the following form:

```
<data link layer> | ./modulator.py | ./gnurtotwelvebit | \  
sudo ./dac_adc_pc | \  
./twelvebittognur | ./demodulator.py | <data link layer>
```

5.3.3 Physical Layer Operation

Figure 5.10 shows random, modulated output on the DAC. Figure 5.11 shows the FFT of this output recombined into a complex signal. This is more revealing than individually computing the FFTs of each channel, since the FFT of the complex signal shows the spectrum that is expected around the carrier frequency after quadrature modulation. This is taken from the output of the DAC without any low-pass filtering or DC blocking.

Figure 5.12 shows the modulated data captured by the receiving ADC and Figure 5.13 its FFT.

5.4 Summary

The steps taken to create the different components of the physical layer and their respective interactions have been discussed. The chosen hardware was seen to have some performance issues that were resolved by applying various code optimisations. The GNU Radio libraries and framework were used to create a modulator and demodulator and interfaced with the driver for the DAC and ADC.

In the next chapter testing and verification of the operation of the physical layer, is done.

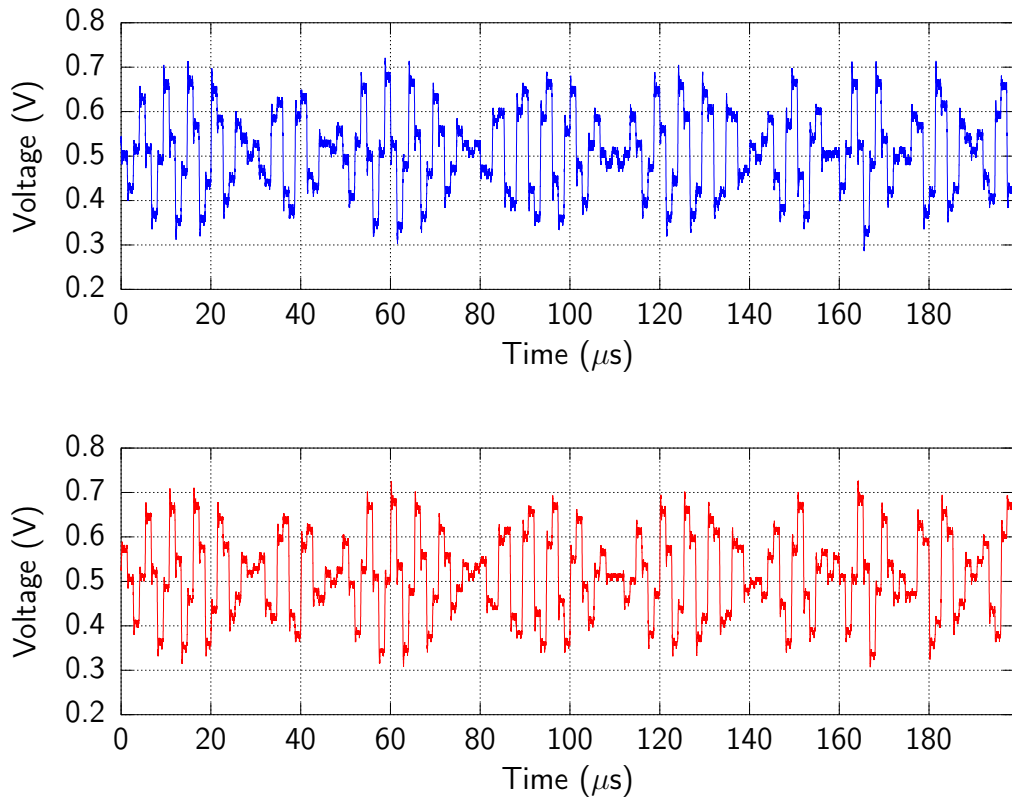


Figure 5.10: *I* and *Q* DAC output.

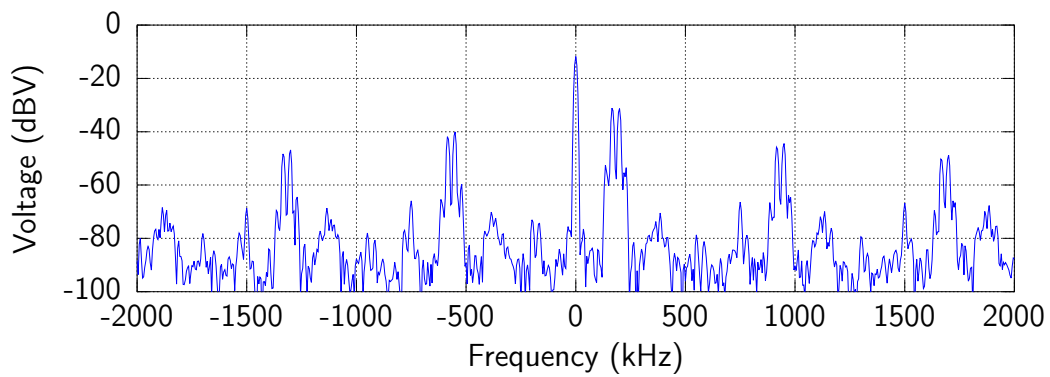


Figure 5.11: FFT of DAC output as complex signal.

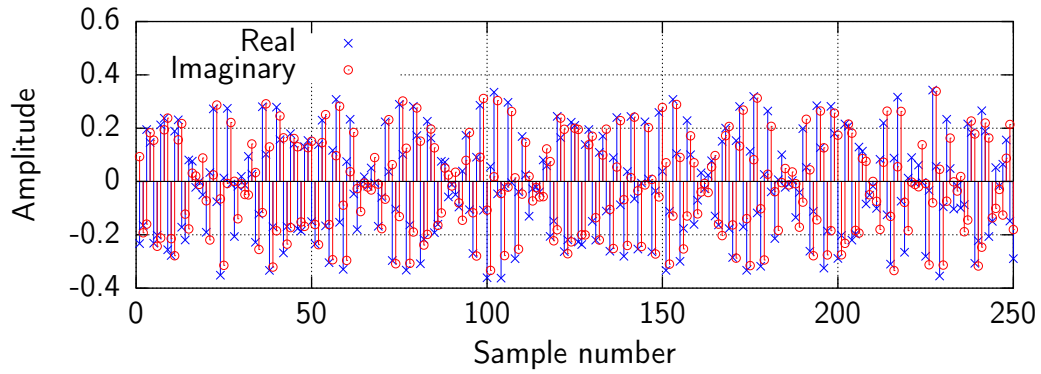


Figure 5.12: Output of ADC.

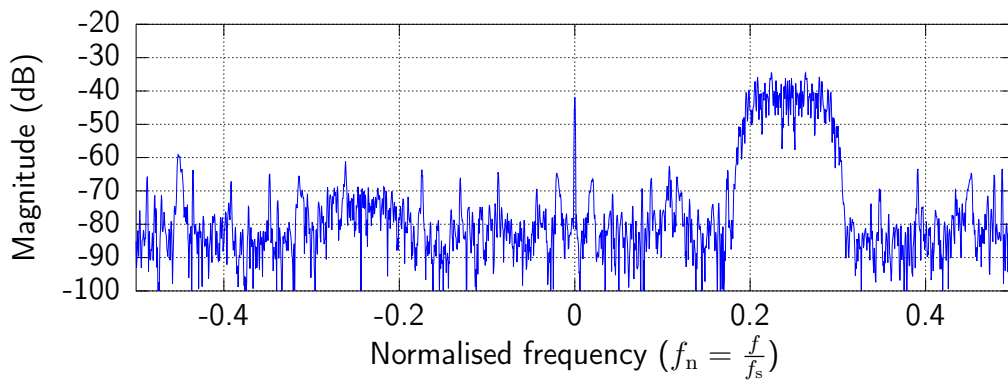


Figure 5.13: FFT of ADC output.

Chapter 6

Testing of Physical Layer

In this chapter the correctness and performance of the physical layer is tested by inspecting the outputs of its individual components and by evaluating its performance under different circumstances.

6.1 GNU Radio Modulation Tests

Here the software modulator and demodulator are tested in several ways:

- By investigating output samples at different points in the modulation and demodulation process.
- By investigating the time taken to achieve a lock on an incoming signal.
- By testing the bit error rate.

6.1.1 Test Different Stages of Modulation and Compare to Theory

This test inputs a stream of random data to the modulator then feeds the output of the modulator to the demodulator. Data from various locations in the modulator and demodulator chain is sampled. Figure 6.1 and Figure 6.2 show the modulator and demodulator flow graph with the sample locations marked with numbered circles. The phase of all plots was adjusted in order to show the same set of symbols in different parts of the modulation and demodulation chain.

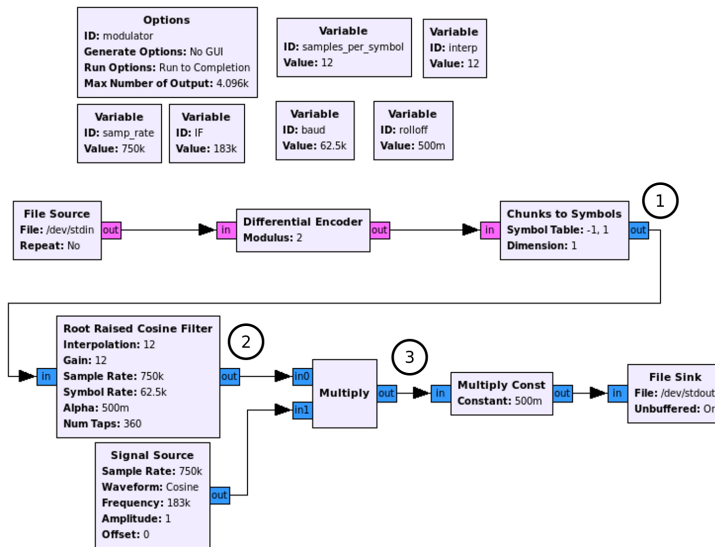


Figure 6.1: GNU Radio modulator flow graph with test sample points marked.

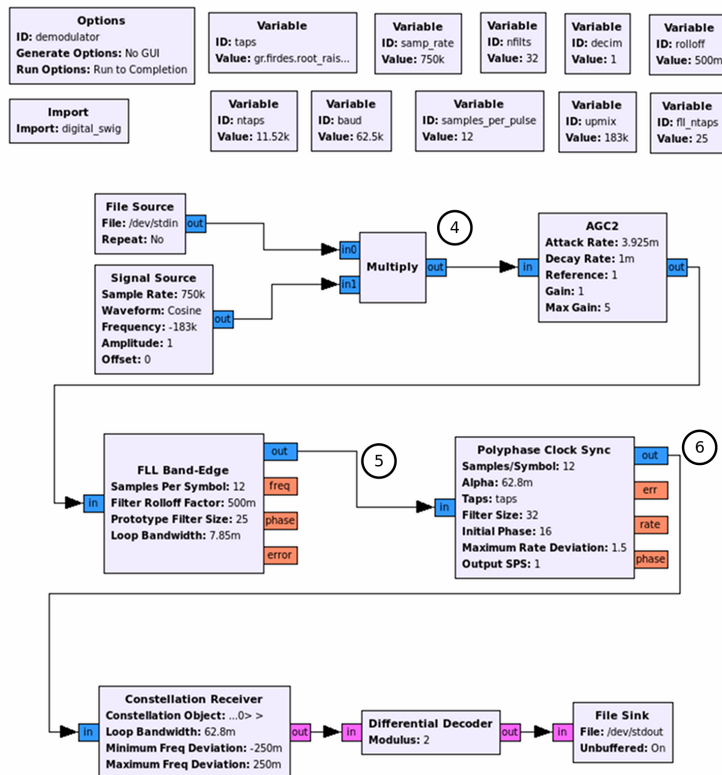


Figure 6.2: GNU Radio demodulator flow graph with test sample points marked.

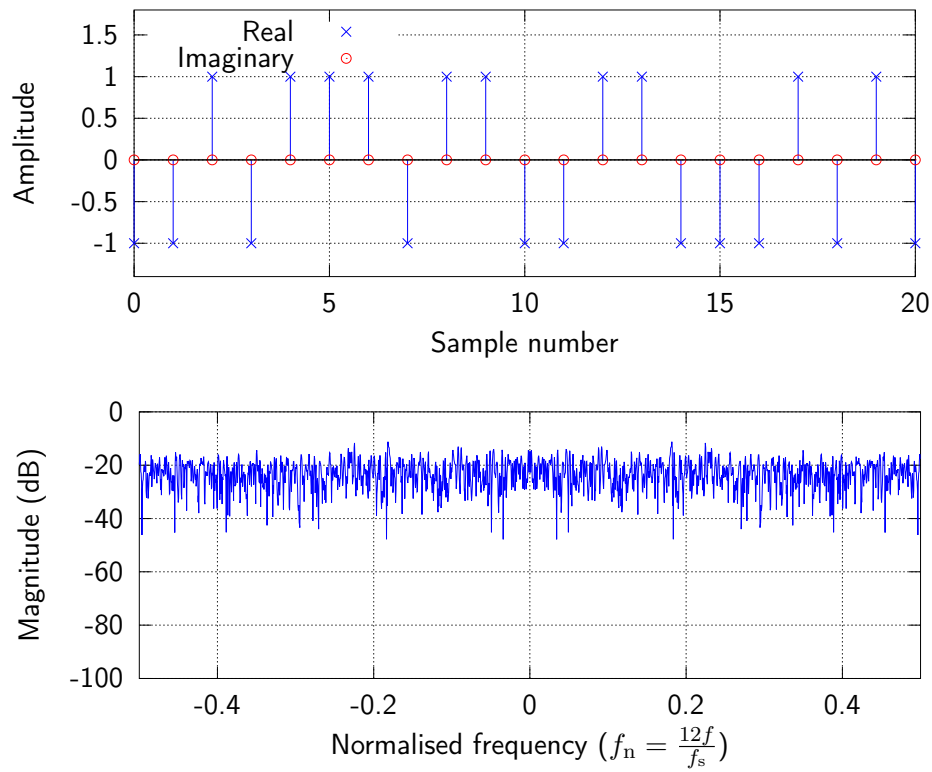


Figure 6.3: Time and frequency domain of differentially encoded symbols. (Sample point 1)

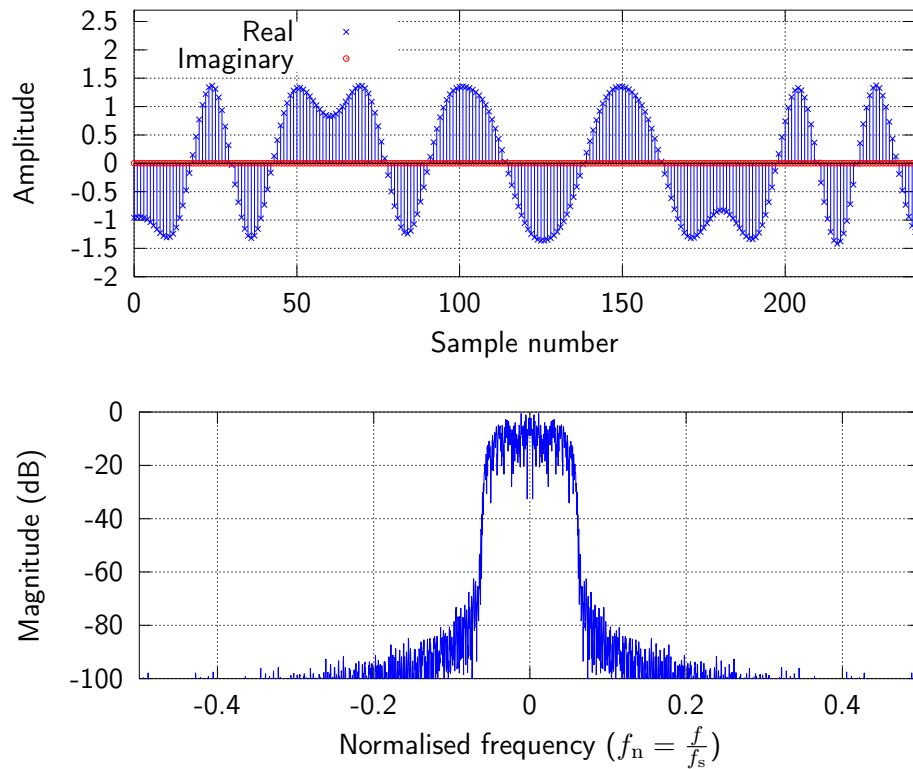


Figure 6.4: Time and frequency domain of root raised cosine filtered symbols. (Sample point 2)

The binary input is 0111 0011 0101 0100 1111. Figure 6.3 shows the output after the symbols have been differentially encoded and converted from 1's and 0's to 1 and -1 . The point sampled is marked as '1' on Figure 6.1. The input bits can still be easily read from this plot. The frequency domain shows the signal spread over all frequencies. This is expected, since no filtering has been performed at this stage.

Figure 6.4 shows the symbols after they have been root raised cosine filtered. The point sampled is marked as '2' on Figure 6.1. It is still possible to make out the binary values at this stage and the relationship to Figure 6.3 can be seen. The filter is also used for interpolation and this can be seen in the increase in the number of samples. The frequency domain now shows the signal with limited bandwidth.

Figure 6.5 shows the signal after it has been mixed away from 0 Hz. The point sampled is marked as '3' on Figure 6.1. It is now much harder to see the original signal in the plot. The plot now shows an imaginary component for the first time, since the frequency components are no longer symmetrical around 0 Hz. The frequency domain plot shows that the signal has moved away from 0 Hz.

Figure 6.6 shows the signal after it has been mixed back down to approximately 0 Hz. The reason for not mixing down to exactly 0 Hz is to emulate a realistic environment and show the proper working of the demodulator when mixing frequencies do not match exactly. The point sampled is marked as '4' on Figure 6.2. Some components of the signal in Figure 6.4 can be seen, but overall the bit values are not readily apparent. In the frequency domain the small offset from 0 Hz is visible.

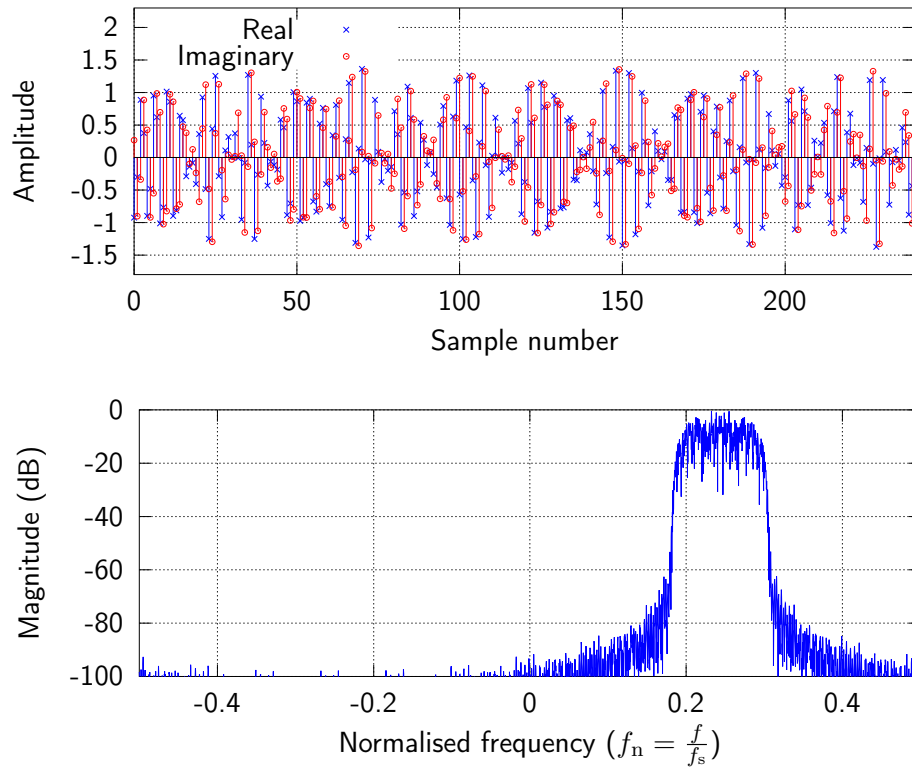


Figure 6.5: Time and frequency domain of symbols after mixing away from 0 Hz. (Sample point 3)

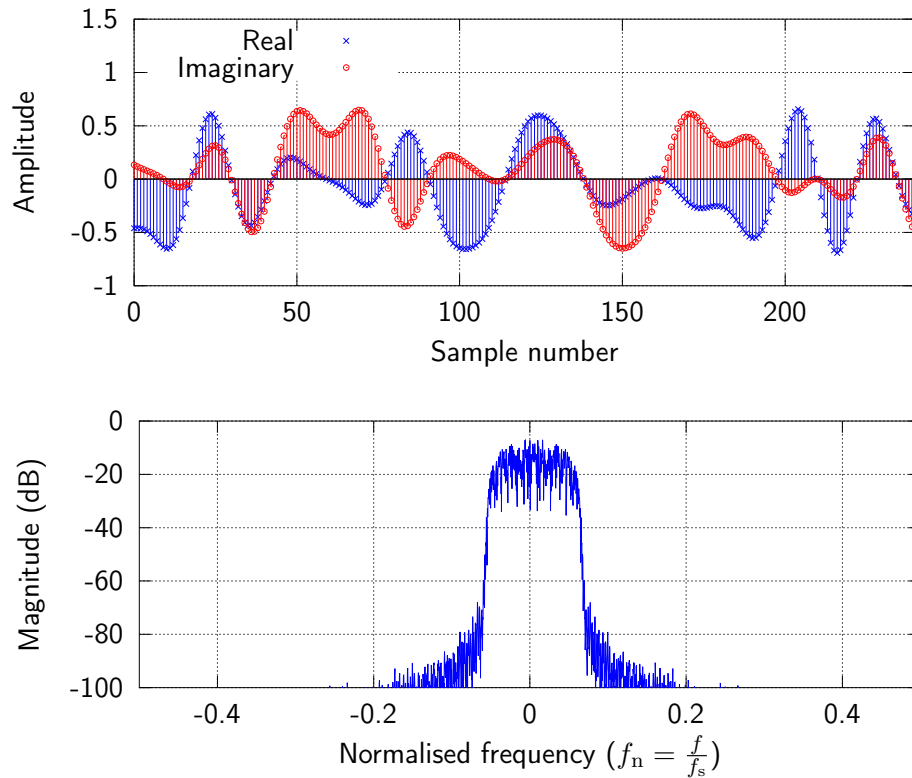


Figure 6.6: Time and frequency domain of symbols after mixing to approximately 0 Hz. (Sample point 4)

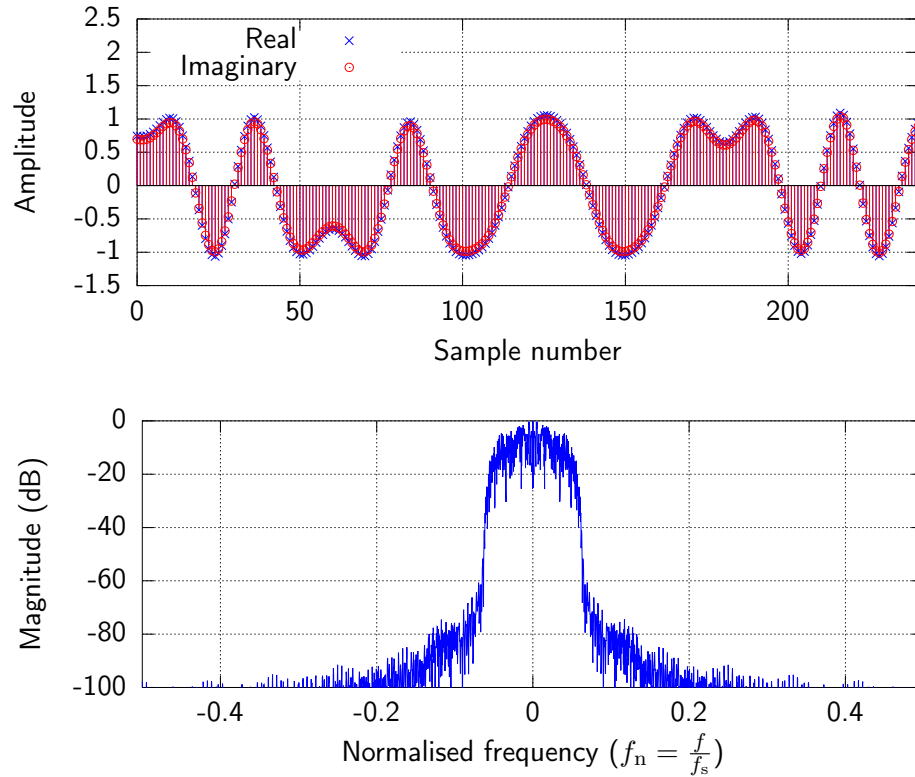


Figure 6.7: Time and frequency domain of symbols after frequency locked loop. (Sample point 5)

Figure 6.7 shows the signal after it has been through the FLL. The point sampled is marked as ‘6’ on Figure 6.2. Coarse correction of the frequency offset has been done and this can be seen in both the frequency and the time domains. The original signal in Figure 6.4 can now easily be seen in the time domain plot.

Figure 6.8 shows the signal after the polyphase clock synchroniser. The point sampled is marked as ‘6’ on Figure 6.2. The signal has been decimated to 1 sample per symbol with the samples being at the optimal sampling point for data recovery. It can clearly be seen that this is the same signal as in Figure 6.3. There is still a complex component at this point and the signal appears to be the inverted compared to the original. This means that there is still some phase offset. This offset will be corrected by the combination of the constellation receiver and differential decoding. As in the original signal in Figure 6.3, the frequency domain shows that the signal is spread over the entire spectrum.

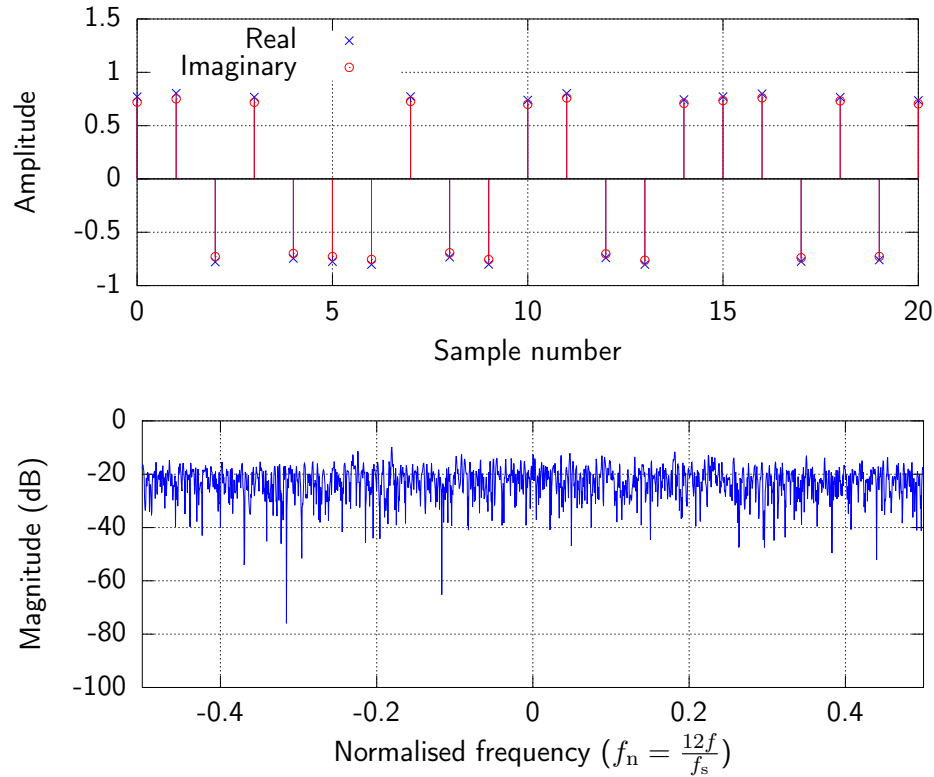


Figure 6.8: Time and frequency domain of symbols after clock synchronisation. (Sample point 6)

The final output in binary produces no interesting plots. The output is 0111 0011 0101 0100 1111. This matches the input.

We have now shown that the modulator and demodulator flow graphs produce sensible output at their intermediate stages. We have also shown that the input data was received at the output.

6.1.2 Test Delay of Demodulator Lock

This tests how long it takes the demodulator to lock on to the frequency and phase of an incoming signal. This is especially useful if the system is to be used without transmitting random data at all times to maintain demodulator lock. It also shows the maximum input frequency offset that the system can reasonably tolerate.

Figure 6.9 shows the time taken by the frequency locked loop to attain an output frequency error of 0 Hz. Since the input frequency error is artificially induced the output error can easily be calculated when compared to the internal

frequency of the FLL. The zero'th sample is referenced to the first symbol output by the FLL. This test shows that under a normalised frequency offset of 0.1 or less the FLL will acquire lock within under 15 000 samples or 125 symbols. Frequency offsets of over 0.15 will take longer to acquire lock. At 0.2 the FLL was not observed to acquire lock within 80×10^6 bits. This test was under optimal conditions with no noise added to the system. These lock times can be decreased by increasing the FLL loop bandwidth. This comes at the cost of adding noise to the system.

The polyphase clock synchroniser was not observed to contribute any significant delay to synchronisation with an incoming signal.

The constellation receiver could also contribute some delay, but there is no straightforward way of measuring this without modifying the block. Since the constellation receiver has the same loop bandwidth as the polyphase clock sync, it is not believed to contribute much delay.

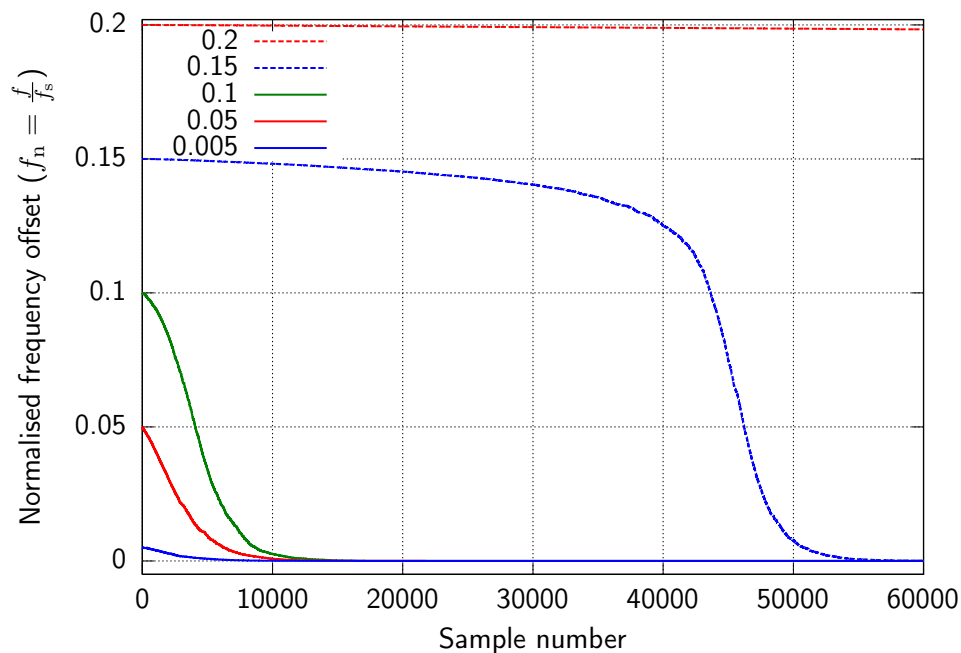


Figure 6.9: Number of samples output versus frequency offset of frequency locked loop for different input frequency offsets.

6.1.3 Modulate and Demodulate Random Data

This test uses the same configuration as described in Section 6.1.1, but with a larger amount of input data. 32 MiB of randomly generated data was given as input to the modulator.

No errors were observed and the input data was recovered.

6.1.4 Modulate and Demodulate Random Data in Presence of Noise

Here the performance of the demodulator is tested in the presence of noise. Gaussian noise of different amplitudes is added to the output of the modulator and the resulting signal is demodulated. The bit error rate is then compared to the noise level. The noise level is measured in $\frac{E_b}{N_0}$, the energy per bit to noise power spectral density ratio. The theoretical bit error rate is calculated, as well as the bit error rate when the signal is demodulated under ideal circumstances with perfect bit synchronisation. 4 MiB of random data was used for this test. This gives just over $3,355 \times 10^7$ test bits.

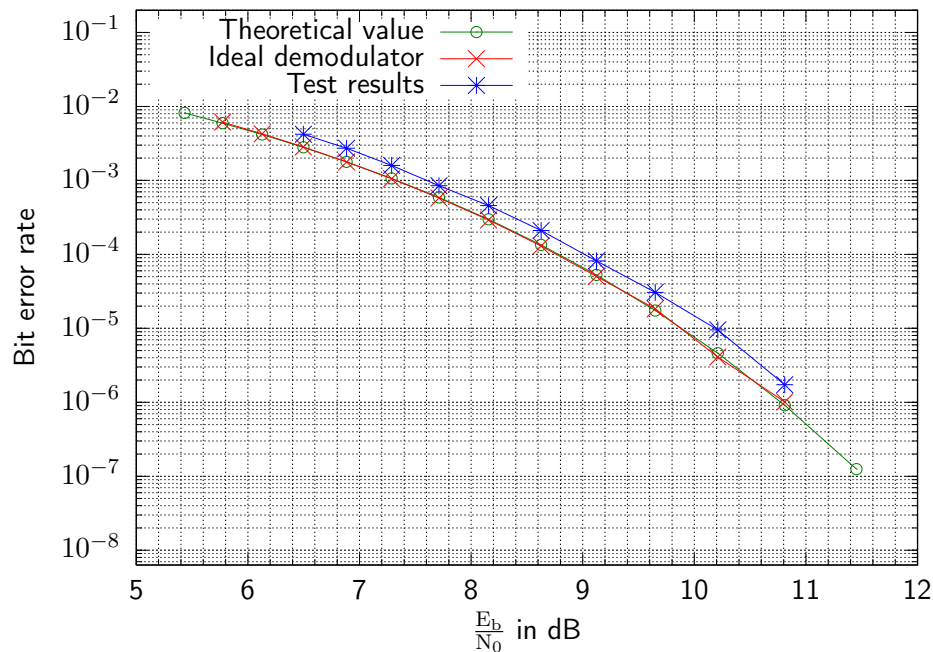


Figure 6.10: Bit error rate versus $\frac{E_b}{N_0}$ for demodulator. Theoretical values and ideal coherently demodulated values included for reference.

Figure 6.10 shows the bit error rate of the demodulator as well as the theoretical bit error rate and the bit error rate of the synchronised coherent demodulator. The demodulator in this system performs slightly worse than the theoretical value. The difference is about 0,2 to 0,4 dB.

6.2 DAC and ADC Tests

These tests try to show that the DACs and ADCs are working correctly. The general method used is to use some known signal to either output to the DAC or sample with the ADC and then to compare the result with the expected result.

The sample rate, f_s , used for both the DAC and the ADC is 750 kHz.

6.2.1 Output Test Waveforms to DAC

Different signals are used as input for the DACs to convert to analogue. The output is then inspected for abnormalities. In all cases the output is measured at the output of the DAC and before being low-pass filtered.

6.2.1.1 Square Wave

This test outputs a 75 kHz ($\frac{f_s}{10}$) square wave on one of the DACs. This is a simple shape and shows that there are no major problems with the DAC. Figure 6.11 shows the output of the DAC and Figure 6.12 shows the FFT of this output. These results are as expected.

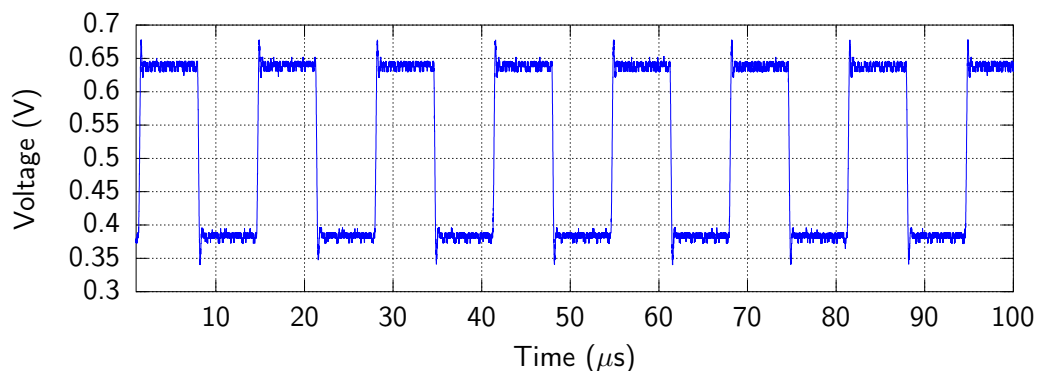


Figure 6.11: Output of DAC if given an 75 kHz square wave as input.

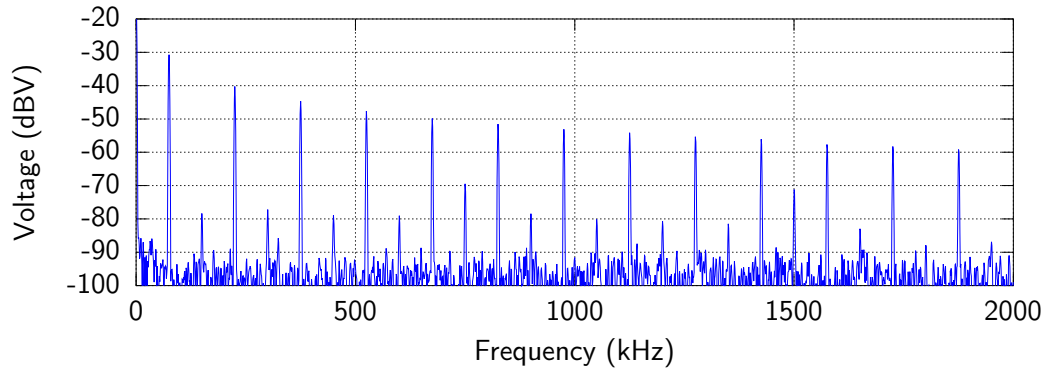


Figure 6.12: FFT of DAC output if given an 75 kHz square wave as input.

6.2.1.2 Sine Wave

This test uses a 75 kHz sine wave as input to the DAC. Figure 6.13 shows the output of the DAC and Figure 6.14 shows the FFT of the output. This test shows that the DAC can output more complex signals. The results show no abnormalities.

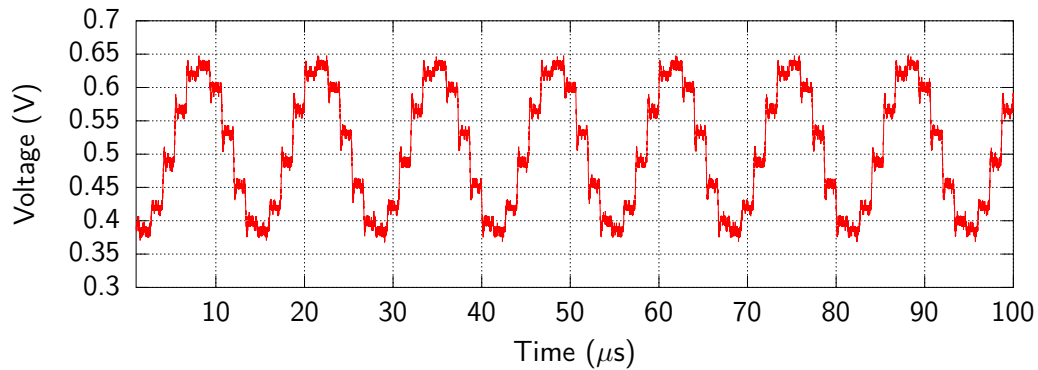


Figure 6.13: Output of DAC if given a 75 kHz sine wave as input.

6.2.1.3 Random Modulated Data

This test uses the software modulator to modulate random data and uses the resulting signal as input for both DACs. This tests the general use case for the DACs. Figure 6.15 shows the output of the DAC given the I signal as

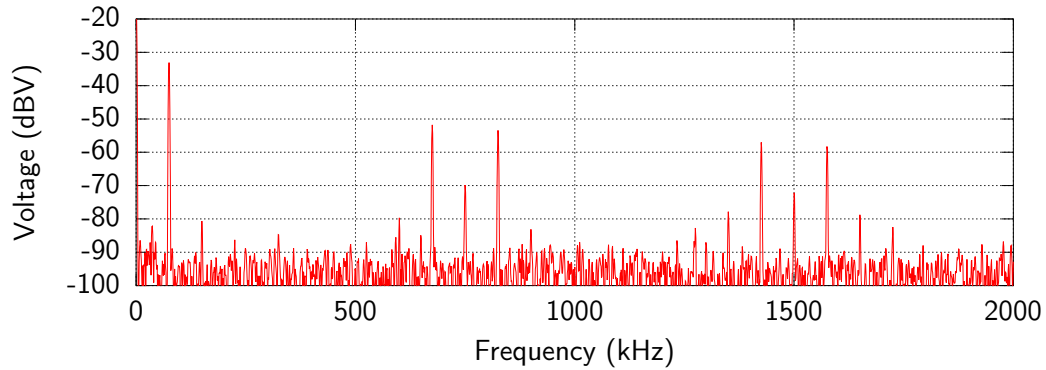


Figure 6.14: FFT of DAC output if given an 75 kHz sine wave as input.

input. Figure 6.16 shows the output of the DAC given the Q signal as input. Figure 6.17 shows the FFT of the output of the DAC outputting the I signal. The FFT of the Q signal has been omitted because it is practically identical to the FFT of the I signal. Figure 6.18 shows the FFT of the output of both DACs if the signal is to be combined into the complex baseband signal. This shows the spectrum that is expected at the output of a quadrature modulator if it were given the I and Q signals as input.

The output signals are as expected and do not show any significant distortion.

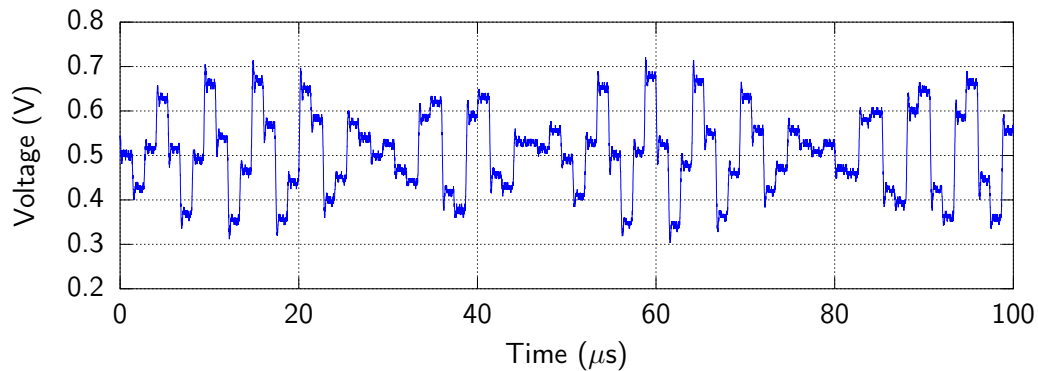


Figure 6.15: Output of I DAC if given modulated random data as input.

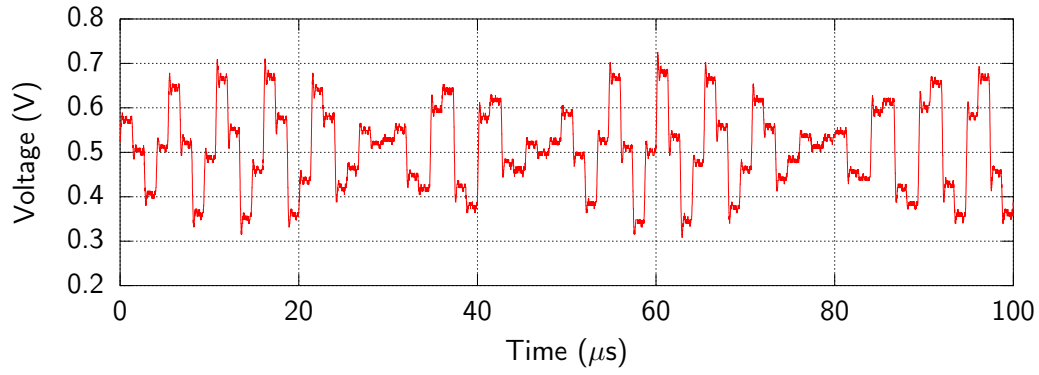


Figure 6.16: Output of Q DAC if given modulated random data as input.

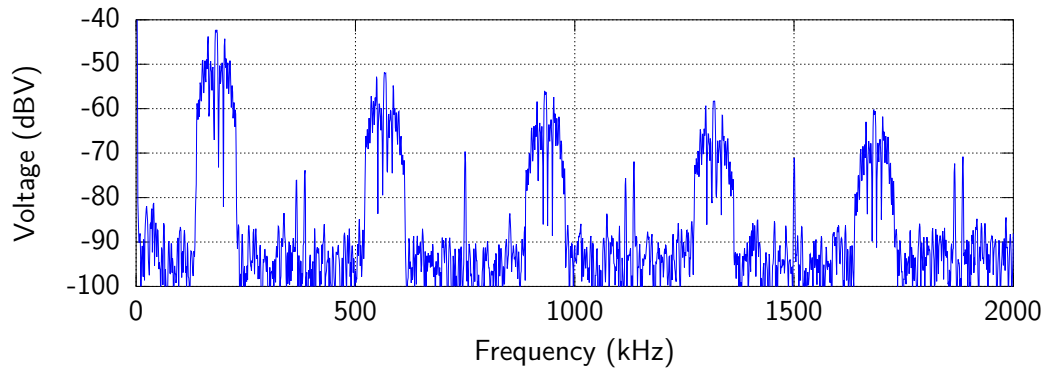


Figure 6.17: FFT of I DAC with modulated random data input.

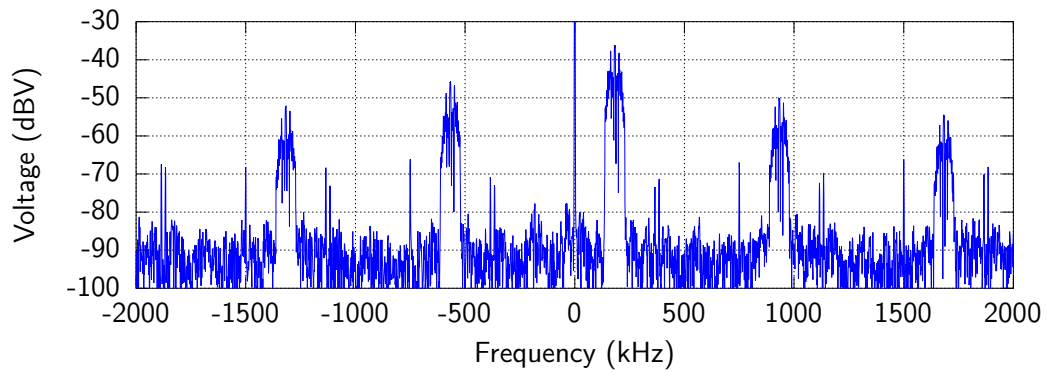


Figure 6.18: FFT of DAC outputs combined into complex baseband.

6.2.2 Sample Test Waveforms with ADC

The basic functionality of the ADC is tested by sampling different signals and comparing the results to the expected results. As with the DAC tests the sampling rate, f_s , is 750 kHz.

6.2.2.1 Square Wave

This test uses a 75 kHz ($\frac{f_s}{10}$), 400 mV_{p-p} square wave as input for the ADC. Aliasing is expected because the bandwidth of the signal extends well beyond $\frac{f_s}{2}$.

Figure 6.19 shows the sampled square wave and Figure 6.20 shows the FFT of these samples. The time domain shows a clear square wave. The frequency domain shows a clear peak at $f_n = 0.1$ and the first harmonic at 0.3. Because of the particular frequency of the signal, aliasing is difficult to see in this case.

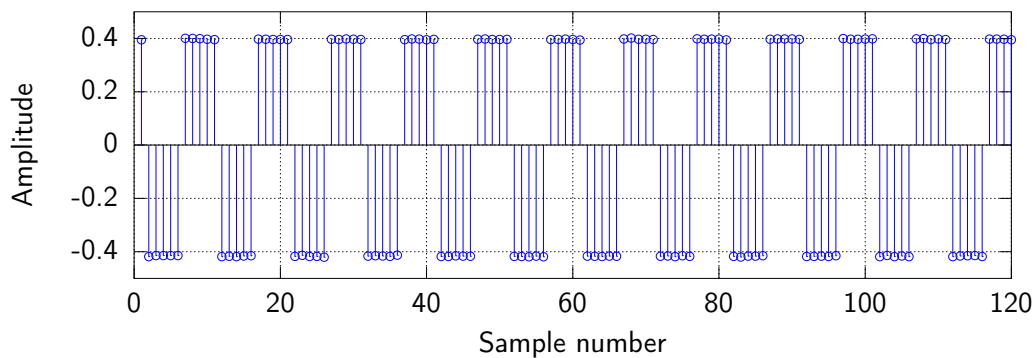


Figure 6.19: Output of ADC sampling a square wave.

6.2.2.2 Sine Wave

This test uses an 75 kHz, 400 mV_{p-p} sine wave as input. The sampled signal is not expected to show any great amount of distortion.

Figure 6.21 shows the sampled sine wave and Figure 6.22 shows its FFT. The frequency domain shows a clear peak at $f_n = 0.1$. The smaller peaks at 0.2 and onwards were also visible on an FFT done by an oscilloscope and are assumed to be imperfections in the signal and not the ADC. The sine wave was sampled sufficiently well.

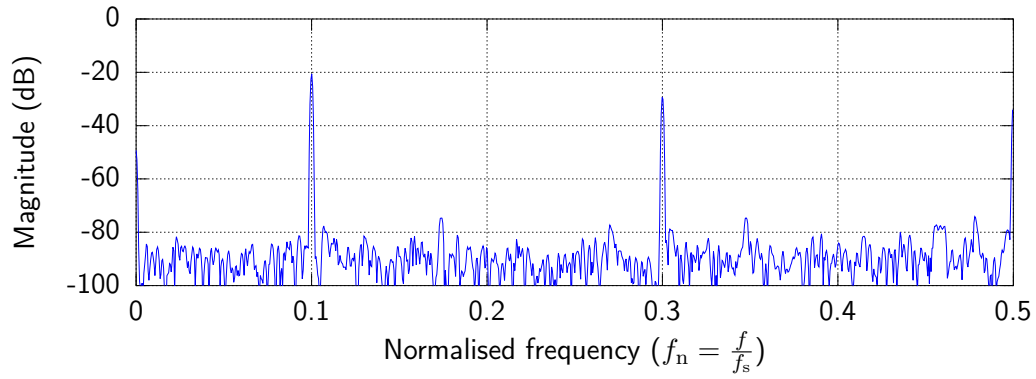


Figure 6.20: FFT of output of ADC sampling a square wave.

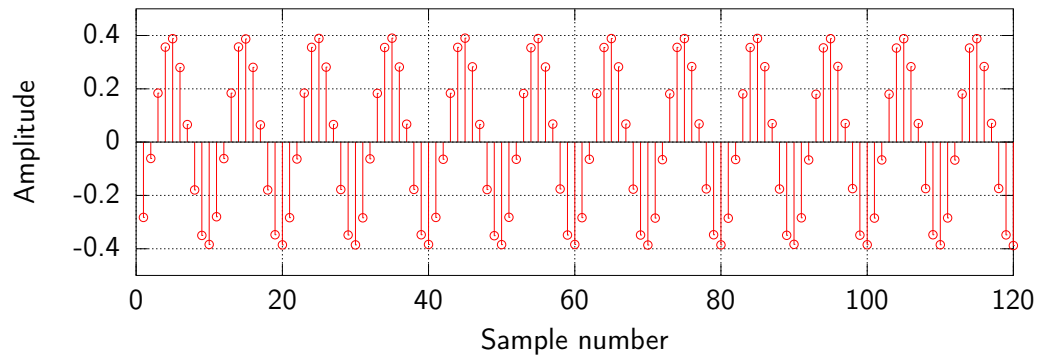


Figure 6.21: Output of ADC sampling a sine wave.

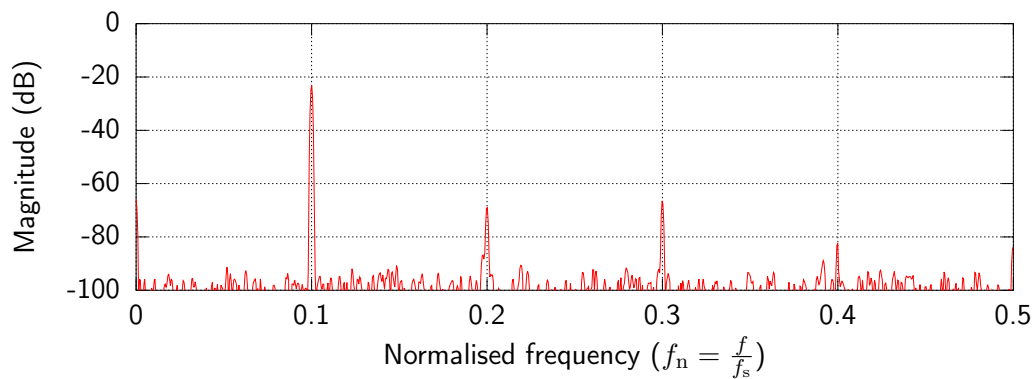


Figure 6.22: FFT of output of ADC sampling a sine wave.

6.2.2.3 Random Modulated Data

This test uses modulated random data produced by one of the DAC pairs as input to the other ADC pair. Since no filtering is applied to the output of the DAC, some aliasing is expected.

Figure 6.23 shows the I and Q sampled signals and Figure 6.24 shows the FFT of this complex signal. The signal is clearly visible. It is expected that the noise floor will lower if proper anti-aliasing filters are used. A second smaller signal is visible at the negative of the frequency of the desired signal. It is suspected that this is either caused by a slight difference between the amplitudes of the I and Q channels or by a slight phase difference between the same channels. The magnitude of this second signal was observed to occasionally vary with time.

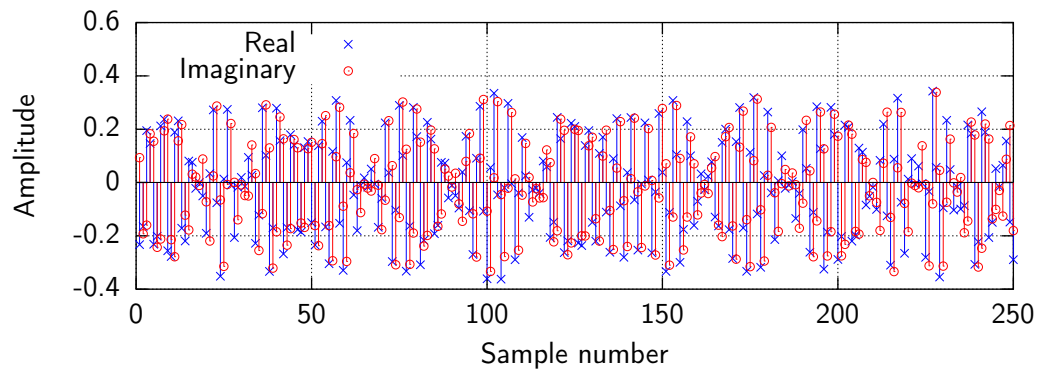


Figure 6.23: Output of ADC sampling modulated random data.

6.3 Tests of Complete Physical Layer

The assembled physical layer is now tested by measuring the bit error rate and frame error rate. Since the design of the radio frequency (RF) components was not part of this project these test are done both with and without the RF components.

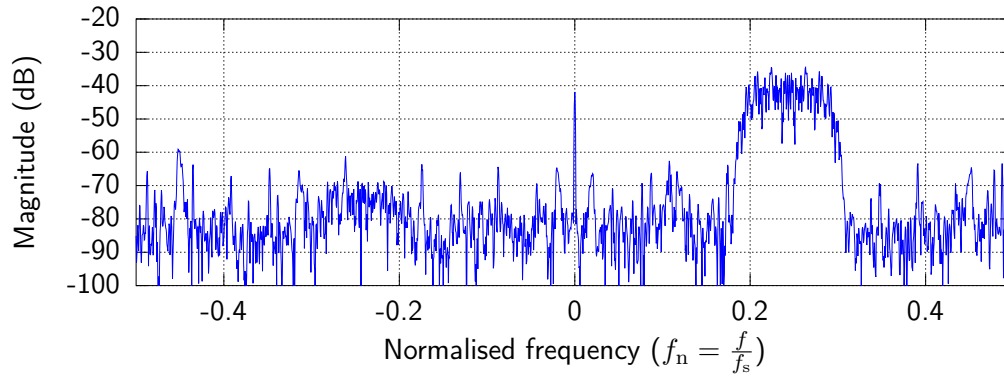


Figure 6.24: FFT of output of ADC sampling random modulated data.

6.3.1 Transmit and Receive Random Data

Here the modulator, demodulator and DAC/ADC pairs are tested. The bit error rate as well as the frame error rate is tested. Appendix A shows the method used to do these tests.

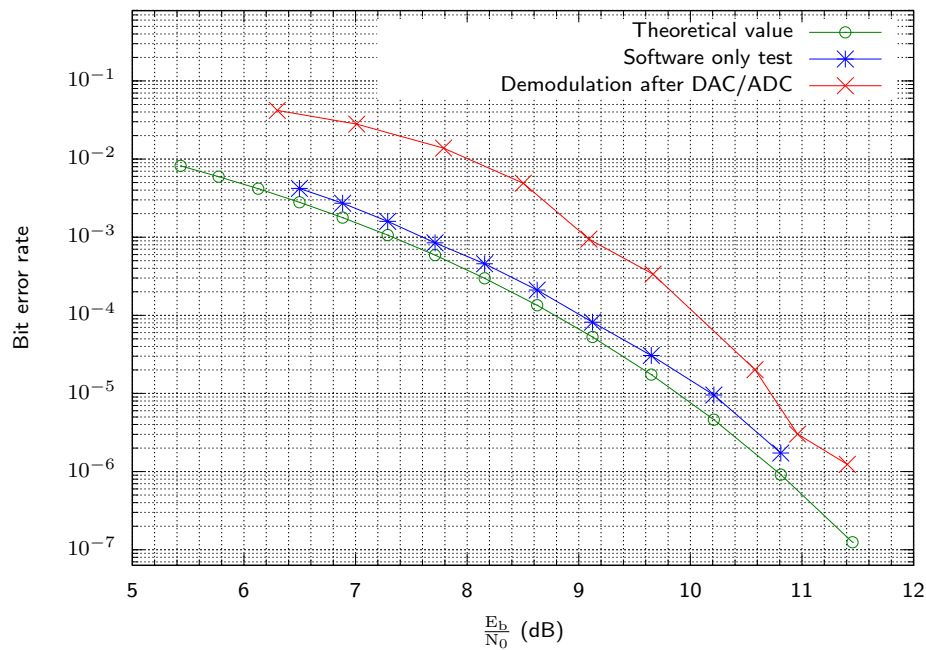


Figure 6.25: Bit error rate of system using DAC connected directly to ADC.

Figure 6.25 shows the bit error rate of the physical layer with the DAC connected directly to the ADC.

The bit error rate is worse than the theoretical bit error rate by between 0.2 dB and 2.5 dB, with the performance gap increasing as $\frac{E_b}{N_0}$ decreases. The performance was expected to be worse than that of the software only demodulation test, since the software test assumes that white Gaussian noise is the only source of signal corruption. This assumption is no longer true, with corruption being added by the DAC process as well as other imperfections in the system, for example slight mismatches between the amplitudes of different DACs and clock jitter.

Measuring bit error rate gives a good indication of the performance of the system, but does not deal well with bit errors caused by errors in the clock recovery process. These errors cause the number of bits output by the demodulator to change, occasionally adding or dropping a bit. This can lead to significant burst errors and because of this it also useful to measure the frame error rate.

Figure 6.26 shows the frame error rate of the system at different $\frac{E_b}{N_0}$ values. The error rates for frames in cases where up to 50 and up to 100 bit errors can be corrected are also shown. The theoretical frame error rates at those values are also given for the non error correcting case. Figure 6.27 shows the theoretical predictions for frame error rates with 0, 50 and 100 bits correctable. Appendix B shows how the theoretical values were calculated.

As can be seen from Figures 6.26 and 6.27, the difference between the theoretical frame error rate and the measured frame error rate is quite large. This difference gives some insight into the timing errors that occur in the system. Timing errors become apparent at $\frac{E_b}{N_0}$ values of under approximately 9.1 dB since no frames with over 50 or 100 errors were predicted to be seen at these levels and yet a fair number of them were observed.

From these results we can also conclude that the system's performance degrades significantly at $\frac{E_b}{N_0}$ values under 9 dB, even with error correction. This is due to non additive white Gaussian noise (AWGN) sources of error.

6.3.2 Transmit and Receive Random Data with RF Hardware

This repeats the test in Section 6.3.1 but with the RF chain also used. This gives a more realistic measure of performance than the DACs directly feeding

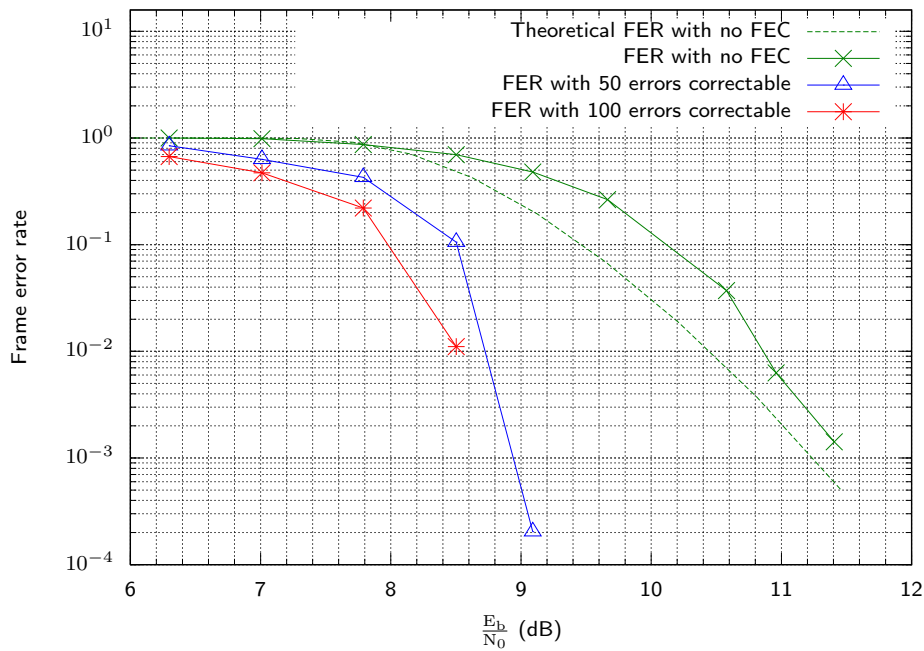


Figure 6.26: Frame error rate of system using DAC connected directly to ADC.

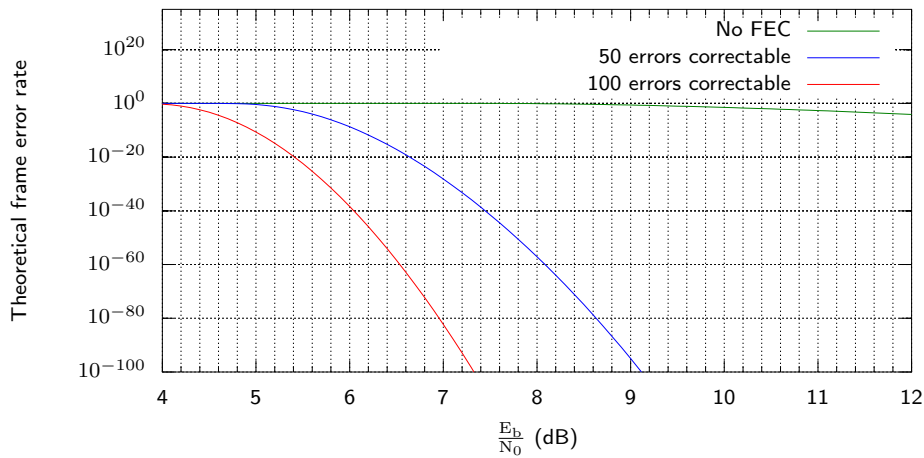


Figure 6.27: Theoretical frame error rates.

into the ADCs. Unlike the previous test, the $\frac{E_b}{N_0}$ value was varied by making use of attenuators instead of a noise generator. The RF components were not designed in this project, but they are nevertheless required for a complete system and thus knowledge of the system's performance with them connected is useful.

Figure 6.28 shows the bit error rate and Figure 6.29 shows the frame error

rate.

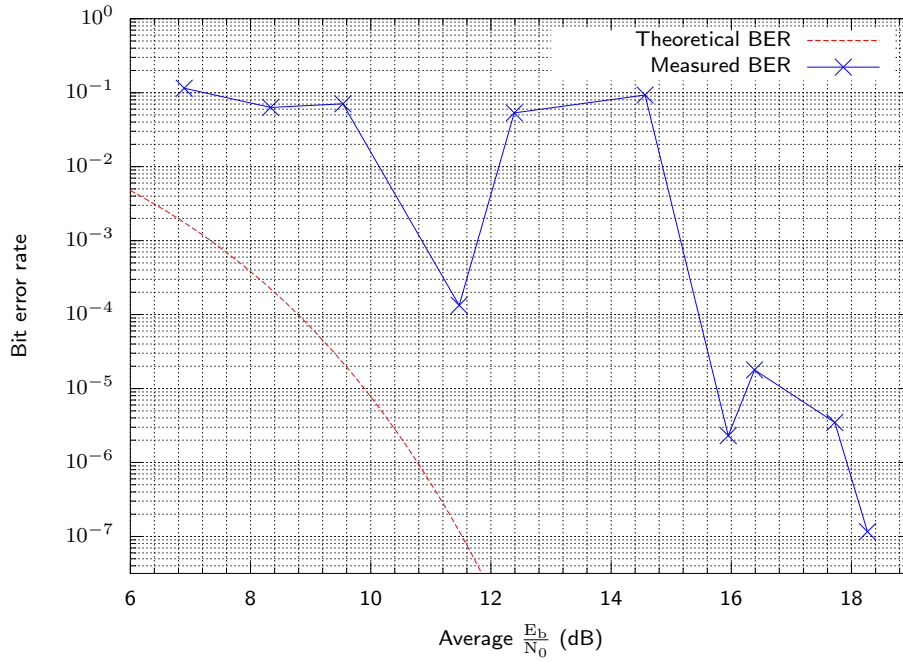


Figure 6.28: Bit error rate of system using RF transmission and reception.

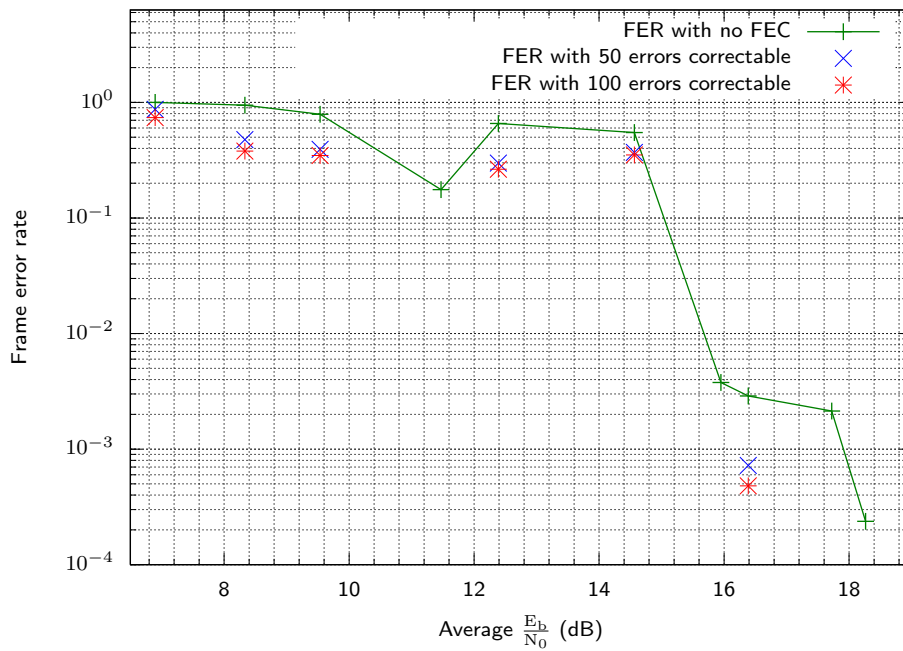


Figure 6.29: Frame error rate of system using RF transmission and reception.

The $\frac{E_b}{N_0}$ values varied significantly during testing and the values given in these two figures is an average. Unlike the previous results, the bit error rates also varied significantly during testing, especially around large jumps in the bit error rate, for example between $\frac{E_b}{N_0}$ values of 14.4 dB and 16 dB. It is expected that the results could vary quite a lot if the tests were run again. They are, nevertheless, useful for making general observations about the system.

The system performs approximately 7 dB worse than without the RF path and communication becomes impractical at $\frac{E_b}{N_0}$ values under 16 dB. The large jump just below 16 dB is thought to be caused by the hardware automatic gain control switching between gain levels and causing clipping at the higher gain.

The frame error rate results give some insight into the odd shape of the bit error rate graph. If a data point exists that shows a frame error rate without FEC, but there are no data points for 50 or 100 correctable errors, then that means there were no frames with that number of errors observed during that test. Knowing this it can be seen that several of the tests showed no frames with 50 or more corrupt bits, most notably at 11.5 dB. A possible explanation could be that the RF hardware did not always achieve the same steady state performance after being powered on. Possible causes of this could be clock offsets being different because of different temperatures. Another possibility is the AGC reacting differently to different input level, possibly even oscillating between states on some input levels and not others.

Since such instability was never observed without the RF hardware it is unlikely to be a software or DAC/ADC issue.

In the defence of the performance of the RF system it should be remembered that the RF components were still in the process of being developed when these tests were performed. It is expected that their performance will improve significantly upon completion of their development.

6.4 Summary

This chapter began by presenting a complete description of the software modulation and demodulation chain and then went on to show the bit error rate of the pure software system, which was shown to be fairly close to ideal. The DAC and ADC were then tested by outputting and sampling various different signals including random modulated data. Both were shown to perform acceptably,

although an increase in sample rate would simplify some other parts of the system design. The chapter concluded with bit error rate tests of the system, both without the RF components connected and with the RF components connected.

It could clearly be seen that every extra component added to the system had a significant effect on bit error rate, with the RF components having the greatest effect. Timing errors were also seen to create far more burst errors than expected from the AWGN channel model. Despite this, the system was still capable of transferring data quite acceptably, although a higher $\frac{E_b}{N_0}$ was required, especially when RF hardware was introduced.

In the next chapter the development of the Data-link layer is described.

Chapter 7

Data-link layer

This chapter shows the detail of the design of the data-link layer. It starts by giving the requirements of the data-link layer. The design of the components that comprise the frame used are covered next. It then presents the structure of a complete frame and, finally, how everything is combined to make a functioning network layer.

7.1 Frame Components

This section details the development of the libraries used to populate some of the fields in the frame header. Section 7.4 shows the complete frame structure. The ASM module, CRC module and randomiser module are software counterparts to the FPGA implementation of the same modules in [19].

7.1.1 Attached Synchronisation Marker

An attached synchronisation marker (ASM) is a sequence of bits used to identify the beginning of a frame. Attaching this sequence is trivial. To recover the frame the ASM mechanism checks for a match on each incoming bit. The sequence chosen is hexadecimal `0x1ACFFC1D`. This ASM sequence is recommended by the CCSDS [20, chap. 8].

This is implemented as a C library and works by having a function that accepts arrays of bits and checks the entire array for matches. A buffer of the last 32 bits is kept so that the system can detect the ASM even if it is split between two different calls of the function. If a match is found the frame

is copied to another buffer and a pointer to this buffer is stored. A second function call checks whether any complete frames are available and returns a pointer to the first available frame. NULL is returned if no frames are available.

7.1.2 Cyclic Redundancy Check

A cyclic redundancy check (CRC) is a very common error detecting code. Its use is necessary when using the system without error correction to detect and drop frames with errors.

The CRC polynomial used is $g(x) = X^{16} + X^{12} + X^5 + 1$, often called CRC-CCITT. The implementation used is based on the lookup table implementation described by Williams [21]. The basic implementation of a CRC is a linear feedback shift register (LFSR) that XORs the entire contents of the register, with its polynomial depending on the value of the oldest bit in the register. This method uses a 256 element lookup table for each 8 bits being shifted out, in order to determine with what the register should be XORed. Since the last 8 bits will always affect the rest of the register in the same way, regardless of the other values in the register, this is a valid optimisation.

This is implemented as a C library with a function that takes a pointer to an array of unsigned 8 bit integers and the length of the array. The CRC checksum is returned as an unsigned 16 bit integer.

7.1.3 BCH Forward Error Correction

BCH codes are error correcting codes that allow precise control of the number of correctable errors.

The library in use was adapted from the BCH library used by the Linux kernel. It was modified to be used as a userspace library by replacing all kernel functions with their userspace equivalent, for example, `kmalloc()` is replaced by `malloc()`. Since this code is GPL licensed, any program that uses it must also be GPL licensed. This should be taken into consideration if the programs described here were to be distributed.

The BCH parameters used are a field order of 12, giving a code length of 4095 bits¹ and the ability to correct 100 errors. 4095 bits can easily be stored in 512 bytes, so we use this to choose the frame size of 512 bytes.

¹Codeword size = $2^{\text{field order}} - 1$

The library provides functions to generate the BCH polynomial from the given parameters, to encode a block of data and to decode a block of data giving a list of error locations if possible.

7.2 Randomiser

A randomiser is required to ensure that enough bit shifts are transmitted that the demodulator can maintain lock on the transmitter clock. Long periods of 1's or 0's that could cause these problems are removed by XORing the data to be transmitted with the output of the randomiser.

The actual bits being used are pseudo-random and generated by a 8-bit LFSR with polynomial $h(x) = x^8 + x^7 + x^5 + x^3 + 1$ with every bit having an initial value of 1. This sequence repeats after every 255 bits. This sequence is recommended by the CCSDS [20, chap. 9].

Since the LFSR is only 8 bits long, every possible state it can take can be saved in a 256 element array, with the index of the array being the current state and the value the next state. This array can then be used as a lookup table to efficiently generate the random sequence.

This is implemented in a C library with a function that returns the next 8 bits of the LFSR as an unsigned 8 bit integer.

7.3 Automatic Repeat Request

The Automatic Repeat reQuest (ARQ) mechanism is responsible for the retransmission of frames lost because of corruption from the channel. Transmitted frames are stored in a buffer and received frames are acknowledged by sending an ACK frame. If an ACK frame is not received within a timeout period the frame is retransmitted. Sequence numbers are added to the frames to ensure that they are received in the correct order and to allow for multiple frames to be transmitted and acknowledged at the same time without ambiguity. If a frame is received out of order it is ACKed and if it has not been received before, it is placed in a buffer until it is the frame with the sequence number that the receiver expects.

The library that implements this is split into two parts, the transmitting part and the receiving part, with some functions that are used by both.

On the transmitting side the library provides the functionality to save frames for possible retransmission and to keep track of which frame should be retransmitted and when the next retransmission could occur. It also provides functions to keep track of the current sequence number. The transmitter side is also responsible for generating ACK packets when the receiver side requests them.

For the receiving side, the ability to save frames to a buffer and to rearrange frames to the same order they were transmitted in is available. The receiver also requests the generation of ACKs from the transmitter process when a packet is received. If an ACK is received this is also communicated to the transmitter process, so that the ACKed frame can be taken out of the timeout queue.

This library is also responsible for the synchronisation of sequence numbers between nodes. This is only done when a node is starting up. The synchronisation mechanism in use is based on the method used in TCP and consists of sending a synchronisation (SYN) frame, waiting for an acknowledgement of this frame and acknowledging the acknowledgement.

7.4 Frame Structure

Table 7.1 shows the frame structure. The frame length is 512 bytes.

7.5 Integration

It will now be described how the different parts of the data-link layer fit together and interact with each other.

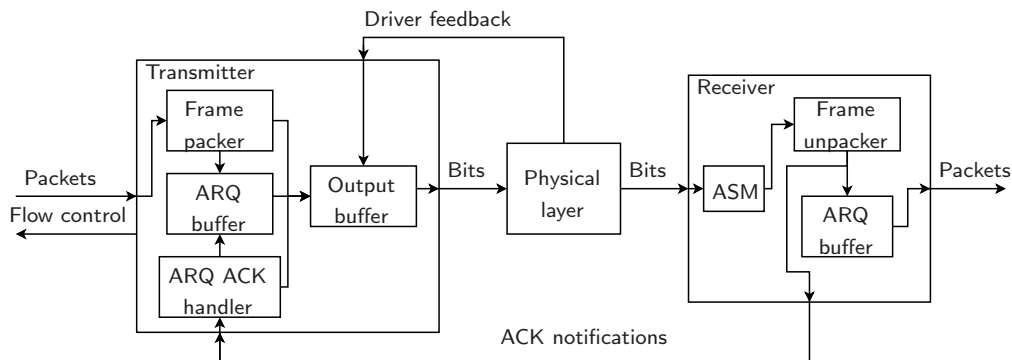
Figure 7.1 shows a high level overview of the data-link layer and also how it relates to the physical layer, specifically the diagram of the physical layer in Figure 5.9. Like the software in the physical layer, most inter-process communication is done by making use of `STDIO`.

Communication from the DAC/ADC driver and between processes in the data-link layer is performed by making use of Linux FIFOs. A Linux FIFO is a Linux pipe that has a name within the file system. The FIFO (First In First Out) appears as a file and can be read from and written to. Unlike a file,

Field	Data type	Note
ASM	Unsigned 32 bit integer	Always hexadecimal 0x1ACFFC1D
Sequence number	Unsigned 32 bit integer	Used for data and ACK
Frame type	Unsigned 8 bit integer	0 - Data 1 - ACK 3 - SYN 4 - SYN ACK 5 - SYN ACK ACK
Miscellaneous	Unsigned 8 bit integer	Only used for sending synchronisation status.
Data length	Unsigned 16 bit integer	
CRC	Unsigned 16 bit integer	
Frame data	Unsigned 8 bit integer array with 355 elements	
BCH check bits	Unsigned 8 bit integer array with 143 elements	

Table 7.1: Frame structure.

and as its name suggests, data written to it will be available for reading in the same order it was written but will disappear from the FIFO once read.

**Figure 7.1:** Data-link layer overview.

7.5.1 Transmitting Side

The transmitting side of the data-link layer is responsible for packing packets received from the higher layers into frames, adding FEC information, and

transmission of the frames by sending them to the physical layer. It also holds a buffer of transmitted frames to allow the retransmission of lost frames. If an acknowledgement for a frame does not arrive during the time out period then the frame is retransmitted.

To accommodate the retransmission of frames, as well as the transmission of new frames, either non-blocking I/O or threads need to be used. Non-blocking I/O by making use of the `select()` system call was chosen because of the researcher's familiarity with this system call. More information about this system call can be found in Section 5.1.5 since the DAC/ADC driver also makes use of it.

Figure 7.2 shows an overview of the operation of the transmitter that covers the most important parts of its operation. For the sake of readability some details such as error handling and the exact details of the synchronisation mechanism have been left out of this diagram.

The transmitter starts by sending a SYN frame after initialisation. It will not attempt to read any packets from the higher layer until it has received a SYN ACK frame and a SYN ACK ACK frame. Alternatively, it will start reading packets from a higher layer if it receives a SYN ACK frame that indicates that the other side has already synchronised at some point. Upon reception of a SYN ACK frame it will then respond with a SYN ACK ACK frame. After this synchronisation both sides start transmitting frames starting at sequence number 0.

The method used to indicate that one side has already synchronised in the past is to use the miscellaneous field in the packet header. This mechanism allows both sides to start up simultaneously, as well as allowing one side to disconnect and reconnect without restarting the other side.

The length of a packet, as well as the packet data, is sent to the transmitter via `STDIO`. When a packet is received from `STDIN` it is placed in a frame along with all the necessary frame header information and FEC. This frame is then saved to the ARQ buffer for possible retransmission and is also randomised, converted to a one bit per byte form and sent to the output buffer.

To prevent overuse of the pipe buffer, as also seen in Section 5.1.5.5, a flow control mechanism was created by means of which the receiving of a packet over `STDIN` is reported back to the higher layers. This mechanism allows for better control of the number of packets that are in the pipe buffer. The reception of a

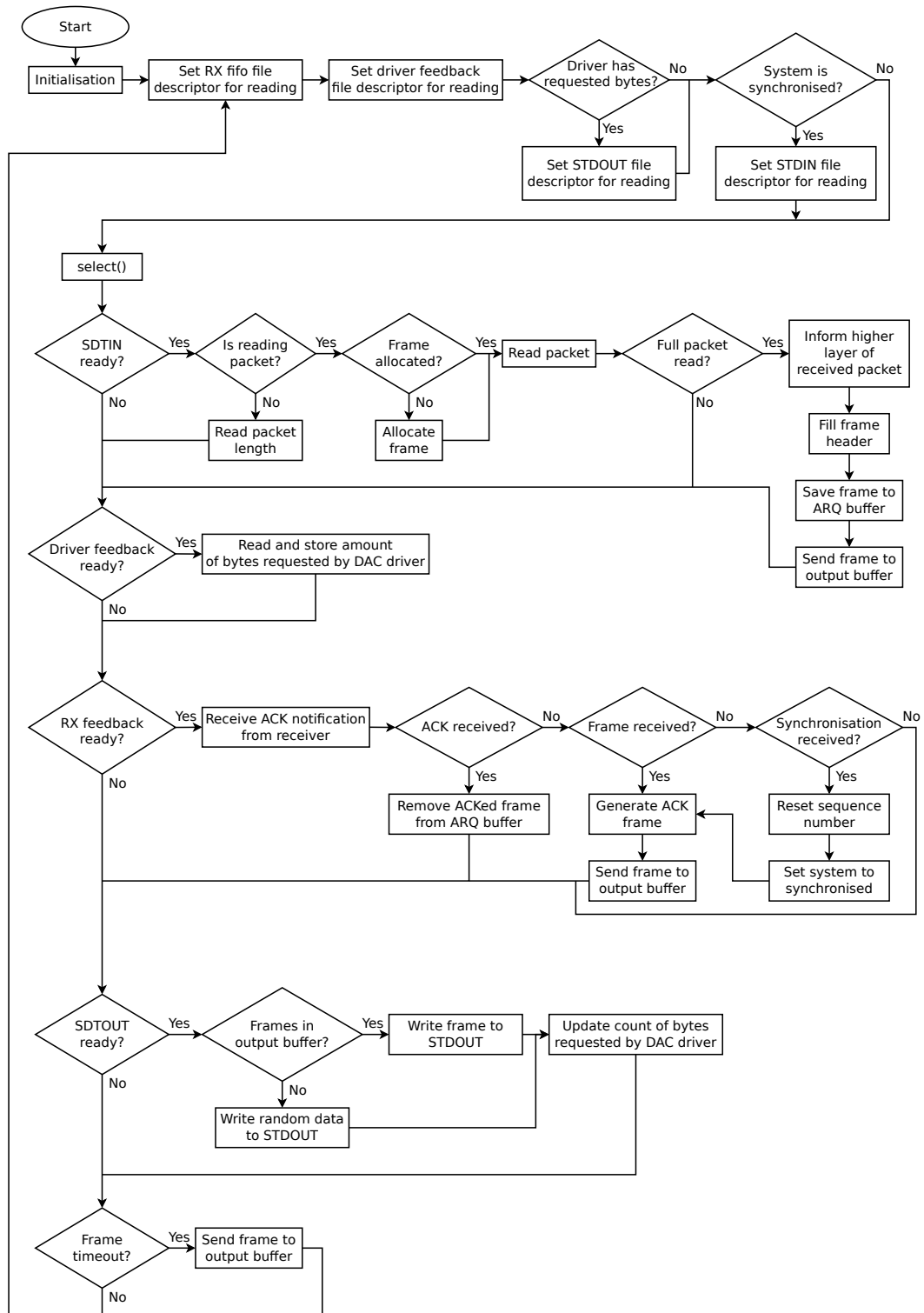


Figure 7.2: Diagram of transmitting part of the data-link layer.

packet is reported by making use of a Linux FIFO.

If driver feedback is received, the count of outstanding bits that should be sent to the driver is updated. This is used for flow control as described in 5.1.5.5.

If a notification is received from the receiving part of the data-link layer then one of several things can happen:

- If it is an ACK then the corresponding frame is removed from the ARQ buffer since it has been successfully received.
- If it is a notification of a received frame then an ACK frame is generated and sent to the output buffer.
- If a synchronisation frame was received, ACK it and set the internal sequence number to 0.
- If a synchronisation ACK is received, ACK it. If it indicated that the other side has synchronised in the past, then assume the system is synchronised.
- If a synchronisation ACK ACK is received and a synchronisation ACK has been received, then assume the system is synchronised, otherwise wait for a synchronisation ACK and then assume the system is synchronised.

If the count of the bits that should be sent to the DAC/ADC driver is large enough to send a frame and one is available in the output buffer then it is sent. If there is no frame to send then random data is sent. This helps the demodulator on the receiving end to maintain a lock on the signal.

If a frame in the ARQ buffer times out then it is randomised and sent to the output buffer for retransmission.

7.5.2 Receiving Side

The receiver is responsible for identifying frames from the stream of bits provided to it from the demodulator, correcting errors in frames, verifying the integrity of frames, ensuring frames are in the correct order and unpacking packets from frames to send on to the higher layers.

Figure 7.3 shows an overview of the receiving part of the data-link layer. As with the transmitter, some details such as error handling were omitted from the diagram.

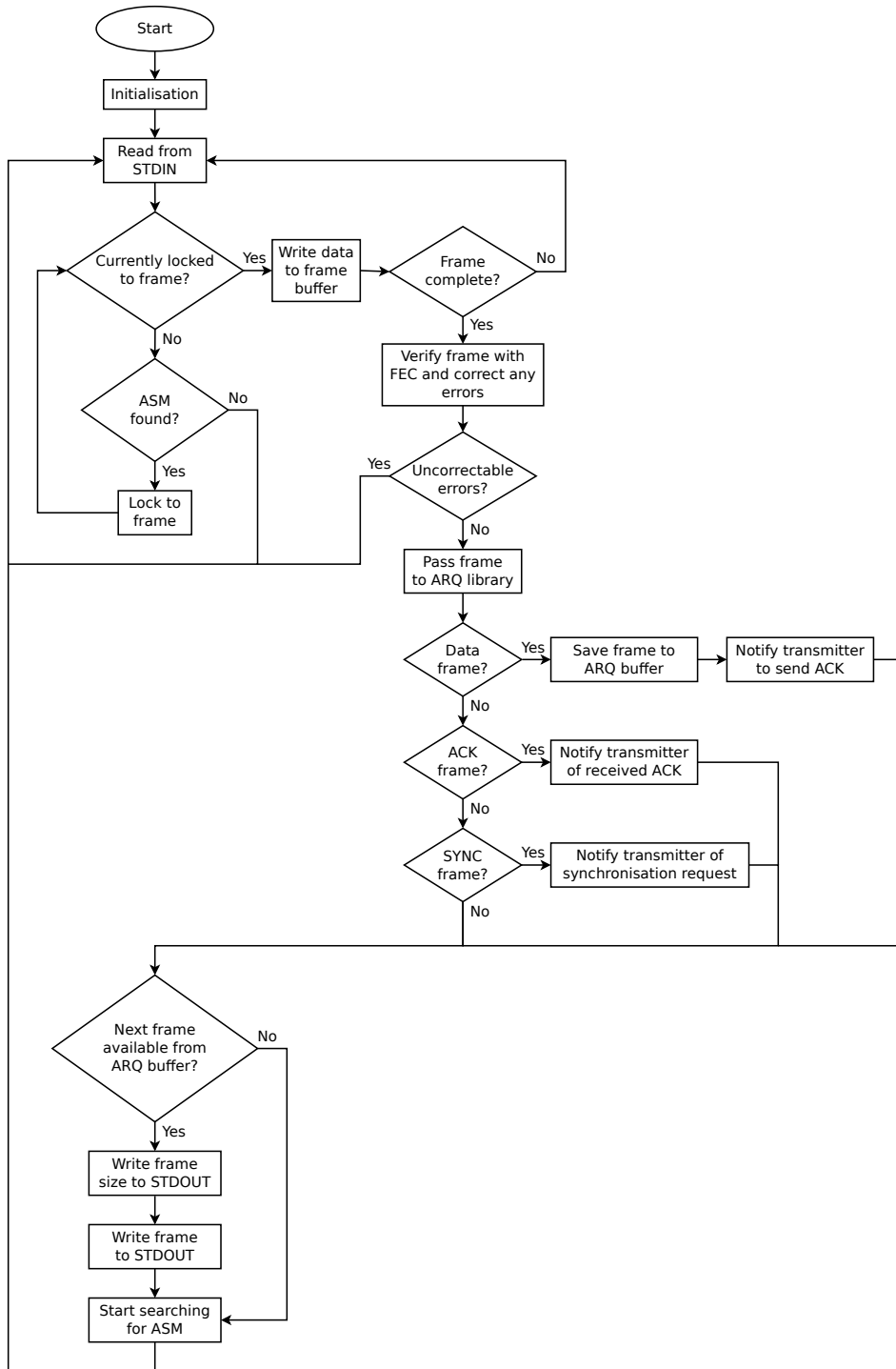


Figure 7.3: Diagram of receiving part of the data-link layer.

Blocking I/O is used in the receiver, unlike the non-blocking I/O which is used by the transmitter.

The default state of the receiver is to block, waiting for bits to arrive from the demodulator over `STDIN`. If an ASM is found in the stream of bits then the rest of what is assumed to be a frame is copied to a buffer.

Once the entire frame has been received it is converted from the one bit per byte to standard byte representation. The randomisation applied at transmission is removed and the FEC code is verified. If there are correctable errors then they are corrected.

The frame is then passed to the ARQ library. Different types of frame are handled differently:

- If a data frame is received then it will be placed in the ARQ buffer if it has not been received before, or discarded if it has. In both cases the transmitter will be notified to send an ACK frame.
- If an ACK frame is received the transmitter will be notified of the received ACK.
- If it is a synchronisation, frame then the transmitter is requested to ACK this frame and the next expected sequence number of the receiver is set to 0.
- If it is an ACK for a SYN frame, then the transmitter is requested to ACK it.
- If it is the ACK for an SYN ACK frame, then the transmitter is informed of this.

If the next sequence number expected by the system is available in the ARQ buffer then the frame data length and data is written out to `STDOUT` and the next expected sequence number is incremented. This is repeated until the frame with the next sequence number is not available.

The receiver then starts reading bits from `STDIN` again.

7.6 Summary

The design of the libraries used to populate the fields in the frame header, as well as the library responsible for retransmission, have been described, as well as how all these components have been integrated into the data-link layer.

In the next chapter the operation of the different components, as well as the data-link layer as a whole, will be verified.

Chapter 8

Testing of Data-link Layer

In this chapter the functionality of the various components that compose the data-link layer as well as the layer as a whole are tested. The test data used was chosen to be of the same size, or some multiple of the size, of a data-link link layer frame. This was done in an attempt to test the libraries under conditions close to those in which they are in intended to be used.

8.1 Attached Synchronisation Marker

The ASM detection code must be tested to ensure that it behaves correctly under all circumstances. It is also tested to see how well it behaves in the presence of noise.

8.1.1 Find ASM in Random Bits

This test uses the library to find an ASM within a random stream of bits and to check whether the bits following the ASM are the correct bits. An array of 40960 bits with 3 ASMs is given to the ASM library.

Result

All 3 ASMs and their following bits were correctly detected.

8.1.2 Find an ASM that is Spread Over Two ASM Finder Library Calls

This test is for the corner case where an ASM is not entirely contained within a single call of the ASM detection function. For example, the first few bits of an ASM are the last few bits of an array being analysed and the last few bits of the ASM are in the next array. The non ASM parts of the data are still random bits.

Result

ASM and the bits following it were correctly detected.

8.1.3 Find Corrupted ASM in Random Bits

This is the same test as in Subsection 8.1.1 but with the ASM corrupted by a certain number of bits.

Result

The ASMs were correctly detected when the number of corrupt bits was below the library tolerance. The bits following the ASM were also correctly detected. It was noted that there were some false positives with higher numbers of tolerated errors. The next test investigates this more thoroughly.

8.1.4 Determine Number of False Positives

This test gives some indication of the number of false positives that the ASM library will find depending on the number of flipped bits it is set to tolerate. The library scans 4096 kilobits of random data with a differing number of tolerated corrupt bits. This data does not contain the ASM.

Result

The library detected no false positives up to 2 tolerated faults and from there the number of detected false positives increased quickly. Table 8.1 summarises the results. This gives a good indication that a tolerance of 2 or perhaps 3 could be used in practice. Any more would waste a large amount of resources

on dealing with false positives. Also, since the library cannot detect overlapping frames, a false positive could lead to a missed frame.

Tolerance	False positives
0	0
1	0
2	0
3	6
4	42
5	189
6	532

Table 8.1: Result of testing ASM library's corruption tolerance versus false positives.

8.2 CRC

The CRC implementation must be tested to ensure that it is correct and produces the expected CRC values. These tests assume that the CRC algorithm itself is correct.

8.2.1 Test the Output of the Library Against a Reference Output

The output of the CRC library for the test string '123456789' is compared to the reference output 0xE5CC [22].

Result

The output of the library matched the reference output.

8.2.2 Attempt to Detect Varying Amounts of Corruption

Create an array of random data and append a CRC of this data so that the total length is equal to that of a frame. Corrupt this array by varying degrees and attempt to detect errors using the CRC. Record all false positives.

Result

Out of 10 000 000 different corrupted arrays:

- 2580 had no errors and were all detected as error free.
- 152 arrays were incorrectly detected as error free.
- All even numbered errors were correctly detected.

Thus 99.998% of errors were detected correctly. This is the percentage of errors CCITT-CRC should be able to detect [23, chap. 5.3.2]. The library is performing as expected.

8.2.3 Test for Detection of Burst Errors

The same test as in Section 8.2.2 but with a single burst error instead of random bit flips. The burst errors tested were of differing lengths, placed at every possible position in the array. 100 randomly generated arrays were tested with burst lengths from 0 to 100 bits. Both burst errors that change bit to 0, and those that change it to 1 were tested.

Result

All burst errors with lengths less than or equal to 16 bits were correctly detected. Most other lengths produced some false positives.

8.3 BCH Forward Error Correction

For this test the BCH library is assumed to be generally working. It was tested to ensure it worked in the current configuration. The BCH code used for testing was configured to correct 100 errors and had a block length of 512 bytes, the same as in Section 7.1.3.

8.3.1 Test the Number of Correctable Errors

Generate several arrays of random bytes and append their BCH check bits. Corrupt these arrays with varying levels of bit flips and attempt to recover the original arrays. 1000 different arrays with 1000 different combinations of errors each were tested. A total of 1 000 000 tests was performed.

Result

All bit errors of 100 or less were properly corrected. Sometimes the data from arrays with more than 100 errors could still be corrected if all the errors in the data could be corrected, even when all the errors in the check bits could not. No false positives were found. It is expected that a more exhaustive search would reveal false positives, but this was deemed unnecessary.

8.4 Randomiser

The randomiser output is compared with its specifications [20, chap. 9]. These tests are only for the correctness of the randomiser implementation. It is assumed that the algorithm itself is correct.

8.4.1 Test if the Correct Sequence is Produced

The specification on which this randomiser is built gives the first 40 bits of output as an example. This test compares the output of the randomiser to those bits.

Result

The output of the randomiser matched the test bits.

8.4.2 Test if Sequence Repeats After 255 Bits

The randomiser sequence should repeat after 255 bits. This tests whether it does.

Result

The randomiser sequence repeats after 255 bits.

8.4.3 Test if Sequence Resets Properly

The randomiser has a function that resets the sequence to its beginning. This tests whether the output bits restart at the beginning of the sequence after a reset.

Result

The sequence properly restarts after a reset.

8.5 Automatic Retry Request

The general working of the ARQ library needs to be verified. It must be shown that frames can be sent from the transmitting side to the receiving side, will be retransmitted when lost and will be received in the correct order. This test will also show the generation of ACKs and the ACKing of frames in the transmitting ARQ buffer. The communication between the transmitting and receiving libraries will be implicitly tested, since the ACKing mechanism depends on it. The synchronisation mechanism is also tested because it is a special case of frame transmission and uses other code than the usual transmission of frames.

8.5.1 Test Basic Transmission of Frames

This tests basic frame transmission. A few frames are inserted into the transmitting ARQ buffer and copies are sent to the receiving ARQ buffer. The ACKs for the frames are generated and also sent. It is then checked that the frames have been properly ACKed and whether the same frames that were sent were received.

Result

The frames sent were properly received and ACKed.

8.5.2 Test the Retransmission of Frames

The same test as in Section 8.5.1 is performed with one frame being lost on purpose. The test waits for the frame to timeout and then sends the frame that the ARQ library produces for retransmission.

Result

The lost frame was retransmitted and all frames were correctly received and were in the correct order.

8.5.3 Test of Whether Synchronisation System Works

To test the synchronisation exchange an instance of the transmitting and receiving ARQ libraries are initialised and a synchronisation sequence is emulated by letting the library generate the necessary frames and pass them between the transmitting and receiving libraries as needed. This tests that the SYN frame is correctly generated, that the SYN frame is ACKed upon reception and that this ACK is ACKed upon reception.

Results

The synchronisation sequence functioned correctly.

8.5.4 Test of Whether Synchronisation System Works with Lost Frames

This tests the retransmission of frames in the synchronisation sequence when frames become lost. Since the sequence is only 3 frames long, the loss of all three frames can be tested. The previous test is repeated 3 times, each time with one of the frames being dropped.

Results

All frames were retransmitted correctly regardless of lost frames.

8.6 Complete Data-link Layer

Here the working of the data-link layer as a whole was tested and an attempt made to quantify its performance. In these tests two sets of transmitting and receiving data-link layers are set up and connect with FIFOs. This simulates two nodes communicating. Data can be piped into one of the transmitting sides and then received via a pipe from the other data-link pair's receiving side. This configuration allows files to be used as test data which, in turn, simplifies the comparison of data output with data input.

8.6.1 Test the Transmission of Data

This test transmits a few hundred megabytes of data and confirms that the same data arrives at the receiving end. This is more data than can be reasonably expected of the system to ever transfer in one session.

Results

The correct data arrived at the receiving end.

8.6.2 Test the Transmission of Data in the Presence of Noise

The same test data as in the previous test was used. The data transmitted between the two nodes was corrupted by a bit error rate of 0.01.

Results

The correct data arrived at the receiving end.

8.6.3 Test the Performance of the Data-link Layer with FEC

The performance of the data-link layer is quantified in this test. The test transmits 10 000 frames with incrementing bit error rate and records the number of transmissions required to transmit all the frames. The results are then given as an average of transmissions per frame. The range of bit error rates tested is where the data-link layer begins to fail. For the sake of thoroughness this test also compares the output data to the input data.

Results

Figure 8.1 shows the results of the test as well as the theoretical results. The data-link layer performed as expected. Appendix B shows the calculation of the theoretical results.

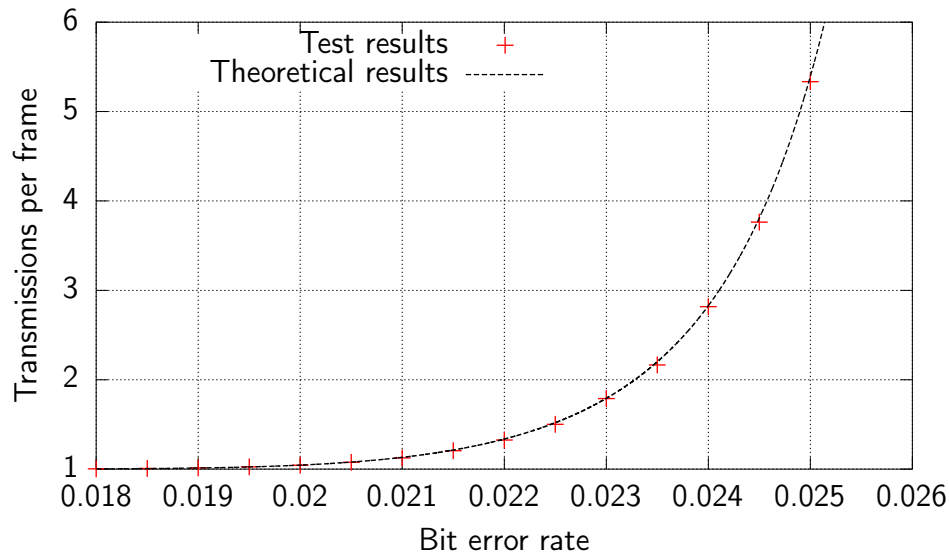


Figure 8.1: Measurement of data-link layer performance in the presence of noise.

8.6.4 Test the Performance of the Data-link Layer Without FEC

This repeats the previous test, but with forward error correction disabled. The bit error rate range has also been adjusted to give relevant results. This mode has a slight advantage over having FEC enabled, in that it can send more data per frame for the same frame size.

Results

Figure 8.2 shows the results of this test. Again the data-link layer performed as predicted. Appendix B shows the calculation of the theoretical results.

8.7 Summary

The workings of the individual components of the data-link layer have been tested and it was shown that all components performed as expected. This gave a reasonable assurance that they were all implemented correctly. The data-link layer as a whole has also been tested and it was found that it functioned correctly under all test circumstances. Lastly, the performance of the data-link

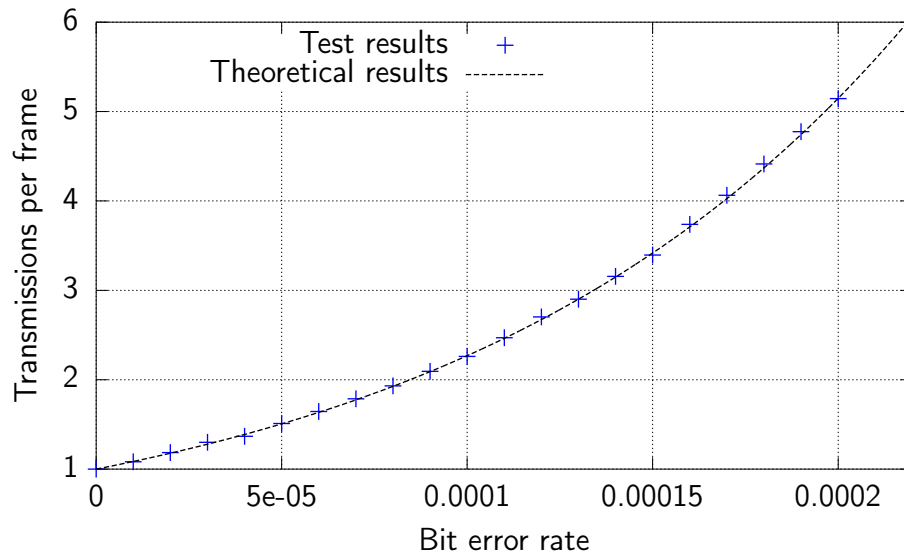


Figure 8.2: Measurement of data-link layer performance in the presence of noise with FEC turned off.

layer was compared to the theoretical performance and was found to be a close match.

In the next chapter the integration of the components of the system is described.

Chapter 9

System Integration

This chapter shows how all the components of the system are connected and how they interact with each other.

9.1 Higher Network Layers

The design of higher layers of the network stack was not within the scope of this project but these layers are nonetheless necessary to test the complete system. The IP stack in Linux was chosen as the higher layers to interface with. Since the Internet makes use of IP there is no shortage of applications to test every aspect of a network over IP. There are also software methods for emulating network hardware and interfacing the emulated hardware to other software.

The TUN/TAP device driver was used to provide the virtual network interface [24]. A TUN interface was chosen since it emulates a network layer device, as opposed to the TAP interface, which emulates the data-link layer. An open source demonstration implementation [25] for using the TUN/TAP device was found and modified to communicate over `STDIO` so that it could interface with the data-link layer.

9.2 Overview of Integrated System

The overview of the system given in Chapter 3 and Chapter 4 described the system from a functional perspective. Figure 9.1 describes the system from a

more practical and software oriented perspective. This figure is a combination of Figure 5.9 and Figure 7.1.

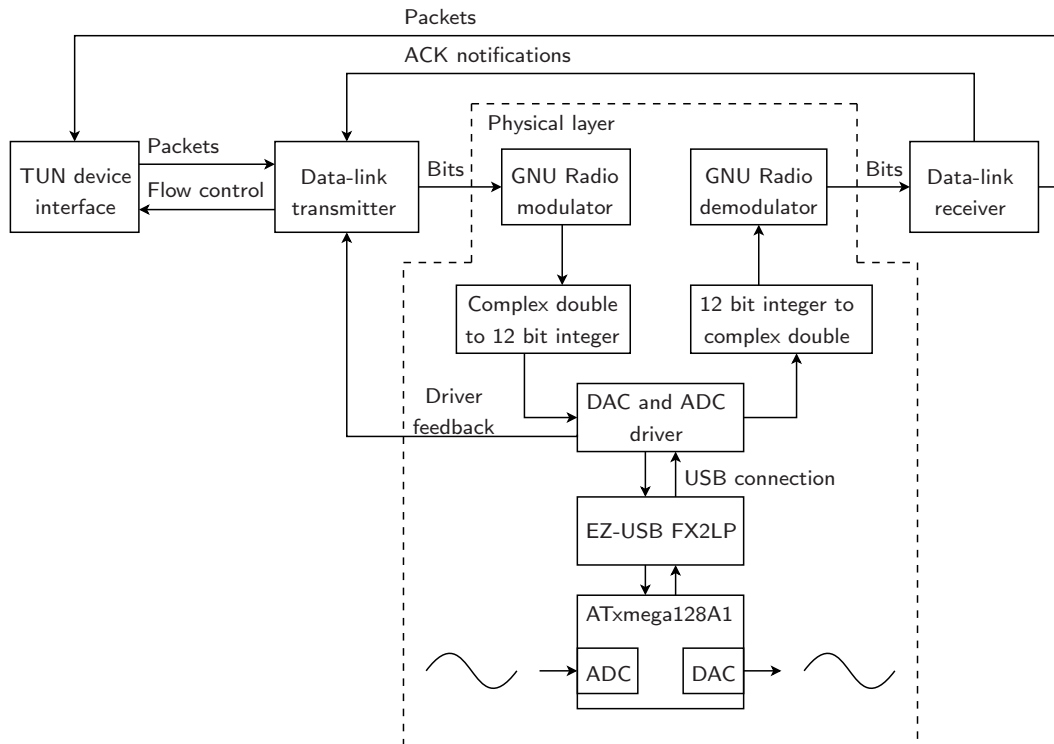


Figure 9.1: Software overview.

Transmission of a packet over this system works as follows:

- The packet is received by the TUN interface and made available to the TUN userspace process. This packet is then written to a pipe.
- The transmitting half of the data-link layer reads the packet from the pipe, encapsulates it in a frame, saves the frame to the ARQ retransmit buffer and writes the frame to a pipe in a one bit per byte format. Flow control information is also sent back to the TUN process via a FIFO.
- The bits are then read from the pipe and modulated by the GNU Radio based modulator into complex baseband BPSK and this is output to a pipe as a `complex double` stream.

- A simple conversion process then reads these `complex double` values and converts them to unsigned 12 bit integers stored as unsigned 16 bit integers.
- The DAC/ADC driver then reads these values and sends them to the DAC over USB. They are then output as voltage signals. The driver also provides flow control information to the transmitting part of the data-link layer by using a FIFO.
- These signals are sent through a quadrature modulator and transmitted over the air to another identical system to be demodulated and fed into an ADC.
- The driver reads these values from the ADC and outputs them into a pipe.
- A conversion process reads these values and converts them from signed 12 bit integer to `complex double` values. These values are output to a pipe.
- The GNU Radio based demodulator then attempts to extract bits from this stream and outputs the bits to a pipe.
- These bits are then read by the receiving half of the data-link layer. It finds the frame by searching for its ASM and then outputs the packet contained in the frame to a pipe. A FIFO is also used to inform the transmitting part of the link-layer of the packets reception, allowing it to send an ACK.
- The TUN process reads the packet and passes it to the TUN interface.

If there are no frames to transmit the data-link process generates random bits to be modulated and transmitted. If an ACK frame is received it is passed on only to the transmitting part of the data-link layer and not to the TUN process.

9.3 Execution Script

A single shell script starts up the system. This script creates the necessary FIFOs, sets up the TUN network interface and then starts the required applications. Appendix C shows the full script.

9.4 Integration with Hardware

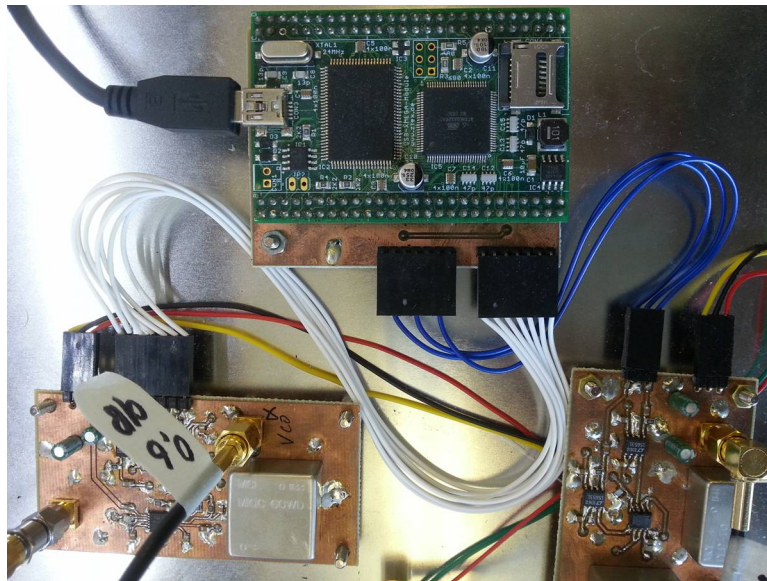


Figure 9.2: Image of the USB-XMEGA connected to the rest of the hardware.

Figure 9.2 shows the USB-XMEGA connected to the quadrature modulator and demodulator. The blue wires carry the output from the two DACs and the white wires carry the differential input to the two ADCs.

9.5 Initial Observations

The system is capable of standard IP based communication. Pings show no packet loss and round trip times of around 500 ms to 600 ms. The high latency comes from a buffering issue with the GNU radio based modulator, as shown in Section 5.1.5.5. Data throughput using TCP as measured by `iperf` is around 14 kbit/s. More thorough tests performed on the system are documented in Chapter 10.

9.6 Summary

The interconnection of the system components has been described and the possibility of efficient data transfer has been determined. In the following chapters the system as a whole will be tested.

Chapter 10

Testing of Complete System

To measure the performance of the complete system the latency, packet loss and throughput was tested. Since this system appears as a Linux network interface, standard network test tools can be used. The `ping` utility tests latency and packet loss and `iperf` measures throughput. The tests were performed at different $\frac{E_b}{N_0}$ values by adding RF attenuators to both transmitting sides. The attenuation values used were 20 dB, 43 dB and 50 dB giving measured $\frac{E_b}{N_0}$ values of 22 dB, 21 dB and 12.5 dB. Accurate measurement of $\frac{E_b}{N_0}$ was limited by the automatic gain control in the RF circuitry.

Because of the instability in the RF hardware seen in Chapter 6, it was expected that rerunning the test could produce slightly varied results. The tests were run over several minutes each, in an attempt to overcome this.

10.1 Latency and Packet Loss Tests

Table 10.1 shows the results of the latency and packet loss tests performed using `ping`.

Attenuation	Minimum latency	Average latency	Maximum latency	Standard deviation	Packet loss
20 dB	426,7 ms	461,3 ms	2538,7 ms	131,9 ms	0 %
43 dB	392,9 ms	7602,1 ms	60733 ms	16618 ms	12%
50 dB	411,5 ms	22111 ms	64894 ms	20263 ms	34%

Table 10.1: Results of latency and packet loss tests.

Making an accurate theoretical prediction about the latency is difficult since the latency of the software, especially the GNU Radio components, is difficult to model. The approximate minimum latency is $3 \times (L_{tx} + L_{software})$ with L_{tx} the transmit time of a data-link frame and $L_{software}$ the software latency. Since a single ping requires both the data-link layer ACK and ping response to be returned the predicted latency is three times the transmit and software latency.

The transmit time for this system with a baud rate of 62.5 kbit/s and a frame size of 4096 bits is 65.54 ms. This gives a minimum latency of 196.6 ms + $3 \times L_{software}$.

The ARQ system will retransmit a frame after 2 seconds if no ACK is received, thus for every retransmission a frame's latency will increase with 2 seconds.

The exact point at which packet loss will begin to appear is also hard to predict because the only buffers capable of dropping packets are in the higher IP layers and are not easy to investigate.

The system performed mostly as expected in these tests with the best performance at 20 dB attenuation and the latency and packet loss increasing as attenuation increased. Packet loss was observed, because the IP stack as well as this system's data-link layer will eventually run out of buffer space if communication is not possible for long enough. Latency increases as the data-link layer attempts to retransmit and correct lost frames. The maximum latency seen at 20 dB is most likely caused by a single data-link retransmission since it is approximately the average latency plus 2 seconds.

The unpredictable nature of this system's error rate can be seen in the large variation in latencies in the higher attenuation tests. This is similar to the unpredictable results seen in Section 6.3.2 where the full physical layer was tested. The high minimum latency, first seen in Chapter 9.5 can also be seen in these tests. Long periods of no communication were seen at both 43 dB and 50 dB attenuation. This is expected in a system with slightly varying performance, as caused by the RF hardware, or by the demodulation software losing and regaining lock.

The minimum latency varied a bit with the lowest minimum latency seen at 43 dB attenuation. The most likely cause for this would be a fluke of the Linux scheduler, where the buffer that causes the usually longer delay happened to be emptier than usual when the ping was sent.

10.2 Throughput Tests

Table 10.2 shows the results of the throughput tests performed using `iperf`. Note that `iperf` calculates the throughput from the TCP payload throughput. For this test there was 303 bytes of TCP payload for every 512 byte frame. The tests were also performed with data being sent in only one direction and with data being sent in both directions simultaneously.

Attenuation	Unidirectional throughput	Bi-directional throughput
20 dB	19,0 kbit/s	8,89 kbit/s 8,62 kbit/s
43 dB	13,3 kbit/s	6.53 kbit/s 7.84 kbit/s
50 dB	6,03 kbit/s	6,75 kbit/s 4,80 kbit/s

Table 10.2: Results of throughput tests.

The theoretical maximum throughput can be calculated as follows: At 750 ksps at 12 samples per bit the maximum raw transfer rate is 62,5 kbit/s. Every 512 byte data-link layer frame has only 303 bytes TCP payload. Also, TCP packets are ACKed and these ACKs are again ACKed by the data-link layer, causing as many data-link ACK frames to be sent as TCP containing frames, halving the effective data rate. This gives a theoretical maximum throughput of 18.49 kbit/s. The bi-directional case is expected to have half the throughput of the unidirectional case since the ACKs generated by the unidirectional case already uses all the available throughput in the return direction.

The results here show values slightly above this maximum. Inspections of the actual data sent showed that the Linux TCP implementation occasionally combined TCP ACKs to acknowledge more than one packet per ACK. This explains the slightly higher than predicted throughput.

Some asymmetry of the two signal paths can be seen from the results of sending data in both directions simultaneously. This is from a combination of component tolerances and attenuator tolerances.

10.3 Summary

The latency and throughput of the system has been tested under different levels of attenuation applied to the transmitters. The system was shown to degrade with an increase in attenuation, as expected. Long periods in which no communication was possible were observed under higher attenuation levels. This was also expected. Even under these conditions, data transfers could still be properly resumed once communication was restored and could still be completed without errors in the delivered data.

In the next chapter this document is concluded and some recommendations for further work are given.

Chapter 11

Conclusions and Recommendations

11.1 Conclusion and Summary

The physical layer is capable of transforming bits to analogue I and Q values as well as transforming I and Q signals back into bits. Performance without the radio frequency components being taken into account is between 0.2 dB and 2 dB worse than the theoretically predicted values. Tests showed that the system should function reasonably above $9 \text{ dB } \frac{E_b}{N_0}$ if 100 bits can be corrected by error correction. These results are approximately 2 dB better than with no error correction. The radio frequency components of the physical layer interfaced correctly with the components developed here, and adequate communication was possible. The performance of the system with RF components included in their present state was significantly worse than without, but this is expected to improve as those components are refined.

The data-link layer is capable of providing reliable transmission. The error correction performed as expected and the data-link layer as a whole matched theoretical predictions with and without error correction.

The system as a whole was able to transfer data reliably at a maximum throughput of about 19 kbits/sec using TCP/IP.

11.2 Work and Development Successfully Completed

The developments and work undertaken in this project and successfully completed can be summarised as follows:

- Fairly modular and flexible data-link layer and physical layer components that could be reused in future telemetry and telecommand applications was developed.
- Demonstrated that a relatively low cost DAC/ADC pair can be used successfully for this type of communication with good results. This was achieved by selectively targeted software optimisation techniques.
- Performed an investigation into using GNU Radio for modulation and demodulation using custom hardware and showed it as being viable.
- Successfully tested and proved the functionality of a fully integrated system.

11.3 Recommendations

The following is a list of known issues or possible improvements:

11.3.1 General

- Investigate using another inter-process communication system to address the latency problems encountered with the use of pipes.

11.3.2 Physical Layer

- Use DACs and ADCs capable of higher sample rates, possibly using a module with enough processing capability to interpolate the samples meant for the DAC and decimate the samples from the ADC. This will mitigate many of the problems encountered with DAC and ADC filtering.
- Consider the use of QPSK instead of BPSK.

- Limit FLL maximum frequency error to prevent locking on DC offset or other signals.
- Use commercial software defined radio hardware for the RF layer.
- Do software modulation and demodulation on a FPGA, for reasons of processing speed and power consumption.

11.3.3 Data-link Layer

- Allow headers to be sent without payload. This would greatly decrease wasted communication potential from ACK frames which contain a blank payload.

List of References

- [1] Verschaeve, T.: Development of a satellite QPSK transceiver with doppler shift compensation. 2013. Unpublished Masters Thesis.
- [2] ITU-T recommendation X.200 - Basic reference model: The basic model. 1994. Available at: <http://www.itu.int/rec/T-REC-X.200-199407-I/en>
- [3] Braden, R.: Requirements for Internet Hosts - Communication Layers. RFC 1122 (INTERNET STANDARD), October 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633. Available at: <http://www.ietf.org/rfc/rfc1122.txt>
- [4] Lathi, B.: *Modern digital and analog communication systems*. 3rd edn. Oxford University Press, Inc., New York, 1998.
- [5] Digital Modulation in Communications Systems - An Introduction. Accessed May 2013. Available at: <http://cp.literature.agilent.com/litweb/pdf/5965-7160E.pdf>
- [6] Narayanan, K.: Baseband equivalent of passband signals. Accessed May 2013. Available at: <http://ecen661tamu.wikidot.com/basebandequivalent>
- [7] Morelos-Zaragoza, R.H.: *The Art of Error Correcting Coding*. Wiley, 2006. ISBN 0470015586.
- [8] USB-XMEGA Module: USB 2.0 Microcontroller Board with Analog Functions. Accessed May 2013. Available at: <http://www.ztex.de/usb-1/usb-xmega-1.0.e.html>
- [9] Documents for ATxmega128A1. Accessed May 2013. Available at: <http://www.atmel.com/devices/atxmega128a1.aspx?tab=documents>

- [10] Missed optimization accessing struct component with integer address. Accessed May 2013.
Available at: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=50448
- [11] libusb website. Accessed June 2013.
Available at: <http://www.libusb.org/>
- [12] GNU Radio Website. Accessed June 2013.
Available at: <http://www.gnuradio.org>
- [13] Harris, F., Venosa, E., Chen, X. and Dick, C.: Band edge filters perform non data-aided carrier and timing synchronization of software defined radio QAM receivers. In: *Wireless Personal Multimedia Communications (WPMC), 2012 15th International Symposium on*, pp. 271–275. 2012. ISSN 1347-6890.
- [14] GNU Radio API documentation for `digital_fll_band_edge_cc`. Accessed June 2013.
Available at: http://gnuradio.org/doc/doxygen-3.6/classdigital__fll__band__edge__cc.html
- [15] Harris, F. and Rice, M.: Multirate digital filters for symbol timing synchronization in software defined radios. *Selected Areas in Communications, IEEE Journal on*, vol. 19, no. 12, pp. 2346–2357, 2001. ISSN 0733-8716.
- [16] GNU Radio API documentation for `gr_pfb_clock_sync_ccf`. Accessed June 2013.
Available at: http://gnuradio.org/doc/doxygen-3.6/classgr__pfb__clock__sync__ccf.html
- [17] GNU Radio API documentation for `digital_constellation_receiver_cb`. Accessed June 2013.
Available at: http://gnuradio.org/doc/doxygen-3.6/classdigital__constellation__receiver__cb.html
- [18] Costas, J.: Synchronous communications. *Proceedings of the IEEE*, vol. 90, no. 8, pp. 1461–1466, 2002. ISSN 0018-9219.
- [19] Wiid, R.: *Implementation of a protocol and channel coding strategy for use in ground-satellite applications*. Master's thesis, Stellenbosch University, 2012.
Available at: <http://hdl.handle.net/10019.1/20346>

- [20] *TM synchronization and channel coding*. Consultative Committee for Space Data Systems (CCSDS), Newport Beach, CA, August 2011.
Available at: <http://public.ccsds.org/publications/archive/131x0b2ec1.pdf>
- [21] Williams, R.N.: A painless guide to CRC error detection algorithms. August 1993.
Available at: http://www.repairfaq.org/filipg/LINK/F_crc_v3.html
- [22] Geluso, J.: CRC16-CCITT. Accessed July 2013.
Available at: <http://srecord.sourceforge.net/crc16-ccitt.html>
- [23] Wicker, S.B.: *Error Control Systems for Digital Communication and Storage*. Prentice-Hall, 1994. ISBN 0132008092.
- [24] Krasnyansky, M.: Universal TUN/TAP device driver. Accessed June 2013.
Available at: <https://www.kernel.org/doc/Documentation/networking/tuntap.txt>
- [25] Brini, D.: A simplistic, simple-minded, naive tunnelling program using tun/tap interfaces and TCP. Accessed June 2013.
Available at: <http://www.cis.syr.edu/~wedu/seed/Labs/VPN/files/simpletun.c>

Appendices

Appendix A

Test System Used for Physical Layer

In this appendix the execution of the BER tests for the physical layer are discussed. We show how $\frac{E_b}{N_0}$ was estimated as well as how it was varied.

A.1 $\frac{E_b}{N_0}$ Estimation

Since there is no way of easily separating the noise from the signal, $\frac{E_b}{N_0}$ cannot be directly calculated. It is possible to estimate its value by comparing the power of the received signal with the power of the noise floor for the same bandwidth. Since the received signal already has noise added, the received signal power can be calculated by subtracting the received noise power.

$$\frac{E_b}{N_0} = \frac{P_{\text{signal}} - P_{\text{noise}}}{P_{\text{noise}}} \quad (\text{A.1.1})$$

P_{signal} and P_{noise} were calculated by taking the RMS values of the signal filtered by a root raised cosine filter and the noise floor filtered by a root raised cosine filter. Since the noise floor was not always flat, the best estimation of P_{noise} was at the negative of the frequency of the signal. This gave rise to another problem that a mirror of the signal is sometimes seen at the negative of the signal frequency and this interfered with the noise measurement. The final method of measuring noise was to take the average of the measured noise power at frequencies right above and below the mirror image of the signal.

This method was used to measure $\frac{E_b}{N_0}$ in Figure 6.10 for both the ideal demodulator, and the actual demodulator so is assumed to give correct results.

A.2 Measuring BER

Bit error rate was measured by transmitting random data with a minimalistic header containing a synchronisation marker, sequence number and error correction. The data size used was 512 bytes so that measured behaviour should match the behaviour of the actual 512 byte data-link frames.

These minimal frames were received and compared to the transmitted frames. The bit error rate as well as frame error rate for differing numbers of correctable errors can then be calculated from this. Only frames that could be reliably identified were taken into account. This should not affect the results, as long as the bit error rate does not vary significantly during a test.

A.3 Test Hardware

Some hardware was required to manipulate $\frac{E_b}{N_0}$. For the complete physical layer with the RF components included, attenuators were used to decrease the power of the transmitted signal thus decreasing $\frac{E_b}{N_0}$. For testing without the RF layer some more hardware was required.

Since the output of the DAC has a DC offset, a DC blocking capacitor was required. This was found to cause the ADC input to have DC drift, therefore a pull down resistor was added to remove that DC drift. Two simple third order resistor capacitor low-pass filters, one for the I channel and one for the Q channel, were constructed to remove the worst of the aliasing problem caused by sampling the unfiltered output of an ADC. To vary $\frac{E_b}{N_0}$ noise was injected to each of the signals by connecting noise generators through resistors to the output of each of the low-pass filters.

Figure A.1 shows this piece of hardware.

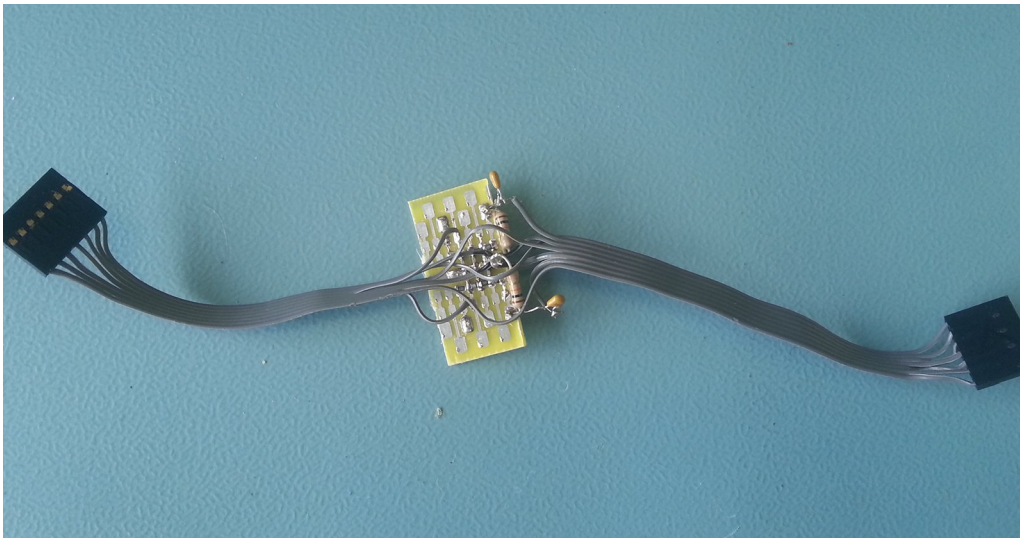


Figure A.1: Image of the circuit used to test the BER of the DAC directly connected to the ADC.

Appendix B

Prediction of Frame Error Rate

In this appendix it is shown how the frame error rate for frames with differing number of correctable bit errors can be calculated. Then it is also shown how to use this to calculate the average transmissions required per frame.

To calculate the probability that a certain number of bits will be flipped in a frame the following formula is used:

$$P(k) = \binom{n}{k} p^k q^{n-k} \quad (\text{B.0.1})$$

Where k is the number of flipped bits, n is the total number of bits, p is the probability that a single bit will flip and q is the probability that a single bit will not flip.

For the case that no error correction is possible a frame will fail if 0 bits are not flipped. Substituting $k = 0$ into Equation B.0.1 simplifies to $P(0) = q^n$ or the bit success rate to the power of the number of bits. This is the probability that the frame does not fail. The probability of the frame failing is $1 - P(0) = 1 - q^n$

For cases where a certain number of errors can be corrected, the frame success probability is the sum of the probabilities that 0 to the *number of correctable* errors happen. Evaluate Equation B.0.1 at every number of correctable errors and add all the results. The frame error probability is then 1 minus the frame success probability.

B.1 Transmissions per Frame

The average number of required transmissions is one divided by the probability of a successful transmission, $\frac{1}{p}$. In this case a successful transmission requires both the frame and its ACK to be transmitted. Using Equation B.0.1 with $n = k = 2$, where n is the total number of transmitted frames, k is the number of successfully transmitted frames, p is the probability of a successful transmission and q is the probability of a failed transmission. The probability of two out of two transmissions being successful is thus the probability of one being successful, squared. Thus the average number of transmissions required is $\frac{1}{(1-\text{Frame error rate})^2}$.

Appendix C

Execution Script of Complete System

```
#!/bin/sh

sudo true

rm netfifo
rm adcfifo
rm arqfifo
rm tapfifo
mkfifo netfifo
mkfifo adcfifo
mkfifo arqfifo
mkfifo tapfifo

sudo ./tunctl -n -d sat0
sudo ./tunctl -u $USER -t sat0 -n
sudo ip addr add 10.244.0.1/24 dev sat0
sudo ip link set dev sat0 mtu 355
sudo ip link set dev sat0 txqueuelen 1
sudo ip link set dev sat0 up

./simpletun -i sat0 < netfifo | ./packer e d l | \
./modulator.py | ./gnurtotwelvebit | \
sudo nice -n -2 ./dac_adc_pc | \
./twelvebittognur | ./demodulator.py | \
./unpacker e d l > netfifo

rm netfifo
rm adcfifo
rm arqfifo
rm tapfifo
./tunctl -n -d sat0
```