
DSP-Based GFSK Demodulator for AAU Ground Station

May 2014

Group 627

Project report





AALBORG UNIVERSITY
STUDENT REPORT

Department of Electronic Systems

Fredrik Bajers Vej 7

DK-9220 Aalborg Ø

<http://es.aau.dk>

Title:

DSP-Based GFSK Demodulator
for AAU Ground Station

Theme:

Embedded Real-Time Signal Processing

Project Period:

Spring Semester 2014

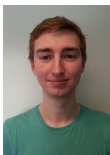
Project Group:

627, 6th Semester

Authors:



Jacob Møller Hjerrild Hansen



Jacob Velling Dueholm



Miklas Strøm Kristoffersen



Søren Bøgeskov Nørgaard

Supervisor:

Thomas Arildsen

Number of Prints:

2

Number of Pages:

110

Date for Submission:

May 28, 2014

Abstract:

The development of a software-defined GFSK demodulator for satellite communication is documented in this report. The existing front-end and communication protocol for communicating with AAUSAT3 is analyzed along with aspects of wireless communication with the specific satellite.

This analysis shows that the demodulator is going to encounter Doppler shifts and induction of noise. The report is centered around the design of the demodulator algorithm, dealing with Doppler and time shift. Following the preliminary analysis, requirements are obtained for the demodulator. These requirements are used in the subsequent design chapters. The Doppler shifts are accounted for by using a linear subtraction model. An implementation of the demodulator is developed in MATLAB at first. This implementation is used to demodulate a simulated GFSK signal, induced with noise, time shift and Doppler shift. These files are simulated in attempt to represent the data received from AAUSAT3. The MATLAB implementation verifies the algorithm and ensures that the demodulation is done as specified. Following the MATLAB implementation, the algorithm is implemented in floating point C and finally in fixed-point on a DSP. The purpose of the DSP implementation is to assess the feasibility of implementing the algorithm on an embedded, real-time platform. The following acceptance test verifies that the DSP implementation is working as specified and ready for optimizing identified subsystems. Lastly a conclusion is drawn from the achieved results.

Preface

Many electronic applications have signal processing as an important part of the system. The theme of this semester is Embedded Real-time Signal Processing. The level is a 6th semester project rated to 20 ECTS-points within Electronics & IT at Aalborg University. The target group of this report are supervisors and future developers of AAU satellites such as students and other interested parties within the School of Information and Communication Technology at the The Faculty of Engineering and Science. The central aspects of this project are dealing with design considerations concerning the Software defined GFSK Modulator, implementation in a DSP processor and algorithm development in order to utilize the DSP structure.

The authors would very much like to thank AAU SATLAB for assisting in making this project happen. A special thanks goes to Troels Jessen for extraordinary assistance and ideas.

Reading Guide

The report is structured in four parts. In each part different aspects of the project will be analyzed.

- I Analysis
- II Design
- III Implementation
- IV Appendix

The Analysis will be dealing with definition of the technology which will be developed and therefore it contains the necessary discussions for a division into sub-systems and for the Requirement Specification, which will result in guidelines and instructions for the Design and Implementation parts. The Design part contains development of the demodulation system by use of MATLAB and floating-point C. The implementation part is an implementation on the DSP with corresponding tests.

Appendix

Appendices are found after the main report as part IV and on the attached CD. The appendices contain all source code, an extended analysis, measurement reports and work sheets. The CD also contains a digital version of the report and the web pages used. A complete list of the CD content can be found in Appendix C.3.6. When referred to the CD the following symbol is used:

 /path/to/stuff

All figures, tables and equations are referred to by the number of the chapter they are used in, followed by a number indicating the number of figure, table or equation in the specific chapter. Hence each figure has a unique number, which is also printed at the bottom of the figure along with a caption. The same applies to tables and equations, the latter of which have no captions. Appendices are referred to by capital letters instead of chapter numbers.

Bibliography

At the very end of the main report, a Bibliography is listed which contains all sources of information used in the report. In the Bibliography books are indicated with author, title, publisher and year. Web pages are indicated with author and title. All information sources are referred to by an abbreviation of the authors' name, e.g., [Lat98]. If the sources reference is written just before a final dot, then it represents the source of information for the sentence in question. If the reference is placed after the dot it represents the source of information for the section in question.

Contents

I	Analysis	1
1	Introduction	2
1.1	Motivation	2
1.2	Initial Problem	3
2	Preliminary Analysis	4
2.1	System Front End	4
2.2	DSP Platform	5
2.3	Spacelink Format	8
2.4	Environment of AAUSAT3 Communication	9
2.5	Choice of Demodulator	15
3	Division of Demodulator Into System Modules	17
3.1	Sampler	17
3.2	Channel Filter	18
3.3	Packet Detection	18
3.4	Mid-Frequency Estimation	22
3.5	Time Synchronization	24
3.6	Symbol Decision	29
4	Interim Conclusion	31
4.1	Problem Statement	31
5	Requirements Specification	32
5.1	Interfaces Between Modules	32
5.2	Overall Requirements	33
5.3	Module Requirements	33
5.4	Delimitations	35

II	Design	36
6	Program Flow	37
7	GFSK Simulation in MATLAB	38
7.1	Simulation Description	38
7.2	GFSK Modulator in MATLAB	38
7.3	GFSK Demodulator in MATLAB	41
8	GFSK Demodulator in C	50
8.1	Sampler	50
8.2	Packet Detection	50
8.3	Estimation of Mid-Frequency	50
8.4	Time Synchronization	51
8.5	Symbol Decision	52
8.6	Example Output	52
9	Channel Filter	53
9.1	Filter Type Considerations	53
9.2	Channel Filter Design	55
9.3	Filter Implementation Concerns	58
9.4	Fixed Point Effects	61
III	Implementation	63
10	GFSK Demodulator in Blackfin C	64
10.1	Sampler	64
10.2	Channel Filter	65
10.3	Packet Detection	67
10.4	Mid-Frequency Estimation	68
10.5	Time Synchronization	70
10.6	Symbol Decision	73
11	Acceptance Test of System	80
11.1	Real-Time Summary	80
11.2	Test of Demodulation	81
12	Conclusion	87

IV Appendix	89
A Quotes From AAUSATLAB	90
A.1 Modulation Index	90
A.2 Packet Transmission and Training Sequence	90
A.3 AAUSAT3 Statistical Information about Reception	90
B Hardware Layout of the Front End	91
C Measurement Reports	93
C.1 UHF Packet From Signal Generator	93
C.2 UHF Packet From AAUSAT3	95
C.3 Acceptance Test With Signal Generator	96
Bibliography	101
Appendix CD	102

Part I

Analysis

Chapter 1

Introduction

1.1 Motivation

Aalborg University has an on-going student project which is dealing with satellite development and manufacturing by mostly electronic engineering students. This project has involved the launch of three cubic satellites (cubesats). The last launch was in 2013 where AAUSAT3 [AAUa] was successfully send into Low Earth Orbit (LEO). AAUSAT3 serves to receive AIS packets from ships and transmit these to the ground stations in Aalborg and Greenland. The packets from AAUSAT3 are received using a hardware based radio. Aalborg University is in the process of designing two new satellites, AAUSAT4 and AAUSAT5. AAUSAT4 will be an improved version of AAUSAT3. In order to improve the data transmission and lower the amount of bit errors the approach in this project is to replace parts of the current hardware based radio with a software defined system, which is designed for receiving the packets transmitted from AAUSAT3. Therefore this project aims to develop a software defined GFSK demodulator. Figure 1.1 shows a graph of the downloaded data for the first eight months of AAUSAT3 orbit. The dots each represent a received packet. Red dots are without bit errors and blue dots contains bit errors. By improving the radio the amount of messages with bit errors could be lowered and by that the amount of error correction code and retransmissions can be reduced. When there is more than 200 bit errors the error coding is non-functioning and the packet is then lost (Reference: Appendix A).

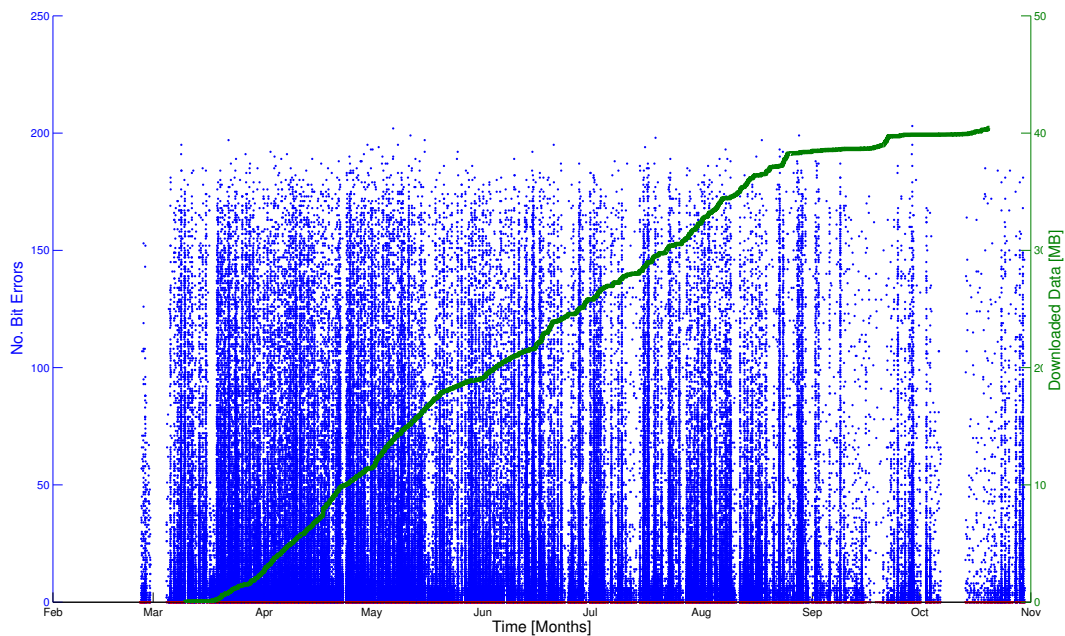


Figure 1.1: Bit Errors of the downloaded data as a function of time.

The SDR system also allows for change of communication protocol as it is implemented in software instead of

hardware. This could be advantageous if implemented in future satellites. The focus and therefore motivation of this study is the signal processing in the Gaussian filtered Frequency Shift Keying (GFSK) demodulator. A recent study has shown that the GFSK demodulator on-board AAUSAT3 has been improved by developing a software defined system which was implemented in the Blackfin BF537 DSP it is therefore of great interest that the developed algorithms in this project is implemented using this DSP.

1.2 Initial Problem

Based on the motivation of this study an initial problem is formed in order to structure the further analysis:

What does it take to design a software defined demodulator for receiving data from AAUSAT3?

To investigate this problem a preliminary analysis will have to clarify the following:

- Existing setup
- RF front end
- Blackfin DSP Platform
- Environment of AAUSAT3 Communication
- Choice of Demodulator

A description of the existing setup allows for implementation with the current receiver. In order to design the demodulator it is needed to know the input this is achieved with an explanation of the RF front end. The Blackfin DSP used in this study is described to prepare for implementation. A description of the environment in which it is possible to communicate with AAUSAT3 contains a description of the GFSK modulation scheme, and the communication channel in which White Gaussian Noise is introduced along with Doppler shift. This prepares the developer for the design of a GFSK demodulator. The choice of demodulator prepares the system to be divided into submodules from which an analysis of each module will make way for the requirements to be specified.

Chapter 2

Preliminary Analysis

2.1 System Front End

This section contains an analysis of the hardware platform used for the ground station radio receiver. This analysis includes an introduction to the RF front-end which receives and downconverts the signals from AAUSAT3. It outlines the hardware and architecture of the Blackfin 537 as well as a short introduction to the fixed-point format. Since the design of the RF front-end and ADC are beyond the scope of this report the focus of the hardware analysis is placed on the interfaces.

2.1.1 Existing Setup of Hardware Based Radio

The communication between satellite and ground station is based on GFSK modulation. At the ground station the hardware based receiver demodulates the signal and passes it on to a PC. This is currently handled by a single chip solution based on a ADF7021 transceiver [?] as shown in Figure 2.1. The antenna is interfacing to an impedance

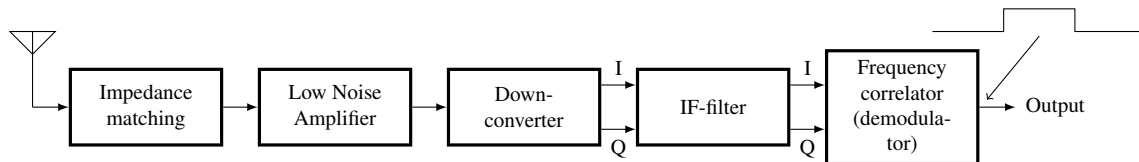


Figure 2.1: Existing setup for the hardware based radio receiver based on the ADF7021

matching block before the low noise amplifier. This is followed by a quadrature down-conversion mixer, splitting the signal into a I - and Q . The IF filter removes negative frequencies and out-of-band RF interferences. The RF signal is downconverted to 200 kHz which the IF filter is centered around. The frequency correlator demodulates the signal into a bitstream. The bitstream is processed by a micro processor to be send by serial communication to a PC. **The analysis**

2.1.2 Front End for the Software Based Radio

For this project the front end consist of the three following parts as shown at Figure 2.2. Impedance matching, signal amplification, and a quadrature down conversion mixer. The down converted signal is more suitable for

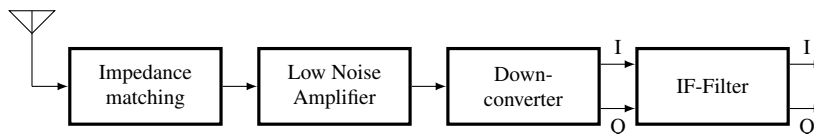


Figure 2.2: Front end for this project

sampling because of a lower sampling rate, and the demodulation can then be done in software on a DSP. Since the demodulation is now done on a DSP, the front end becomes simpler. The ADF7020 is still used, as the front-end for the software based demodulator.

2.2 DSP Platform

This section describes the architecture of the BF537 as well as the operation system available for programming this particular DSP in SATLAB.

As mentioned in the motivation, a Blackfin BF537 is already used in Aalborg University Satellite Laboratory (SATLAB) for development of the receiver for the Automatic Identification System (AIS) module on AAUSAT3. Therefore, an AIS board, similar to that in the satellite, has been chosen as the platform for implementation in this project.

This board consists of:

- Analog Devices ADF7020 ISM band transceiver chip.
- Analog Devices AD7262 ADC.
- Analog Devices Blackfin BF537 DSP.
- 32 MiB memory. The lower 16 MiB is used by uClinux, 16 MiB available to the programmer.
- SD card for data and program storage.
- Miscellaneous other chips for internal CAN communication in the satellite—this is not relevant to this project.

2.2.1 Memory Management and uClinux

The uClinux operating system is developed for systems with no MMU, ultimately meaning that a process run under uClinux cannot increase the size of available memory at runtime as a traditional Linux process [McC]. It is not possible for processes running under the uClinux system to run on a sample-by-sample basis. Instead, a larger block of data is sampled and processed while the next block is fetched—this approach is further described in Section 3.1. Since uClinux is used, the implementation is done in C using GCC by the GNU Blackfin toolchain. The use of assembly planned for optimizing critical regions.

2.2.2 Hardware Layout

The hardware layout of the DSP is briefly introduced in this section and in appendix B. One of the important aspects of the transceiver/ADC interface is that the inputs pins to the ADC is taken from two internal test pins on a test output of the IF filter. The intermediate frequency of these signals is at 200 kHz. The interconnection of the hardware blocks are shown in Figure 2.3. Some more detailed block diagrams and descriptions can be found in the Appendix B.

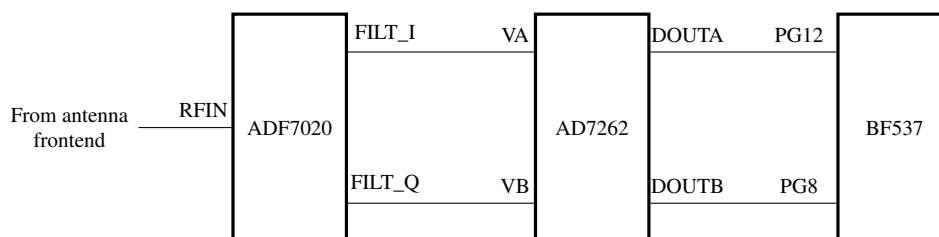


Figure 2.3: Hardware data interconnections. SPI clock and chip select are left out.

2.2.3 Blackfin Architecture

The Analog Devices Blackfin is a 16 bit DSP. As shown on Figure 2.4. It has 16×16 b general purpose registers, that can be combined into 8×32 b registers. It has two ALUs and accumulators and is therefore capable of performing two multiply-accumulate (MAC) instructions per cycle, if two 16 b datastreams combined into 32 b registers as illustrated in Figure 2.4. The accumulators are 40 b wide.

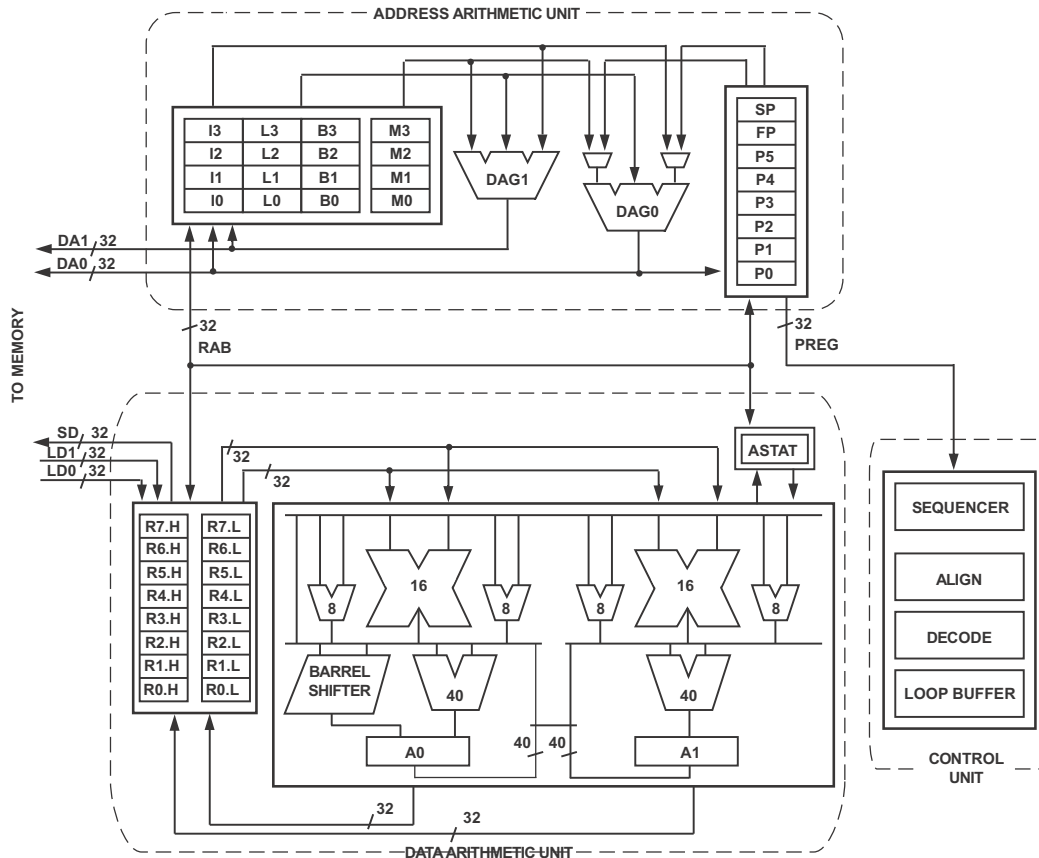


Figure 2.4: The core of the Blackfin BF537.[Anab]

For data access, the DSP has two separate memory lines: One for program memory and one for data memory. This makes it possible to fetch an instruction and a piece of data at the same time. For automatic address generation, the processor is capable of doing circular buffers and bit-reversed addressing. These more advanced techniques are used in the libraries that come with the Blackfin, and can be further exploited by writing in Assembly.

2.2.4 Fixed-Point Representation

In this section, the steps needed to go from a floating-point implementation to a fixed-point implementation will be described. This step is needed when going from a PC floating point implementation to the implementation of the Blackfin.

The whole theory will not be described here, but rather, the practical techniques used when going from floating-point to fixed-point will be described.

When working in floating point, multiplication and addition is scaled automatically. This is because a floating point number consists of a signed *mantissa* with a sign-bit and a constant one-bit above the fraction mark, along with a number of bits below the fraction mark as shown in Figure 2.5. This means, that the mantissa can not be greater than 2—if it does, the exponent will increase.

Like so, the floating point automatically adjusts the exponent to have a constant one above the fraction mark.

In contrast, a fixed point representation does not automatically adjust. Therefore, the programmer must make

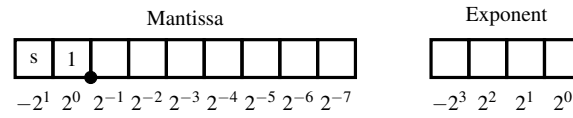


Figure 2.5: Floating point format.

sure the numbers are within the range of the data type used.

There is no actual fraction mark in a fixed point implementation, but one can be assumed by using a few rules when doing computations. Two different numbers can therefore be represented by the same bit sequence. An integer number is said to be in Q0 format. If, for example, the 7 lowest bits are assumed to be below the fraction mark, the format is called Q7.

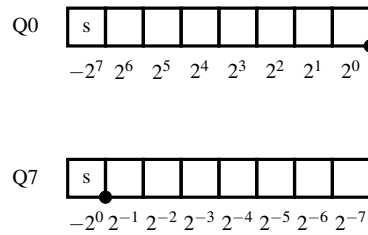


Figure 2.6: Fixed point format in 8 b data type.

To convert a floating point number from floating point to Q_x , the following formula

$$x_{Qx} = \lfloor 2^x \cdot x_{\text{float}} \rfloor \quad (2.1)$$

is used. To go the opposite way,

$$x_{\text{float}} = 2^{-x} \cdot x_{Qx} \quad (2.2)$$

These two corresponds to integer left and right shifting, $\ll$ and $\gg$ in C, respectively.

In Listing 2.1 is an example of these calculations in C. They show the overflow that occurs if the fixed-point number becomes too large to for its data type. It also shows that larger x in Q_x , the more of the original number's decimals will be preserved.

```
float x_float = 3.2225;
int8_t x_Q0 = x_float * pow(2,0); // 3
int8_t x_Q1 = x_float * pow(2,1); // 6
int8_t x_Q2 = x_float * pow(2,2); // 12
int8_t x_Q3 = x_float * pow(2,3); // 25
int8_t x_Q4 = x_float * pow(2,4); // 51
int8_t x_Q5 = x_float * pow(2,5); // 103
int8_t x_Q6 = x_float * pow(2,6); // -50 (overflow)
int8_t x_Q7 = x_float * pow(2,7); // -100 (overflow)

float y_f10 = x_Q0 * pow(2,0); // 3.000000
float y_f11 = x_Q1 * pow(2,-1); // 3.000000
float y_f12 = x_Q2 * pow(2,-2); // 3.000000
float y_f13 = x_Q3 * pow(2,-3); // 3.125000
float y_f14 = x_Q4 * pow(2,-4); // 3.187500
float y_f15 = x_Q5 * pow(2,-5); // 3.218750 (best precision)
float y_f16 = x_Q6 * pow(2,-6); // -0.781250 (from overflow)
float y_f17 = x_Q7 * pow(2,-7); // -0.781250 (from overflow)
```

Listing 2.1: Conversion from floating point to fixed point in C.

In Table 2.1 are shown some basic rules one needs to keep in mind when doing calculations. Assume two numbers in Q_x and Q_y format respectively are used where $x < y$.

The Blackfin BF537 is a 16 b DSP, and it's default word length is therefore 16 b. When programming it, the standard data type is `fract16`, which is a signed 16 b type. This data type is used by the default libraries that come with the Blackfin. When 16 b is not enough, the data type `fract32` is the 32 b equivalent to `fract16`.

Operation	Resulting format	Notes
$Qx \cdot Qy$	$Q(x + y)$	Output should probably be scaled down to Qx or Qy .
$Qx + Qy$	Qx	Qy must first be shifted (down) to Qx . Be aware of overflow!

Table 2.1: Rules when doing calculations in fixed-point numbers.

2.3 Spacelink Format

The protocol for communication with AAUSAT3 is developed at Aalborg University and is called Can Space Protocol (CSP)[AAUb]. This protocol has been developed without any detailed documentation until now. A typical CSP

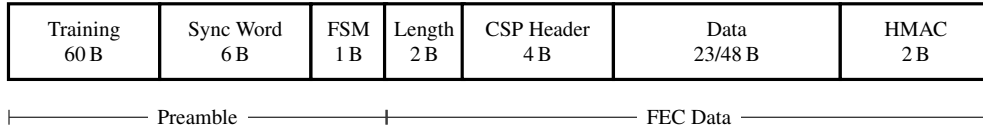


Figure 2.7: CSP packet[AAUb].

packet format is shown in Figure 2.7. In the beginning of a transmission, a sufficiently large training sequence defined by a time period of 200 ms which is 60 B at $R = 2400$ b/s. The training sequence is an alternating sequence of 0's and 1's. Each packet then contains a training sequence. The training sequence makes it possible to synchronize carrier frequency, which changes with Doppler shift for each packet, before actual data is received. A back-to-back

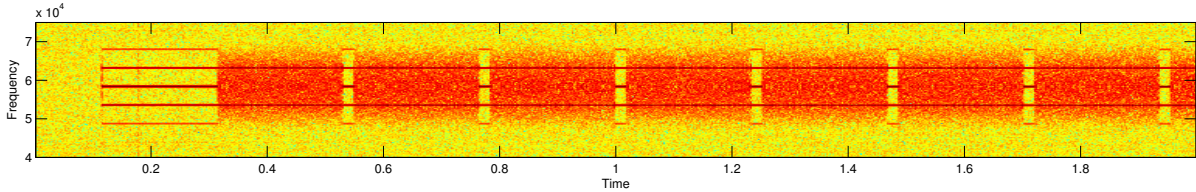


Figure 2.8: Recording of a burst from a pass of AAUSAT3, sampled with 192 kHz..

transmission mode can be set. In this case, packets are sent directly after one another and without losing energy in between (Appendix A). A recording from AAUSAT3, Appendix C.2, shows that the training sequence in between is reduced to only 20 ms in between packets in this mode.

The next thing that is transmitted is a synchronization word. The sync word tells when the training sequence has ended and identifies that it is the satellite ID. For AAUSAT3, the sync word is OZ3CUB.

The data part of a CSP packet can be of two lengths: 31 B and 92 B. The FSM (Frame Size Marker) tells whether a small ($0xA6$) or a large ($0x59$) data packet should be expected.

The data part of a packet is FEC (Forward Error Correction) coded. The data itself is outside the scope of this project, as any decoding is done in the next step from the demodulator. In short, the data field contains the length of the data (23 B or 84 B), a CSP header, the actual data, and a Hash-based Message Authentication Code (HMAC). After the FEC is applied the total length of the FEC Data is either 128 B or 250 B [AAUb].

The technical data of the transmission has been taken from [AAUb]:

- Frequency: 437.425 MHz.
- Frequency deviation: 9600 Hz.
- Modulation scheme: Gaussian 2FSK ($BT_b = 0.5$).
- Modulation index: Variable, typically $h = 1$ (See Appendix A).
- Bit rate: 1.2 kb/s to 19.2 kb/s. Variable, typically 9600 b/s.
- Callsign: OZ3CUB.

- Forward Error Correction:
 - Outer code: Rate $1/2$, $k = 7$ convolutional code.
 - Inner code: $(255, 223)$ Reed-Solomon code.

As mentioned in Section 2.1.2, the received carrier frequency of 437.425 MHz is downconverted by the ADF7020 to make it easier to handle in a DSP. The intermediate frequency is located at 200 kHz.

2.4 Environment of AAUSAT3 Communication

For the developer to be able to design a GFSK Demodulator this section serves to define the environment in which the communication with AAUSAT3 is carried out. The first part is a description and an analysis of the GFSK modulation form. Since the propagation through the air leads to a received signal induced with noise the modelling of noise follows in this section. Finally, an analysis of the Doppler shift from which the signal will suffer at the receiver end is defined the last subsection.

2.4.1 GFSK Modulation Scheme

For the design of the demodulator to be specified it is of great importance to understand the principles of modulation scheme which is utilized for communicating with AAUSAT3. Therefore the GFSK will be described in the following. GFSK is an abbreviation for Gaussian filtered Frequency Shift Keying, or rather GCPFSK—Gaussian filtered Continuous Phase Frequency Shift Keying.

Frequency Shift Keying (FSK) used for transmitting binary data over a wireless channel. The binary data is represented as two different frequencies—one for logic 1 and one for logic 0. The difference between the logic 1 and the logic 0 frequency is determined by the modulation index, h , along with the symbol duration time, T_b

$$\Delta f = |f_1 - f_0| = \frac{h}{T_b} \quad (2.3)$$

where

h is the modulation index. In an AAUSAT transmission, this is typically $h = 1$ but is in fact variable. For the case of MSK (Minimum Shift Keying), $h = 1/2$. See Appendix A.

f_1 is frequency of a logic 1.

f_0 is frequency of a logic 0.

T_b is the symbol duration time, $T_b = 1/R$.

An example of an FSK signal is shown in Figure 2.9.

Signal Representation

Mathematically, an FSK signal can be described as a sinusoid with a frequency, f_c (the carrier frequency), where the frequency is increased or decreased when sending a 0 or 1, respectively. For FSK the modulated signal, $s(t)$ is[HM01]

$$s(t) = \sqrt{\frac{2E_b}{T_b}} \cos[2\pi f_c t + \phi(t)] \quad (2.4)$$

where

E_b is the energy contained in each symbol.

T_b is the symbol duration time.

f_c is the carrier frequency.

$\phi(t)$ is the phase signal that increases or decreases frequency of transmitting ones and zero, respectively: For 1: $\phi(t) = \phi(0) + h\pi t/T_b$, for 0: For 1: $\phi(t) = \phi(0) - h\pi t/T_b$.

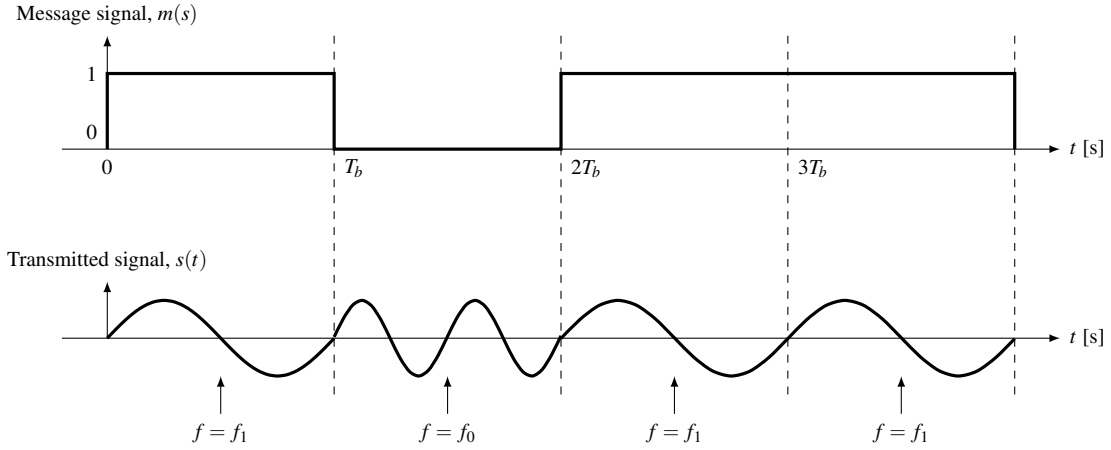


Figure 2.9: Example of FSK modulated signal. Binary message signal on top and modulated signal below. It is seen that the frequency is lower— f_0 —at a 0-transmission than at a 1-transmission— f_1 . Here the modulation index is $h = 1$

$\phi(0)$ is the phase when entering a new symbol. Adding this term assures the phase is continuous. Note that this changes for every symbol—that is, this term in the previous symbol is $\phi(-T_b)$.

When implementing an FSK modulator it may be advantageous to represent Equation 2.4 as simpler sinusoids, splitting up the carrier signal and the symbol determining the signal. Knowing the trigonometric identity $\cos(A + B) = \cos(A)\cos(B) - \sin(A)\sin(B)$, this is [HM10]

$$s(t) = \sqrt{\frac{2E_b}{T_b}} \cos[\phi(t)] \cos[2\pi f_c t] - \sqrt{\frac{2E_b}{T_b}} \sin[\phi(t)] \sin[2\pi f_c t] \quad (2.5)$$

$$= I(t) \cos[2\pi f_c t] - Q(t) \sin[2\pi f_c t] \quad (2.6)$$

where

$I(t)$ is the in phase component, $I(t) = \sqrt{2E_b/T_b} \cos[\phi(t)]$.

$Q(t)$ is the quadrature component, $Q(t) = \sqrt{2E_b/T_b} \sin[\phi(t)]$.

Another convenient notation of the above signals, is complex exponential notation. This notation also splits up carrier and signal. Using Euler's formula, it is seen that

$$\begin{aligned} s(t) &= \sqrt{\frac{2E_b}{T_b}} \cos[2\pi f_c t + \phi(t)] \\ &= \frac{\sqrt{2E_b/T_b}}{2} \left(e^{j[2\pi f_c t + \phi(t)]} + e^{-j[2\pi f_c t + \phi(t)]} \right) \\ &= \left(\frac{\sqrt{2E_b/T_b}}{2} e^{j\phi(t)} \right) e^{j2\pi f_c t} + \left(\frac{\sqrt{2E_b/T_b}}{2} e^{-j\phi(t)} \right) e^{-j2\pi f_c t} \end{aligned} \quad (2.7)$$

Using this notation makes it possible to look at the amplitude and phase of the signal instead of the waveform in trigonometric form. Knowing complex notation it is seen that multiplying with an exponential term changes the frequency, so moving the baseband signal to around the passband (upconverting) is done simply by multiplying with an exponential term. Similarly, downconversion (from passband to baseband) can be done by multiplying with an exponential term with the opposite sign.

When looking at the spectrum of the signal, the amplitude will be the sum of the amplitude of the positive and the negative frequency part or double the amplitude of either part.

$$|s(t)| = 2 \frac{\sqrt{2E_b/T_b}}{2} = \sqrt{\frac{2E_b}{T_b}} \quad (2.8)$$

The phase of the signal is a measure of frequency. Looking at positive frequencies, the phase of the baseband signal is

$$\angle s_{\text{baseband}}(t) = \phi(t) \quad (2.9)$$

and the phase of the passband signal is

$$\angle s(t) = \phi(t) + 2\pi f_c t \quad (2.10)$$

From these equations it is easy to see that to downconvert the passband signal, $2\pi f_c t$ must be subtracted from the phase of the passband signal to obtain the baseband signal. This corresponds to multiplying with $\exp(-j2\pi f_c t)$.

Gaussian Filter

When a symbol changes from, e.g., 0 to 1, the frequency changes from one to another. To smoothen this transition, a filter is added to shape the binary sequence into a continuous curve, stabilizing the frequency variations in time[Rap02].

In GFSK, a Gaussian filter is used. The advantage of the Gaussian filter is that its impulse response has very little overshoot while, at the same time, it has a narrow bandwidth. A Gaussian filter has the impulse response shown on Figure 2.10 and is mathematically represented as[Rap02]

$$h(t) = \frac{\sqrt{\pi}}{\alpha} \exp\left(-\frac{\pi^2 t^2}{\alpha^2}\right) \quad (2.11)$$

where

α is $\sqrt{\ln 2}/\sqrt{2}B$

B is the -3 dB bandwidth, defined by BT_b (the BT product) where T_b is the same as above.

t is the time.

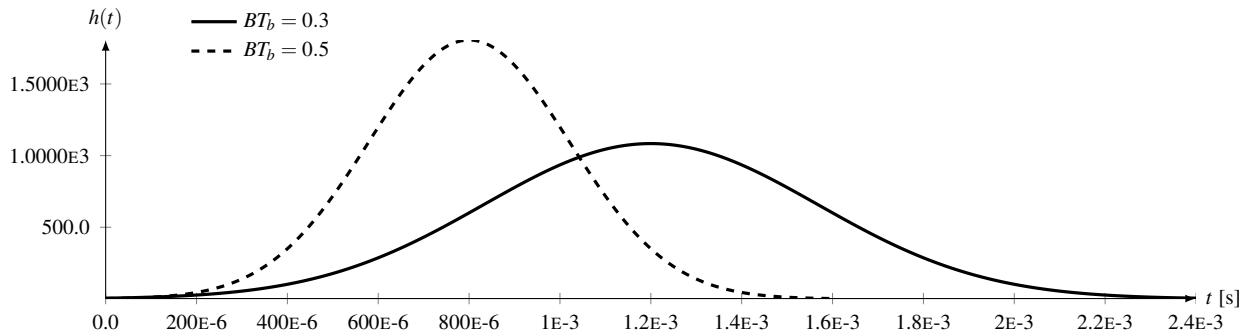


Figure 2.10: Gaussian filter impulse response for different values of the BT product.

From FSK to GFSK

The above equations are for FSK modulation. To go from FSK to GFSK the input sequence only has to be filtered through a Gaussian filter when $\phi(t)$ is calculated.

To calculate $\phi(t)$ for FSK is simple: Just use the formula $\phi(t) = \phi(0) \pm h\pi t/T_b$ from Equation 2.4—for a 1 symbol the positive part is used, for a 0 symbol the negative part is used. However, because a filter is now added, the distinction between 0 and 1 is not this clear. Therefore, another method is used for calculating $\phi(t)$.

The variable $\phi(t)$ for FSK is basically a straight line from $\phi(0)$ with a slope of $h\pi/T_b$. If a bit sequence is represented as a NRZ sequence where a zero symbol is -1 and a one symbol is 1 , this corresponds to integrating and scaling this bit sequence, $m_{\text{NRZ}}(t)$, as shown in Figure 2.11

$$\phi(t) = \phi(0) + \frac{h\pi}{T_b} \int_0^{T_b} m_{\text{NRZ}}(t) dt \quad (2.12)$$

where $\phi(0)$ is still responsible for keeping the phase continuous (note again, that the limits from 0 to T_b mean the beginning and end of *each* symbol—not the absolute beginning of a sequence). Using this definition instead, $m_{\text{NRZ}}(t)$ can be replaced by a NRZ sequence that is filtered through a Gaussian filter, thus, making Equation 2.5 usable for GFSK as well.

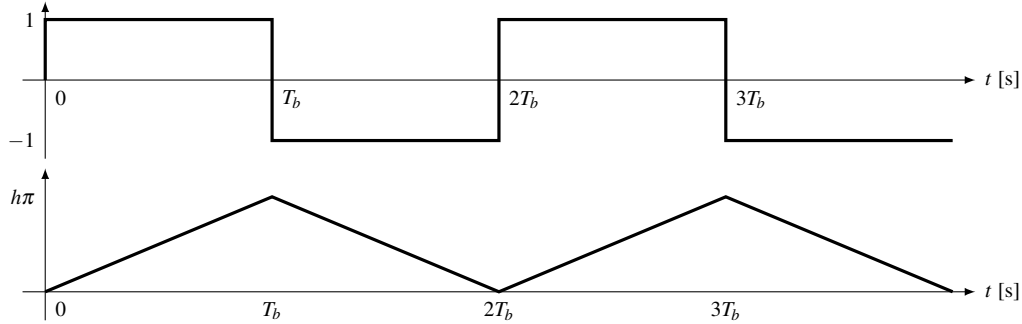


Figure 2.11: To calculate $\phi(t)$ when the input sequence is filtered, an integral and scaling may be used. Above is shown integration of an unfiltered NRZ sequence to show the correlation to Equation 2.4. Above: NRZ signal $m_{\text{NRZ}}(t)$. Below: Integrated and scaled signal $\phi(t)$.

2.4.2 White Gaussian Noise

Signals are exposed to noise when transmitting data on a wireless channel. In satellite communication the noise is thermal noise of the receiver front end [LF07]. Therefore, it is needed to model the noise to be able to simulate the communication link. The noise is often modelled as additive white Gaussian noise (AWGN) [HM10]. This model does not take channel fading, frequency selectivity, dispersion or non-linearity into account. The noise is called white because $w(t)$ has a flat power spectrum with height:

$$S_w(f) = \frac{N_0}{2} \quad (2.13)$$

where

N_0 is the power spectral density of the noise.

The noise is Gaussian distributed with zero-mean. As seen from Figure 2.12, the received signal at the band-pass filter is a sum of the transmitted signal, $s(t)$, and the noise, $w(t)$. It is assumed that the band-pass filter is ideal

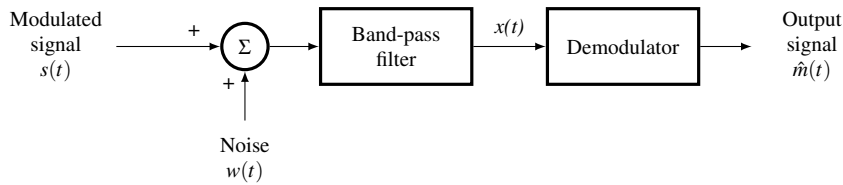


Figure 2.12: Noisy receiver model.

having a bandwidth equal to the transmission bandwidth B_T of $s(t)$ and a mid-band frequency equal to the carrier frequency f_c . The filtered noise $n(t)$ is narrowband and can be expressed as:

$$n(t) = n_I(t) \cos(2\pi f_c t) - n_Q(t) \sin(2\pi f_c t) \quad (2.14)$$

where

$n_I(t)$ is the in-phase noise component.

$n_Q(t)$ is the quadrature noise component.

As seen from Equation 2.14, the filtered noise is modelled in a similar way as the modulated signal itself (as described in Section 2.4.1) having an in-phase and a quadrature component. The filtered signal, $x(t)$, available for demodulation is:

$$x(t) = s(t) + n(t) \quad (2.15)$$

The average power of the filtered noise is equal to $N_0 B_T$ as seen in Figure 2.13. Signal-to-noise ratio (SNR) is

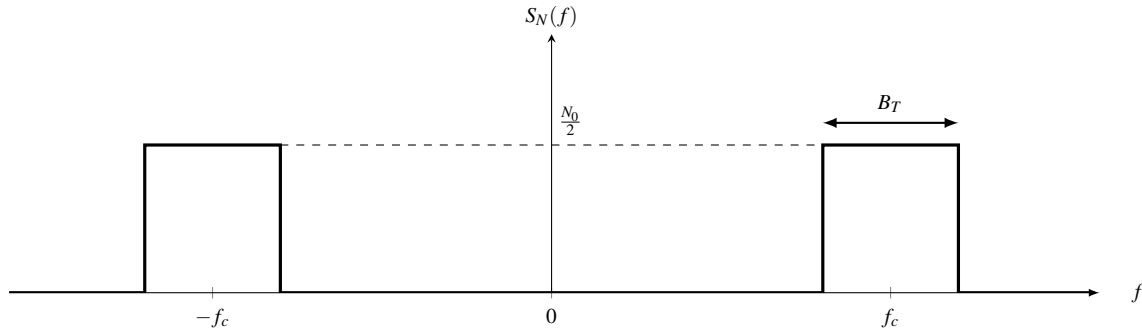


Figure 2.13: Spectrum of band-limited white noise.

defined as the relationship between the average power of the modulated signal, $s(t)$, and the average power of the filtered noise, $n(t)$.

$$\text{SNR} = \frac{P_s}{P_n} \quad (2.16)$$

Therefore, when simulating noise in a communication system, the average power of the noise is decided according to the average power of the signal and a wanted SNR. This makes it possible to test the receiver with respect to a requirement for a minimum SNR at which the receiver is able to decode the signal.

The noise is normally distributed with mean $\mu = 0$ and variance σ^2 also expressed as $W \sim \mathcal{N}(\mu, \sigma^2)$. The variance is given as:

$$\text{Var}(X) = E[(X - \mu)^2] = E[X^2] = \sigma^2 \quad (2.17)$$

This is also the average power of the noise meaning that:

$$P_n = \sigma_n^2 = \frac{P_s}{\text{SNR}} \quad (2.18)$$

Considering a standard normal distribution, $X \sim \mathcal{N}(0, 1)$, where the variance is equal to one. Scaling the variance to the wanted level yields that [Ros09]:

$$\frac{P_s}{\text{SNR}} \text{Var}(X) = \text{Var}\left(\sqrt{\frac{P_s}{\text{SNR}}} X\right) \quad (2.19)$$

The equation above shows that the standard normal distributed noise must be scaled by the square root of the quotient of the average signal power and the wanted SNR to obtain the wanted average noise power.

The standard normal distributed noise is produced in MATLAB using the `randn()` function which generates Gaussian distributed random values. However the random vector needs to be of zero-mean with variance equal to one. This is done as seen in Listing 2.2.

```

1 SNR_dB = 10; % Wanted SNR in dB
2 t = 0:0.001:1; % Time-vector
3 s = sin(2*pi*2.*t); % Test-signal
4 L = length(s); % Length of signal
5 SNR = 10^(SNR_dB/10); % Calculate SNR
6 Ps = sum(s.^2)/L; % Average power of signal
7 w = randn(1, L); % Generate random noise vector

```

```

8 | w = w - sum(w)/L;           % Ensure zero-mean
9 | w = w./sqrt(sum(w.^2)/L);   % Make average power (variance) = one
10 | w = w.*sqrt(Ps/SNR);       % Scale noise to wanted level
11 | s = s + w;                 % Add noise to signal

```

Listing 2.2: MATLAB-script that adds noise with specific SNR to a signal.

Running the script listed above gives a result as seen in Figure 2.14. The noise will of course vary each time the script is used since the noise is generated from random values. It will however be AWGN with zero-mean and an average power of P_s/SNR .

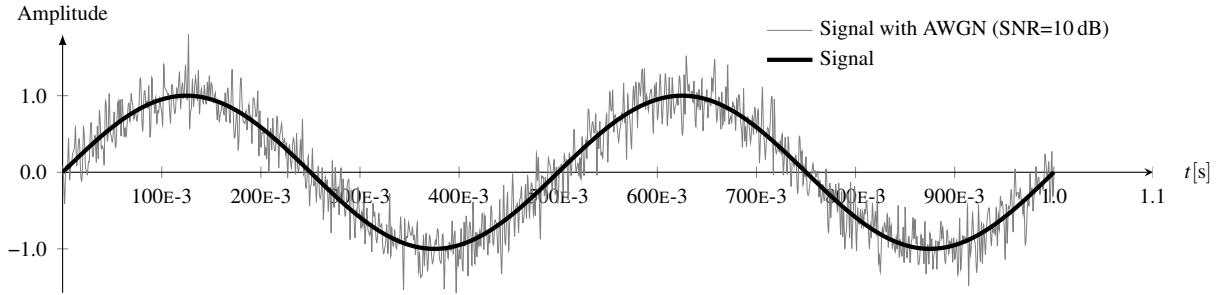


Figure 2.14: Example of signal exposed to AWGN with an SNR of 10 dB.

The addition of noise as described in this section is utilized in the end of Section 7.2 where the channel between modulator and demodulator is to be simulated. As shortly mentioned this model of noise does not take e.g. channel fading into account which can possibly result in errors when implementing the system with a real wireless channel. The AWGN model does however fulfill the scope of this study.

2.4.3 Doppler Shift

When communicating with non-geostationary satellites such as CubeSats, the relative velocity v_{rel} between the transmitting satellite and the receiving ground station will cause a change in frequency at the receiver known as Doppler shift. The sign of v_{rel} and thereby the Doppler shift will be positive when the satellite moves closer to the ground station and negative when moving away. The velocity vectors between the transmitter and receiver is thereby noted v_t and v_r .

The Doppler shift f_{Doppler} can be expressed as seen in Equation 2.20 [Skl01], where the signal wavelength is given by $\lambda = v_{\text{medium}}/f_c$, the propagation velocity, in this case free space and thereby the speed of light noted $c \approx 3.0\text{E}8\text{ m/s}$, divided by the frequency of the wave known as the carrier frequency.

$$f_{\text{Doppler}} = \frac{v_{\text{rel}}}{\lambda} = f_c \frac{v_{\text{rel}}}{c} = f_c \frac{v_t + v_r}{c} \quad (2.20)$$

The velocity component v_t is found by a projection of v_{sat} down on v_t , see Figure 2.15, under the assumption that the satellite height is constant. The velocity of the satellite is 7500 m/s, and the height is 800 km accordingly to [GPR]. This gives the expression

$$v_t = -v_{\text{sat}} \cos(180^\circ - \beta) \quad (2.21)$$

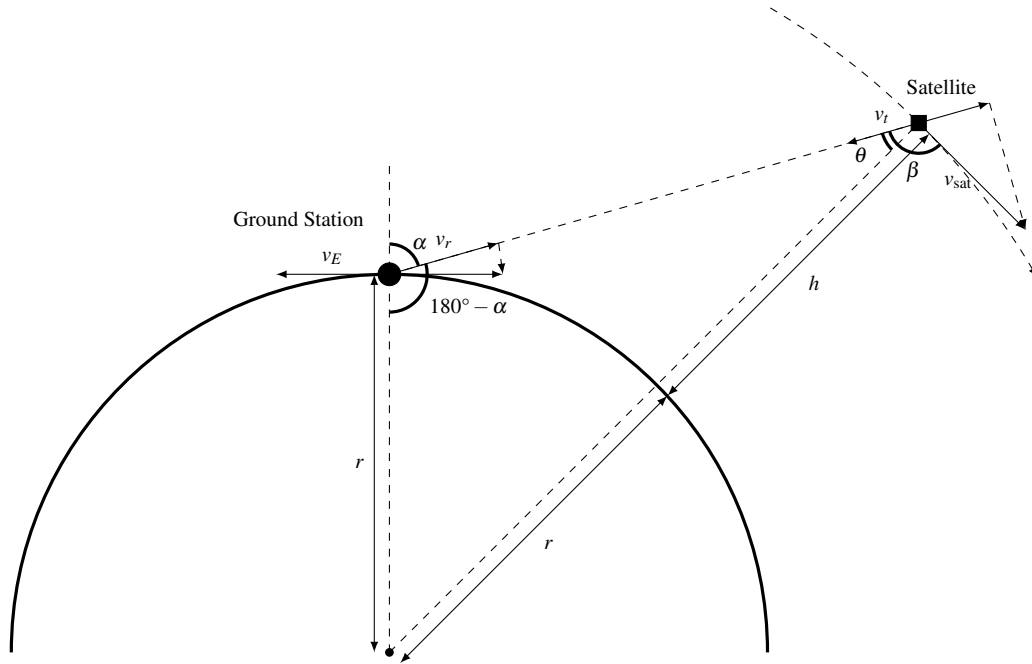
where β is the angle between the velocity vectors v_t and v_{sat} described by

$$\beta = 90^\circ + \theta \quad (2.22)$$

Here θ can be found by the law of sines, given the orbital height h for the satellite and radius of the earth $r = 6371\text{ km}$.

$$\theta = \sin^{-1} \left(r \frac{\sin(180^\circ - \alpha)}{r + h} \right) \quad (2.23)$$

The Doppler shift is evaluated with an angle from -90° to 90° as seen in Figure 2.16. MATLAB script can be found on the CD

Figure 2.15: Illustration of α , v_t , and v_r with the satellite in an arbitrary position.

 /code/matlab_gfsk_workbench/demodulator/dopplershift.m

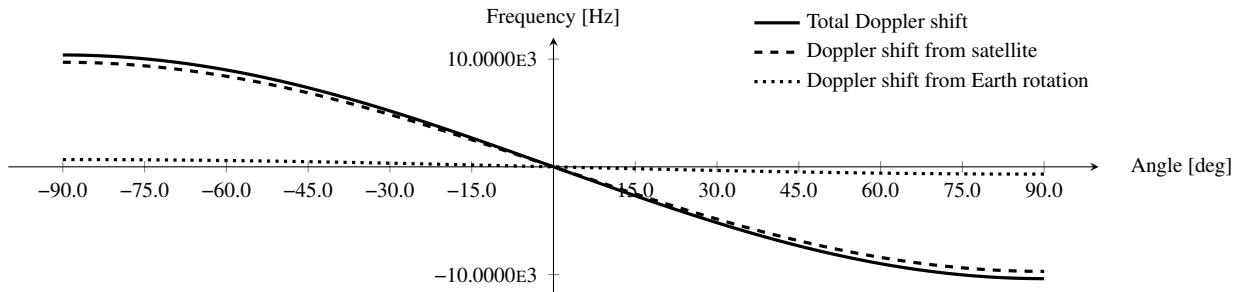
The Doppler shift caused by the satellite is shown in Figure 2.16, with a maximum frequency shift of ± 9.7 kHz.

The Doppler shift caused by the Earth rotation is found likewise. Here v_r is found projecting v_E down on v_r .

$$v_r = -v_E \cos(90^\circ - \alpha) \quad (2.24)$$

Worst case is found when v_E is largest, which is at equator, and the satellite flies directly opposite the earth's rotation. Here v_E is found, knowing the Earth rotates one round per day, which equals $(2\pi 6378 \text{ km}) / (24 \cdot 60 \cdot 60) \text{ s} = 463.8 \text{ m/s}$. The Doppler shift caused only by the Earth rotation is shown at Figure 2.16 with a maximum of 675 Hz and is therefore very small compared to the Doppler shift of the satellite movement.

The total Doppler shift can be seen at Figure 2.16, with a maximum of 10.4 kHz of which the SDR as minimum has to account for.

Figure 2.16: Doppler shift of AAUSAT3 with $v_{\text{sat}} = 7500 \text{ m/s}$ and $h = 800 \text{ km}$.

2.5 Choice of Demodulator

As described in Section 2.4.1 the differentiated phase is also an expression of a frequency change. Therefore the design of a software defined demodulator can be designed by either looking at the phase or the frequency. A brief

analysis of two different designs is compared in the following. This is done as a comparison between the newly developed AIS FSK Demodulator on-board the satellite and an FSK demodulator which uses the phase shift for decision making.

2.5.1 FSK Demodulator by Frequency Analysis

The AIS FSK demodulator samples and filters only the I channel. When a packet is detected, the signal is filtered by a low- and high-pass filter respectively, followed by envelope detectors, as shown in Figure 2.17. The bit decision is made by subtracting the two envelopes, giving a positive or negative value depending on the frequency of the input.

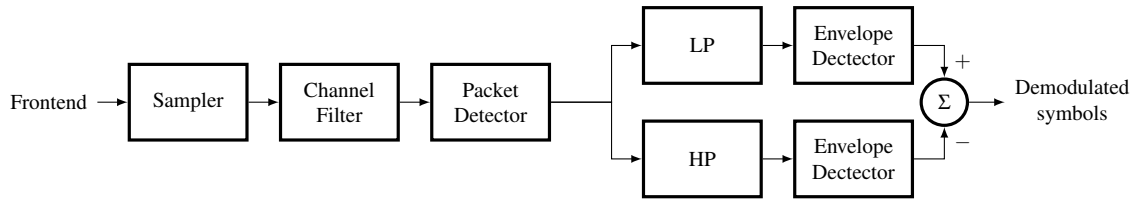


Figure 2.17: Detection of FSK Signals by frequency analysis.

2.5.2 FSK Demodulator by Phase Analysis

Another approach is to design a demodulator utilizing the phase of the complex signal. The solution has been proposed by SATLAB and is depicted at Figure 2.18. The I and Q channel are sampled and filtered, where only one channel is used for packet detection. The phase signal can be calculated using both I and Q . To determine the start of the first bit time synchronization is needed, from whereon the bit decision can be done by comparing the current phase value with the previous as seen in Figure 2.11, where a positive sign corresponds to 1 and a negative sign to 0. It is though important that the phase signal have an insignificant slope when comparing the two values, of which the frequency estimator and frequency translation is used to obtain a phase signal centred around zero.

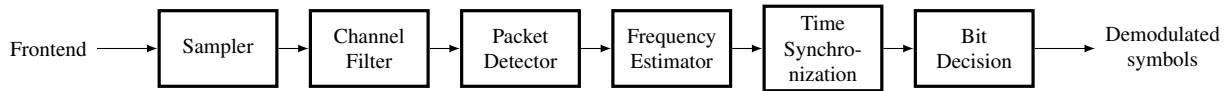


Figure 2.18: Demodulator utilizing phase change.

The software defined solutions are much alike, since the frequency and phase of a signal are two sides of the same coin. Both demodulators have the opportunity of including soft-bit decision, which may be desired for further development. It has already been shown that AIS demodulator works, and it is therefore chosen to demodulate by using the phase. This is also an opportunity to make something that have not been done before, and a possible improvement for the downlink.

This chapter has outlined the existing hardware demodulator and the new interface between the software demodulator and the front-end. The DSP platform of the BF537 has been analyzed along with the uClinux operating system. A specification and analysis of the space link format and aspects when communicating with AAUSAT3 has shown that the received signal will suffer from Doppler shift and noise. The choice of demodulator has been taken which leads to a well defined system which will be divided into sub-systems in the following.

Chapter 3

Division of Demodulator Into System Modules

Now that it has been decided to develop a demodulator which demodulates by utilization of both the I and Q part of the complex signal. It is now possible to define the building blocks of this system. These modules can be developed simultaneously and they will be described on a functional level in the following sections. Firstly, each sub-system will be defined by a short analysis. Some of these will then be followed up with comparisons between different solutions for filling out the defined functionality. The overall modules which are to be defined are:

- Sampler
- Channel Filter
- Packet Detection
- Mid Frequency Estimator
- Time Synchronization
- Bit Decision

The sampler will retrieve the data from the ADC on a block by block basis as described in Section 3.1. When the sampler has finished sampling a block and the data is ready for processing it should be passed through a channel filter. After this, a packet detector algorithm is run on the filtered data. Once the packet detector has located a packet start the Doppler shift should be estimated by the Mid Frequency Estimator. The estimated Doppler shift is used to by the block called Time Synchronization to translate the frequency of the some of the alternating phase to a frequency of 0 Hz and then carry out a time synchronization in order for the symbol decision to know the location of the individual data symbols. The Bit Decision will decide the binary output by knowing the phase shift per bit period and knowing the data rate.

Next, the Sampler will be described in more detail. Through the following the other modules are described.

3.1 Sampler

The sampler will retrieve data from the ADC on a block by block basis, which means that while a block is processed a new block is buffered. Therefore a real-time constraint has to be well defined. This is done in the following. The time it takes to process a block is not allowed to exceed the time it takes before a new block is ready for processing. Making the block size large gives much time and data for processing. However when the block size is increased so is the memory usage which might be troublesome.

From previous student satellite projects a sampling and buffering system has been established on the Blackfin board. The system utilizes the parameters as listed in Table 3.1.

Parameter	Description	Value
f_{clk}	DSP clock frequency	412 000 000 Hz
f_s	Sampling frequency	758 272 Hz
f_{adccclk}	Clock frequency of ADC	25 781 250 Hz
Block size	Amount of data in one block	2 MiB
Complex samples	Complex samples in one block	256 KiS

Table 3.1: Parameters used in the system.

3.1.1 Sample Block Definitions

As described in the data sheet [Anaa] the sample block consists of 20 zeros, 12 in-phase component data symbols, 20 zeros and 12 quadrature component data symbols, which are to be sorted when buffering to memory. Therefore, to achieve a full 16 bit resolution the 12 bit data is shifted four bits left. In other words the last 12 bits of each 32 bit sample block is used as I and Q data, respectively.

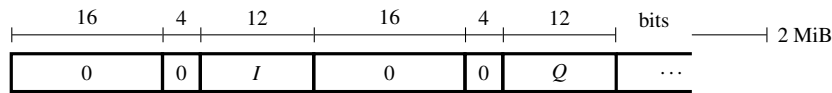


Figure 3.1: Sampling of data [TBT13].

3.1.2 Buffering and Ready Time

Three memory blocks for buffering data from the ADC has each been allocated an area of 2 MiB. In such a block it is therefore possible to store 512 KiB I and Q samples directly from the ADC. Since the data format is 16 bit this makes room for 256 Ki of usable samples, in each of the defined blocks. With a sampling frequency of 758 272 Hz and a bit rate of 2400 b/s a total of 316 samples/symbol are available. With a packet length of 1272 symbols (for a short packet) 401 952 samples are needed to buffer one packet. As only 262 144 samples are available in one buffer this seems unreachable. With a bit rate of 9600 bit/s a total of 79 samples per symbol are available. This gives a total of approximately 100 000 samples/packet.

With M samples the buffer is ready every T_{buf} seconds:

$$T_{\text{buf}} = \frac{M}{f_s} = \frac{256 \text{ KiS}}{758 272 \text{ Hz}} \approx 346 \text{ ms} \quad (3.1)$$

From this follows that the maximum cycles available for processing one block is:

$$\text{max cycles} = f_{\text{clk}} \cdot T_{\text{buf}} = 412 \text{ MHz} \cdot 346 \text{ ms} \approx 142 433 491 \text{ cycles} \quad (3.2)$$

This is effectively the number of cycles available to the demodulation process, if the demodulation is to run in real-time.

3.2 Channel Filter

The pre-processing channel filter is responsible for isolating the wanted channel. The considerations regarding the channel filter is described in the beginning of design of the filter in Section 9.

3.3 Packet Detection

The packet detection has the job of detecting whether the incoming data contains a packet or not. By having this ability, the rest of the system does not have to demodulate constantly, therefore reducing the computational load. As described earlier, the channel filter reduces the noise in the incoming packets. For that reason the packet detection

has its input from the channel filter. A wide variation of methods for detection of packets are available. Some of these are single sliding window, double sliding window, windowed FFTs and cross-correlation with training sequence. In this section, the algorithm used for packet detection is designed and described. This is done as a comparison between the windowed FFT method and the double sliding window method.

3.3.1 Example Using Windowed FFTs

A signal is modulated and exposed to noise as seen in Figure 3.2. It is important that the noise is scaled according to the average power of the packets which does not include the zero-signal. The signal contains approximately 160 000 samples.

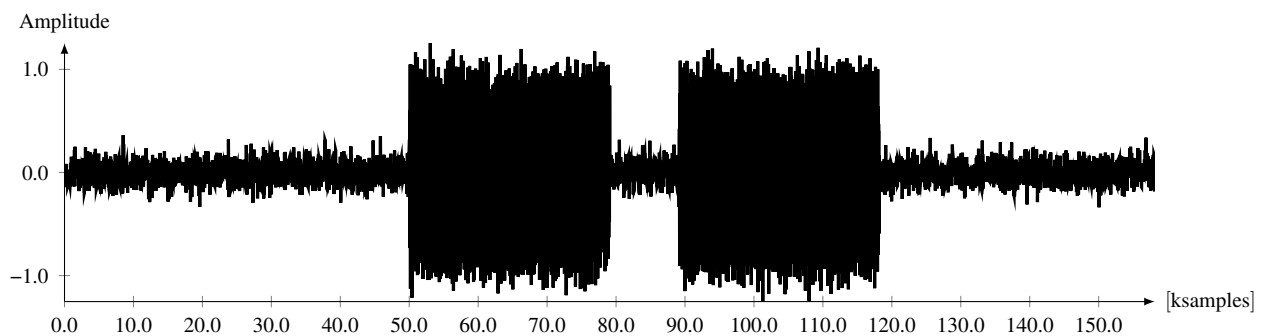


Figure 3.2: Noisy signal which contains two packets (SNR=5 dB).

The buffer is split into windows containing 64 samples. An FFT is calculated for each window. The value and location of the highest value of the FFT is saved. The highest value of each FFT is shown in Figure 3.3. Note from

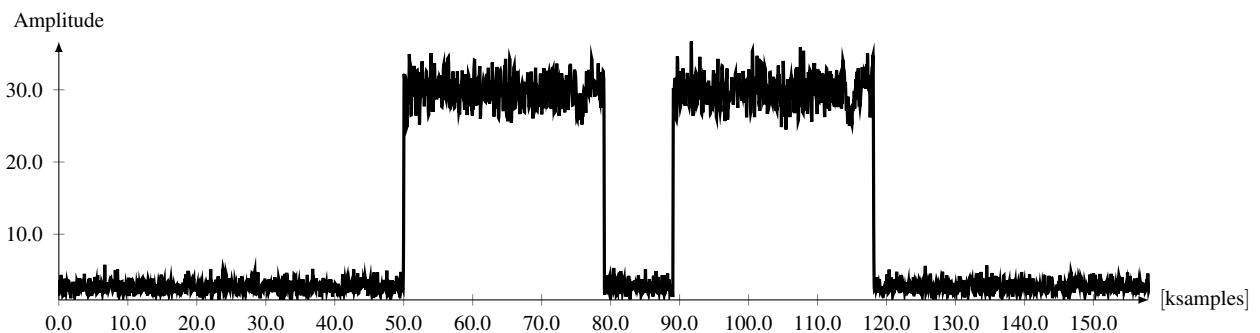


Figure 3.3: Amplitude of maximum of 64-point spaced FFTs.

the figure that when a packet is present the maximum value of the FFT gets high compared to when no packet is present. However as the modulated signal used in this case does not contain any other communicating systems a real world implementation might have relatively high FFT peaks even when a wanted packet is not present. This means that a packet can be detected using the windowed FFTs, but it needs to make sure it is within the right channel. A simple implementation checks whether the maximum FFT value is above a threshold and whether the frequency of the maximum FFT value is within the expected bandwidth. If this is the case then a packet has started.

```

1 % Length of FFTs
2 fft_length = 64;
3 % Generate frequency axis
4 f = 0:fs/fft_length:fs-1;
5 % Windowed FFTs
6 for (i = 1:fft_length:length(s_hat)-fft_length)
7     % Make FFT of current window
8     s_hat_fft = abs(fft(s_hat(1,i:fft_length+i-1)));
9     % Find maximum value and location of the FFT
10    [max_ampl, argmax] = max(s_hat_fft);
11    % Check if maximum value is above threshold

```

```

12     if(max_ampl>fft_length/10)
13         % Check if the maximum value is located at a wanted frequency
14         if(f(argmax)>160e3 && f(argmax)<220e3)
15             % Save location of the start of the packet
16             pstart = k*fft_length -fft_length;
17             break;
18         end
19     end
20 end

```

Listing 3.1: MATLAB-script that detects the start of a packet using windowed FFTs.

The packet detector shown in Listing 3.1 is used to produce the data in Figure 3.3 with the difference that the code stops at the detection of a packet.

The frequency resolution of the 64-point FFT is:

$$f_{\text{res}} = \frac{f_s}{N} = \frac{758272 \text{ Hz}}{64} = 11848 \text{ Hz} \quad (3.3)$$

This is a rather low resolution but as the packet detector only needs an approximate peak frequency this is found to be sufficient. To estimate the computational complexity the theory as described in Section 3.4 is used. The worst case total amount of FFT computations in one block is then:

$$\begin{aligned}
 \text{FFT cycles} &= 2N \log_2(N) \cdot \frac{\text{Block Size}}{N} \\
 &= 128 \log_2(64) \cdot \frac{256 \cdot 1024}{64} \\
 &= 3145728
 \end{aligned} \quad (3.4)$$

This is not much compared to the maximum 142 433 491 cycles available which indicates that the packet detector is able to run in real-time. However the FFTs have a spacing of 64 samples which might be an insufficient resolution when the start of the packet has to be estimated. Sliding the 64-point FFT one sample at a time needs more computations than the DSP is able to handle within one block period. Another disadvantage of this method is that it is difficult to define the threshold, which the maximum value of the FFT has to exceed, in a way such that it correctly sorts out packets no matter SNR.

3.3.2 Using Double Sliding Window

Double sliding window (DSW) is a method that requires less computations to achieve a one-sample sample resolution, since the packet is detected in time domain and not the frequency domain. Furthermore it results in a relationship between the average power of two windows which means that the threshold can be defined in a more proper manner than that of the case in the FFT example above.

As the name suggests the DSW method uses two sliding windows placed next to each other. The energy within one window is:

$$E_W(n) = \frac{1}{N} \sum_{i=n-N+1}^n x_i^2 \quad (3.5)$$

where

x is the sampled signal

n is the number of the current sample

N is the window length

The relationship between the energy of the two windows is:

$$\text{Ratio}(n) = \frac{E_W(n)}{E_W(n-N)} \quad (3.6)$$

By sliding the windows through the samples of the received signal this ratio will get large as the first window contains a packet and the second window contains noise only. When both windows contain some of a packet the ratio will be approximately one. As seen in Figure 3.4 the windows contain old samples which means that when

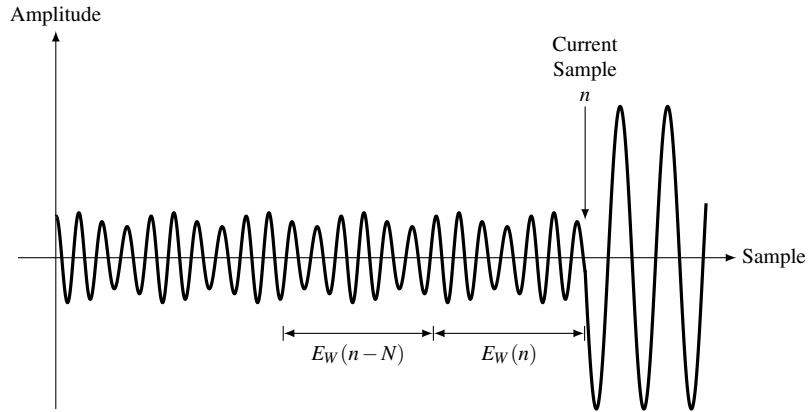


Figure 3.4: Example of signal and placement of current sample and the two windows.

a peak in the ratio is detected the actual placement of the start of the packet is at $n - N$. In the figure the ratio is approximately one as both windows contain what might be thought of as noise. The ratio might vary from one even though both windows contain noise. This variation depends on the size of the windows as larger windows will make the mean of the noise go to zero and the variance to σ^2 as described in Section 2.4.2. Smaller windows of noise do not with certainty have zero-mean and the variance might vary between two windows. This means that the ratio will vary. Thus, to secure a stable ratio when only noise is present large windows are required. However as the windows get larger the packet detector gets more vulnerable to short time intervals between packets. As seen in

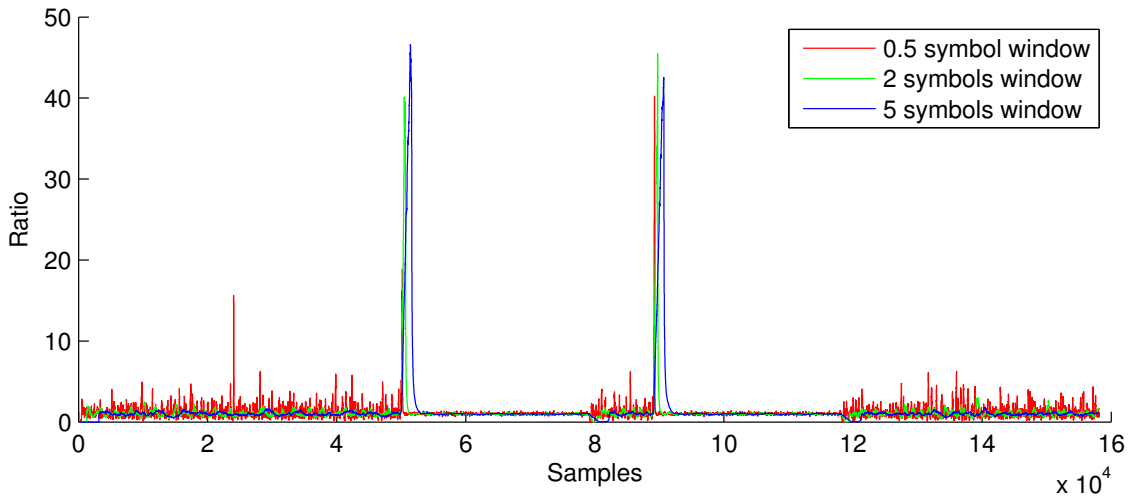


Figure 3.5: DSW with different sizes applied to the received signal shown in Figure 3.2.

Figure 3.5 the ratio during only noise is rather unstable when using window sizes of half a symbol while window sizes of two and five symbols are relatively stable. Thus, choosing $N = 2f_s T_b$ secures a stable ratio which will get high at the arrival of a packet. A smaller SNR will, as shown in Figure 3.6, not make the ratio more unstable during noise. It will however decrease the value of the ratio when a packet starts. This means that the threshold which the ratio has to exceed needs to be set carefully to secure that packets with various SNRs are all detected.

The window as shown in Equation 3.5 is implemented as:

$$E_W(n) = E_W(n-1) - \frac{x_{n-N}^2}{N} + \frac{x_n^2}{N} \quad (3.7)$$

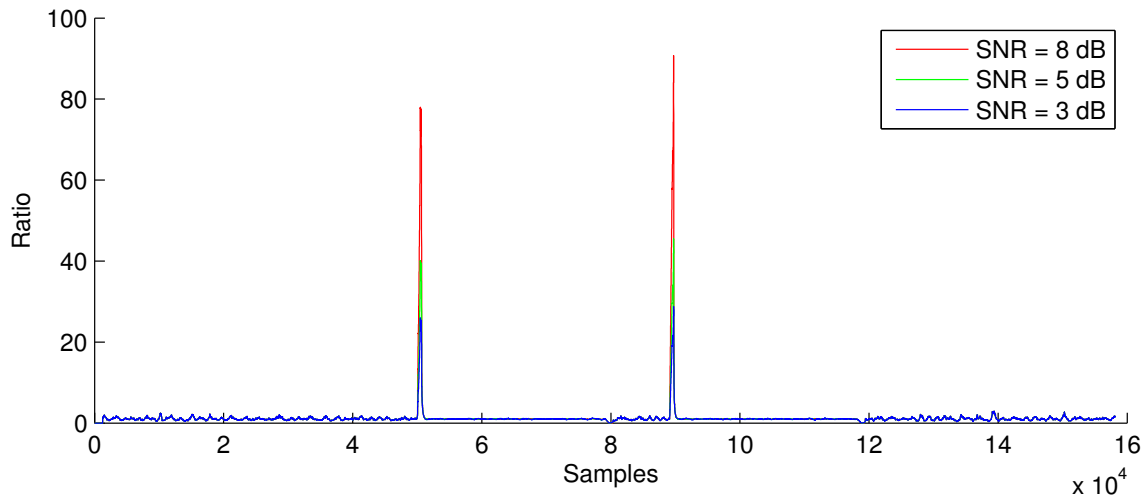


Figure 3.6: DSW with different SNRs using a window size of two symbols.

By adding the energy of the next sample and subtracting the energy from the very last sample, the amount of calculations in each step is reduced considerably. As seen in Listing 3.2 using this method requires that a start-window is calculated before the window starts sliding.

```

1 % Window size
2 N = Tb*fs*2;
3 % Compute the energy of the two start-windows
4 e2 = sum(real(s_hat(1:N)).^2)/N;
5 e1 = sum(real(s_hat(N+1:2*N)).^2)/N;
6 % Slide the windows
7 for i = 2*N+1:length(s_hat)
8     e2 = e2 - real(s_hat(i-2*N+2))^2/N + real(s_hat(i-N))^2/N;
9     e1 = e1 - real(s_hat(i-N+1))^2/N + real(s_hat(i))^2/N;
10    ratio = e1/e2;
11    % Check if ratio exceeds threshold
12    if(ratio > threshold)
13        % A packet is detected at i-N
14        pstart = i-N;
15        break;
16    end
17 end

```

Listing 3.2: MATLAB-script that detects the start of a packet using DSW.

Based on these considerations the double sliding window is implemented and tested in the C program 8.2.

3.4 Mid-Frequency Estimation

As seen from Figure 7.6 the frequency spectrum of a training sequence has some very characteristic peaks. The most interesting peak when dealing with mid-frequency estimation is the one in the middle of the 0-frequency, f_0 , and the 1-frequency, f_1 . The mid-frequency of the received signal might not be equal to the original carrier frequency. This is due to Doppler shift. Thus, to make use of the phase signal the mid-frequency has to be estimated during the training sequence using an FFT. The mid-frequency can be estimated from the FFT using one or more peaks. Using more peaks results in a more robust solution.

3.4.1 Real-Time FFT

It is important to make sure that the FFT, which is used to estimate the mid-frequency, does not occupy too much of the available computation time. This is due to the fact that it will make the algorithm impossible to run in real-time on the DSP. Therefore a trade-off between accuracy and execution time has to be made.

An important factor in the accuracy and the amount of used computation time is the number of points, N , used

in the FFT. The resulting frequency resolution, f_{res} , is given as:

$$f_{\text{res}} = \frac{f_s}{N} \quad (3.8)$$

The discrete Fourier transform is given as:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}, \quad 0 \leq k \leq N-1 \quad (3.9)$$

where

W_N is the twiddle factor $W_N = e^{-j(2\pi/N)}$

The FFT takes advantage of the symmetry and periodicity of W_N^{kn} . It is computed using butterflies as explained in [OS10]. In Figure 3.7 such a butterfly is shown. The butterfly is scaled to prevent overflow in the fixed point numbers. Note that the twiddle factor is simply noted as W as the purpose of the figure is to show the necessary steps within one butterfly and not the exact computations of a full FFT. A total of $\frac{N}{2} \log_2(N)$ butterflies have to be

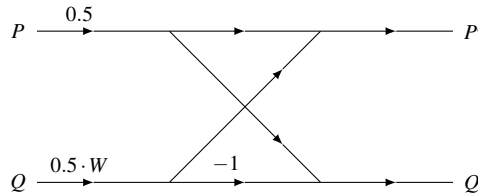


Figure 3.7: Simplified butterfly computation scaled to prevent overflow.

computed for an N-point FFT. One butterfly consists of the two equations:

$$\begin{aligned} P' &= \frac{1}{2}(P + QW) \\ Q' &= \frac{1}{2}(P - QW) \end{aligned} \quad (3.10)$$

As seen from the equations the butterfly contains one complex multiplication and two complex additions/subtractions. The complex multiplication is expressed as:

$$\begin{aligned} QW &= (Q_r + jQ_i)(W_r + jW_i) \\ &= (Q_r W_r - Q_i W_i) + j(Q_i W_r + Q_r W_i) \end{aligned} \quad (3.11)$$

where

X_r is the real part of the number X

X_i is the imaginary part of the number X

This means that the two equations are calculated as:

$$\begin{aligned} P' &= \frac{1}{2} \left([P_r + (Q_r W_r - Q_i W_i)] + j [P_i + (Q_i W_r + Q_r W_i)] \right) \\ Q' &= \frac{1}{2} \left([P_r - (Q_r W_r - Q_i W_i)] + j [P_i - (Q_i W_r + Q_r W_i)] \right) \end{aligned} \quad (3.12)$$

A total of four multiplications and eight additions has to be computed. Scaling by a half does not cost anything since it can be done by shifting the bits one to the right.

The BF537 is able to perform two 16-bit additions in one cycle [GK07]. The same applies to multiplications and MACs. From this follows that one butterfly can be computed in four cycles. This means that the cycles needed to compute an N -point FFT on a BF537 is at least (assuming that the FFT on the BF537 is fully optimized):

$$\begin{aligned} \text{FFT cycles} &= \frac{N}{2} \log_2(N) \cdot \text{butterfly} \\ &= 2N \log_2(N) \end{aligned} \quad (3.13)$$

Lastly, the cycles used for the FFT is compared to the total amount of cycles available:

$$\text{FFT usage} = \frac{\text{FFT cycles}}{\text{max cycles}} \cdot 100 \quad (3.14)$$

It should be noted that the calculations do not take e.g. time used for data transport into account. That is, these equations are rough estimates and should be used only as guide lines of the properties of the FFT.

In the table below are the properties of various N -point FFTs with the block as described in Section 3.1.

N [samples]	f_{res} [Hz]	T_{buf} [ms]	max cycles [kcycles]	FFT cycles [kcycles]	FFT usage [%]
256	2962	346	142.433	4	0.003
512	1481	346	142.433	9	0.006
1024	740.5	346	142.433	20	0.014
2048	370.3	346	142.433	45	0.032
4096	185.1	346	142.433	98	0.069
8192	92.6	346	142.433	213	0.150
16384	46.3	346	142.433	459	0.322

Table 3.2: Properties of the FFT when N varies.

According to the results above a 16 384-point FFT with a frequency resolution of 46.3 Hz will use 0.3 % of the cycles available. Therefore the size of the FFT does not seem critical to the real-time aspect of the implementation.

3.5 Time Synchronization

The time synchronization block is the module between the mid-frequency estimator and the bit decision. The estimated mid-frequency is used by this module to translate the frequency of the alternating phase of the training sequence to a frequency of 0 Hz, followed by a time synchronization so that it is possible to pass a time marker for the bit decision. The details of this functionality will be described in the following section. Firstly, some frequency translation techniques are analyzed which can be used to downconvert the signal. These are followed by an analysis of two methods for the actual time synchronization. Finding the first extremum is one of the methods. The other is cross-correlation with a pre-computed training sequence.

3.5.1 Frequency Translation

The phase of the received signal will as explained have a slope because of the carrier. For the demodulator to be able to decide the binary values of the signal $s(t)$ it is necessary to downconvert the signal from the interval of the mid-frequency down to 0 hertz. When the signal has been downconverted calculation overflows are easier avoided and the difference in the phase $\phi(t)$ is easier to detect by bitwise analysis. In order to carry out the necessary down conversion one can apply different methods to do so. Some of the applied ones in this project will be described in the following.

Methods of Down Conversion

When translating the frequency of a signal, it is basically a trade-off between precision in frequency and computational complexity. For inducing frequency translation to $f_s/2$ and $f_s/4$ simple techniques can be applied such as

decimation or data assignment and signal routing. For a more precise translation to 0 hertz it is possible to multiply by a complex exponential. The last method studied is a model for linear subtraction. These techniques will be described in the following, mainly in complex domain as already introduced in Section 2.4.

Complex Exponential Multiplication

As carried out in the MATLAB Demodulator in Section 7.3.6 a precise translation to 0 hertz is applied by the frequency shift property when doing multiplication with complex exponentials. When the frequency of interest is the mid-frequency f_c the complex signal $s(t)$ is described as:

$$s(t) = e^{j\omega t} = e^{j2\pi f_c t} \quad (3.15)$$

This signal is downconverted to DC by applying the estimated mid-frequency $f_{c,est}$ with the assumption that $f_{c,est} = f_c$:

$$s_{DC}(t) = e^{j2\pi f_c t} \cdot e^{-j2\pi f_{c,est} t} = e^{j2\pi(f_c - f_{c,est})t} = e^0 = 1 \quad (3.16)$$

And thereby the signal is now located at 0 hertz and the phase is easily analyzed as shown in the MATLAB Demodulator. In the Blackfin DSP this method is not appropriate due to the fixed point calculations and the dependency of the math library.

Mixing Scheme for Inducing Frequency Translation to $f_s/4$

With a sample rate at $f_s = 758272$ Hz the carrier frequency is located in the interval around $f_s/4$. Without Doppler shift the carrier is at $f_c = 200$ kHz which implies that it could be useful to translate by $f_s/4$. This can be done by looking at the quadrature Q and in-phase I parts of the signal. This description is shown in the discrete domain. The intermediate signal of interest is defined as.

$$s_n = I_n + jQ_n \quad (3.17)$$

In order to down convert by $-f_s/4$ the I and Q vectors need to rotate samplewise in the clockwise direction by $-f_s/4 = -\pi/2$. This is shown in Figure 3.8 and shown in the following equation.

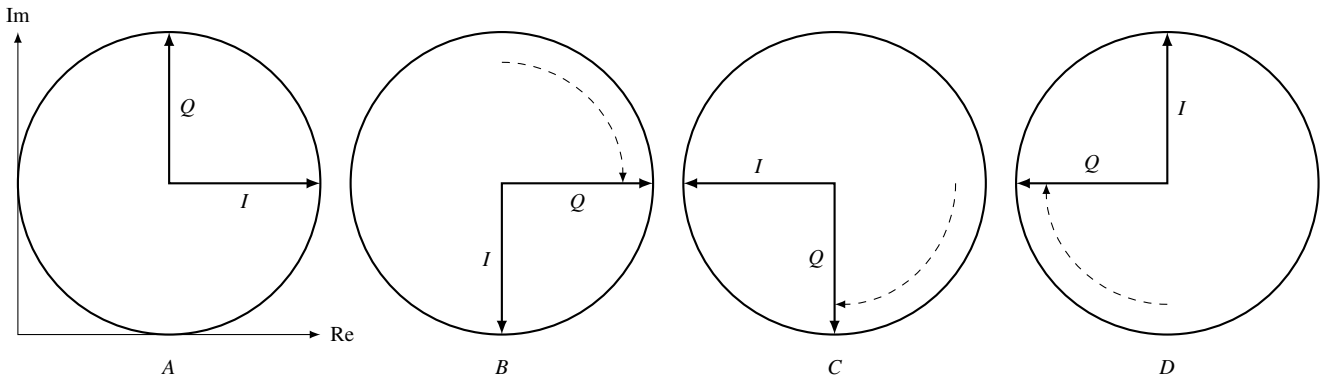


Figure 3.8: Downconversion by $-f_s/4$ visualized with Q and I vectors in unit circle

$$\begin{aligned} s_0 &= I_0 + jQ_0 \\ s_1 &= Q_1 - jI_1 \\ s_2 &= -I_2 - jQ_2 \\ s_3 &= -Q_3 + jI_3 \end{aligned} \quad (3.18)$$

This algorithm will then be repeated for the full sequence which it is desired to down convert. Therefore the downconversion is accomplished merely with data assignment and signal routing without doing any multiplications [Lyo12].

This method differ from the other methods explained as the signal is downconverted by a fixed amount. It does not depend on the Doppler shift which makes it a more imprecise method. The resulting phase signal after applying this method will still have a slope caused by the difference between $f_s/4$ and the actual mid-frequency.

Frequency Translation by Linear Subtraction

The last method described for frequency translation in this study is the one implemented in the Blackfin as described in Section 10.5. It differs from the other methods as it requires that the phase of the signal is calculated before the frequency is translated. It basically corresponds to downconverting by multiplication with a complex exponential term as described above, Section 3.5.1 but now the downconversion is done directly on the phase/time instead of in the amplitude/time domain. Because the phase is computed anyway, the computations might as well be done here. The calculations in the phase domain is lighter than in the amplitude/time domain because frequency shifting is done by simple *subtraction*.

When looking at the phase of the signal the slope, which can be calculated using the estimated mid-frequency, has to be subtracted. It is known that the first part of a packet is training sequence. The training sequence is an alternating sequence which is already utilized for estimating the mid-frequency $f_{c,est}$. The phase $\phi(t)$ is described in Section 2.4.1 as:

$$\phi(t) = \phi(0) \pm \frac{h\pi}{T_b} t \quad (3.19)$$

This phase signal is an alternating signal in the 60B training sequence. It has a slope, a , of:

$$\angle s(t) = \phi(t) + a \cdot t, \quad a = 2\pi f_{c,est} \quad (3.20)$$

This slope will samplewise be ($t = 1/f_s$):

$$a \cdot t = \frac{2\pi f_{c,est}}{f_s}, \quad t = \frac{1}{f_s} \quad (3.21)$$

This procedure realizes a simple downconversion by utilizing the following algorithm:

$$\phi_{n,down} = \phi_n - a \cdot n = \phi_n - \frac{2\pi f_{c,est}}{f_s} \cdot n \quad (3.22)$$

As seen in Figure 3.9 this method removes the slope caused by the carrier and the Doppler shift.

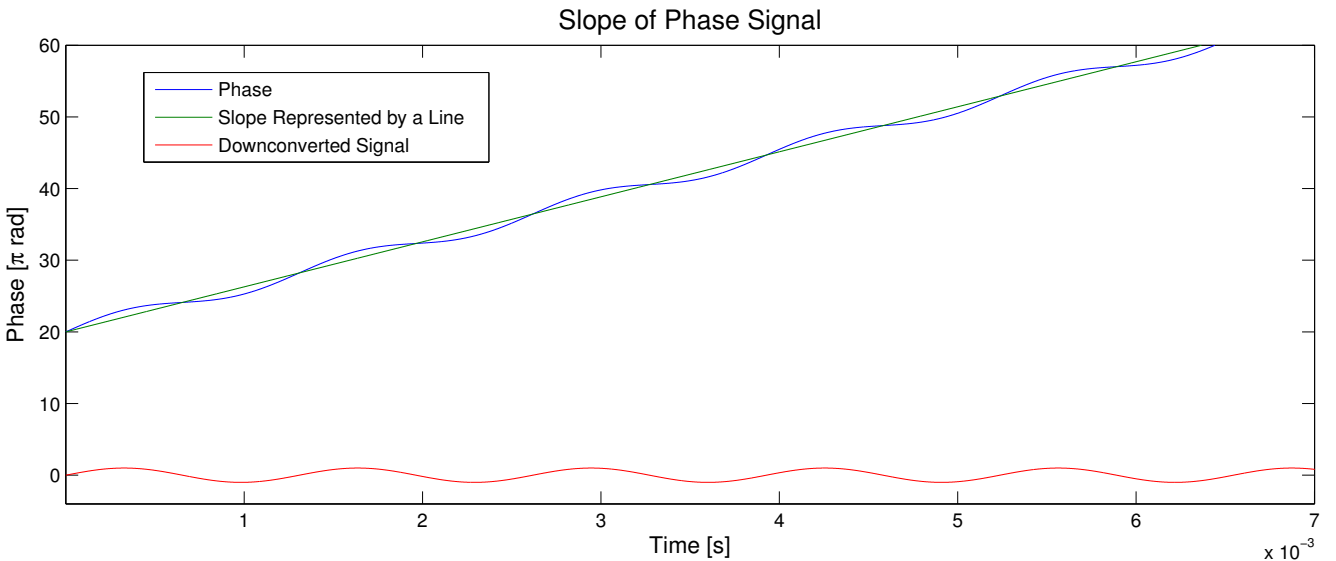


Figure 3.9: Model of the principles in the linear subtraction model

A symbolwise downconversion ($t = T_b$) instead of a samplewise is of the form:

$$\phi_{n,\text{down}} = \phi_n - 2\pi f_{c,\text{est}} T_b \cdot n \quad (3.23)$$

The equation above is used when deciding symbols as will be explained in Section 3.6

3.5.2 Finding First Extremum

When estimating the placements of symbols in the signal the downconverted phase of the signal is used. It is obtained using either of the methods described in the section above (Section 3.5.1). Assuming that the phase of the signal has no slope because of the carrier, as a result of the frequency translation, a simple method for time synchronization is to find the first extremum. As the start of the packet is training sequence the phase of the signal used in the time synchronization will contain maxima when a one is transmitted followed by a minima when a zero is transmitted. Thus, finding the first extrema results in the first available one or zero in the training sequence.

However, if the phase of the signal has a slope because the mid-frequency estimation results in a not precise enough result, this method will be inaccurate. Another problem arises if the phase of the signal is noisy. In this case the extremum found might be slightly off, resulting in a possibly erroneously time synchronization.

3.5.3 Cross-Correlation

Cross-correlation is a technique for determining a point of maximum similarity between two signals, i.e., how much one of the signals must be moved for the two signal to be as similar as possible. Because the phase is altering as shown in Figure 3.10a, it looks a lot like a triangle wave with a distance of T_b between its extrema. The extrema of the phase signal can therefore be found by generating a such triangle wave and cross-correlating it with the phase signal. Using a triangle signal will not result in the ideal cross-correlation since the FSK signal is Gaussian filtered. It will however have the characteristics as wanted.

The cross-correlation function, $\psi_{y_1 y_2}(\tau)$, is for real numbers defined as [Lat98]:

$$\psi_{y_1 y_2}(\tau) = \int_{-\infty}^{\infty} y_1(t) y_2(t + \tau) dt \quad (3.24)$$

where

$y_1(t)$ is the first signal—the triangle wave.

$y_2(t)$ is the second signal—the phase signal.

The cross correlation function shows how well the two functions, $y_1(t)$ and $y_2(t)$ align when $y_2(t)$ is advanced by τ . The value of τ where the maximum of $\psi_{y_1 y_2}(\tau)$ is found is the point where $y_1(t)$ and $y_2(t)$ are most similar. The cross correlation between triangle wave and the downconverted phase is shown in Figure 3.10c.

The cross-correlation function can be evaluated in $0 \leq \tau \leq 2T_b$ because the first phase minimum should be located within this interval. Assuming the triangle signal starts at a minimum, the minimum of the phase signal is located at

$$t_0 = \arg \max \psi_{y_1 y_2}(\tau) \quad (3.25)$$

Moving the phase signal according to t_0 is shown in Figure 3.10e. As seen from Figure 3.10f the cross-correlation works even if the mid-frequency estimates a frequency which is slightly of the actual mid-frequency. However the maximum of the cross-correlation is lower when the phase signal has a slope as seen from Figure 3.10c and Figure 3.10d.

MATLAB and the Blackfin DSP libraries both contain functions for doing cross correlation. An example of using MATLABs `crosscorr()` function found on the CD:

 /code/matlab_correlation/

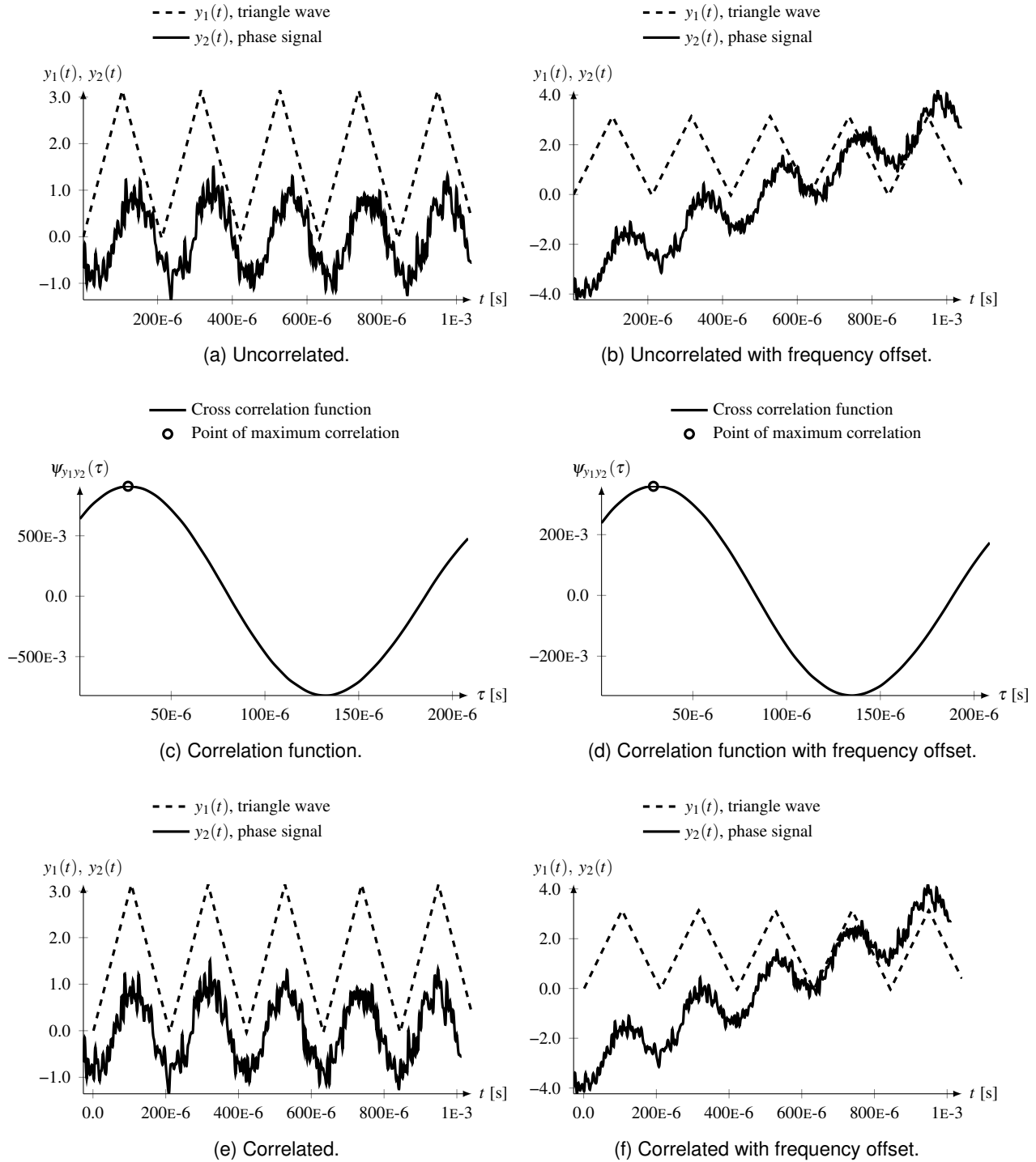


Figure 3.10: Cross-correlation of downconverted phase signal in left column. Cross-correlation of downconverted phase signal with a mid-frequency estimated 1 kHz to low in the right column. The phase signal is of a packet received from AAUSAT3 (see Appendix C.2)

3.6 Symbol Decision

As a result of the mid-frequency estimation and the time synchronization it is possible to decide the value of each symbol in the signal. The symbols are equally spaced with T_b .

This study deals with a modulation index, h , equal to one as described in Section 2.4.1. This is of great interest when having to decide which symbols are present. If the signal is not downconverted to 0 Hz the carrier will add a positive phase change of $2\pi f_c T_b$ after T_b seconds as shown in Equation 2.10. This is neglected in the following. Hereby only looking at the contribution from the symbols. With a modulation index $h = 1$ a phase change of π will be present after T_b when transmitting a one. Similarly a zero will result in a phase change of $-\pi$ after T_b seconds. With $h = 0.5$ the phase change will be $\pm\pi/2$. It can be concluded that this difference is significant to the decision by looking at the unit circles in Figure 3.11. As seen in the figure the difference between a one and a zero after T_b

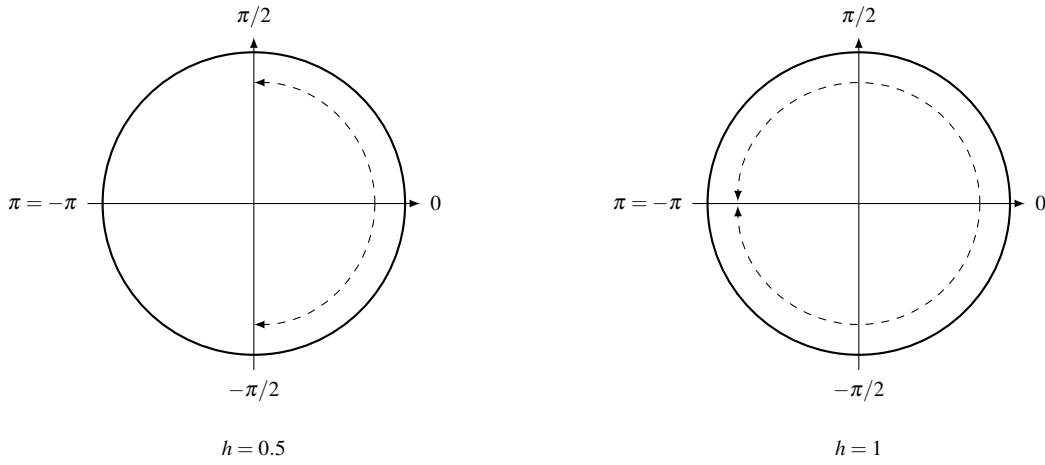


Figure 3.11: Phase change after T_b when transmitting a zero or one symbol. Note that the phase does not necessarily start at zero and that the contribution from the carrier is neglected.

seconds at $h = 0.5$ is π . However with $h = 1$ there is a difference of 2π after T_b seconds. This means that if the phase is not unwrapped the decision after T_b will be troublesome. Thus, when using $h = 1$ the decision must be made after $T_b/2$ (if the phase is not unwrapped) where the phase change will be approximately $\pm\pi/2$.

Using this method, the value of a symbol, $\hat{m}_n$, can be estimated using only the phase signal. For $h = 0.5$ this can be done as follows:

$$\hat{m}_n = \begin{cases} 1 & \text{if } \hat{\phi}_{n+1} - \hat{\phi}_n > 0 \\ 0 & \text{if } \hat{\phi}_{n+1} - \hat{\phi}_n < 0 \\ \text{undefined} & \text{otherwise} \end{cases} \quad (3.26)$$

where

$\hat{\phi}_n$ Is the phase evaluated at the beginning of the current symbol.

$\hat{\phi}_{n+1}$ Is the phase evaluated at the end of the current symbol.

For $h = 1$:

$$\hat{m}_n = \begin{cases} 1 & \text{if } \hat{\phi}_{n+0.5} - \hat{\phi}_n > 0 \\ 0 & \text{if } \hat{\phi}_{n+0.5} - \hat{\phi}_n < 0 \\ \text{undefined} & \text{otherwise} \end{cases} \quad (3.27)$$

where

$\hat{\phi}_n$ Is the phase evaluated at the beginning of the current symbol.

$\hat{\phi}_{n+0.5}$ Is the phase evaluated half the way through the current symbol.

Note that the method described above utilizes hard decision. Another method for decision is soft decision which results in a probability that the symbol has a given value.

3.6.1 Worst Case Symbol Decision

This section will describe the situation when the estimated mid-frequency is not precise. One situation is that the estimated mid-frequency is lower than the actual mid-frequency. This means that the phase of the signal, downconverted using the estimated mid-frequency, has a positive slope. In this case chances are that zeros are erroneously decided to be ones. Using the hard decision as described above and $h = 1$ the largest slope allowed is when the symbolwise slope is not higher than π . This means that the contributions from the zero and the slope cancels out giving the undefined state. This corresponds to an error in the mid-frequency of (see Equation 3.23):

$$\pi > 2\pi f_{\text{err}} T_b \quad \Rightarrow \quad f_{\text{err}} < \frac{1}{2T_b} \quad (3.28)$$

With $T_b = 1/9600$ seconds the error is not allowed to be equal to or exceed 4800 Hz. Like wise the error is not allowed to be equal to or exceed -4800 Hz since this would result in ones decided to be zeros. This worst case scenario assumes perfect time synchronization and no noise disturbances.

The system has now been divided into sub-systems which can be developed at the same time once the interfaces are well defined. Some solutions have been proposed and compared in this chapter and it is therefore a problem statement is defined in the next chapter and this is followed by requirement specification for the full system.

Chapter 4

Interim Conclusion

During the Problem Analysis the basis for the following development and the choice of the design have been described. In order to develop a software defined demodulator, the existing hardware solution has been investigated along with the Blackfin Processor which defines the data format and states that the use of fixed point signal processing is essential when developing and implementing the SDR Demodulator, and also it has been chosen to interface to the front-end by sampling the I and Q signals. It is investigated that the channel frequency spectrum alters due to Doppler shift and the channel filter has to cover this resulting range. It is from the same chapter shown that the induced noise of the received signal can be modeled with white noise. It is from an analysis of the CAN Space Protocol defined when the time and frequency correlation procedures is to be carried out.

Finally, the platform that has been chosen for implementation will be the DSP Blackfin 537 Processor in close corporation with the uClinux implemented as the operating system in the DSP.

4.1 Problem Statement

Now that the existing system has been analyzed and the following problem statement serves to improve this for implementation:

How is a software defined GFSK demodulation designed in the Blackfin DSP?

The design involves the following parts:

- Simulation of Modulator and Demodulator in MATLAB as a proof of concept in non real-time, floating point.
- Designing the sub-systems which are defined in the analysis.
- Designing the demodulator in floating point C.
- Implementing the demodulator in fixed-point on the DSP.

Chapter 5

Requirements Specification

In this chapter the requirements specification for the demodulator is presented. This leads to subsystem requirements which will be used in the following design process. It contains the requirements to the overall system as well as for every subsystem. The requirements to the overall system are obtained from the previous chapter, whereas the requirements for the subsystems are derived from internal interfaces and also from simulations in 7.3.

5.1 Interfaces Between Modules

This section defines the signals and protocols between the different subsystems. This is done to clarify the interfaces in order to define the requirements to each subsystem. These interfaces are specific to the Blackfin implementation, and may vary in the intermediate design steps in MATLAB and PC C.

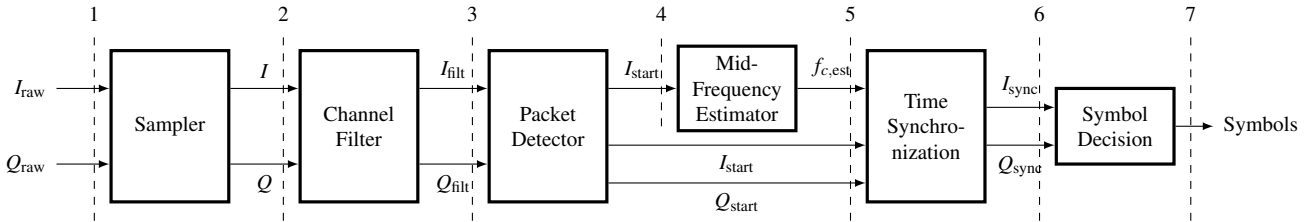


Figure 5.1: Interfaces between modules

Interface 1 This interface consists of the sampled data from the ADC in 12 bit signed integers. This data is sampled with $f_s = 758.272\text{ kHz}$ and contains the GFSK modulated signal at the intermediate mid-frequency of approximately 200 kHz. The mid-frequency of this signal is affected by the Doppler effect shifting it in the range $\pm 10.4\text{ kHz}$. The Can Space Protocol signals are GFSK modulated with a modulation index h in the interval from 0.5 to 1. The data rate will be in the interval from 2400 b/s to 19.2 kb/s.

Interface 2 The channel filter contains a 8th order Butterworth band-pass filter. The channel filter inputs the 16 b signed I and Q signals separately, and filters these one at a time. The bandwidth of the filter allows for all packets with Doppler shift to pass. The input is 12 b and the upper four excess zeros are shifted out for the remaining program to use the full range of the signal, i.e., 16 b.

Interface 3 The packet detector only inputs the filtered I channel, I_{filt} , since the energy in one channel is sufficient for detecting a packet. The band centered around 200 kHz is passed through to the packet detector. This way, only packets on the desired channel will be detected.

Interface 4 The mid-frequency estimator is not active until the packet detector notifies that a packet is found. It receives the address of the packet start from the packet detector. If a packet is detected, the rest of the demodulating thread is run, which means that the packet detector decides if the mid-frequency estimator, time synchronization,

and symbol decision should run. If a packet is not detected, no demodulation will be performed on the currently sampled block.

Interface 5 The mid-frequency estimator passes the estimated mid-frequency on to the time synchronizer. The I_{start} and the Q_{start} signal is passed on to the time synchronizer.

Interface 6 The time synchronized version of I_{start} and Q_{start} are passed on to the symbol decision block.

Interface 7 The output of the symbol decision are passed on to a ground station logging system where they are decoded and logged. This is outside the scope of this study and will not be described in greater detail.

5.2 Overall Requirements

The demodulator has to demodulate a received packet with less than 200 symbol errors (see Appendix A) at an SNR down to 8 dB. The demodulation of a packet has to be done before a new memory block is ready as described in Section 3.1, to respect the real-time requirement. The total amount of cycles per memory block is 142 433 491 as seen from Equation 3.2.

#	Parameter	Min.	Max.	Unit	Conditions
1	Symbol errors		200		<ul style="list-style-type: none"> • When demodulating a packet • $\text{SNR} \geq 8 \text{ dB}$
2	Cycles used		142 433 491	cycles	<ul style="list-style-type: none"> • $T_{\text{buf}} = 384 \text{ ms}$

Table 5.1: Requirements for the overall system

5.3 Module Requirements

In this section the requirements for each submodule is defined. All the requirements are derived from the analysis. Each requirement has an individual number which makes it easy to specify and reference to it from the individual tests. The individual tests are all specified and carried out in the implementation of each individual module on the blackfin in Part III.

5.3.1 Sampler

The sampler has to be compatible with the ADC, being able to receive data as specified in Section 3.1. The data has to be sorted in an in-phase signal and a quadrature signal.

5.3.2 Channel Filter

The filter should attenuate 80 dB/decade below and above the lower and upper cutoff frequencies respectively. See Section 7.3.3. The lower and upper -3 dB cutoff frequencies should be located at:

$$\begin{aligned}
 f_{\text{lower}} &= 200 \text{ kHz} - 10.4 \text{ kHz} - \frac{h}{2T_b} = 184.8 \text{ kHz} \\
 f_{\text{upper}} &= 200 \text{ kHz} + 10.4 \text{ kHz} + \frac{h}{2T_b} = 215.2 \text{ kHz}
 \end{aligned} \tag{5.1}$$

where

$$T_b = 1/9600 \text{ s}$$

$$h = 1$$

The impulse response should settle within 1% in less than the symbol period, T_b , to avoid symbol interference. Because the demodulator utilizes the phase, the group delay should be maximally flat.

#	Parameter	Min.	Max.	Unit	Conditions
1	Filter order	8th		order	• Butterworth band-pass filter
2	Lower cutoff frequency		184.8	kHz	• $h = 1$ • $T_b = 1/9600$
3	Upper cutoff frequency	215.2		kHz	• $h = 1$ • $T_b = 1/9600$
4	Impulse response settling time		105	μ s	• Settling within 1% • $T_b = 1/9600$

Table 5.2: Requirements for the channel filter module

5.3.3 Packet Detection

The double sliding windows should be at least 2 symbols long, as described in Section 3.3.2, to ensure a stable ratio during noise only. A new packet should be detected within the training sequence with 95% confidence, with a SNR down to 8 dB. The packet should be detected before the training sequence ends, giving room for the mid-frequency estimation and the time synchronization. Hence, the packet detection should have a precision within two symbols.

#	Parameter	Min.	Max.	Unit	Conditions
1	Success rate	95		%	• $\text{SNR} \geq 8 \text{ dB}$
2	Precision of detected packets	2		symbols	• $\text{SNR} \geq 8 \text{ dB}$

Table 5.3: Requirements for the packet detection module

5.3.4 Mid-Frequency Estimation

The mid-frequency should be estimated within ± 4800 Hz, as shown in Section 3.6.1.

#	Parameter	Min.	Max.	Unit	Conditions
1	Estimated mid-frequency error		4800	Hz	

Table 5.4: Requirements for the mid-frequency estimator module

5.3.5 Time Synchronization

Assuming the mid-frequency is estimated perfectly, the first symbol must be found within a precision of 50% of a symbol period, corresponding to $79 \cdot 50\% \approx 39$ samples, to make sure the symbol decision is done correctly. However as the mid-frequency might not be estimated perfectly, the first symbol must be found within a precision of 5% of a symbol period, corresponding to $79 \cdot 5\% \approx 4$ samples.

#	Parameter	Min.	Max.	Unit	Conditions
1	First symbol accuracy	4		samples	• Estimated mid-frequency error less than 4800 Hz

Table 5.5: Requirements for the time synchronization module

5.3.6 Symbol decision

The symbol decision module must be able to specify T_b with a precision such that a packet can be demodulated without errors due to drifting in the moment of decision. Furthermore a sync word must be correctly detected and an FSM decoded. The data symbols must be demodulated with 95% success rate with an SNR down to 8 dB.

#	Parameter	Min.	Max.	Unit	Conditions
1	Success rate of demodulation	95		%	<ul style="list-style-type: none"> • Estimated mid-frequency error less than 4800 Hz • $\text{SNR} \geq 8 \text{ dB}$

Table 5.6: Requirements for the symbol decision module

5.4 Delimitations

For the prototype which is described in this report some delimitations are naturally to be defined due to the present stage of the development.

- The demodulator should be compatible with data rates from 2400 b/s to 19 200 b/s, and modulation index from 0.5 to 1. All tests documented in this report are carried out with a data rate of 9600 b/s and a modulation index $h = 1$. Thus, changes have to be implemented in order to demodulate at other data rates and modulation indexes.
- A complete packet should be within one memory block of 2 MiB, or else the packet is discarded.
- Tests are only carried out for demodulating single packets, neglecting back-to-back packets.
- Errors in the sync word and FSM are not accounted for, making the detection of these a simple check of equality.

Part II

Design

Chapter 6

Program Flow

In Figure 6.1, the individual blocks of the demodulator and the flow between them are shown. During this part, the individual blocks will be developed, but the flow will not necessarily follow that on the figure. The total flow will later be described during the implementation in Part III. The figure should serve as a reference to keep track of the steps in the development.

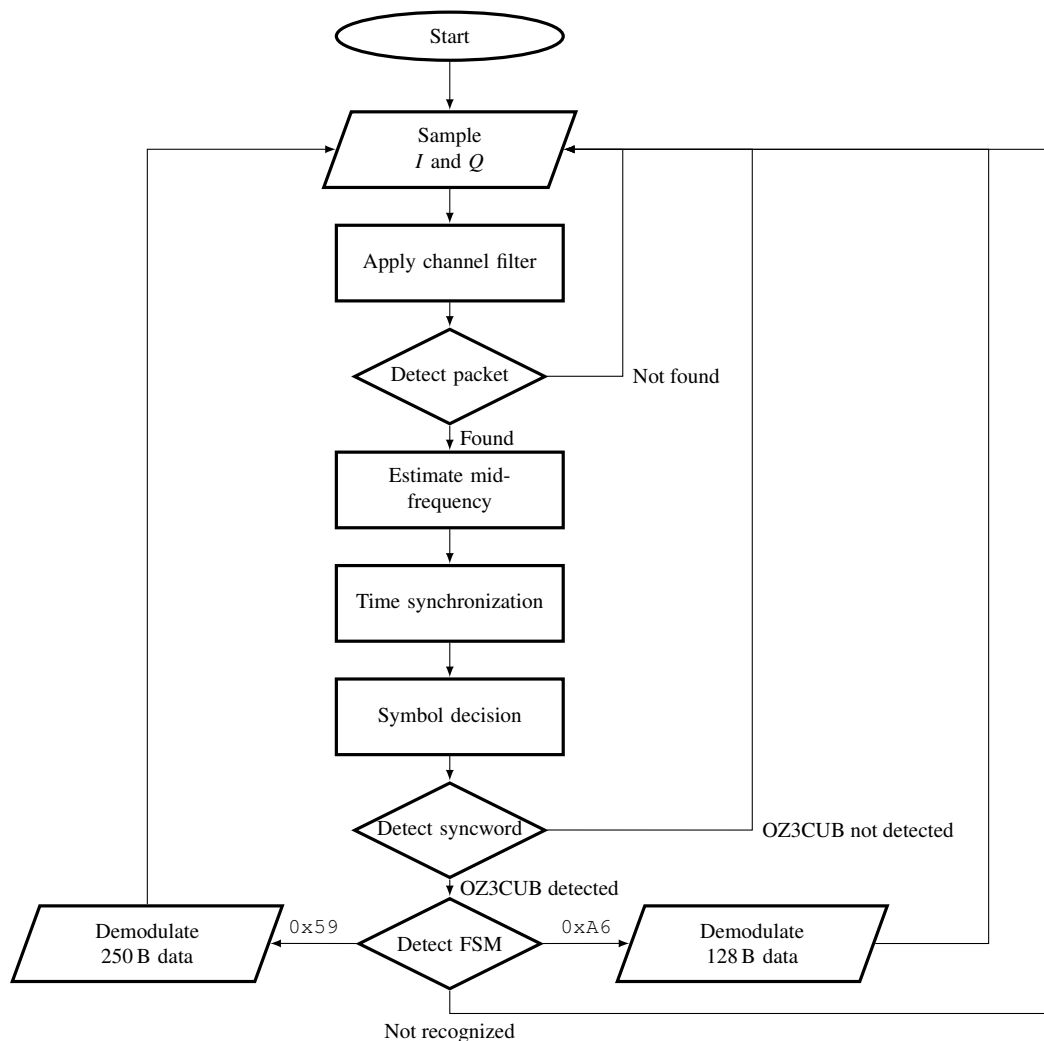


Figure 6.1: Program flow after implementation.

Chapter 7

GFSK Simulation in MATLAB

The MATLAB simulation chapter goes through steps taken in the development of the GFSK demodulator. The approach is to simulate the signal behavior from the binary sequence input to the transmitter, all the way through the signal chain, to the received binary sequence which is output at the ground station. Firstly, the modulator simulation is described, which serves as the input for the demodulator which is described afterwards.

7.1 Simulation Description

To confirm the analysis of this project a simulation will now be designed in MATLAB. The transmission of a binary message which utilizes the GFSK modulation scheme will in the following simulation be described, along with results and descriptions of the methods used. A block diagram serves to give an overview in Figure 7.1. This block diagram represents the signal chain during transmission from satellite to ground. The message $m(t)$

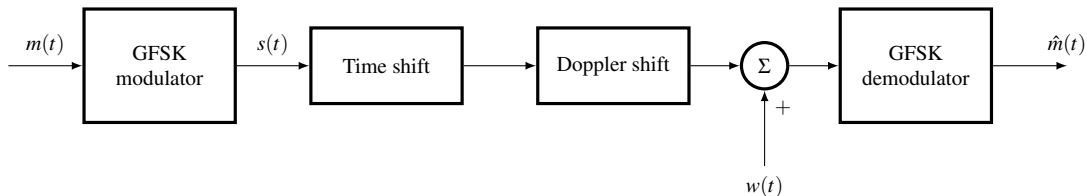


Figure 7.1: GFSK test bench for MATLAB simulations.

is a binary signal input to the modulator which outputs a modulated waveform $s(t)$ into the transmission channel (the air space). In this channel AWGN noise, a time shift, and a Doppler shift is added to the signal. Lastly the receiver end contains the GFSK demodulator and outputs the decoded binary sequence. In the following sections the simulation is designed block by block, from input to output, as shown in the figure of overview. First, the Modulator simulation is carried out.

7.2 GFSK Modulator in MATLAB

This section describes the functionality of the GFSK modulator block simulated in MATLAB. The approach is to present results from the MATLAB simulation program which is found on the CD:

 /code/matlab_gfsk_workbench/modulator/

Note: The output of the simulation is at an intermediate carrier frequency (IF) of 200 kHz. When the signal is transmitted through the air, the upconverted from this to 437.425 MHz (RF). Because Doppler shift is converted up and down with the rest of the baseband signal, the Doppler shift is preserved when simulating at IF.

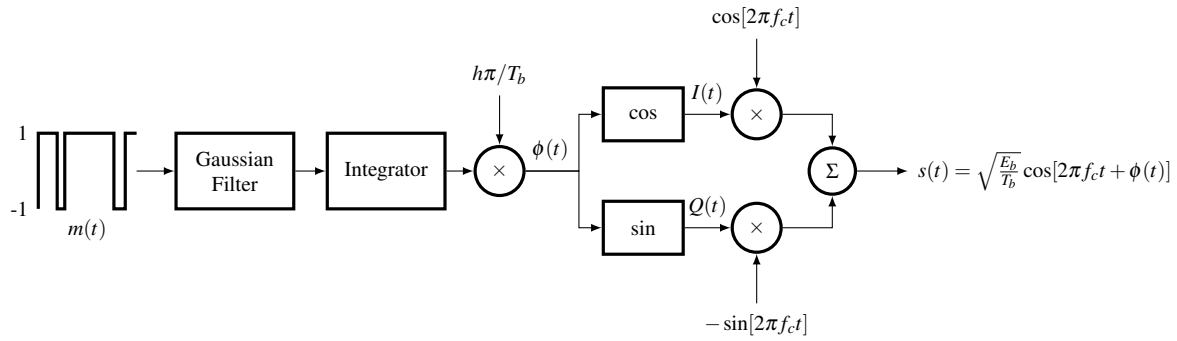


Figure 7.2: GFSK modulation block diagram of the MATLAB simulation. The output is IF at 200 kHz.

As an example in this documentation, the following data is input to the simulation:

- $m(t) = [0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1]$
- $T_b = 1/9600 \text{ s}$
- $f_c = 200 \text{ kHz}$
- $BT_b = 0.5$

The following explanations are based on a presentation of the designed signals that are written as symbols in Figure 7.2. These signals will be graphed: $m(t)$, $\phi(t)$, $I(t)$, $Q(t)$, $s(t)$.

First, the binary $m(t)$ signal is designed by converting a binary input to a NRZ signal. Next scenario is to generate the phase signal $\phi(t)$. This is done by convolving the $m(t)$ with a Gaussian filter. Finally, the $\phi(t)$ signal is generated by integration followed by scaling, by multiplication with $\pi h/T_b$. This result is shown in Figure 7.3. Note that the convolved output is shown without the delay caused by convolving in order to give a better overview of the link between $m(t)$ and $\phi(t)$. The phase signal $\phi(t)$ is generated to produce the desired phase change per symbol, as described in Section 2.4.1.

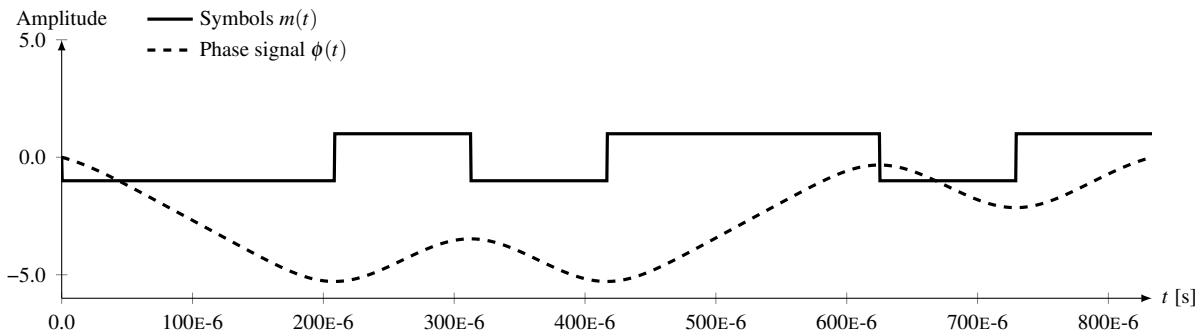
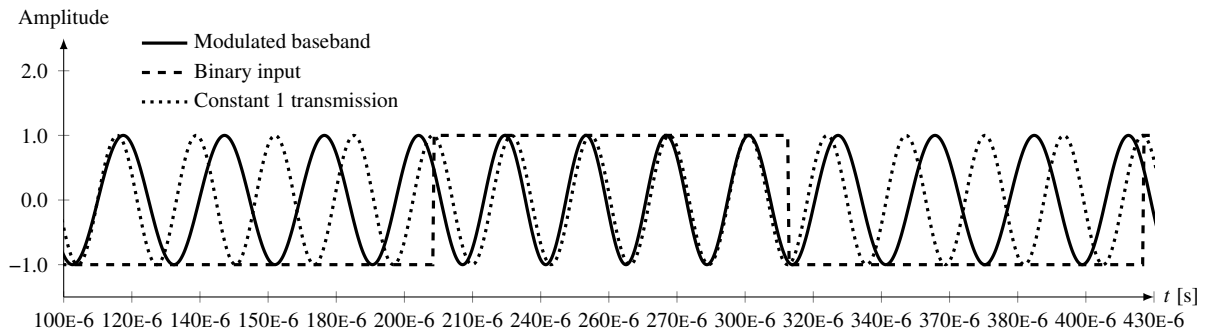


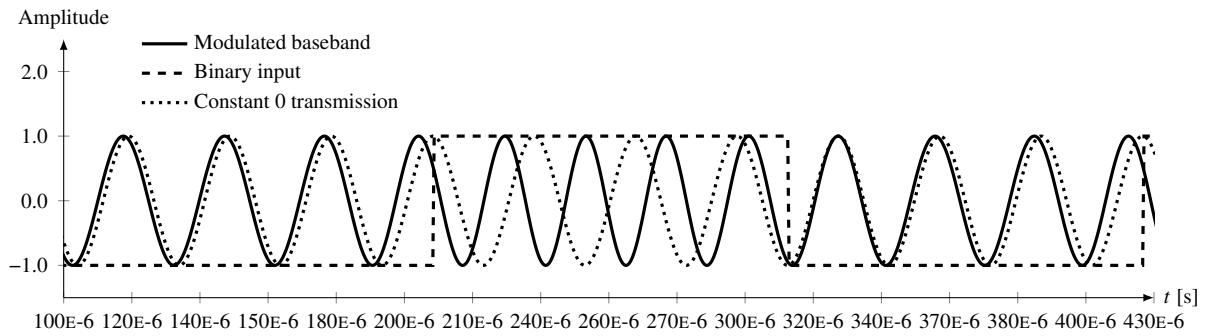
Figure 7.3: $m(t)$ input and calculated phase signal $\phi(t)$.

The phase signal $\phi(t)$ is then split into a real and imaginary part, by applying a $\sin()$ and a $\cos()$ respectively. The $I(t)$ and $Q(t)$ outputs are shown in 7.5. Finally the GFSK passband output $s(t)$ is generated by multiplying $I(t)$ and $Q(t)$ respectively with $\sin$ and $\cos$ containing the carrier frequency and then subtracting the products. The baseband output is shown in time domain in Figure 7.4 to confirm the frequency change and constant envelope of the baseband.

By carrying out this simulation it has been demonstrated that when changing the phase of the modulated signal, it includes a change in frequency. An FFT of a training sequence along with a only-1-signal and an only-0-signal is shown in Figure 7.6. When designing the modulator as described in the GFSK theory Section 2.4.1, the modulated signal will vary the frequency around the carrier frequency, exactly as described in 2.4.1. That is: If the message is



(a) Reference signal represents logic one.



(b) Reference signal represents logic zero.

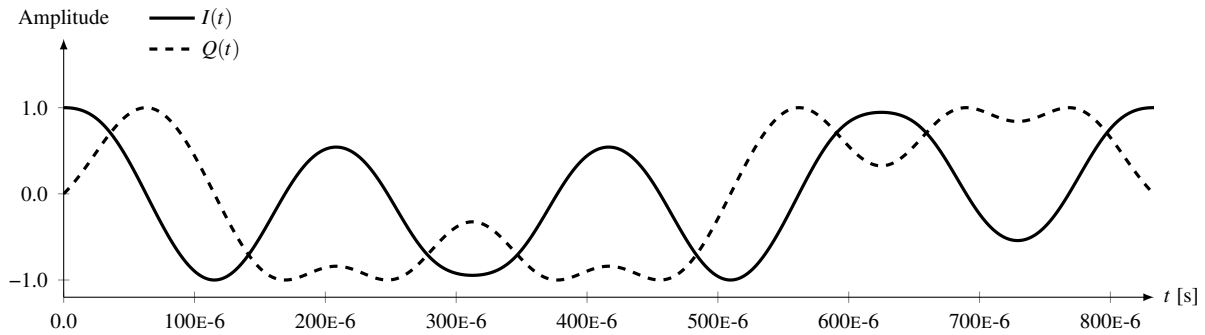
Figure 7.4: Modulated Baseband $s(t)$, compared to quasistatic frequencies which represents logical zero and one.

Figure 7.5: In-phase and quadrature components.

a logic 1, $\phi(t) = \phi(0) + h\pi t/T_b$, where the modulation index is typically $h = 1$ for AAUSAT. If the symbol is a 0, $\phi(t) = \phi(0) - h\pi t/T_b$. The theory used for designing this simulation was further elaborated in section 2.4.1.

7.2.1 Complex Signal Representation

As described in Section 2.4.1, the modulation can also be carried out using complex numbers. As this has proved helpful in designing the demodulator, it has been decided to update the modulator so it outputs a complex passband signal.

The original idea of using the signals $I(t)$ and $Q(t)$, seen on Figure 7.2, has been modified. The result is seen on Figure 7.7.

When designing the demodulator, the IF band data will be generated using this implementation.

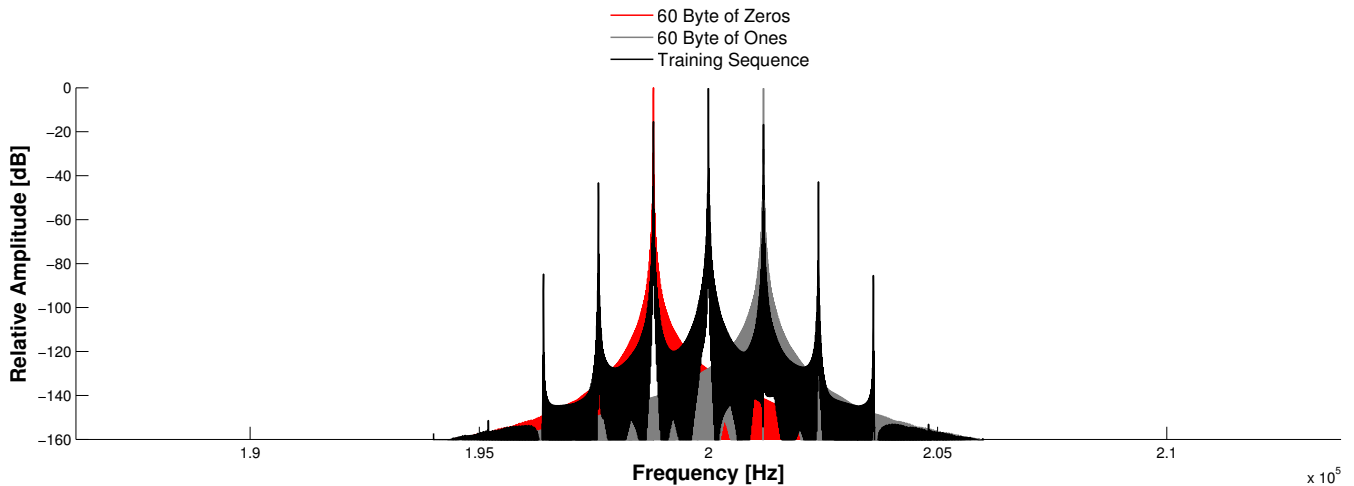


Figure 7.6: FFT of GFSK baseband at Modulator output.

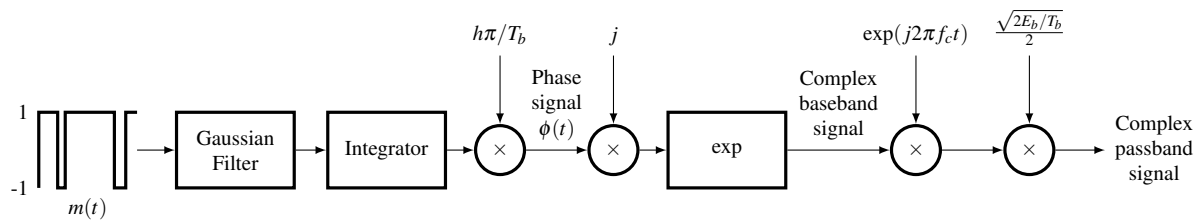


Figure 7.7: GFSK modulator with complex passband output.

7.3 GFSK Demodulator in MATLAB

Now the modulator has been simulated so that it is able to convert a binary input sequence into a GFSK-modulated signal. Following Figure 7.1, the next step is to model time shift, Doppler shift, external noise, and demodulation. This has been combined into one block that can be found on the CD:

 /code/matlab_gfsk_workbench/demodulator/

This MATLAB implementation serves to be a basis for a later implementation in C. It will operate on a whole buffer containing a packet and will therefore not be a real-time implementation and it contains no design of the sampler block.

The complex signal to be demodulated is generated with the MATLAB script from Section 7.2. The steps taken to model the channel are:

1. Add a **time shift** to the modulated signal.
2. Add a random **Doppler shift** to the modulated signal.
3. Add random **white Gaussian noise** to the signal.

The steps taken to model the receiver are:

1. Apply a **channel filter** to filter out the band the 0 frequency, the 1 frequency, and carrier frequency is expected to be within.
2. Estimate **mid-frequency**.
3. **Downconvert** signal by the carrier frequency so that a 0 symbol is a negative frequency and a 1 symbol is a positive frequency, equally spaced around 0 Hz.
4. Convert the complex baseband signal into a real **phase signal**.

5. Predict time location of symbols and **demodulate**.

These steps will be described below.

7.3.1 Time Shift, Doppler Shift, and White Gaussian Noise

These three steps will be done first to create a signal that realistically resembles a signal received from a satellite.

The time shift is added so the demodulator will be designed to detect a signal that does not have a specific starting point, as the modulator outputs by default. This is simulated by adding a delay in the beginning of the modulator output. The delay should not be so long that the first symbol is lost. A symbol period is, as seen from Section 2.3, at least

$$T_{b,\min} = \frac{1}{19200} = 52.08 \mu\text{s} \quad (7.1)$$

In simulation, the delay will be a number of samples. As the sampling frequency, described in Section 2.2, is 758 272 Hz, a symbol's duration is

$$T_{b,\min,\text{samples}} = f_s T_{b,\min} = \frac{758272}{19200} = 39.49 \text{ samples} \quad (7.2)$$

The delay has been randomly created to be between 1 and $\lfloor f_s T_b \rfloor$ samples, because f_s and T_b are defined. The delay will then be dynamic based on these parameters.

A Doppler shift is easily added by multiplying with a complex exponential function, as described in Section 2.4.1. The maximum Doppler shift was calculated in Section 2.4.3 to be $\pm 10.4 \text{ kHz}$. For simulation, the Doppler shift is therefore randomly generated within these limits.

White Gaussian noise will be added based on a given SNR, as described in Section 2.4.2. The noise, that is added, is complex. An FFT of a training sequence with added noise is shown in Figure 7.8.

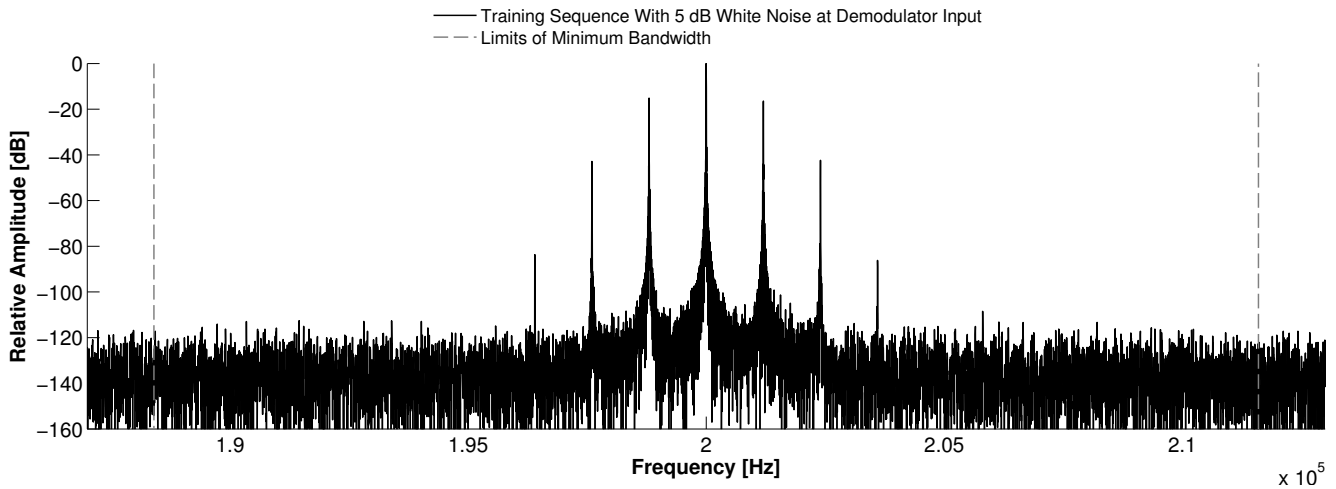


Figure 7.8: FFT of training sequence at Demodulator Input with noise added.

The MATLAB code that does these steps can be seen in Listing 7.1. Note: The code is split into several files.

```

1 %% gfsk_demod.m
2 repro      = 1; % Set to 1 if reproducible Doppler, noise and delay are wanted
3 fd         = 10.4e3;
4 SNR        = 5;
5 fs         = 758272;
6 Ts         = 1/fs;
7 Tb         = 1/2400;
8 Tb         = round(Tb/Ts)*Ts; % Change bit period to fit sample rate
9 fc         = 200e3;
10 h          = 1;
11 BT         = 0.5;

```

```

12
13 % Training sequence is defined as 60 B of 0 1 0 1 0 1 0 1
14 input = [trainingsequence];
15
16 %% Acquire Modulated Signal
17 addpath(' ../modulator/ ');
18 [t_full, s] = gfsk_mod(input, Tb, fc, h, BT);
19
20 %% Apply Random Dopplershift.
21 if repro==1
22     % rng(4) % Good at worst-case, dop=10kHz
23     rng(1)
24 end
25 [s_hat, fd_random] = dopplershift(s,t_full,fd);
26
27 %% Add additive white Gaussian noise
28 [s_hat] = add_complex_noise(s_hat,SNR);
29
30 %% Add time delay
31 delay = 1 + floor(rand*fs*Tb); % Random delay: 1 <= delay <= fs*Tb
32 if delay > 0
33     t = [t_full(1:length(t_full)-delay)];
34     s_hat = [s_hat(delay:length(s_hat)-1)];
35 end
36
37 %% SUBFILES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
38 %% add_complex_noise.m
39 function [output] = add_complex_noise(s, SNR_dB)
40     L = length(s); % Length of signal
41     Ps = s*s'/L; % Average power of signal
42     SNR = 10^(SNR_dB/10); % Calculate SNR
43     w = 1/sqrt(2).*(randn(1,L) + 1j*randn(1,L)); % Generate random noise vector
44     w = w - sum(w)/L; % Ensure zero-mean
45     w = w./sqrt(w*w'/L); % Make average power (variance) one
46     w = sqrt(Ps/SNR).*w; % Scale noise to wanted level
47     output = s + w; % Add noise to signal
48 end
49
50 %% dopplershift.m
51 function [s_dop, fd] = dopplershift(s, t, size_of_shift)
52     factor = rand*range([-1 1])+min([-1 1]);
53     fd = factor * size_of_shift;
54     s_dop = s .* exp(2j*pi*fd*t);
55 end

```

Listing 7.1: MATLAB script generate a GFSK modulated signal with time shift, Doppler shift, and 5 dB SNR white Gaussian noise

The output of this procedure is complex. The real part is shown in Figure 7.9.

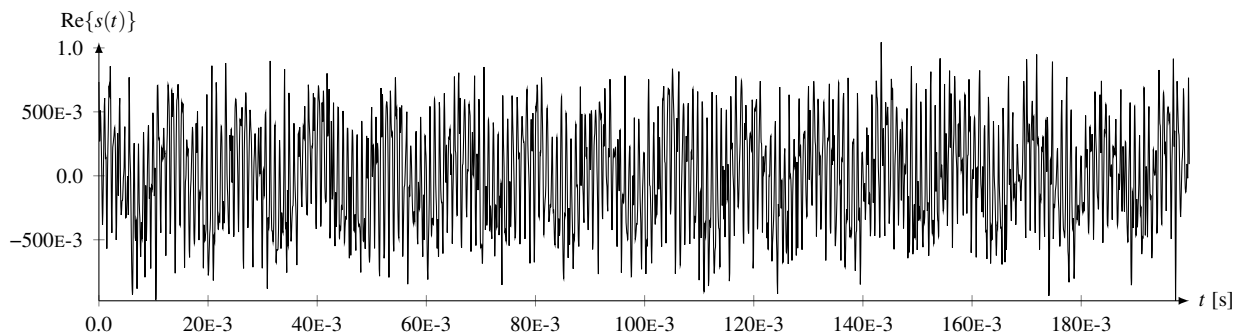


Figure 7.9: Real part of a modulated training sequence with Time shift, Doppler shift, and 5 dB SNR Gaussian white noise added.

7.3.2 Sampler

In the MATLAB implementation, no sampling is performed. The input data is obtained directly from the modulator function described in Section 7.2.

7.3.3 Channel Filter

First step in extracting data from the signal above is to add a channel band-pass filter so that only the frequencies around the 0 symbol and the 1 symbol are included. Through experimentation and experience from [TBT13] it has been chosen to use a 8th order Butterworth filter.

The cutoff frequencies for the filter must be centered around the mid-frequency. This is 200 kHz, as described in Section 2.2. As described in Section 2.4.1, the 0 symbol frequency and the 1 symbol frequency are separated by

$$\Delta f = \frac{h}{T_b} \quad (7.3)$$

The filter's lower cutoff frequency should therefore have room for $\Delta f/2$ Hz below 200 kHz. Besides this, there should be bandwidth enough so the maximum Doppler shift, 10.4 kHz, can be handled. The cutoff frequencies are therefore

$$\begin{aligned} f_{\text{lower}} &= 200\text{E}3 - 10.4\text{E}3 - \frac{\Delta f}{2} = 200\text{E}3 - 10.4\text{E}3 - \frac{h}{2T_b} \\ f_{\text{upper}} &= 200\text{E}3 + 10.4\text{E}3 + \frac{\Delta f}{2} = 200\text{E}3 + 10.4\text{E}3 + \frac{h}{2T_b} \end{aligned} \quad (7.4)$$

For this simulation it has been chosen to calculate the filter coefficients in MATLAB and use the build in `filter()` function to filter the signal. The filtered output is seen in Figure 7.10

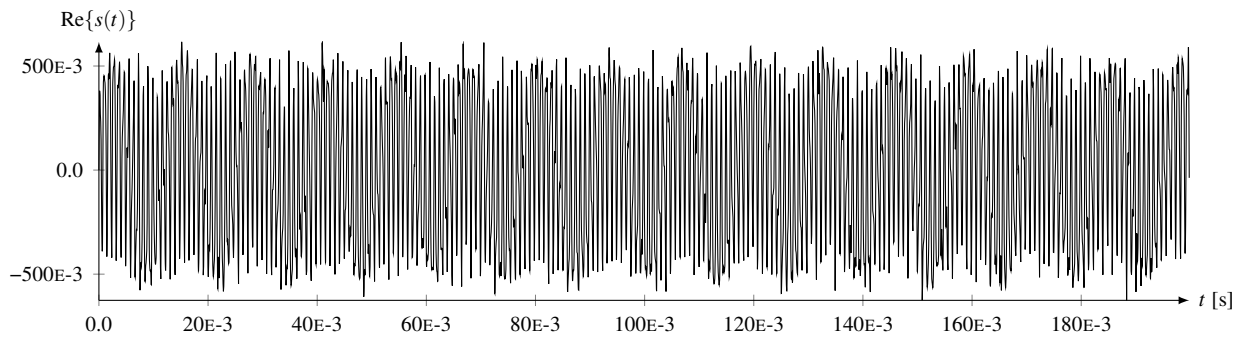


Figure 7.10: The signal from Figure 7.9 filtered through a 4th order Butterworth band-pass filter.

The MATLAB code that does the filtering is shown in Listing 7.2. It continues from Listing 7.1.

```
1 %% gfsk_demod.m
2 %% Filter with bandpass at 200 kHz
3 [b,a] = butter(4,[fc-fd-h/(2*Tb), fc+fd+h/(h*Tb)]/(fs/2),'bandpass');
4 s_hat = filter(b,a,s_hat);
```

Listing 7.2: MATLAB script filtering the modulated signal with distortion. Continues from Listing 7.1.

7.3.4 Packet Detection

In the MATLAB implementation, the packet is generated by the MATLAB modulator from Section 7.2. No space is added before the packet so therefore no packet detection is implemented in this chapter. The first packet detector is implemented in Section 8.2.

7.3.5 Mid-Frequency Estimation

The next step is to downconvert the signal so that the mid-frequency is at 0 Hz, the 0 symbol frequency is negative, and the 1 symbol frequency is positive. This is done by first doing an FFT of the filtered signal and finding the

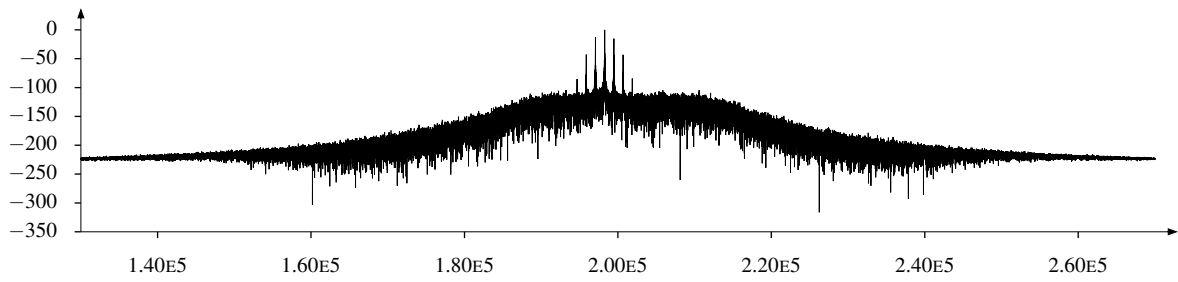


Figure 7.11: FFT of filtered signal generated in the previous steps.

frequency at which the largest peak is located. This frequency corresponds to the carrier frequency.

An FFT of the noisy, filtered signal from above, can be seen in Figure 7.11. Here the largest peak (the carrier) can be seen as well as two smaller peaks corresponding to the 0 symbol frequency and the 1 symbol frequency. The effect of the filter can also be seen here as well as the Doppler shift which shifts the three peaks a bit to the left of 200 kHz.

The MATLAB script for estimating the carrier frequency can be seen in Listing 7.3. Note: The code is split into several files.

```

1 %% gfsk_demod.m
2 %% FFT analysis to find carrier
3 [s_hat_fft, s_hat_fft_db, f] = fft_(s_hat, Ts);
4 [max_ampl, argmax] = max(s_hat_fft);
5 fc_est = ceil(f(argmax(1)));
6
7 %% SUBFILES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
8 %% fft_.m
9 function [fft_sig, fft_sig_db, f_axis] = fft_(sig, Ts)
10     L = length(sig);
11     NFFT = 2^nextpow2(L);
12     fft_long = fft(sig, NFFT)/L;
13     fft_sig = 2*abs(fft_long(1:NFFT/2+1));
14     fft_sig = fft_sig/max(fft_sig);
15     fft_sig_db = 20*log10(fft_sig);
16     f_axis = 0:(1/Ts)/NFFT:(1/Ts)/2;
17 end

```

Listing 7.3: MATLAB script for estimating carrier frequency, f_c .

7.3.6 Time Synchronization

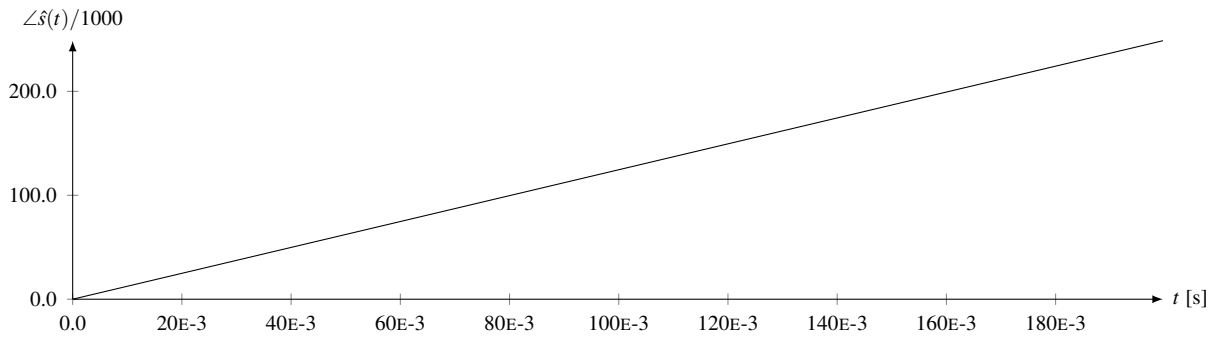
As described in Section 3.5, symbol time synchronization can be achieved by downconverting the signal so it is centered at 0 Hz. The further details of this will be described here.

Downconversion and Conversion to Phase Signal

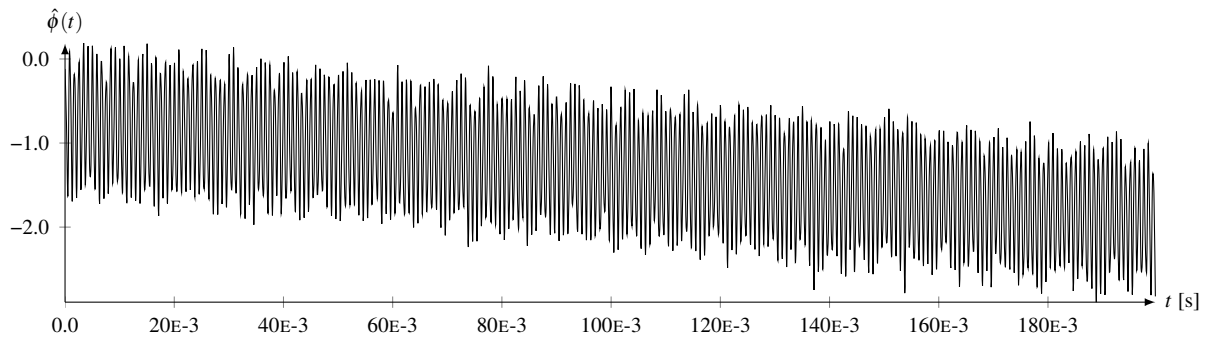
After the carrier frequency has been estimated, the signal can be downconverted as previously described. This is done by multiplying each complex sample by an exponential term as described in Section 3.5.1. If the signal this far is denoted $\hat{s}(t)$ and the carrier frequency estimated with the FFT is denoted $f_{c,est}$, the downconverted signal is found as

$$\hat{s}_{down}(t) = \hat{s}(t)e^{-j2\pi f_{c,est}t} \quad (7.5)$$

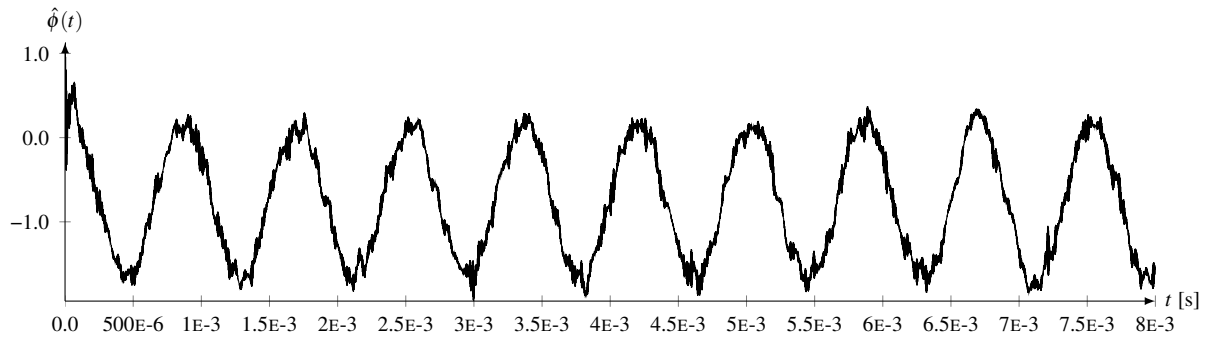
Now the signal is ready to be demodulated. The way this is done is by looking at the phase of the downconverted signal as it will be described shortly. Therefore, the signal is converted to a phase signal using the MATLAB function `angle()`. The outcome of this command corresponds to taking the arctan of the imaginary part of the signal over the real part of the signal. As the outcome of this is wrapping around at π and $-\pi$, the output is not continuous at these points. The function `unwrap()` makes up for this and makes the phase continuous.



(a) Before downconversion.



(b) After downconversion.



(c) After downconversion (zoomed in).

Figure 7.12: Phase of received, filtered signal before and after downconversion.

The phase signal before and after downconversion is shown in Figure 7.12. The MATLAB code for downconversion and conversion to phase signal is seen in Listing 7.4. This corresponds to the following:

$$\hat{\phi} = \angle \hat{s}_{\text{down}} \quad (7.6)$$

```
1 %% gfsk_demod.m
2 s_hat = s_hat .* exp(-2j*pi*fc_est*t);
3 phi_hat = unwrap(angle(s_hat));
```

Listing 7.4: MATLAB script for downconverting and converting to phase signal.

Finally, the downconverted signal shown in Figure 7.12b and 7.12c can be demodulated. When working with a phase signal it is known that if the slope is positive, the frequency is positive meaning a 1 symbol was transmitted and vice versa.

By investigating Figure 7.12c it is seen that during the training sequence, the slope is changing at very distinct points in time—the points where a new symbol is received. By comparing the received values at these points in time, the slope can be computed and therefore the symbols can be demodulated.

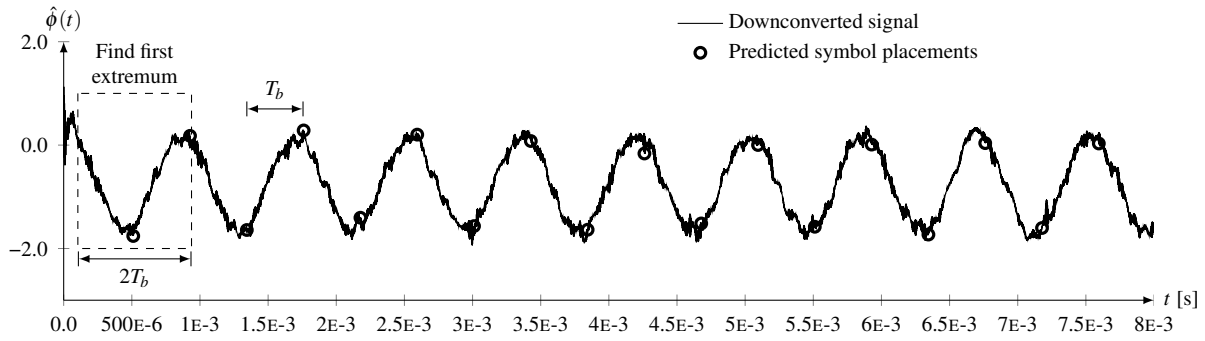


Figure 7.13: Phase of downconverted signal with annotations explaining how demodulation is performed.

The spacing between the symbols is already known as T_b . Therefore, only one change in slope (say, the first minimum) needs to be found and then all other points can be computed.

The resulting points have been plotted in Figure 7.13, and this figure may therefore help in understanding how the first extremum is found.

Because the training sequence is alternating between 0 and 1 each T_b seconds, both a maximum and a minimum should be located within any time window of $2T_b$. As it can barely be seen on Figure 7.13, the first part of the signal contains a lot of ups and downs. It has therefore been chosen to look for the first extremum in the window from $T_b/4$ to $9T_b/4$.

To find both minima and maxima, the absolute value of the phase signal is processed. The procedure for finding the peak is to move and compare three points:

- One center point. This is the candidate for an extrema.
- One $T_b/4$ below the center point.
- One $T_b/4$ above the center point.

Now three situations may arise:

1. $(b) < (a)$ and $(a) < (c)$: On a positive slope.
2. $(b) > (a)$ and $(a) > (c)$: On a negative slope.
3. $(b) < (a)$ and $(a) > (c)$: A peak is within (b) and (c).

Now, the windows is iterated through to find situation 3. This subwindow is then iterated through to find the largest value, and this, $t_{\text{first peak}}$ is where the first peak is located.

7.3.7 Symbol Decision

Because all symbols are equally spaced by T_b , all future points can be determined as

$$t_n = t_0 + nT_b \quad n = 0, 1, \dots \quad (7.7)$$

These points are shown in Figure 7.13. Because the slope determines the value of each symbol, the demodulation can now be performed by comparing two peak points at a time. The first symbol is a special case because it does not have a previous sample to be compared to. Instead, it is compared to a value previously (10 %) in the same symbol, to that way get an idea of the slope.

$$\hat{m}_0 = \begin{cases} 1 & \text{if } \hat{\phi}(t_0 - 0.1T_b) < \hat{\phi}(t_0) \\ 0 & \text{if } \hat{\phi}(t_0 - 0.1T_b) > \hat{\phi}(t_0) \end{cases}$$

$$\hat{m}_n = \begin{cases} 1 & \text{if } \hat{\phi}(t_{n-1}) < \hat{\phi}(t_n) \\ 0 & \text{if } \hat{\phi}(t_{n-1}) > \hat{\phi}(t_n) \end{cases} \quad n = 1, 2, \dots \quad (7.8)$$

```

1 %% gfsk_demod.m
2 % Find first extremum
3 t_0=1;
4 for i = 1+ceil(1/4*Tb*fs):9/4*Tb*fs
5     if abs(phi_hat(i)) - abs(phi_hat(i-ceil(1/4*Tb*fs))) > 0 ...
6         && abs(phi_hat(i+ceil(1/4*Tb*fs))) - abs(phi_hat(i)) < 0
7
8         t_0 = i;
9         peak = abs(phi_hat(t_0));
10        t_0_tmp = t_0;
11        for j = t_0:t_0+ceil(1/2*Tb*fs)
12            if abs(phi_hat(j)) > peak
13                peak = abs(phi_hat(j));
14                t_0_tmp = j;
15            end
16        end
17        t_0 = t_0_tmp;
18        break;
19    end
20 end
21
22 % Find first bit. Magic number "-3" is to compensate for delay.
23 t_0+ceil((length(input)-1)*Tb*fs)
24 t_i = t(t_0) : t(round(Tb*fs+1)) : t(t_0+ceil((length(input)-3)*Tb*fs));
25 phi_hat_i=phi_hat(t_0: round(Tb*fs) : t_0+ceil((length(input)-3)*Tb*fs));
26
27 % Hard demodulation -- First symbol
28 m_hat = zeros(1,length(phi_hat_i));
29 if phi_hat(t_0) < phi_hat(t_0-ceil(0.1*Tb*fs))
30     m_hat(1) = 0;
31 else
32     m_hat(1) = 1;
33 end
34
35 % All other symbols
36 for i = 2:length(phi_hat_i)
37     if phi_hat_i(i) < phi_hat_i(i-1)
38         m_hat(i) = 0;
39     else
40         m_hat(i) = 1;
41     end
42 end
43 end

```

7.3.8 Example Output

[illegible]

Listing 7.6: Input and output of the code presented above. The input is a 60 B training sequence.

It is seen that two symbols are missing. This is because of the “magic number” -3 in Listing 7.5. The reason for this number is to compensate for the number of samples that are missing when adding the delay (the delay is generated by cutting off some of the signal). This is not optimal but because the general demodulator is working, it has been decided to continue development in C from here and tackle the problem later on.

The simulations in MATLAB has served as a proof of concept for the chosen design of a demodulator which is able to demodulate a GFSK modulated package that is induced with noise and suffers from Doppler shift. Therefore the designed sub-systems can will now be translated in to C, which will translate some of the designs directly from MATLAB, while other needs a redesign. This is done in the next chapter.

Chapter 8

GFSK Demodulator in C

Now that the fundamentals of demodulating a signal has been tested in MATLAB, the next step is to implement a more complete program in C running on a PC. This is tested and verified in floating point precision before implementing on the fixed point Blackfin DSP. This implementation likewise reads and writes to files and is not working in real-time.

The MATLAB code is translated function wise into it's C complement. The C implementation is very like the MATLAB code. The changes as well as some new aspects, are documented below. The parts that are very similar to the MATLAB implementation will not be described in detail. Instead, refer to the code implementation and the MATLAB sections.

8.1 Sampler

No actual sampling is used in the this code. However an emulation of the sampling method used on the Blackfin is implemented reading from files. To make it more like the actual case, where the input data is continuously received from the ADC, without knowing the beginning of the packet, a signal generator is used to simulate a DLIST1 packet, sent somewhere in a 6 seconds window (see Appendix C.1).

8.2 Packet Detection

In the MATLAB demodulator the packet location were already known, thereby no need of a packet detector. The start of the packet is in this PC C-code unknown, and a packet detector using double sliding windows is chosen as tested in Section 3.3.

The implementation of the double sliding window, can be found in the folder:

 /code/c_gfsk_workbench/pdetect.c

When the start of a packet is detected, three full buffers are filled to ensure a whole packet is obtained before demodulating.

8.3 Estimation of Mid-Frequency

Estimation of mid-frequency is done in a very similar way to the MATLAB implementation in Section 7.3.5. However, this implementation involves performing an FFT to determine mid frequencies. This is done with a built-in MATLAB function, and must be replaced for C implementation.

Because the Blackfin comes with an FFT library, it has been chosen to use a library—Kiss FFT[Bor]—for PC implementation. The implementation can be found on the CD:

 /code/c_gfsk/workbench/gfsk_demod.c

8.4 Time Synchronization

Downconversion and conversion to phase signal is likewise done in a similar way to the MATLAB implementation in Section 7.3.6.

Three MATLAB functions, `exp()`, `angle()`, and `unwrap()`, were used to do this. The first two have similar functions in the standard C libraries `math.h` and `complex.h`. A comparison of these two are shown in Table 8.1 (note: `creal()` and `cimag()` extracts the real and imaginary part of a complex signal, respectively).

MATLAB function	Similar C function
<code>exp(s)</code>	<code>cexp(s)</code>
<code>angle(s)</code>	<code>atan2(cimag(s), creal(s))</code>

Table 8.1: Comparison of MATLAB and C functions for downconversion and conversion to phase signal. Here, s is a complex float signal.

The last function, `unwrap()`, is not available in any standard C libraries. It is invoked on the output of the `atan2()` function to remove the 2π jumps that occur when the input exceeds the interval $-\pi \leq \phi \leq \pi$.

A function by [Vin] has been used to do unwrapping. It is used in a loop and compares the current phase with the previous phase to observe if a large jump has occurred. If this is the case, the phase is unwrapped to remove the jump. This function is found on the CD:

 /code/c_gfsk_workbench/unwrap.c

The function for downconversion and phase conversion is also found on the CD:

 /code/c_gfsk_workbench/gfsk_demod.c

The actual time synchronization uses the find first extremum method as described in Section 3.5.2. Because the first symbol sent in the transmission is a zero (See Section 2.3), only the first minimum within $2T_b$ is found—not the first extremum as in the MATLAB code as seen in Section 7.3.6.

Furthermore, for simplicity it has been decided to traverse all values within the window marked with a dashed line in Figure 8.1, and find the minimum instead of doing it in two steps as it was done in the MATLAB implementation.

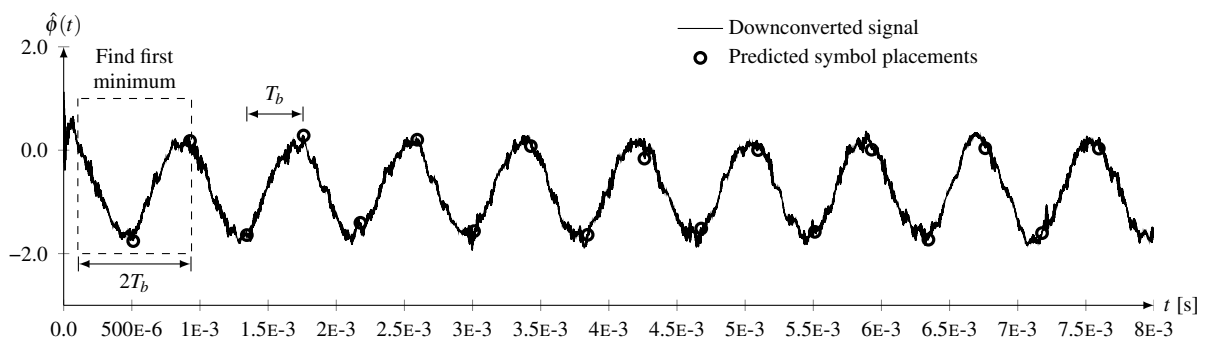


Figure 8.1: Demodulation process in C implementation. The figure is the same as Figure 7.13. In C, the first minimum is found instead of the first extremum.

This part of the program is found on the CD:

 /code/c_gfsk_workbench/gfsk_demod.c

Chapter 9

Channel Filter

In this section the filter design is carried out. Through tests in the simulations of the demodulator it is shown that a fourth order Butterworth band-pass filter is sufficient, in order to reduce enough noise for the system to be able to demodulate the message signal. In the end of this section the filter is designed for implementation on the Blackfin such that it is compliant with the Q15 data format. For this chosen design the cut off frequencies are chosen from the data rate of 9600 b/s, therefore the assumption is made that this rate is stationary. This assumption has been made to simplify the system, since this is the most common rate used for the space link and the principles in the development process is equivalent for variable data rate.

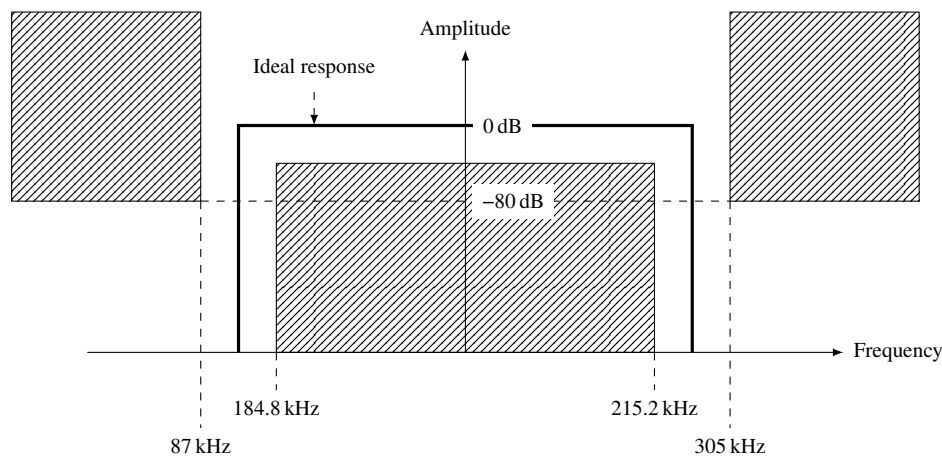


Figure 9.1: Channel filter specifications

Since the 8th order Butterworth filter is shown to be sufficient, the frequency characteristics will be compared to other filter types in order to investigate if some calculations can be left out. With the real-time aspect taken into account, the phase response and impulse is investigated. The impulse response is also investigated to avoid ISI (Intersymbol Interference).

9.1 Filter Type Considerations

When designing filters to run in a DSP there is basically two ways of doing so. The filter can be designed as FIR (Finite Impulse Response) or IIR (Infinite Impulse Response). FIR can be designed to have linear phase, but requires more computations than IIR in order to fulfill the same requirements to the frequency response [OS10, page 607]. Even though the Blackfin is optimized with circular buffers, FIR will still be computational heavier than IIR, and it will use more memory to address the coefficients. Furthermore IIR has the risk of being unstable because of feedback, especially with fixed point coefficients, where FIR is guaranteed stability. MATLABs FDATool shows that a 8'th order band-pass Butterworth IIR filter corresponds to a 58'th order FIR, in terms of frequency response.

To ensure the filter meets the real-time aspects IIR is chosen for implementing on the Blackfin DSP.

Since it is chosen to design an IIR filter and since the Butterworth filter does not have the most extreme dropoff, the Butterworth filter is compared with to Chebyshev Type 1 and the Elliptical filter type. Since they both have a steeper roll off, a lower order is needed and thereby fewer computations.

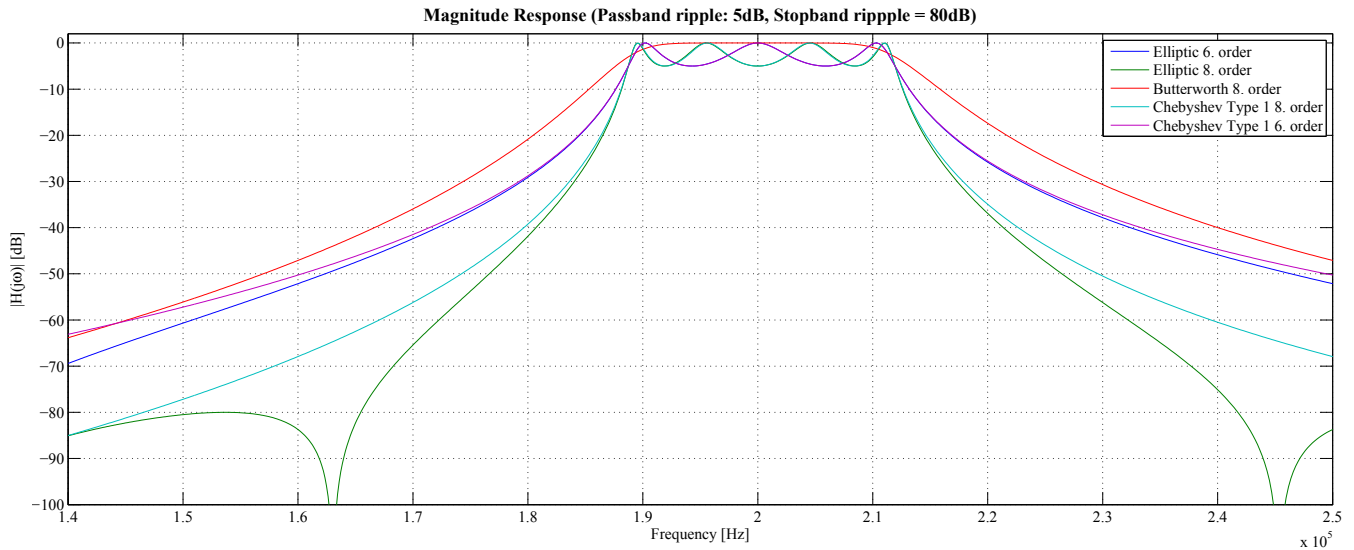


Figure 9.2: Comparison of frequency responses for different filter types.

The important aspects of the comparisons is to decide if the ripple of these filters and the phase response is compliant with the chosen demodulator type. As it can be seen in Figure 9.2 the frequency responses of the elliptic and the Chebyshev are both steeper than the Butterworth, even with a lower 6'th order designs. The Chebyshev and the elliptic filter are very similar, with the biggest different being Chebyshev have more attenuation in the far outer band. When considering the phase response in Figure 9.3 it is noted that the prize for the calculation efficiency of

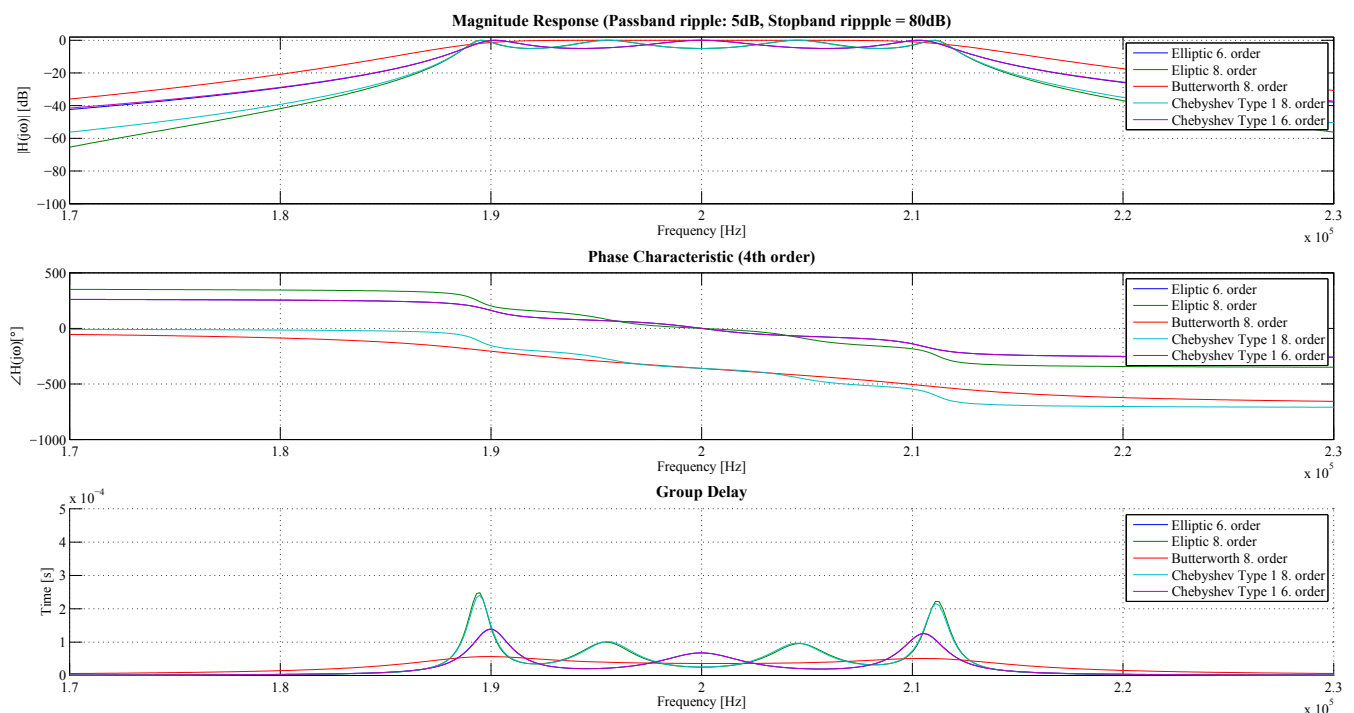


Figure 9.3: Phase Response Comparisons

implementing Chebyshev or Elliptical filters, is the group delay, which is proportional to the amount of ripple in

the passband. Since the focus is to design a demodulator, which demodulates the phase of the intermediate signal, it is very important to have a constant group delay, which is the characteristics of the Butterworth filter. Therefore it is chosen to design a 8'th order Butterworth band-pass filter.

9.2 Channel Filter Design

In the following section the design of the 8th order Butterworth band-pass filter is presented.

An analog 4th order lowpass filter is designed and transformed into a 8th order band-pass filter through well known procedures. The analog filter is translated into the digital domain and brought into cascades for implementation on the DSP.

Low Pass Design

The Butterworth LP-filter prototype may analytically be described in the normalized domain, as a squared magnitude function:

$$|H(j\omega)|^2 = \frac{1}{1 + \left(\frac{\omega}{\omega_c}\right)^{2N}} \bigg|_{\omega_c=1} = \frac{1}{1 + \omega^{2N}} \quad (9.1)$$

where

ω is the frequency variable.

ω_c is the cut off frequency.

N is the filter order.

To be able to optimize with respect to the filter order, the expression of the necessary order can be defined from deriving the attenuation α , from the Butterworth gain in Equation 9.1.

$$10\log(|H(j\omega)|^2) = -10\log(1 + \omega^{2N}) \Rightarrow \frac{\alpha_{dB}}{10} = \log(1 + \omega^{2N}) \Rightarrow \omega^{2N} = 10^{\frac{\alpha_{dB}}{10}} - 1 \quad (9.2)$$

The ratio between the stop band and the pass band edges will express the filter order, so the ratio is put up.

$$\frac{\omega_s^{2N}}{\omega_p^{2N}} = \frac{10^{\frac{\alpha_{s,dB}}{10}} - 1}{10^{\frac{\alpha_{p,dB}}{10}} - 1} \Rightarrow 2N\log\left(\frac{\omega_s}{\omega_p}\right) = \log\left(\frac{10^{\frac{\alpha_{s,dB}}{10}} - 1}{10^{\frac{\alpha_{p,dB}}{10}} - 1}\right) \Rightarrow N \geq \log\left(\frac{10^{\frac{\alpha_{s,dB}}{10}} - 1}{10^{\frac{\alpha_{p,dB}}{10}} - 1}\right) \cdot 2\log\left(\frac{\omega_s}{\omega_p}\right)^{-1} \quad (9.3)$$

where

ω_s is the stop band frequency.

ω_p is the pass band frequency.

$\alpha_{s,dB}$ is the stop band attenuation.

$\alpha_{p,dB}$ is the pass band attenuation.

Pole Location

The poles of the Butterworth LP-filter are to be derived from the roots of the denominator of the expression $|H(j\omega)|^2$, and because $s = j\omega$ it follows that $\omega = s/j$.

$$1 + \omega^{2N} \big|_{\omega=s/j} = 0 \Rightarrow \left(\frac{s^2}{j^2}\right)^N = -1 \Rightarrow (-s^2)^N = -1 = e^{j\pi} \quad (9.4)$$

and by all the rotations around the unit circle, with k as the counter. [Su02, page 54]

$$(-s^2)_k^N = e^{j\pi + j2\pi k} \Rightarrow s_k = \exp \left(j \left(\frac{\pi + 2\pi k}{2N} - \frac{\pi}{2} \right) \right) \quad (9.5)$$

For the 4th order lowpass filter there are $2N$ equally distributed poles on the unit circle. Due to stability, only the left half plane poles are of interest. These poles are calculated from $k = N$ to $2N - 1$ For the 4th order filter these are located in:

$$s_5 = -\frac{\sqrt{2-\sqrt{2}}}{2} + j\frac{\sqrt{2+\sqrt{2}}}{2} = -0.38 + j0.92 \quad (9.6)$$

$$s_6 = -\frac{\sqrt{2+\sqrt{2}}}{2} + j\frac{\sqrt{2-\sqrt{2}}}{2} = -0.92 + j0.38 \quad (9.7)$$

$$s_7 = -\frac{\sqrt{2+\sqrt{2}}}{2} - j\frac{\sqrt{2-\sqrt{2}}}{2} = -0.92 - j0.38 \quad (9.8)$$

$$s_8 = -\frac{\sqrt{2-\sqrt{2}}}{2} - j\frac{\sqrt{2+\sqrt{2}}}{2} = -0.38 - j0.92 \quad (9.9)$$

Because the poles always come in conjugate pairs (if not real), They can be put on the standard form for second order polynomials as follows:

$$H(s) = \frac{1}{(s + s_5)(s + s_6)(s + s_7)(s + s_8)} \quad (9.10)$$

$$= \frac{1}{(s^2 + 2\text{Re}\{s_5\}s + |s_5|^2)(s^2 + 2\text{Re}\{s_6\}s + |s_6|^2)} \quad (9.11)$$

$$= \frac{1}{s^4 + 2.61s^3 + 3.41s^2 + 2.61s + 1} \quad (9.12)$$

$$(9.13)$$

The analog Butterworth 4th order low pass prototype is now defined.

Band-Pass Transformation

Now that the lowpass prototype has been defined, the band-pass prototype will be developed. This is carried out by transforming the lowpass with a substitution of the s variable [Su02, page 80].

$$\begin{aligned} H(s) &= \frac{1}{s^4 + 2.61s^3 + 3.41s^2 + 2.61s + 1} \bigg|_{s = \frac{s^2 + \Omega_0^2}{BS}} \\ &= \frac{1}{\left(\frac{s^2 + \Omega_0^2}{BS}\right)^4 + 2.61\left(\frac{s^2 + \Omega_0^2}{BS}\right)^3 + 3.41\left(\frac{s^2 + \Omega_0^2}{BS}\right)^2 + 2.61\left(\frac{s^2 + \Omega_0^2}{BS}\right) + 1} \end{aligned} \quad (9.14)$$

where

Ω_0 is the center frequency.

B is the bandwidth.

S is the frequency variable.

By expanding the denominator and substituting the coefficients, the Butterworth band-pass polynomial is derived.

$$H(S) = \frac{B^4 S^4}{aS^8 + bS^7 + cS^6 + dS^5 + eS^4 + fS^3 + gS^2 + hS + i} \quad (9.15)$$

where

$$\begin{aligned}
a &= 1 \\
b &= 2.61B \\
c &= 3.41B^2 + 4\Omega_0^2 \\
d &= 2.61B(B^2 + 3\Omega_0^2) \\
e &= B^4 + 6.82B^2\Omega_0^2 + 6\Omega_0^4 \\
f &= 2.61B(B^2 + 3\Omega_0^2)\Omega_0^2 \\
g &= 3.41B^2\Omega_0^4 + 4\Omega_0^6 \\
h &= 2.61B\Omega_0^6 \\
i &= \Omega_0^8
\end{aligned}$$

Bilinear Transformation

To transform the analog filter into the digital domain either the impulse invariant method or the bilinear transform can be used. The impulse invariant method will cause aliasing, which is not accepted in the channel filter. The bilinear method causes no aliasing, but some amount of distorted frequency response. Appropriately, most of this distortion is fixed by pre-warping the critical frequencies. Therefore the bilinear method will be applied.

By having a system in the s -domain ($s = j\omega$), which is desired to map into the z -domain ($z = e^{j\omega}$), there will firstly be a short definition of the link between the two domains, by starting in the z -domain and transforming to the s -domain.

$$z = e^{j\omega} = e^{jT_s\Omega} = e^{T_ss} \Rightarrow \ln(z) = \ln(e^{T_ss}) \Rightarrow s = \frac{\ln(z)}{T_s} \quad (9.16)$$

where

Ω is the analog frequency variable. [rad/s]

ω is the digital frequency variable. [rad/sample]

T_s is the sample period. [s]

From this relationship, it is shown that the s -domain system, is represented in z -domain by the natural logarithm divided by the sample period. Therefore the bilinear transformation is a Taylor approximation of the natural logarithm of z divided by T_s . The bilinear transformation consists of only the first term in the Taylor polynomial, which is showed in Equation 9.17 [OS10, page 533].

$$s = \frac{2}{T_s} \cdot \frac{z-1}{z+1} = \frac{2}{T_s} \cdot \frac{1-z^{-1}}{1+z^{-1}} \quad (9.17)$$

The bilinear transformation maps the frequencies from $-\infty < \Omega < \infty$ to $-\pi < \omega < \pi$. It preserves stability from the left half-plane to the interior of the unit circle. Due to the non-linearity of this transformation a pre-warping of the critical frequencies is necessary before the transformation is carried out. The non-linear relation between the analog and digital frequencies is expressed as [OS10, page 534]

$$\Omega = \frac{2}{T_s} \tan\left(\frac{\omega}{2}\right) \quad (9.18)$$

Therefore, when designing the specified filter bands, the cut-off frequencies must be pre-warped and then used for defining the z -domain center frequency by first transforming these into the normalized digital domain.

$$\omega_{\text{lower}} = \frac{\Omega_{\text{lower}}}{f_s} \quad \wedge \quad \omega_{\text{upper}} = \frac{\Omega_{\text{upper}}}{f_s} \quad [\text{rad/sample}] \quad (9.19)$$

The lower and upper edge frequencies can now be pre-warped.

$$\Omega_{\text{lower-prewarp}} = \frac{2}{T_s} \cdot \tan\left(\frac{\omega_{\text{lower}}}{2}\right) \quad \wedge \quad \Omega_{\text{upper-prewarp}} = \frac{2}{T_s} \cdot \tan\left(\frac{\omega_{\text{upper}}}{2}\right) \quad [\text{rad/s}] \quad (9.20)$$

The pre-warped cut-off frequencies are then inserted in the expression for the center frequency, which is the geometric mean.

$$\Omega_{0\text{-prewarp}} = \sqrt{\Omega_{\text{lower-prewarp}} \cdot \Omega_{\text{upper-prewarp}}} \quad (9.21)$$

And the pre-warped bandwidth is calculated.

$$B_{\text{prewarp}} = \Omega_{\text{upper-prewarp}} - \Omega_{\text{lower-prewarp}} \quad (9.22)$$

Finally the pre-warped center frequency and bandwidth will then be applied to the bilinear transformation, showed in Equation 9.23. Note that this equation is showed as being non pre-warped even though the bandwidth and center frequency are both pre-warped.

$$H(z) = \frac{B^4 S^4}{aS^8 + bS^7 + cS^6 + dS^5 + eS^4 + fS^3 + gS^2 + hS + i} \bigg|_{S=\frac{2}{T_s} \cdot \frac{z-1}{z+1}} \quad (9.23)$$

where

$$a = 1$$

$$b = 2.61B$$

$$c = 3.41B^2 + 4\Omega_0^2$$

$$d = 2.61B(B^2 + 3\Omega_0^2)$$

$$e = B^4 + 6.82B^2\Omega_0^2 + 6\Omega_0^4$$

$$f = 2.61B(B^2 + 3\Omega_0^2)\Omega_0^2$$

$$g = 3.41B^2\Omega_0^4 + 4\Omega_0^6$$

$$h = 2.61B\Omega_0^6$$

$$i = \Omega_0^8$$

This gives the transfer function for the 8th order band-pass filter, normalized to coefficient $a_0 = 1$.

$$H(z) = 67.263E-6 \cdot \frac{1 - 4z^{-2} + 6z^{-4} - 4z^{-6} + z^{-8}}{1 + 0.65z^{-1} + 3.65z^{-2} + 1.73z^{-3} + 4.89z^{-4} + 1.52z^{-5} + 2.84z^{-6} + 0.44z^{-7} + 0.60z^{-8}} \quad (9.24)$$

Before implementing on the DSP several decisions has to be made, such as structure and coefficients quantization, which will be considered in the following sections.

9.3 Filter Implementation Concerns

Concerning implementation, direct form I, direct form II, and cascaded form has been evaluated with respect to computational complexity, memory usage, and quantization sensitivity.

The Blackfin DSP got a 40 b accumulator for calculations, whereby the registers are used as 16 b or 32 b for double precision. The direct form I has the better SNR since the calculations can be done in one line, only truncating when storing the output to register. An example is shown for a 2nd order filter:

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) - a_1y(n-1) - a_2y(n-2) \quad (9.25)$$

Thereby becoming more resilient to overflow, since 2's complement allow overflow in the temporary calculations due to wraparound, as long the final output value is within the original dynamic range. Direct form II on the other hand, uses the minimum amount of memory elements $\max(M, N)$ [PM96, page 520], compared to direct form I $(M + N + 1)$ memory elements, with M being the number of poles and N being the number of zeros. However the calculations are done in two steps, resulting in the double truncation noise.

$$w(n) = x(n) - a_1w(n-1) - a_2w(n-2) \quad (9.26)$$

$$y(n) = b_0w(n) + b_1w(n-1) + b_2w(n-2) \quad (9.27)$$

The SNR is assumed less important than memory usage, thus direct form II is preferred of the two.

To reduce the sensitivity of coefficient quantisation, the filter is implemented in a cascade structure consisting of second order sections [KLT06, page 153]. With cascade form, variations in one parameter only affects the related section, where variations in direct form parameters influence all the poles. Therefore cascaded form is preferred in DSP implementation, especially for higher order filters. Each section generates roundoff error, which is propagated to the next stage. The total roundoff noise will depend on the specific ordering and pairing of poles. To prevent overflow either scale factors or another Q format with higher dynamic range can be applied. Scale factors can be used at each section to avoid overflow, at the cost of lower SNR. Based on these advantages the filter is brought to cascade form, consisting of 2nd order sections implemented in direct form II.

9.3.1 Cascade Form

To obtain the 2nd order sections, the poles and zeros are obtained from the 8th order band-pass filter Equation 9.24. These are paired and ordered, which can be done in $(N!)^2$ different ways, each giving a different noise results when working with finite precision. The pole/zero are located at Figure 9.6.

$$p_1 = -0.168 + j0.949 \quad |p_1| = 0.9643 \quad (9.28)$$

$$p_2 = -0.168 - j0.949 \quad |p_2| = 0.9643 \quad (9.29)$$

$$p_3 = 0.002 + j0.964 \quad |p_3| = 0.9637 \quad (9.30)$$

$$p_4 = 0.002 - j0.964 \quad |p_4| = 0.9637 \quad (9.31)$$

$$p_5 = -0.113 + j0.908 \quad |p_5| = 0.9149 \quad (9.32)$$

$$p_6 = -0.113 - j0.908 \quad |p_6| = 0.9149 \quad (9.33)$$

$$p_7 = -0.046 + j0.913 \quad |p_7| = 0.9144 \quad (9.34)$$

$$p_8 = -0.046 - j0.913 \quad |p_8| = 0.9144 \quad (9.35)$$

The pairing of poles and zeroes are done accordingly to the thumb rule saying: the pole closest to the unit circle is paired with the closest zero [OS10, page 480]. This is done for all the poles. Pairing a pole with the closest zero reduces the peak gain as much as possible, thereby reducing the possibility of overflow. Each biquad section are formed by a complex pole with the paired zeroes. This gives the following four biquads.

$$H(z) = \frac{(s - z_1)(s - z_2)}{(s - p_n)(s - p_n^*)} = \frac{b_0 + b_1z^{-1} + b_2z^{-2}}{a_0 + a_1z^{-1} + a_2z^{-2}} \quad (9.36)$$

$$H_1(z) = \frac{(z - (-1))(z - (-1))}{(z - p_1)(z - p_2)} = \frac{1 + 2z^{-1} + z^{-2}}{1 + 0.337z^{-1} + 0.930z^{-2}} \quad (9.37)$$

$$H_2(z) = \frac{(z - 1)(z - 1)}{(z - p_3)(z - p_4)} = \frac{1 - 2z^{-1} + z^{-2}}{1 - 0.005z^{-1} + 0.929z^{-2}} \quad (9.38)$$

$$H_3(z) = \frac{(z - (-1))(z - (-1))}{(z - p_5)(z - p_6)} = \frac{1 + 2z^{-1} + z^{-2}}{1 + 0.227z^{-1} + 0.837z^{-2}} \quad (9.39)$$

$$H_4(z) = \frac{(z - 1)(z - 1)}{(z - p_7)(z - p_8)} = \frac{1 - 2z^{-1} + z^{-2}}{1 + 0.091z^{-1} + 0.836z^{-2}} \quad (9.40)$$

To order the biquads a compromise has to be made. If the output noise variance spectrum is wanted low, the poles close to the unit circle is placed at the beginning of the cascade. If the signal level is wanted high, the poles close to the unit circle is placed at the end of the cascade. Both implementation are tested in MATLAB's FDATOOL shown in Figure 9.4, where the noise spectrum is almost equal for the two fixed point implementations. The least to most peaked method is chosen since this method has the best SNR.

$$H_{\text{cas}}(z) = H_4(z)H_3(z)H_2(z)H_1(z) \quad (9.41)$$

This gives the following cumulative frequency response. MATLAB script can be found on the CD

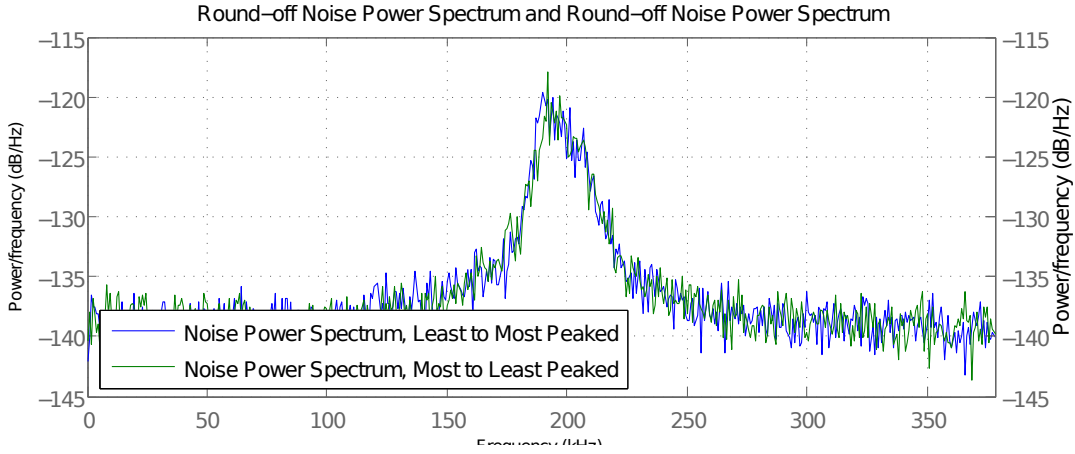


Figure 9.4: Noise spectrum showing two different orderings.

 /code/matlab_filterdesign/chan_filt_coef.m

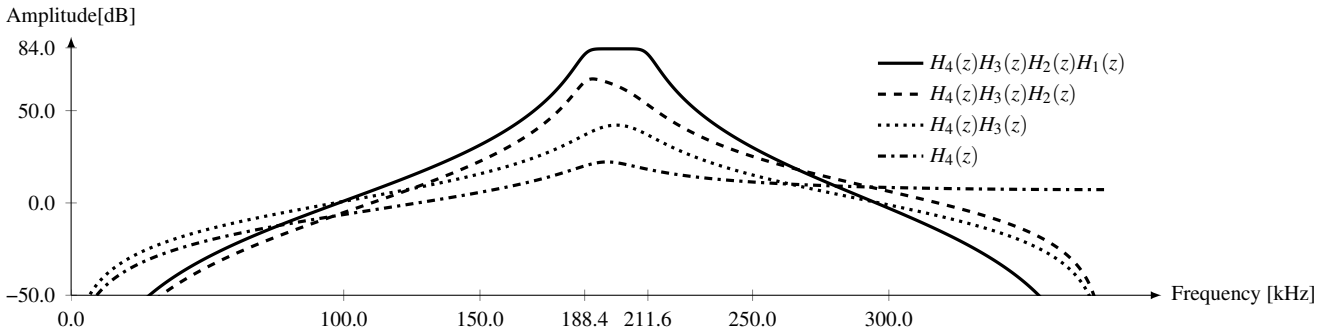


Figure 9.5: Cascaded filter without gain considerations

From Figure 9.5 it is seen the filter got a high gain in the passband which almost certainly will give overflow. Scaling is therefore needed and adjusted to give the best possible SNR without overflow. If the gain factor is placed at the first section only, the input signal level is reduced (gain less than one). To get an overall gain of one in the filter, the following sections will need a higher gain, amplifying the quantization noise [OS10, page 477]. Instead the gain is distributed throughout the sections, avoiding overflow in each stage by ensuring a gain less than one.

$$s_1 \max |H_4(e^{j\omega})| < 1 \Rightarrow s_1 = 0.078 \quad (9.42)$$

$$s_1 s_2 \max |H_4(e^{j\omega}) H_3(e^{j\omega})| < 1 \Rightarrow s_2 = 0.101 \quad (9.43)$$

$$s_1 s_2 s_3 \max |H_4(e^{j\omega}) H_3(e^{j\omega}) H_2(e^{j\omega})| < 1 \Rightarrow s_3 = 0.056 \quad (9.44)$$

$$s_1 s_2 s_3 s_4 \max |H_4(e^{j\omega}) H_3(e^{j\omega}) H_2(e^{j\omega}) H_1(e^{j\omega})| < 1 \Rightarrow s_4 = 0.153 \quad (9.45)$$

The transfer function for the 8th order Butterworth band-pass to be implemented in cascades can then be de-

scribed with its coefficients as:

$$H_{\text{cas}}(z) = s_1 H_4(z) s_2 H_3(z) s_3 H_2(z) s_4 H_1(z) \quad (9.46)$$

$$s_1 H_4(z) = \frac{0.078 - 0.155z^{-1} + 0.078z^{-2}}{1 + 0.091z^{-1} + 0.836z^{-2}} \quad (9.47)$$

$$s_2 H_3(z) = \frac{0.101 + 0.202z^{-1} + 0.101z^{-2}}{1 + 0.227z^{-1} + 0.837z^{-2}} \quad (9.48)$$

$$s_3 H_2(z) = \frac{0.056 - 0.112z^{-1} + 0.056z^{-2}}{1 - 0.005z^{-1} + 0.929z^{-2}} \quad (9.49)$$

$$s_4 H_1(z) = \frac{0.153 + 0.306z^{-1} + 0.153z^{-2}}{1 + 0.337z^{-1} + 0.930z^{-2}} \quad (9.50)$$

Even though the filter is scaled to have an overall gain of one, overflow can still occur in addition nodes [GK07, page 246]. Inputs of high values is therefore still a risk of overflow where scaling of the input signal might be needed. Overflow in IIR filters are highly damaging due to the recursive nature of the filter. A way to limit the error of overflow is to use saturation.

9.4 Fixed Point Effects

For implementation on the fixed point DSP, the double-precision floating point filter coefficients has to be quantized. The filter coefficients is chosen to be implemented as Q15, giving the maximum precision of 2^{-15} in expense of dynamic range ($1 - 2^{-15}; -1$). If both coefficients and data is in Q15 format multiplications will not overflow, unless both operators are -1 . Even with the high precision there will be truncation/rounding. The quantization effect is examined using MATLAB's FDATOOL, to ensure the filter still fulfills the requirements and still being stable.

It is seen from Figure 9.6 that the high precision quantization almost has no effect on the pole placement. The filter is thereby still stable since the poles are within the unit circle.

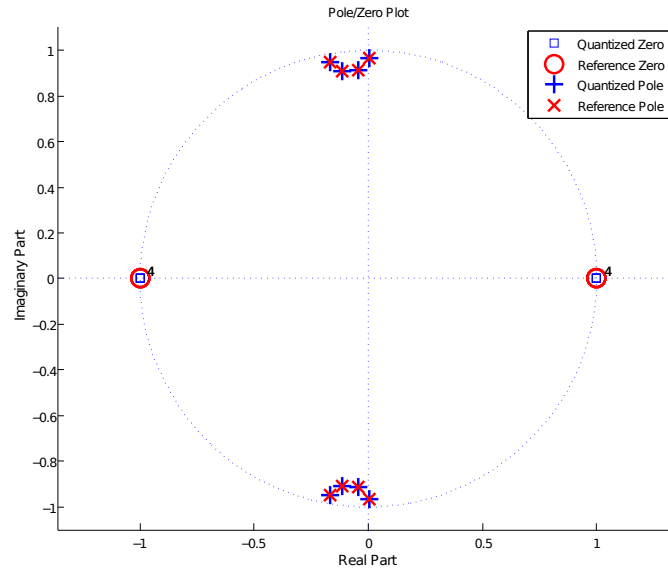


Figure 9.6: Pole-Zero map of quantised filter and reference.

Likewise the simulated frequency response is shown to be near identical to the reference as seen in Figure 9.6, and the filter still fulfills the requirements.

Before the filter design can be confirmed the impulse response is examined. With the real-time aspect in mind, the impulse response needs to be shorter than one bit-period, T_b , to avoid ISI (Intersymbol Interference). The filters impulse response is shown in Figure 9.8.

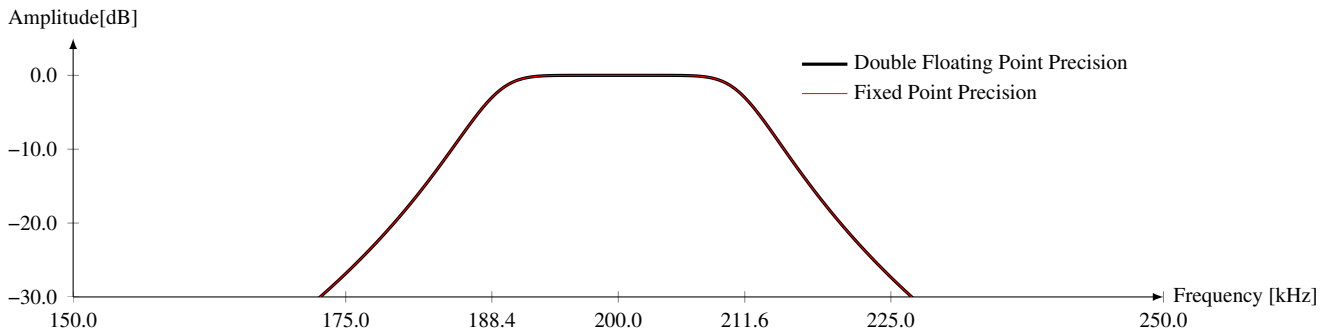


Figure 9.7: Cascaded unity gain filter, in both double floating point precision and fixed point precision

Due to the impulse response requirement listed in Table 5.2, the settling time has to be within 1% of one symbol period. Since one bit period is given by $T_b = \frac{1}{9600 \text{ b/s}} = 105 \mu\text{s/b} \approx 79 \text{ samples per symbol}$, it is seen from Figure 9.8 that the filter fulfills the requirement with a small margin.

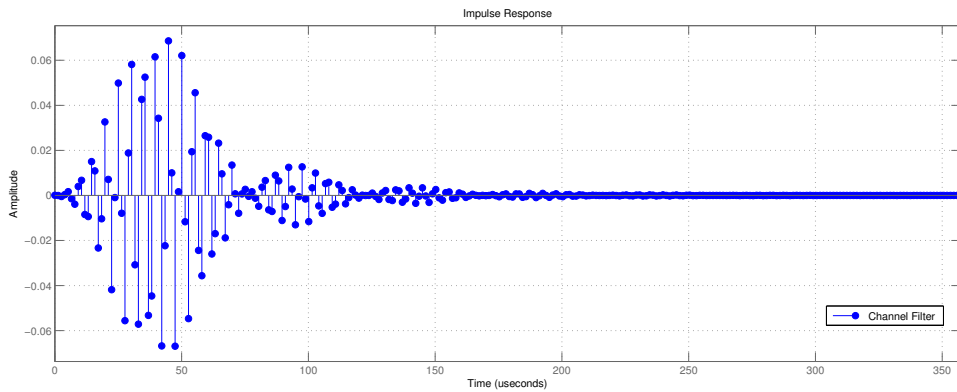


Figure 9.8: Impulse response of 8th order band-pass filter

To see if the filter is able to run in real-time, the computation complexity is examined for filtering of one channel. The filter takes 5 multiplications and 4 additions for each biquad as seen from Equation 9.27, giving a total of 20 multiplications and 16 additions per sample. Since the DSP is able to perform 2 multiplications and 2 additions in each cycle, in a ideal case, 10 cycles is needed per sample, for a fully optimized algorithm, neglecting data gathering. Since a buffer contains 256 KiS from the one channel, the total number of MAC operations is calculated as:

$$\text{MACs} = 10 \text{ cycles} \cdot 256 \text{ KiS} = 2621440 \text{ cycles} \quad (9.51)$$

Compared to max cycles available as seen from Equation 3.2, the filtering of one channel occupy the DSP in the following amount.

$$\text{Filter usage} = \frac{2621440 \text{ cycles}}{142433491 \text{ cycles}} = 1.8 \% \quad (9.52)$$

The filter should thereby be able to run in real-time, under the assumption that the optimal design is implemented and it is only necessary to filter one of the needed channels. The optimal real-time filter would be designed in Assembly, which the would be possible by directly translating the C filter which has been proposed here. Since all the sub-systems now have been designed they will be implemented on the Blackfin DSP in the following chapter.

Part III

Implementation

Chapter 10

GFSK Demodulator in Blackfin C

Now that the demodulator has been successfully executed and tested in floating-point C on a PC with files as input and output, the next step is to get it working on the Blackfin DSP. There are several challenges in doing so. First of all, the Blackfin is a fixed-point processor; second, the input should be sampled from the ADC and not, as before, from a file; and third, all computations should be done in real-time as specified in the requirement specification. This section describes the changes made from implementation in PC C to the final prototype in Blackfin C. The descriptions in this chapter will be structured from the interface diagram shown in figure 5.1, describing one sub-system at a time. Along with the description of each sub-system follows an individual test with emulated signals from the signal generator. The tests are all structured into the 5 items:

- Purpose
- Test Method
- Expected Result
- Result
- Conclusion of Test

10.1 Sampler

The sampler is described in Section 3.1. In this prototype implementation the I and Q data are extracted and placed in an I -buffer and a Q -buffer respectively. The code is shown in Listing 10.1.

```
1 uint32_t *buf32 = (uint32_t *)buffer; //buffer is the raw buffer
2 fract16 *buf16I; //Pointer to free memory
3 fract16 *buf16Q; //Pointer to free memory
4 while (n < SAMBLOCK/2) { // SAMBLOCK = 2 MiB
5     buf16I[i] = (fract16)((buf32[n] << 4)&0xFFFF);
6     n++;
7     buf16Q[i] = (fract16)((buf32[n] << 4)&0xFFFF);
8     n++;
9     i++;
10 }
```

Listing 10.1: Extracting I and Q data from raw buffer

10.1.1 Test of Input Sorting

Purpose

This test has the purpose of verifying that the sorting is done correctly, so that the data has the correct format as input to the main program as specified in Section 5.3.1.

Test Method

The first 32 samples of size 16 b from the raw buffer are investigated in this test. Every second of these samples, counting from sample 2, should be discarded. The test setup is shown in Appendix C.3.

Expected Result

Every fourth sample counting from 1 should be in the *I* file. Every fourth sample counting from 3 should be in the *Q* file.

Result

A hex dump of the raw buffer, the *I*, and the *Q* file are shown in Listing 10.2.

```

1  # Raw buffer
2  0000000: e10f 0000 d90f 0000 e10f 0000 2f00 0000  ...../...
3  0000010: 3000 0000 1b00 0000 0f00 0000 db0f 0000  0.....
4  0000020: e00f 0000 0300 0000 0a00 0000 2200 0000  ....."...
5  0000030: 1d00 0000 fb0f 0000 fc0f 0000 f40f 0000  .....
6
7  # I file
8  0000000: 90fd f002 b001 b0fd 3000 2002 b0ff 40ff  .....0. ...@.
9
10 # Q file
11 0000000: 10fe 10fe 0003 f000 00fe a000 d001 c0ff  .....

```

Listing 10.2: Hex dump of the first 32 samples from the raw buffer and the corresponding samples from the *I* and *Q* files. The dump is made with the `xxd` program running on a Linux PC.

The bits in the listing are ordered so that, for example, `e10f` should be read as the 16 b number `0fe1`. Four left-shifts corresponds to removing the left hex number and appending a zero from the right.

The Average number of clock cycles taken to sort the raw buffer into an *I* and a *Q* buffer: 109 263 467.

Conclusion of the Test

It has been found out, that the *Q* samples come before the *I* samples. Therefore it is clear that the buffer is split up into *I* and *Q* samples as expected. A very important result from this test is that number of instructions from this operation of moving data is extremely high, especially when it is considered that this operation not is needed, since the channel filter can be used to sort the input.

10.2 Channel Filter

The channel filter is implemented in direct form II cascades as specified in Section 9.3. The filter coefficients is read from a binary file under initialization and pointed to in the function call. The filtering calculations are shown in Listing 10.3. With the input being either the *I* or the *Q* channel, the input buffer is looped through sample by sample. Both filter coefficients and data are in Q15 format. For each iteration the 16×16 b multiplication gives a 32 b result with two sign bits, of which a left shift needed. To store the 32 b value to a 16 b register, the data is shifted 16 places to the right, resulting in a total of 15 shifts to the right. Lastly the output is stored and the delay line is updated for the next iteration.

```

1  /**
2   * Butterworth 8th order direct form II biquad band-pass filter.
3   * @param input Pointer to input data location.
4   * @param output Pointer to output location.
5   * @param coeff Pointer to coefficient location.
6   * @return void
7   */
8  void biquad_df2(fract16 *input, fract16 *output, fract16 *coeff, uint32_t N) {
9      fract16 xn=0; // data in Q15
10     fract16 y1=0; fract16 y2=0; fract16 y3=0; fract16 y4=0;
11     fract16 w1n1=0; fract16 w1n2=0; fract16 w2n1=0; fract16 w2n2=0;
12     fract16 w2n=0; fract16 w2n1=0; fract16 w2n2=0;

```

```

13     fract16 w3n=0; fract16 w3n1=0; fract16 w3n2 = 0;
14     fract16 w4n=0; fract16 w4n1=0; fract16 w4n2 = 0;
15
16     int i;
17     for (i = 0; i < N; i++) {
18
19         xn = input[i];
20
21         w1n = ((xn<<15) - coeff[3]*w1n1 - coeff[4]*w1n2) >> 15;
22         y1 = (coeff[0]*w1n + coeff[1]*w1n1 + coeff[2]*w1n2) >> 15;
23
24         w2n = ((y1<<15) - coeff[9]*w2n1 - coeff[5]*w2n2) >> 15;
25         y2 = (coeff[6]*w2n + coeff[7]*w2n1 + coeff[8]*w2n2) >> 15;
26
27         w3n = ((y2<<15) - coeff[13]*w3n1 - coeff[14]*w3n2) >> 15;
28         y3 = (coeff[10]*w3n + coeff[11]*w3n1 + coeff[12]*w3n2) >> 15;
29
30         w4n = ((y3<<15) - coeff[19]*w4n1 - coeff[15]*w4n2) >> 15;
31         y4 = (coeff[16]*w4n + coeff[17]*w4n1 + coeff[18]*w4n2) >> 15;
32
33         output[i] = y4;
34
35         w4n2 = w4n1;
36         w4n1 = w4n;
37         w3n2 = w3n1;
38         w3n1 = w3n;
39         w2n2 = w2n1;
40         w2n1 = w2n;
41         w1n2 = w1n1;
42         w1n1 = w1n;
43     }
44 }

```

Listing 10.3: 8'th order band-pass filter implemented in DF2 biquads.

10.2.1 Test of Channel Filter

Purpose

The test is done in order to test the requirements as listed in Section 5.3.2. This includes testing the frequency characteristics of the channel filter.

Test Method

The filter is tested with a normalized Q15 sweep to derive the frequency response.

Expected Result

The cutoff frequencies of the channel filter are expected to be placed as in the fixed point design.

Result

Conclusion of the Test

The channel filter is proven to reduce noise at unwanted frequencies, as shown in Figure 10.1.

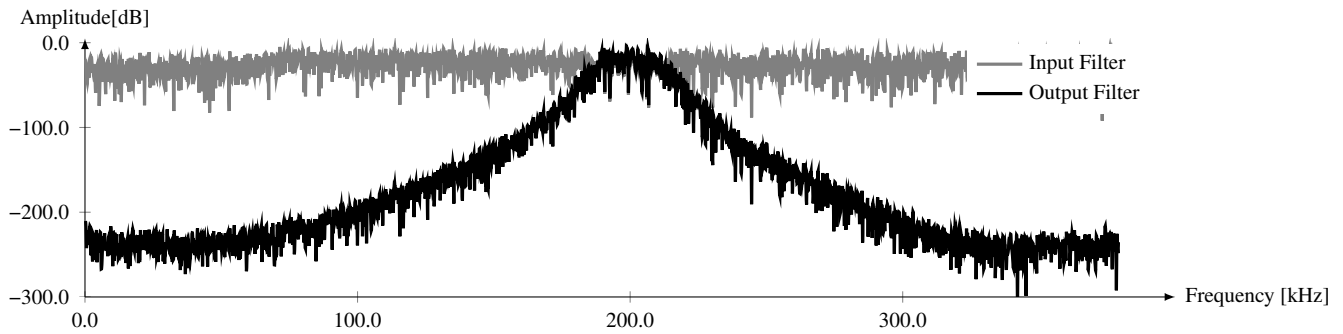


Figure 10.1: The implemented filter tested with an input of a normalized Q15 sweep.

10.3 Packet Detection

Packet detection on the Blackfin is done in the same way as in the floating-point implementation described in Section 8.2. Some minor changes have been made, though.

The buffer is now stored in fixed-point and the packet must be detected using fixed-point, Q15 arithmetic. This means, that 15 right-shifts must be performed after every multiplication and one must be aware of overflow, as described in Section 2.2.4.

10.3.1 Test of Packet Detector Precision

The packet detector is tested due to the requirements as specified in Table 5.3.

Purpose

When using double sliding windows the packet should be detected within the window length, or else the energy in both windows will equal and no packet will be detected. The purpose of the test is thereby to see if the packet detector finds the packet within 2 symbols.

Test Method

A packet is sent from the signal generator, and the packet detector outputs the sample at which the packet start has been detected. This is done without the channel filter. The input is written to a file for manually checking the packet start for comparison. Note the test is done under the condition that a whole packet is contained within a buffer. The test setup is shown in Appendix C.3.

Expected Result

The found packet is expected to be within two symbols, or else the packet will not be found at all.

Result

The packet detector finds the packet in sample number 141 931, which in time corresponds to $141931/f_s = 141931/758272 = 0.187\text{s}$. The packet is manually found by inspecting Figure 10.2 to be at sample 141 828, giving a difference of 103 samples. This is less than the required two symbols being 158 samples.

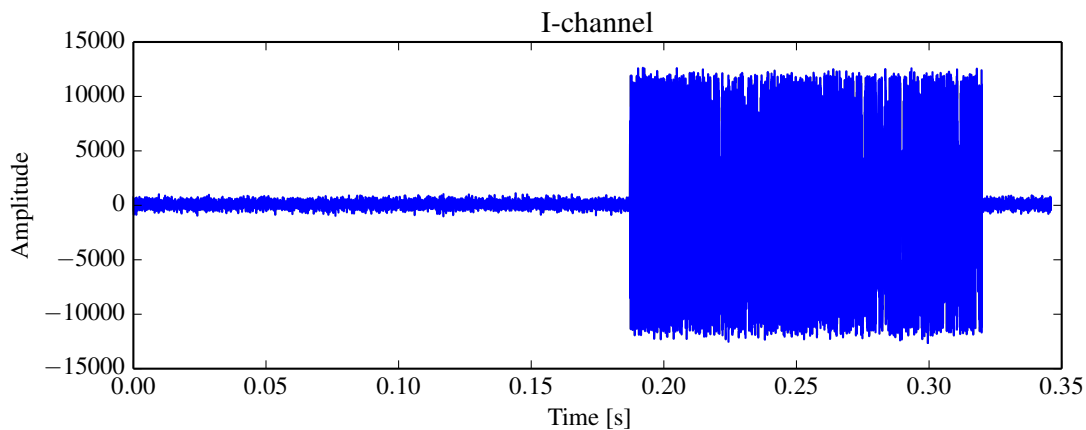


Figure 10.2: Input to the packet detector under test.

Conclusion of the Test

Since the packet is detected within two symbols the packet detection is shown to work.

10.4 Mid-Frequency Estimation

When a packet has been detected a mid-frequency estimation is made. This is done using the principles as described in Section 3.4. It is known that the start of a packet is training sequence as explained in Section 2.3. The first part of the in-phase packet signal is used as input in the mid-frequency estimation. The output is a mid-frequency in Hz.

The first thing to do is to initialize a twiddle-table using the `twidffttrad2_fr16` function from the DSP run-time library [Anac, page 4-173]. This is only done once.

```
1 static complex_fract16 twiddle[FFTLEN/2];
2 twidffttrad2_fr16(twiddle, FFTLEN);
```

Listing 10.4: Snippet for initializing twiddle-table

When a packet is detected the first thing needed is to copy `FFTLEN` amount of the in-phase signal into `fftin`. `FFTLEN` is the number of points in the FFT and `fftin` is the input to the FFT. The memory copying is done using the `memcpy` function from the C run-time library [Anac, page 3-226]. Note from the code in Listing 10.6 that the pointer to the data is the pointer to the start of the in-phase signal plus the sample number at which a packet is detected plus five additional symbols (SPS is the number of samples per symbol). The five extra symbols serve as a safety margin that makes sure the training sequence has reached a stable situation.

The FFT is computed using the `rfft_fr16` function from the DSP run-time library [Anac, page 4-163].

```
1 void rfft_fr16( const fract16      input[],
2                complex_fract16    output[],
3                const complex_fract16 twiddle_table[],
4                int                  twiddle_stride,
5                int                  fft_size,
6                int                  *block_exponent,
7                int                  scale_method);
```

Listing 10.5: `rfft_fr16` function call

It is an FFT that uses a real input and thereby takes advantage of a zero-valued imaginary part to half the amount of multiplications in the butterfly. The function is used as seen from in Listing 10.6, `input[]` is self-explanatory, `fftout` is the complex output, `twiddle_table[]` is the pre-calculated twiddle-table, `twiddle_stride` is set to 1 to use the full twiddle-table, `fft_size` is the number of points in the FFT, `*block_exponent` is not used but indicates how the signal has been scaled, and `scale_method` equal to 1 indicates the use of static

scaling. The size of the FFT is as described in Section 3.4 not crucial to the real-time aspect. It is therefore chosen to make a 8192-point FFT.

Since the output of the FFT is complex it is needed to calculate the absolute values. This is done using the `cabsasm` function from an earlier AAUSAT study. It is a function written in Assembly for optimal use on the DSP. Note from the code in Listing 10.6 that only half of the output of the FFT is used. Thus, removing the mirrored output.

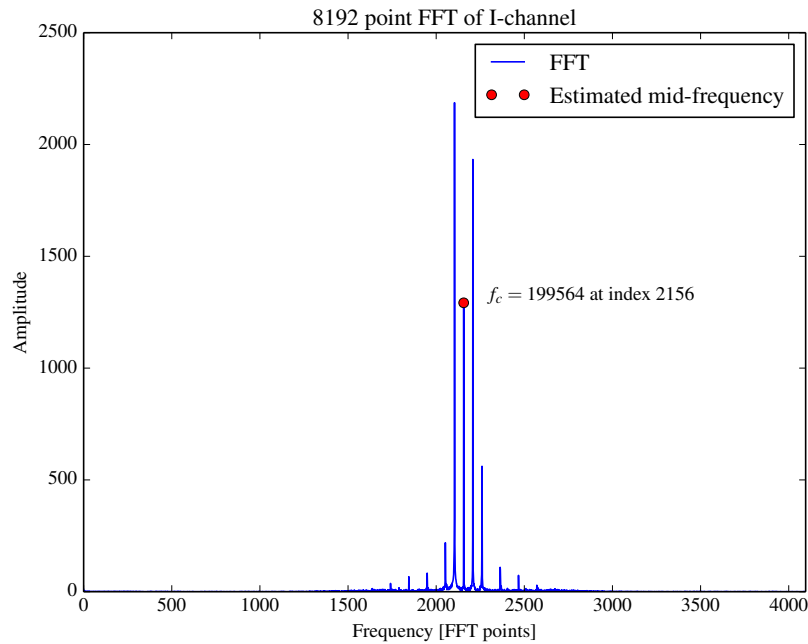


Figure 10.3: 8192-point FFT computed on the Blackfin. The red dot shows the location of the estimated mid-frequency.

As described in Section 3.4 the mid-frequency may be estimated using more than one peak. As seen in Figure 10.3 the zero- and one-frequency have the highest peaks. The mid-frequency is placed in the middle of these. The FFT as shown in the figure is computed on the Blackfin using the start of a packet as described above. A packet is received from the signal generator as described in Appendix C.1. To optimize the process of locating peaks, the peaks should only be searched for within an acceptable frequency range since there is no reason to check for peaks outside the channel.

Finally the argument of the estimated mid-frequency—call it x —is converted to Hz using the relation:

$$f(x) = x \frac{f_s}{\text{FFTLEN}} \quad (10.1)$$

where $f_s = 758272$ Hz. To achieve maximum precision the sampling frequency is shifted nine times right:

$$f(x) = x \frac{758272 \gg 9}{8192 \gg 9} = x \frac{1481}{2^4} = (x \cdot 1481) \gg 4 \quad (10.2)$$

The result is stored in `fract32` to contain numbers around 200 000 Hz.

```
1 int be;
2 fract32 fc;
3 memcpy(fftin, (fract16*)(buf16I+pstart+5*SPS), FFTLEN);
4 rfft_frl6(fftin, fftout, twiddle, 1, FFTLEN, &be, 1);
5 cabsasm(fftout, fftabs, FFTLEN/2);
6 fc = argmax(fftabs, FFTLEN/2);
7 fc = (fc * 1481) >> 4; // (fc*(758272>>9)) / (FFTLEN>>9)
```

Listing 10.6: Snippet for estimating the mid-frequency

As shown in Figure 10.3 the output from the mid-frequency estimator could for example be $f_c = 199564$ Hz.

10.4.1 Test of Mid-Frequency Estimator

Purpose

This test has the purpose of verifying that the mid-frequency estimation is within the limits as specified in Table 5.4.

Test Method

A start of a packet is detected using the packet detector module. Then a mid-frequency estimation is performed on the first part of the packet. The estimated mid-frequency is compared to the actual mid-frequency. The test setup is shown in Appendix C.3.

Expected Result

A mid-frequency estimated within 4800 Hz of the actual mid-frequency is expected as a result of this test. As a 8192-point FFT is used, the mid-frequency estimate has a resolution of 92.6 Hz as seen in Table 3.2.

Result

The test was done twice with the same result. The actual mid-frequency was located at 199 567 Hz. In both tests the output of the mid-frequency estimator was 199 564 Hz.

Conclusion of the Test

As seen from the results the mid-frequency estimator is able to estimate the mid-frequency within the requirement.

10.5 Time Synchronization

After the mid-frequency of the signal has been estimated it is needed to estimate the placement of symbols in time. This is done using the principles as described in Section 3.5. As a result of the time synchronization the placement of each symbol in the packet is known allowing for decision of symbols.

The time synchronization implemented differ from the one used in the MATLAB program. Instead of locating the first minimum of the downconverted phase-signal a cross-correlation with a pre-computed training phase-signal is performed. Using this method makes the time synchronization more robust against errors in the mid-frequency estimation. If the signal is downconverted using a wrong mid-frequency the phase-signal will have a slope making the find-first-minimum-method inaccurate. In that case the cross-correlation is a better choice as described in Section 3.5.

To be able to cross-correlate with the training signal the phase of the downconverted received signal has to be calculated. The signal is downconverted using linear subtraction as described in Section 3.5.1. This means that the phase of the signal is computed before it is downconverted.

The phase signal is found by using the `arg_fr16` function from the DSP run-time library [Anac, page 4-55]. It takes a complex input and outputs values from 0 to 32767 in a similar way as the `atan2` function. The `arg_fr16` function wraps the phase signal from 0 to 2π from which it follows that a value of 32767 corresponds to 2π . The angle is shifted up once, hereby intentionally overflowing the `fract16` format using the full range from -32768 to 32767. This makes the result wrap from $-\pi$ to π for which 32767 corresponds to π .

To downconvert the phase signal the slope as shown in Equation 3.20 is used. The slope noted a is determined by $\frac{2\pi f_c}{f_s}$ where $\pi \equiv 32768 = 2^{15}$ and $f_s = 758272$ Hz. The intermediate value $\frac{2\pi}{f_s}$ is shifted up by 16 to give a better precision, shifting down by the same factor after multiplying with f_c . This means that to calculate a the following is done:

$$a = \frac{2 \cdot 2^{15} \cdot f_c}{f_s} = \left(\left(\frac{2^{16}}{f_s} \ll 16 \right) f_c \right) \gg 16 \approx (5664 \cdot f_c) \gg 16 \quad (10.3)$$

With expected mid-frequencies around 200 kHz the resulting a will be approximately 17 000. When subtracting the

slope the `fract16` will overflow but as the phase-signal is using full range this is going to be correctly wrapped.

To secure that no wrapping occurs in the downconverted phase-signal in the time period where the correlation is performed, the signal is shifted one bit to the right and unwrapped using a method similar to the one used in Section 8.4. The code is shown in Listing 10.7.

```

1  fract16 *phi = (fract16 *)LPMEM;
2  fract16 *phi_down = (fract16 *)DFDMEM;
3  fract16 a = (5664*fc) >> 16; // ((2^16<<16)/758272 * fc) >>16;
4  for (i=1; i < CORRLen; i++) {
5      phi[i] = arg_fr16(input[i]) << 1;
6      phi_down[i] = (phi[i] - (i*a)) >> 1;
7      phi_down[i] = unwrap(phi_down[i-1], phi_down[i]); // wraps at 8192
8  }

```

Listing 10.7: Snippet for downconverting and calculating the phase signal

Time synchronization is performed in the file found on the CD:

 /code/bf_gfsk_workbench/subsystems/gnd/program/dnow/dnow.c

The downconverted phase-signal is cross-correlated with a triangle wave. An example of a downconverted phase-signal and a generated triangle wave is shown in Figure 10.4a. As seen from the figure the spacing between symbols, T_b , in the generated triangle wave is the same as the spacing between symbols in the downconverted phase-signal. A `CORRLen` of 20 symbols is used. Both signals are computed on the DSP using a packet received from the signal generator as described in Appendix C.1.

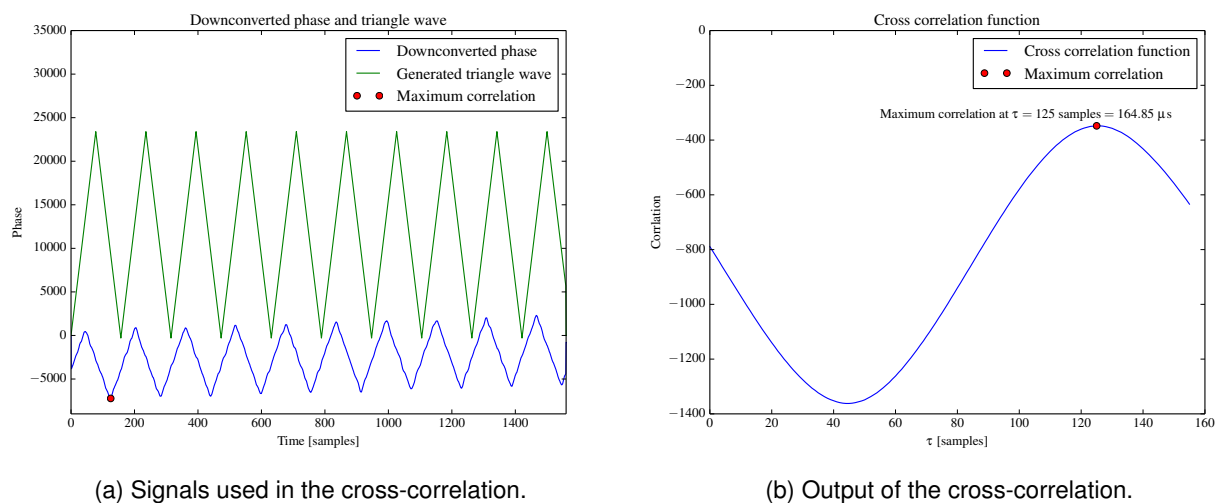


Figure 10.4: Figure (a) shows the green generated triangle wave, the blue downconverted phase-signal and the output location marked with a red dot. Figure (b) shows the output of the cross-correlation and the located maximum marked with a red dot.

The triangle wave is generated as shown in Listing 10.8.

```

1  int generate_triangle(fract16 *output)
2  {
3      fract16 i;
4      fract16 sign = 1;
5      fract16 cnt = 0;
6      for (i = 0; i < CORRLen; i++) {
7          if (F_SAMPLE/BITRATE <= cnt || cnt < 0 )
8              sign = -sign;
9          output[i] = cnt*300;
10         cnt += sign;
11     }
12     return 0;

```

13  }

Listing 10.8: Snippet for generating a triangle wave

The code generating the triangle wave can be found on the CD:

 `/code/bf_gfsk_workbench/subsystems/gnd/program/gfsk/spmath.c`

Finally the cross-correlation is done using the `crosscorr_fr16` function from the DSP run-time library [Anac, page 4-99]. The function takes two input signals, in this case `triang` and `phi_down`. The input signals have the length `CORRLEN`. The cross-correlation is calculated for `LAGS` elements and stored in `cc`. The argument of the maximum of `cc` is the location of the nearest minimum since the triangle wave starts with a minimum. An example of an output from the cross-correlation computed on the Blackfin is shown in Figure 10.4b. The code is shown in Listing 10.9.

```
1 crosscorr_fr16(triang, phi_down, CORRLEN, LAGS, cc);
2 corrmx = argmax(cc, LAGS);
```

Listing 10.9: Snippet for performing cross-correlation and finding time delay

The code for doing cross correlation can be found on the CD:

 `/code/bf_gfsk_workbench/subsystems/gnd/program/dnow/dnow.c`

The result of this time synchronization is the location of the first minimum in the phase-signal. In Figure 10.4a this output is showed using the red dot. The remaining symbols are spaced with T_b .

10.5.1 Test of Time Synchronization

Purpose

This test has the purpose of verifying that the time synchronization respects the requirements as specified in Table 5.5. The time synchronization must be able to detect the first symbol with a precision down to four samples.

Test Method

A start of a packet is detected using the packet detector module. The mid-frequency estimation is performed on the first part of the packet. Then the time synchronization is performed and the output location is noted.

Expected Result

It is expected that the time synchronization locates a zero in the training sequence within a range of 4 samples.

Result

The start of the packet was detected at sample 219 041. The start of the data used in the time synchronization is at sample 219 436. The output of the time synchronization was 131 samples. This added to the start of the packet plus the five symbols safety margin is the location of the first zero. Thus, the first zero should be located at sample 219 567 which corresponds to a time of 0.28956 seconds as seen in Figure 10.5. The blue phase signal in the figure is computed with a Python script using the recorded in-phase and quadrature signals. It serves as a measure of precision of the output location.

The plotting script is found on the CD:

 `/data/acceptancetesting/006_timesync_accuracy_finding_first_minimum/plotwithsy`

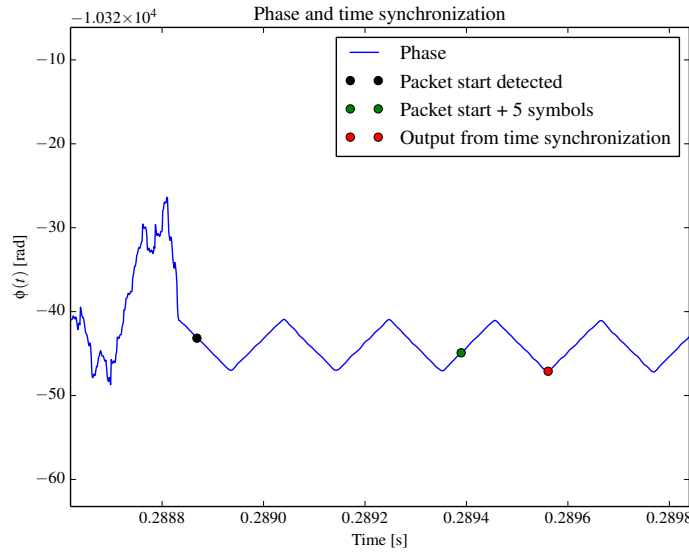


Figure 10.5: The black dot is the location of the packet start detected. The green dot is the packet start plus five symbols of safety margin. The red dot is the output location of the time synchronization. The blue phase signal is generated in MATLAB from the in-phase and quadrature output of the demodulator.

Conclusion of the Test

It seems reasonable to verify that the output location of the time synchronization is within a wanted sample range. It is however a rough confirmation that it is within a range of four samples when concluded from Figure 10.5.

A test of the time synchronization with an error in the estimated mid-frequency is implied in the test as shown in Section 10.6.2. It can be seen from Figure 10.7a that the first zero is located with the same precision as in the test above. Thus it can be concluded that the time synchronization works within the required limits of error in the estimated mid-frequency.

10.6 Symbol Decision

After the position of the first symbol, t_0 , is determined, the position of future symbols will be placed at

$$t_n = t_0 + nT_b \quad n = 1, 2, \dots \quad (10.4)$$

The phase is evaluated at these points in time. The symbol decision is done by unwrapping the phase as described in Section 3.6.

The above method of determining the position of future symbols is assuming that the symbol distance, T_b , is known exactly. However, the actual value of T_b , for a bit rate of 9600 b/s and a sampling rate of 758 272 b/s, is

$$T_b = \frac{f_s}{R} = \frac{758272}{9600} = 78.9867 \text{ samples/symbol} \quad (10.5)$$

This is very close to, but not exactly, 79 samples/symbol. This means that if $T_b \equiv 79$, the point of which the phase is evaluated will drift as n becomes larger. After a number of symbols, n_{drift} , the drift will result in a whole symbol will have been skipped. For the above values, n_{drift} is

$$\begin{aligned} 79n &= 78.9867(n+1) \\ \frac{n}{n+1} &= \frac{78.9867}{79} \\ n_{\text{drift}} = n &= \frac{78.9867/79}{1 - 78.9867/79} = 5938.85 \text{ symbols} \end{aligned} \quad (10.6)$$

Including 200 ms of training sequence, the longest data packet (250 B), a 6 B sync word, and a 1 B frame size marker, the number of symbols per packet is

$$n_{\text{packet}} = 200 \cdot 8 + 250 \cdot 8 + 6 \cdot 8 + 8 = 3976 \text{ symbols} \quad (10.7)$$

This means that t_n will have drifted more than half a symbol at the end of a packet. This will cause the wrong symbol decision to be made.

A better estimate of T_b can be made by varying it between 78 samples/symbol and 79 samples/symbol so the average will be closer to 78.9867 samples/symbol. By choosing every 100 T_b to be 78 and the rest to be 79, the average will be

$$T_b = \frac{78 + 99 \cdot 79}{100} = 78.99 \text{ samples} \quad (10.8)$$

which will cause the drift of one symbol to occur at

$$n_{\text{drift}} = \frac{78.9867/78.99}{1 - 78.9867/78.99} = 23935.36 \text{ symbols} \quad (10.9)$$

By using this technique, the drift of one symbol will occur after $23935.36/3976 \approx 6$ packets with full training sequence. Because t_0 is reevaluated each time a new packet is detected, the drifting effect should be eliminated when this method is used.

As a first try $T_b = 79$ samples/symbols is chosen. In case it becomes necessary, the method described above can be implemented in C as shown in Listing 10.10.

```

1  int n;
2  int N = NUMBER_OF_SYMBOLS_TO_DEMODULATE;
3  int SPS; // SPS: Samples per symbol
4  int t[N];
5
6  t[0] = OUTPUT_FROM_CROSSCORRELATOR;
7  for (n = 1; n < N; n++) {
8      if (n % 100)
9          SPS = 79;
10     else
11         SPS = 78;
12     t[n] = t[0] + n*Tb;
13 }
```

Listing 10.10: Handling T_b drift in C. Here SPS corresponds to T_b samples/second.

The decision utilizes the symbolwise slope as shown in Equation 3.23. This means that the phase is computed at every T_b spaced sample instead of every sample. The phase at T_b is compared to the phase at $T_b + T_b/2$ subtracted by the slope added by the mid-frequency as shown in Listing 10.11 just as done in the time synchronization above. The method used is described in Section 3.6.

```

1  tn = t0;
2  for (i = 0, n = 0; i < plength; i++) {
3      tn += SPS;
4      half_period = SPS/2;
5      phi_old = (arg_fr16(s[tn])) << 1;
6      phi_new = (arg_fr16(s[tn+half_period])) << 1;
7      phi_new -= (fract16)(half_period)*a;
8      phi_new -= phi_old;
9
10     if (0 < phi_new)
11         bits[i] = 1;
12     else
13         bits[i] = 0;
14
15     // Search for sync word here...
16 }
```

Listing 10.11: Snippet for decision of symbols

The search for sync word is not shown here but is implemented in the program:

```
 /code/bf_gfsk_workbench/subsystems/gnd/program/dnow/dnow.c
```

It is implemented as a simple check whether six bytes of the signal are equal to the sync word. This means that it does not handle symbol errors in the sync word meaning that a sync word with one or more symbol errors will not be recognized. When a sync word is located the FSM is demodulated to achieve the length of the packet. After this the rest of the packet can be demodulated.

10.6.1 Test of Decision Drifting in Symbol Decision

Purpose

The purpose of this test is to see if a short packet can be demodulated without errors due to drifting in the moment of decision, as required from Section 5.3.6. The drifting is caused by an inexact symbol period as specified in Section 10.6.

Test Method

A packet is sent to the demodulator, and the moments of decision are marked. The plot is examined to see if the moment of decision has drifted more than half a symbol over the duration of a short packet.

Expected Result

It is expected to demodulate 6000 symbols without errors based on Equation 10.6. It should therefore be able to demodulate a short packet with a length of 1272 symbols as specified in Appendix C.1.

Result

The symbol detections starts five symbols from the packet start too ensure a stable signal level as seen at Figure 10.6. The blue phase signal in the figure is computed with a Python script using the recorded in-phase and quadrature signals. It serves as a measure of precision of the output locations. The result of the test shows, that the moment of decision has not drifted more than half a symbol period for a short packet.

The plotting script is found on the CD:

```
 /data/acceptancetesting/007_symboldecision_drifting/plotwithsymbols.py
```

Conclusion of the Test

It is shown that a short packet can be demodulated without decision errors, with the current precision of T_b . It has not been tested with a long packet, but the symbol period can be specified with higher precision as described in Section 10.6 if needed.

10.6.2 Test of Symbol Decision With Slope

Purpose

The purpose of this test is to verify that the symbol decision works even with an error in the estimated mid-frequency.

Test Method

A start of a packet is detected using the packet detector module. The mid-frequency estimation is performed on the first part of the packet. An error of 4000 Hz is added to the estimated mid-frequency. The time synchronization is performed and finally the moments of decision are stored. The test setup is shown in Appendix C.3.

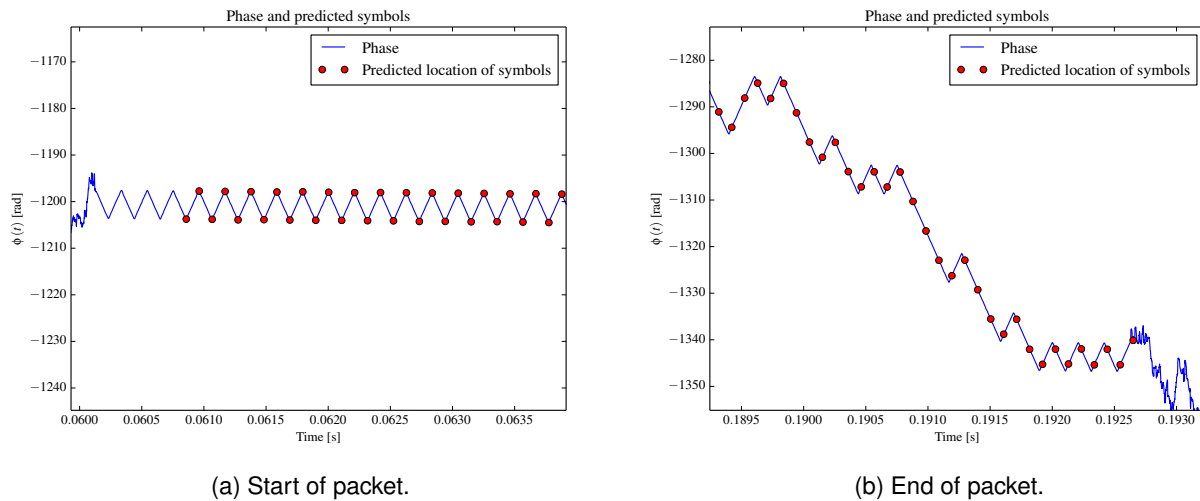


Figure 10.6: The predicted location of symbols generated in the symbol decision module marked with red dots. The blue phase signal is generated in MATLAB from the in-phase and quadrature output of the demodulator.

Expected Result

It is expected to demodulate the short packet with a length of 1272 symbols without errors due to a combination of drifting and slope.

Result

The result of the test is shown in Figure 10.7. Note that the full packet is not in the buffer resulting in a near end of packet in Figure 10.7a as only 1188 symbols are located in the buffer. The blue phase signal in the figure is computed with a Python script using the recorded in-phase and quadrature signals as well as the erroneously estimated mid-frequency. It serves as a measure of precision of the output locations.

The plotting script is found on the CD:

 /data/acceptancetesting/008_symboldecision_slope_4000Hz/plotwithsymbols.py

Conclusion of the Test

It can be concluded from Figure 10.7a that the time synchronization is correct. Furthermore Figure 10.7b shows that the moment of decision drifts less than half of a symbol as was the same conclusion from the previous test. Looking at the decisions near the end of the packet reveals that they will be correct. However it can be seen that transmitted ones are nearly decided to be zeros as a result of the drifting and the slope. Thus, the demodulation of a long packet with imprecise T_b and an error in the mid-frequency estimation will contain errors, where T_b has to be specified with higher precision as described in Section 10.6.

10.6.3 Test of Symbol Decision Finding Sync Word

Purpose

The sync word indicates the end of the training sequence and ID of the packet as specified in the Section 2.3. The symbol decision thereby has to recognize the sync word, in order to demodulate a packet. The test examines if the module is able to recognize the sync word. Likewise it is examined if the module recognizes the FSM.

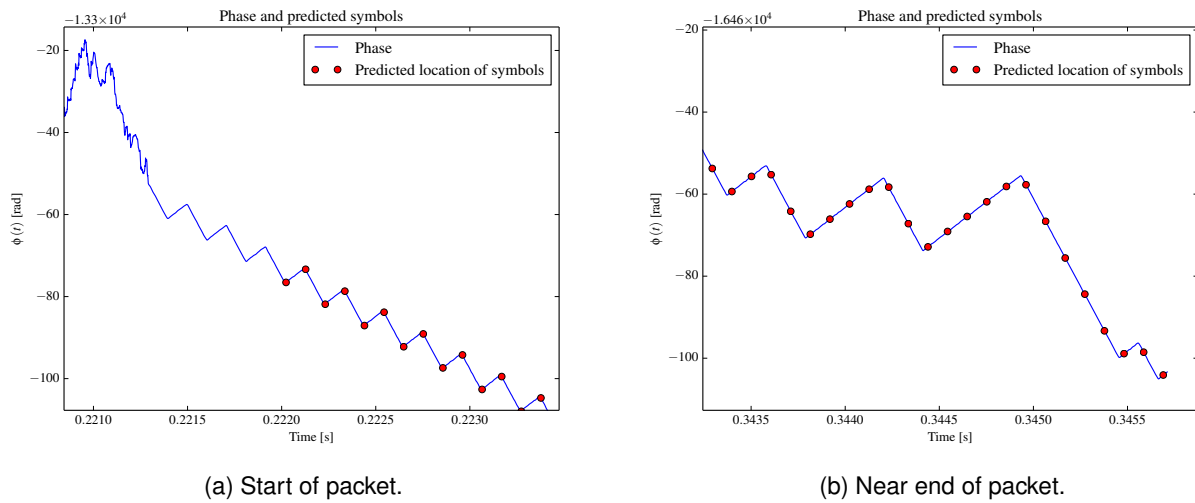


Figure 10.7: The predicted location of symbols generated in the symbol decision module marked with red dots. The blue phase signal is generated in MATLAB from the in-phase and quadrature output of the demodulator.

Test Method

In this test the full demodulator program is executed with the signal generator sending a short packet. The program locates the sync word, and is checked with the input data. After the sync word, the FSM is decoded and matched with the FSM of a short packet.

Under the test it was observed that the signal generators deviation setting apparently is the frequency from mid-frequency to f_0 and f_1 respectively. The deviation is through this report described as the frequency between f_0 and f_1 . The test is performed with the signal generator setting at 4800, giving a frequency difference between f_0 and f_1 at 9600 Hz, as shown by a FFT in Figure 10.8.

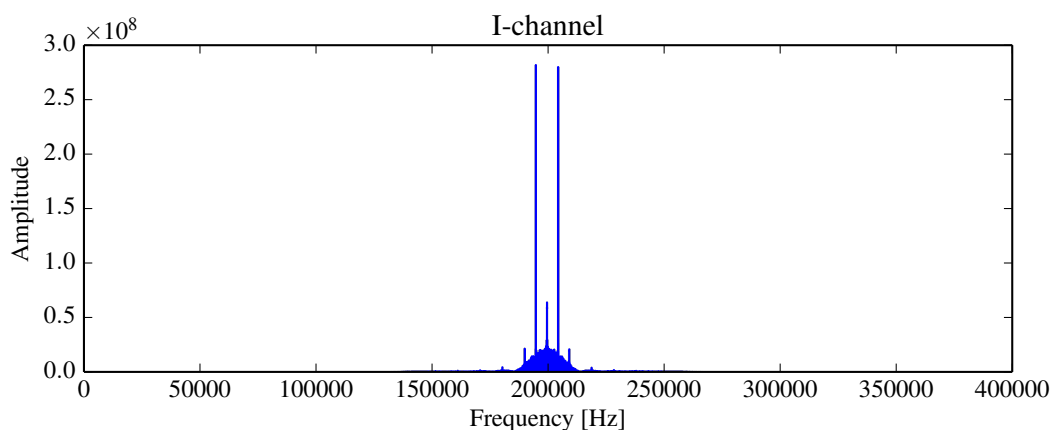


Figure 10.8: FFT of a packet from the signal generator with deviation set to 4800

Expected Result

It is expected to find the sync word OZ4CUB (ASCII), and the FSM A6 (hex).

Result

The program locates the sync word start at sample 147284, corresponding to 0.194 s with a sampling rate of 758272 Hz. This is verified by the spectrogram at Figure 10.9.

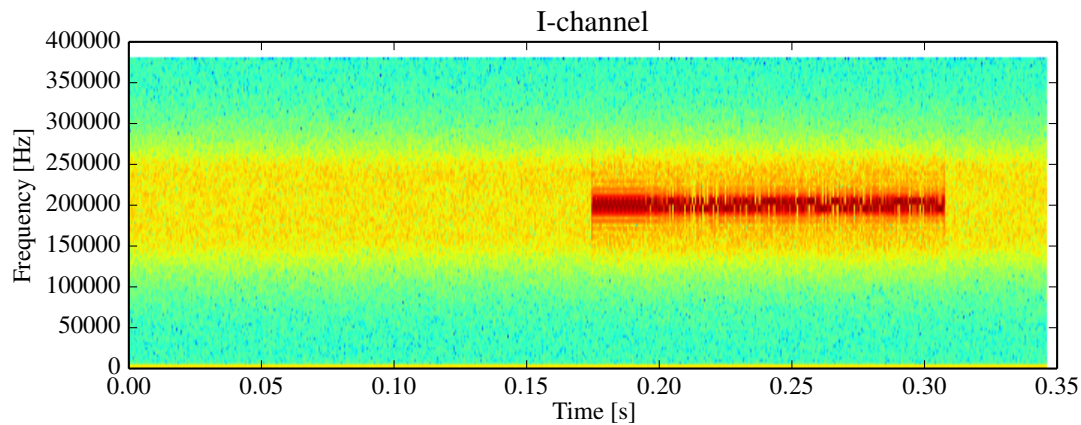


Figure 10.9: Spectrogram of a packet from the signal generator with deviation set to 4800

The FSM is decoded to be A6.

Conclusion of the Test

The test shows that the sync word and FSM can be found correctly.

10.6.4 Test of Symbol Decision

Purpose

The purpose of this test is to verify the decisions made in the symbol decision module. Whereas the first two tests concerned the locations of decision this last test confirms the actual decisions.

Test Method

The demodulator receives a packet from the signal generator which it demodulates. The demodulated symbols are compared to the symbols actually transmitted.

As explained in the previous test this test is also performed with a deviation of 4800.

Expected Result

It is expected that the demodulated packet corresponds to the one transmitted.

Result

The actual DLIST1 packet is shown in Listing C.1. The demodulated symbols without training sequence, sync word and FSM are shown in Listing 10.12.

```
8C 1A 48 CE F5 FF 9E 4D A0 C8 27 B1 18 AF 07 D0 D1 90 CD C0 A2 14 29 AF 13 1F 10 42 54 19
46 54 47 EE C9 C3 A2 A6 CE 5E A2 6B 40 73 10 AD 7E 72 CF 74 BB 1F 4D B4 31 5A 59 04 76 A4
56 1F 5A C9 A5 93 F5 DD 3A D1 68 9A 04 1D 02 46 61 7F E9 6F E8 1B 83 01 67 C5 52 7A BC 4F
CE C0 25 27 A8 AF 31 57 2F 70 1D B0 AC E8 92 AE 69 BE DA F4 34 88 7F 77 DE CC F3 E0 BC B9
8A 9B D6 66 68 94 22 55
```

Listing 10.12: DLIST1 packet demodulated in hex format

The demodulator outputs the correct symbols with no errors.

Conclusion of the Test

The first two tests concluded that the moments of decision in the short packet respected the requirements. In the third test the sync word and FSM were correctly located. Finally this last test verifies the decisions made throughout the short packet.

Throughout this chapter the modules of the demodulator have been implemented and tested accordingly. The tests are performed using a signal generator and show that the implemented demodulator modules respect the requirements. Drifting in the moment of decision does however seem to become a problem when dealing with long packets. Following these tests are the tests of the overall system.

Chapter 11

Acceptance Test of System

This chapter describes the tests and results made in order to test the requirements as specified in Section 5.2. The test serves to show how well the implemented system is able to keep up with its real-time constraints and to show how well the system is able to demodulate data received from an actual satellite.

11.1 Real-Time Summary

Now that the demodulator has demodulated a packet, optimization is needed in order to run in real-time. The total amount of cycles available is 142 433 491 cycles as specified in Equation 3.2.

Each module is tested for DSP usage as seen in Table 11.1, to get an overview and point out the most critical modules. The cycles is measured using the cycle count register, which is read before and after execution of each module. Note the cycles used for channel filtering are both the *I* and *Q* channel.

Module	Cycles Used	DSP usage in %
Sampler	109 263 467	77
Channel Filter	295 467 922	207
Packet Detector	26 435 831	19
Mid-Frequency Estimator	14 322 454	10
Time Synchronization	26 638 224	19
Symbol Decision	4 587 500	3
Total		335

Table 11.1: Cycle measurement before optimization

From the table it is quickly seen that the demodulator is not able to run in real-time without optimization. The largest consumer of DSP usage is the channel filter, using more than double the total amount of cycles by itself. The theoretical cycles used for one channel is calculated in Equation 9.52 to 1.8 %. This is far from the measured value, proving that much of the DSP usage is used for other than arithmetic's, such as data transfer. Likewise the theoretical 8192 point FFT usage was calculated in Table 3.2 to 0.15 %, which is used in the mid-frequency estimation among other calculations. The result is thereby not directly comparable, but the same tendency is shown, with the measured values being far off. Generally programs written in Blackfin C is far from optimal. The filter can be optimized by implementing in assembly, whereas a previous developed assembly filter is used as comparison. The assembly filter is measured to 86 913 124 cycles corresponding to 61.0 % filtering both *I* and *Q* channel.

As described in Section 10.1.1, the sampler is sorting the data to get rid of the interleaved zeros. An alternative is to move the pointer accordingly, skipping the zeros, and use the channel filter to sort things out. Thereby considerably reducing the cycle count.

If the system does not sort the *I* and *Q* as a stand alone operation before filtering and by implementing the filter

in Assembly the total cycle cost for a short packet comes down to 112%. This is significantly closer to real-time, though still above the limit, and therefore either the packet detector or time synchronization has to be optimized as well according to these measurements, when it is a must to use both I and Q for demodulation without downsampling.

11.2 Test of Demodulation

The purpose of this section is to show the performance of the developed demodulator as a final system. This is done by demodulating three packets received from AAUSAT3, as described in Appendix C.2. This test shows how well the system fulfills its purpose as a receiver for AAUSAT3.

The demodulation is done on the Blackfin using files as input and output.

This test is related to the overall requirements in Table 5.1.

A few changes have been made from the module tests described in Chapter 10. The distance between symbols has proved to be somewhat smaller than $T_b = 78.99$ samples/s, as suggested in Section 10.6. By trying different combinations T_b was found to be ≈ 78.63 samples/s. The method also described in Section 10.6 was used to avoid drifting. By altering the number of samples per symbol in a manner as follows, the drifting was resolved.

$$\text{SPS} = [78 \ 78 \ 80 \ 78 \ 78 \ 80 \ 78 \ 78 \ 80 \ 78 \ 78 \ 80 \ 78 \ 78 \ 80 \ 78 \ 78 \ 80 \ 78] \quad (11.1)$$

As briefly described in Appendix C.2, the recording from AAUSAT3 was done with a sampling rate of 192 kHz and therefore it is up-sampled to represent a sample rate of 758 272 kHz and the mid-frequency is moved from 50 kHz to 200 kHz before processing the recording on the Blackfin. This was done in MATLAB with the `resample()` function and by multiplying with a complex exponential, to move the mid-frequency to 200 kHz.

Three parts have been extracted from the recording of a pass: One with strong signal, one with medium signal, and one with weak signal. The signal strength is not measured by SNR. Rather, the signals are taken from the most and the least noisy parts of the recording. The signal is up-converted by the same frequency ratio so that the effect of Doppler shift will be present to two of the recordings (the medium and the weak).

Below are some phase plots with the points of symbol decision. The phase itself is generated in post-processing from the I and Q files output after filtering on the Blackfin. The position of the bits are plotted as dots on top of the phase signal to show the drifting in the moment of decision, as in Section 10.6. This illustrates how precise the symbols are predicted.

The extracted data from each signal are printed in hex. A comparison of the symbols, as in the module tests, can not be performed because the packet from the satellite is unknown. Because the sync word and FSM are known, these are used as a measure of successful demodulation.

Because of the long training sequence and because long packets were sent during the recorded transmission, a whole packet with full training sequence could not be represented in one buffer of 2 MiB. Therefore, the first part of the training sequence is discarded. This means that the packet detector could not be tested since the packet would always start at zero.

11.2.1 Strong Signal

Output from program:

- $f_c = 199\,981$ Hz.
- Correlation maximum: Sample 40.
- Sync word located at: Symbol 947.
- FSM is 0x59.

The mid-frequency was successfully determined. The sync word was successfully found and the FSM was decoded to show the packet was large. A spectrogram of the recorded, up-sampled packet as well as an FFT of the training

sequence is shown in Figure 11.1. The phase of the start of the training sequence and the end of the packet is shown in Figure 11.2. Here it is seen that no severe decision drifting occurs during a packet¹. The output data is shown in Listing 11.1. By comparison of the last byte of output data and the last byte of data in the figure it can be concluded that the Blackfin implemented demodulator outputs the correct symbols.

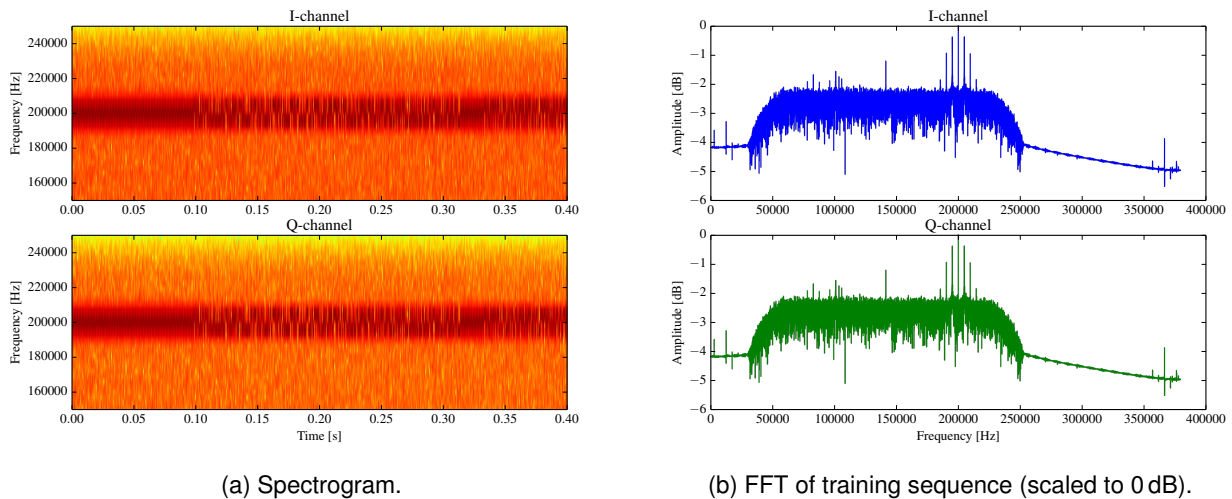


Figure 11.1: Packet received with strong signal—not filtered.

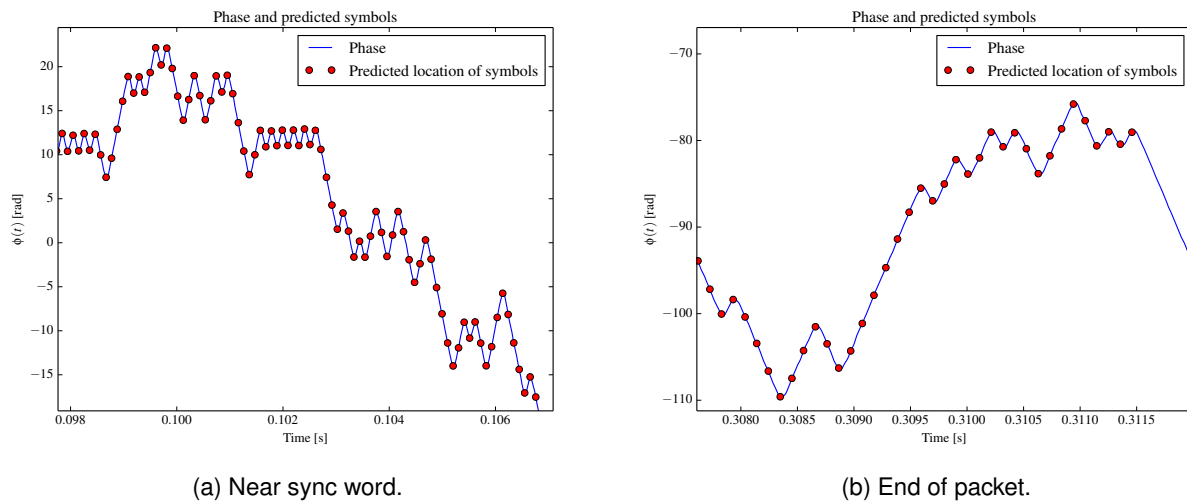


Figure 11.2: The predicted location of symbols generated in the symbol decision module marked with red dots. The blue phase signal is generated in Python from the in-phase and quadrature output of the demodulator.

```

8C 1A 70 83 0F 4F AB AC 04 D3 30 1B 32 6F 04 80 D5 0D 71 F7 5F 48 8A 8F FB F5 8E DC 4D D1
B2 BD 84 EE 75 34 73 78 8F 84 4B 38 D4 C6 86 1E C2 02 CF 4F 93 35 C7 3F C0 DE F4 86 57 CE
49 DF 10 74 09 C0 FC 5E 09 03 8E 1F DD C1 3A 3D C8 D2 7C 08 D2 71 A7 8 D1 22 1F E6 DF 76
91 8C 44 E3 EF 02 0F 81 40 B0 80 51 5B FA E4 50 A6 87 BD C3 FC E2 FE 33 BD 90 19 C2 02 F4
5C A9 D8 07 D3 B9 D7 6E 01 9F 7D 5D 4D 1C DE 16 0E 47 F9 BA 9D B6 18 A2 A8 32 09 03 6F 1E
51 FE 60 58 8B 21 63 8C EF BD 4A 30 67 E7 1A 46 DE DB 80 8B 93 B2 1B 4D 06 0F E4 F6 62 21
CB 6E 3A 56 58 45 F1 80 7E 53 D4 FD 77 AF 6C AF 6F 28 D8 C9 0E 17 A0 E7 C0 B5 52 AD 34 CC
05 7E 2C 85 80 FA E4 87 2D 61 44 C2 CF 42 D0 9B F7 46 2B 2D 59 0A 6B 10 24 B2 1B 78 D2 49
E9 A2 50 65 CD 84 39 FD B4 E5

```

Listing 11.1: Demodulated symbols from a packet received with strong signal.

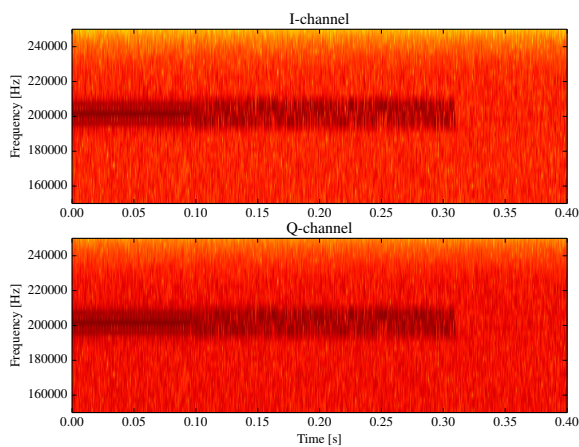
¹Closer inspection than just at the beginning and the end has been performed to verify that no drifting occurs during the whole packet

11.2.2 Medium Signal

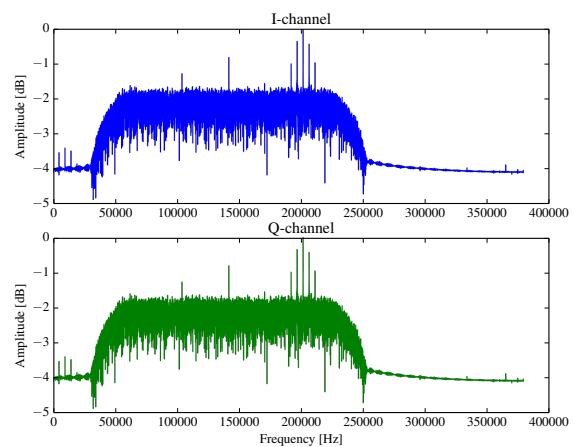
Output from program:

- $f_c = 201\,462\text{ Hz}$.
- Correlation maximum: Sample 117.
- Sync word located at: Symbol 916.
- FSM is 0×59 .

Again, the sync word and the FSM are indicating that a long packet is successfully found. Again, a spectrogram and an FFT of the training sequence are shown Figure 11.3. The phase and predicted symbol locations are plotted in Figure 11.4. Here it is apparent that the signal is induced with more noise and actually determining whether drifting occurs is a bit harder. However, by looking at the peaks of the phase, no drifting or errors in demodulation seems to be a problem. The output data is shown in Listing 11.2. By comparison of the last byte of output data and the last byte of data in the figure it can be concluded that the Blackfin implemented demodulator seems to make one symbol error in the third last decision.



(a) Spectrogram.



(b) FFT of training sequence (scaled to 0 dB).

Figure 11.3: Packet received with medium signal—not filtered.

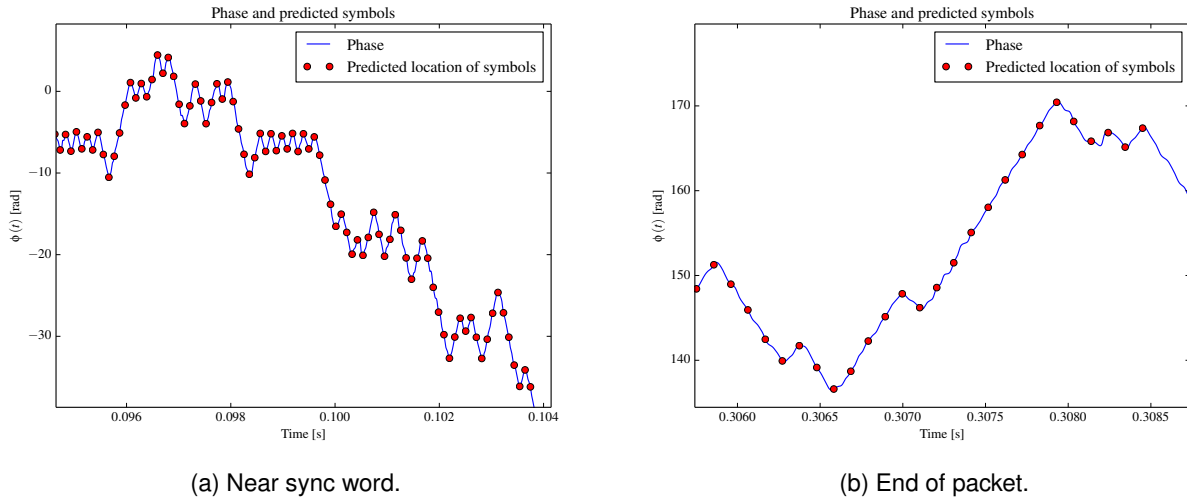


Figure 11.4: The predicted location of symbols generated in the symbol decision module marked with red dots. The blue phase signal is generated in Python from the in-phase and quadrature output of the demodulator.

```

8C 1A 70 83 0F 4F 90 6B 04 DD C1 DB 32 6F 04 80 D5 0D 71 F7 5F 48 8A 8F FB FB F2 85 26 D1
B2 BD 84 EE 43 51 DF 78 58 01 67 1C 28 5C 46 10 33 C2 CF 4F 44 6A F4 01 61 DB 80 88 F6 59
E8 DB 76 2E FF 44 5F 15 9E 54 FB CD 17 FD FD D2 D9 9F 23 21 E9 25 55 ED 20 E2 27 93 84 48
66 C3 1D 02 9C 7E BC F0 5C C2 4B 21 5B FA E4 5E 7A DE D6 C3 FC E2 FE 33 89 F5 B5 C2 D5 71
73 0A A8 5A 13 B7 26 AE 01 9F 73 94 19 A9 AA CC 41 3C 84 20 4A C6 15 00 09 A8 FB A9 DB 8E
7A B1 D8 6F F1 40 51 58 AA EF B5 7D 44 CF 5B 86 E5 1F 31 6C D0 DF 83 BD 72 73 57 87 7E 03
D8 D6 7F E7 A5 03 60 BC CB 95 EA F2 B8 6E BE 00 31 DD 64 95 BA 85 F6 97 E1 AD B0 9A 62 CD
87 52 DF F1 AC C7 4D E7 DA 5C 37 86 3C 8A 06 ED BB 63 4C F6 E5 11 2F 6A 11 D7 8C 87 A7 1F
43 4C 5D 89 67 87 EE 13 DF E1

```

Listing 11.2: Demodulated symbols from a packet received with medium signal.

11.2.3 Weak Signal

Output from program:

- $f_c = 201\,554\text{Hz}$.
- Correlation maximum: Sample 0.
- Sync word located at: Symbol 2802 (NOTE: Sync word of a second packet!).
- FSM is 0x59 (NOTE: FSM of a second packet!).

In this test the first sync word was not detected. As seen on Figure 11.5 and Figure 11.6, the signal is induced with severe noise which makes symbol decision more difficult. The sync word might be almost correct but as a single error in the sync word will cause the demodulator to discard the packet the first packet of the signal is not demodulated. As a result of this some error correction might prove help full when searching for sync word in weak signals. However, as it may be seen from the spectrogram in Figure 11.5a, a second, short, training sequence is present near the end of the recording. The sync word and FSM following this is detected but naturally, not all the data is found. The data is shown in Listing 11.3.

The data itself could not be confirmed as the packet sent from the satellite is unknown. However, from observing the phase of the signal, it was verified that no drifting in the moment of decision occurred and from observing the phase graphs, the data of the strong and medium signal looked to be correct.

A packet could not be demodulated in real-time. Suggestions have been put up on how to make real-time demodulation possible, mainly by reducing the amount of memory and by making the channel filter more efficient.

Chapter 12

Conclusion

The starting point for this study is based on the assumption that a AAU radio receiver can be improved, by exchanging hardware modules with software modules. This assumption led to a problem analysis with an analysis of the GFSK modulation scheme, the existing hardware front-end, the space link format and a recording of a satellite pass. Furthermore, the DSP platform and the environment of communicating with AAUSAT3 were investigated. This analysis was concluded with a definition of the system into sub-modules and an analysis of different design methods for each sub system.

From the preliminary analysis a problem statement was formulated, which is as follows:

How is a software defined GFSK demodulation designed on the Blackfin DSP?

To fulfill this question a list of requirements has been set for the product, and for each requirement a test has been carried out. The results are summarized here.

The developed demodulator prototype is implemented on the Blackfin BF537, which utilizes the uClinux operating system. A system diagram has been developed as seen in Figure 6.1 with the purpose to clarify subsystem interfaces and provide a clear overview of the designed system and how to translate it into software which can be implemented on the BF537 platform. The main system is run as two threads: Sampler and Demodulator. The demodulator system can be considered as 5 elements: Channel Filter, Packet Detection, Mid Frequency Estimation, Time Synchronization, and the Symbol Decision.

The sampler is considered as the input that handles the ADC interface.

The demodulator processes the received packets, on a block-by-block basis, by demodulating with calculations on the complex input signal.

The channel filter is designed as an 8th order band-pass filter. The filter was implemented in the C programming language, designed for optimization by a translation to Assembly.

A packet detector has been implemented. The demodulator captures a buffer from the sampler interface, then applies the filter on the buffer and the output is send through the packet detector. The packet detector must be triggered for the rest of the demodulation to execute.

The Mid Frequency Estimation is based on an 8192 –length FFT. The peaks of the FFT are used to determine the mid-frequency when a packet is detected.

The Time Synchronization features cross-correlation with a pre-computed triangle wave. The first zero of the training sequence is located from which the symbols are spaced with T_b .

The Symbol Decision is done by using the first phase change from the time synchronizer and knowing the distance between the symbols. The distance between symbols is varied to avoid drifting when demodulating the symbols. The Doppler effect is counteracted with a linear subtraction model. Demodulation is done by observing if the phase changes in a positive or negative direction, yielding a 1 or a 0, respectively.

The implementation of the subsystems has been done in a systematic fashion, where each block is implemented, one at a time. First in MATLAB, then in C and finally in fixed-point Blackfin C.

A complete acceptance test has been made, in order to systematically clarify which of the requirements are compliant. The full program and all subsystems were tested on the Blackfin 537 DSP with an external signal generator (Rhode & Schwartz SME 03E). Here it was found that the prototype meets the most relevant requirements in the specifications. The full system is verified as required fully in non real-time, due to the great amount of data and segments of inefficient code. The tests show exactly which sub-systems are the real-time bottlenecks of the system.

Correct demodulation of packets from AAUSAT3 is verified at three SNRs for one specific pass. One with strong signal, one with medium signal, and one with weak signal. The signal strength is not measured by SNR. Rather, the signals are taken from the parts of the recording with the strongest, weakest, and medium signal strengths. Thus, the demodulator is tested using real scenarios of SNRs received.

The tests show that the system is demodulating as specified. The demodulating system may, with real-time improvements in the sub-systems as documented, ensure that the quality is improved in the satellite receiver.

Part IV

Appendix

Appendix A

Quotes From AAUSATLAB

A.1 Modulation Index

AAUSAT3 COM bruger ikke GMSK, men GFSK (GCPFSK, helt eksakt) og variabelt modulation-index. Vi kører typisk med modulationsindex $h = 1$. Hvis vi går ned på $h = 1/2$, koster det sensitivitet med den nuværende modtager, formentlig fordi frekvensestimeringen bliver mere kritisk. —Troels Jessen.

A.2 Packet Transmission and Training Sequence

I kan forvente at pakker kommer umiddelbart efter hinanden, altså uden at power forsvinder mellem pakkerne. Samtidig er træningssekvensen i dette tilfælde kraftigt reduceret. Der er ikke noget handshake på jeres niveau. I kan forvente at pakker kommer ind når som helst.

Jep, højeste datarate er (ca.) 20 kb/s, da den maksimale båndbredde vi har til rådighed er 25 kHz. I praksis fungerer kommunikationen dårligt når vi går over 10 kb/s, men det skulle det gerne komme til, når vi begynder at bruge jeres modtager. —Troels Jessen.

A.3 AAUSAT3 Statistical Information about Reception

Jeg har lavet en figur med label på begge akser - venstre er antal af korrigerede bit fejl i en pakke - var der 0 er der en rød prik, ellers blå prik og højre er total dl. data fra AAUSAT3. Rød er uden bitfejl og blå er med. Så igen, hvis man kan gøre radioen bedre vil man sænke mængden af beskeder med bitfejl og derfor kunne reducere mængden af kodning eller mængden af retransmissioner (over 200 bitfejl så kan vi ikke afkode, men mister pakken). —Jesper A. Larsen

Appendix B

Hardware Layout of the Front End

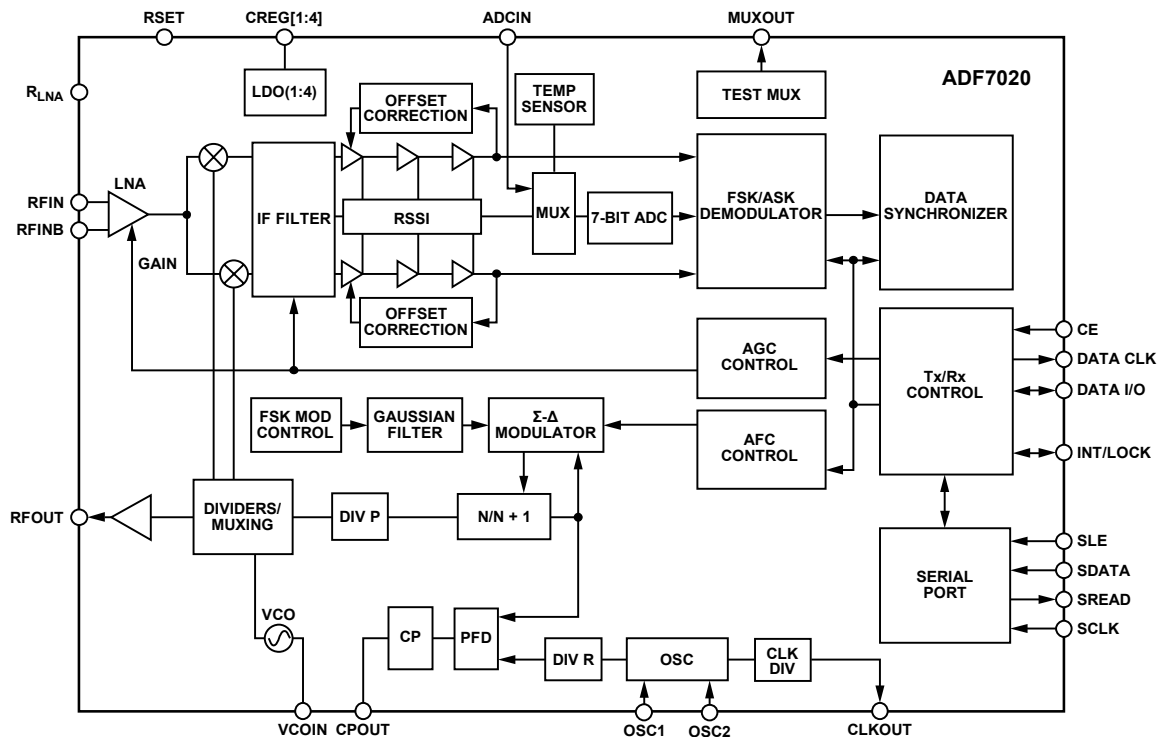


Figure B.1: Functional block diagram Analog Devices ADF7020 ISM band transceiver chip.[Anad]

A functional block diagram of the Analog Devices ADF7020 (transceiver), AD7262 (ADC), and BF537 (DSP) can be found on Figures B.1, B.2, and B.3 respectively. These chips are interconnected as follows. An overview of these are presented in Figure 2.3.

- The antenna is connected to the RF input of the ADF7020. This chip has two sets of differential test pins—one set for the *I* and one for the *Q* channel—connected just after the internal IF filter (see Figure B.1; the pins are not marked on this because they are normally only used for testing). At this point, the received signal is converted down from a carrier at 437.425 MHz to 200 kHz and filtered with an IF filter. This filter can, in software, be adjusted to have a bandwidth between 100 kHz and 200 kHz[Anad].
- The test pins mentioned above are connected to the A and the B input on the AD7262 ADC, respectively. This is a dual input-dual output ADC, and is capable of sending the digital signal to the Blackfin via two SPI connections (see Figure B.2).
- The two SPI lines (DOUTA and DOUTB), along with clock (SCLK) and chip select ($\overline{CS}$) are connected to

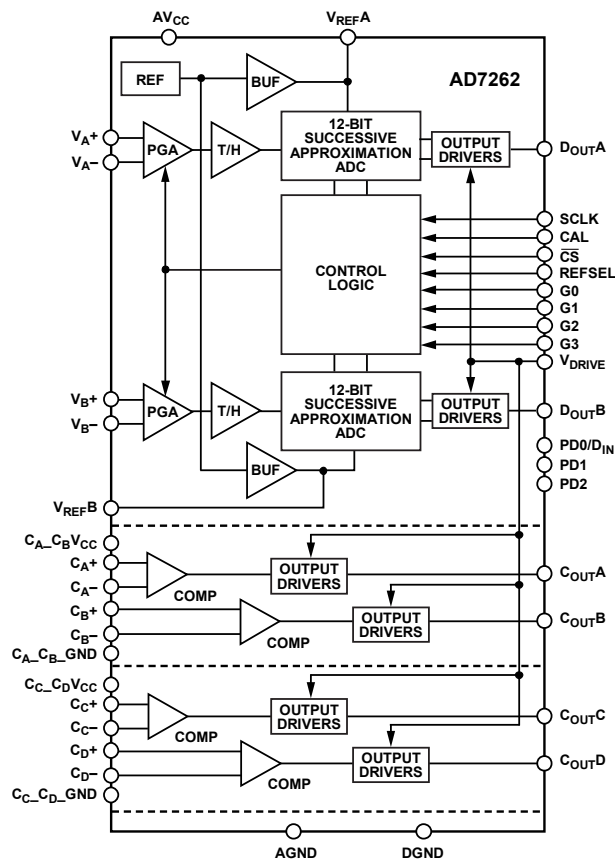


Figure B.2: Functional block diagram Analog Devices AD7262 analog-to-digital converter.[Anaa]

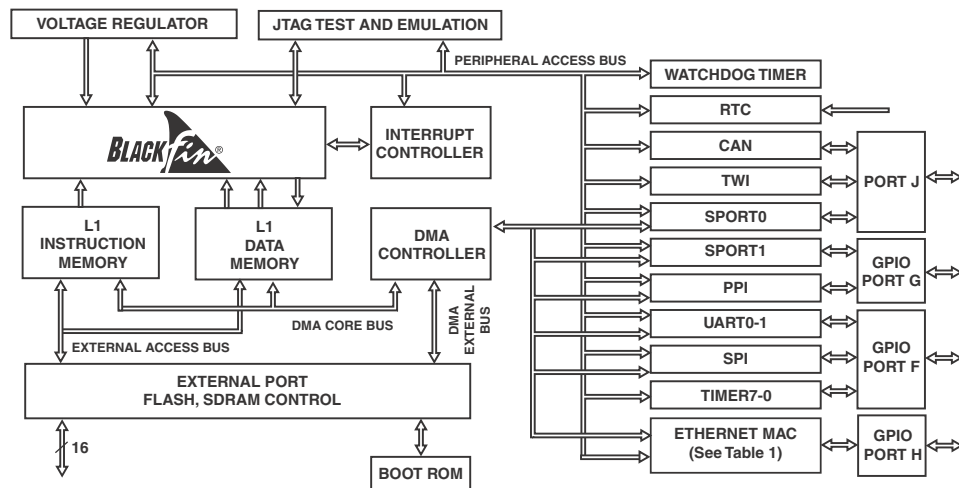


Figure B.3: Functional block diagram of Blackfin BF537.[Anab]

port G on the Blackfin BF537 (see Figure B.3). The SPORT connection is controlled in software on the Blackfin.

The interconnections will not be described in greater detail.

Appendix C

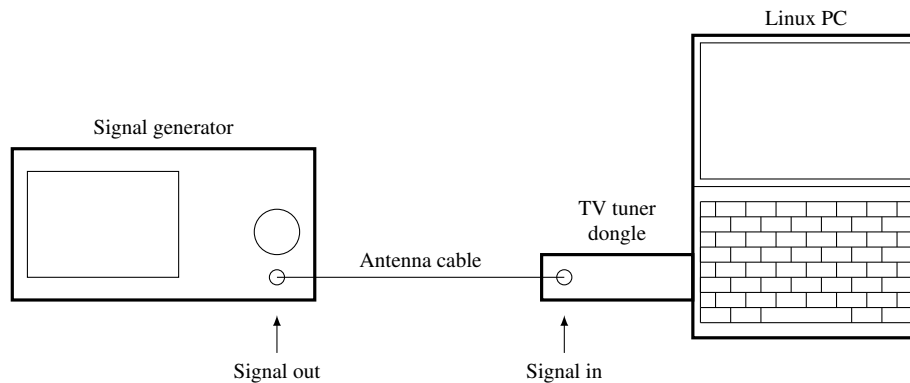
Measurement Reports

C.1 UHF Packet From Signal Generator

C.1.1 Purpose

Obtain real life data like that sent from AAUSAT3. This data can be used to test the designed demodulator.

C.1.2 Test Setup



C.1.3 Equipment Used

Instrument	AAU number	Manufacturer, type, etc.
Signal generator	DEMO-RS-0609	Rhode&Schwartz SME 03E
TV tuner dongle	N/A	Rafael Micro R820T tuner, RTL SDR DVB-T+DAB+FM

C.1.4 Test Procedure

TV dongle is plugged into a Linux PC running the `rtl_sdr` software. This software is set up as follows:

- Sample rate: $f_s = 2.048 \text{ MHz}$
- Number of samples: $6f_s$ samples ($= 6 \text{ s}$)
- Gain: 20 dB
- Frequency to tune to: 437.225 MHz (200 kHz below the carrier of the transmitted signal below. The carrier will therefore be appearing at around 200 kHz in the output signal).

Several measurements is made with different modulation schemes: FSK, GFSK, and GMSK. The reason for this is, that the GFSK modulation on the signal generator only supports a bit rate higher than 10 kb/s, which in

unfortunate as the bit rate most often transmitted by AAUSAT3 is 2400 b/s. The FSK and GMSK measurements are made at these lower bit rates.

Setting	FSK	GFSK	GMSK
Frequency	437.425 MHz	437.425 MHz	437.425 MHz
Level	−60 dBm	−60 dBm	−60 dBm
Source	DATA	DATA	DATA
Prbs	POS/COUP	POS/COUP	POS/COUP
Level attenuation mode	Norm	Norm	Norm
Level attenuation	60 dB	60 dB	60 dB
Mode	Single	Single	Single
Deviation	1.2 kb/s, 9.6 kb/s	14 kHz	N/A
Bit rate	2.4 kb/s, 4.8 kb/s, 9.6 kb/s, 19.2 kb/s	10 kb/s, 19.2 kb/s	2.4 kb/s, 4.8 kb/s, 9.6 kb/s, 19.2 kb/s
Filter	Off	$BT = 0.5$	$BT = 0.5$
Diff encoder	N/A	N/A	Off
Mod polarity	Norm	Norm	Norm
Select list	DLIST1	DLIST1	DLIST1

Packet

The packet, DLIST1, has a length of 159 B. The first 24 B are training sequence and the rest are of a random packet with sync word OZ4CUB (in hex, 4F 5A 34 43 55 42) and FSM 0xA6 (short packet—see Section 2.3). The packet is shown in Listing C.1 in hex format.

```

55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 4F 5A 34 43 55 42
A6 8C 1A 48 CE F5 FF 9E 4D A0 C8 27 B1 18 AF 07 D0 D1 90 CD C0 A2 14 29 AF 13 1F 10 42 54
19 46 54 47 EE C9 C3 A2 A6 CE 5E A2 6B 40 73 10 AD 7E 72 CF 74 BB 1F 4D B4 31 5A 59 04 76
A4 56 1F 5A C9 A5 93 F5 DD 3A D1 68 9A 04 1D 02 46 61 7F E9 6F E8 1B 83 01 67 C5 52 7A BC
4F CE C0 25 27 A8 AF 31 57 2F 70 1D B0 AC E8 92 AE 69 BE DA F4 34 88 7F 77 DE CC F3 E0 BC
B9 8A 9B D6 66 68 94 22 55

```

Listing C.1: DLIST1 packet from AAUSATLAB signal generator in hex format

C.1.5 Results

The resulting files are found on the CD:

 /data/signal_generator/recordings/

The files found here are in binary format (unsigned 8 b integers) of interleaved I and Q samples.

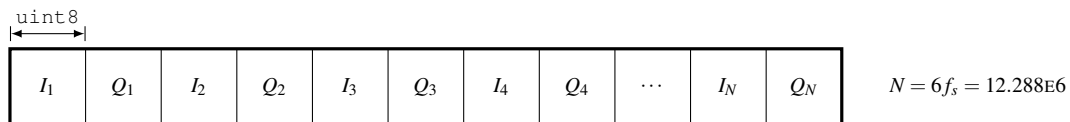


Figure C.1: Output data format from `rtl_sdr`.

C.1.6 Conclusion

The data was collected with success. It now needs to be downsampled from 2 048 000 samples/s to 758 272 samples/s, corresponding to the sampling frequency of the Blackfin board.

A small program has been created to downconvert the recordings and add Gaussian white noise corresponding to Section 7.3.1. The time shift is not added, as a packet is received at a random time in the recording. The Doppler shift is emulated by the lack of precision in the cheap TV tuner dongle. This C program is found on the CD:

 /code/data/signal_generator/dat2csv.c

C.1.7 Uncertainty of Measurements

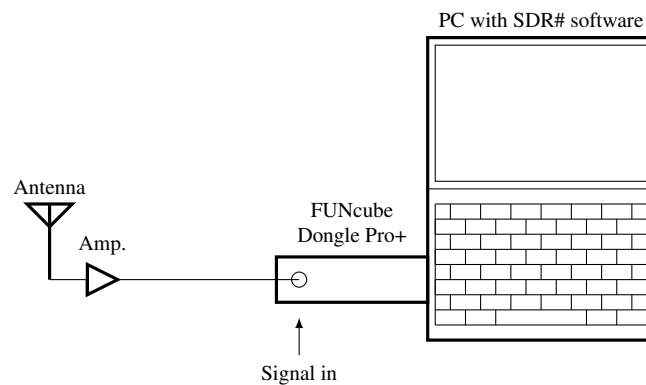
The TV dongle is a cheap model and may therefore not be very accurate in terms of tuning frequency. It should be expected that the carrier frequency is off compared to the expected 200 kHz.

C.2 UHF Packet From AAUSAT3

C.2.1 Purpose

Obtain real data to verify some assumptions made on various parts of the satellite communication. Also, to make it possible to test a solution with a real life packet before trying the system on satellite pass in real-time.

C.2.2 Test Setup



C.2.3 Equipment Used

Instrument	AAU number	Manufacturer, type, etc.
SDR antenna receiver	N/A	FUNcube Dongle Pro+
Antenna	N/A	N/A
Antenna amplifier	N/A	N/A

C.2.4 Test Procedure

The existing front used by AAUSATLAB to communicate with its satellites was used. The FUNcube Dongle Pro+ was connected to a PC running the SDR# software. Because this software records in audio files and the dongle is limited to a similar range, the sampling rate and the bandwidth was reduced. The following settings were used:

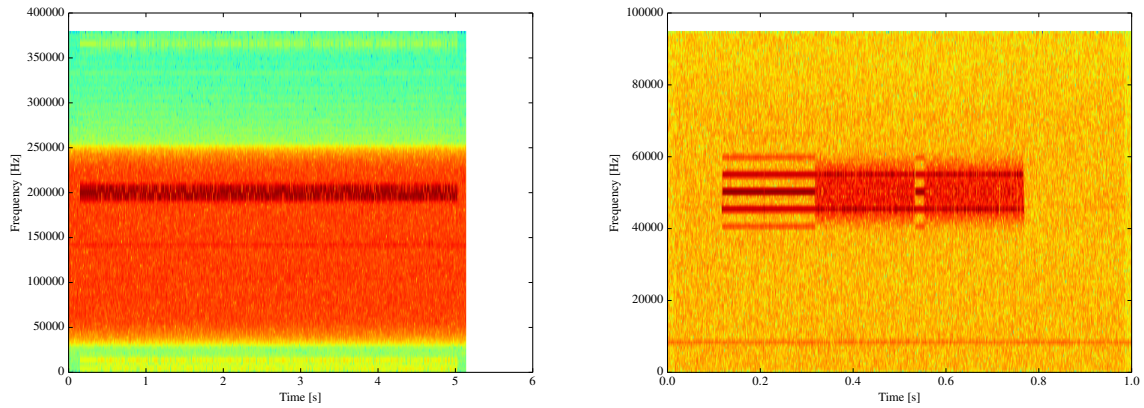
- Frequency: 437.375 MHz
- Sampling rate: 192 000 kHz

This puts the carrier frequency of the expected signal at 50 kHz. To be able to use this data for realistic testing, the carrier should be at 200 kHz and the sampling rate should be 758 272 Hz. To achieve this, the recording is upsampled in MATLAB afterwards.

C.2.5 Results

A good pass lasting 12 minutes was recorded. A spectrogram of a packet, upsampled in MATLAB, is shown in Figure C.2a. This packet is used for further testing. The whole recording is found on the CD:

 /data/AAUSAT3recording.wav



(a) A packet recorded from a pass of AAUSAT3, upsampled in MATLAB and with carrier adjusted to 200 kHz. (b) A packet recorded from a pass of AAUSAT3, upsampled to $758272/4$ and with carrier adjusted to 50 kHz.

Figure C.2: Spectrogram of recordings from AAUSAT3 pass.

The output .wav file has a set of stereo channels where one contains the I samples and one contains the Q samples.

Another packet is shown in Figure C.2b. This is only upsampled to $758272/4$ and with the carrier adjusted to 50 kHz. Here two packets sent back-to-back are shown. It is seen that the first packet has a training sequence lasting 200 ms where the second only has one lasting 20 ms. The packet in Figure C.2a has a training sequence lasting 200 ms. From this it is clear, that the training sequence is shorter in back-to-back mode.

C.2.6 Conclusion

Data from a pass was successfully recorded and upsampled. This data has the same sampling rate as the ADC on the Blackfin board as well as a carrier frequency located close to 200 kHz as it would be when using the Blackfin front-end. The data can therefore be used to emulate a real-life signal when developing and testing on the Blackfin.

C.2.7 Uncertainty of Measurements

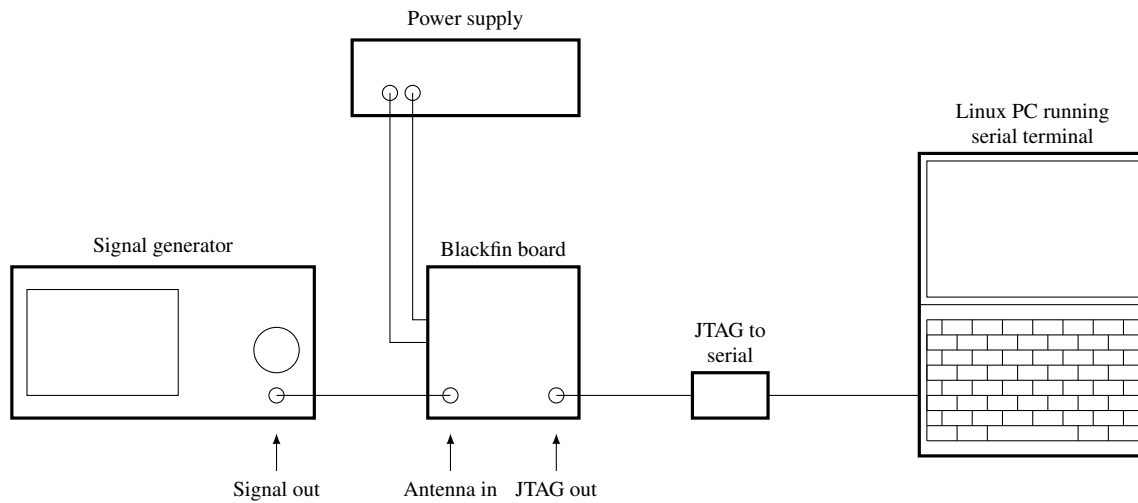
The upsampling process may have resulted in some aliasing on the output signal. A more optimal solution would be to record a pass with the Blackfin and ADC front-end to be sure the sampling rate is correct.

C.3 Acceptance Test With Signal Generator

C.3.1 Purpose

To verify that how well all submodules and the overall system works.

C.3.2 Test Setup



C.3.3 Equipment Used

Instrument	AAU number	Manufacturer, type, etc.
Signal generator	DEMO-RS-0609	Rhode&Schwartz SME 03E
Blackfin board	N/A	AAUSAT3 AIS2 board
Power supply	64339	Hameg HM7042-5
JTAG to serial converter	N/A	JAL/AAUSAT3

C.3.4 Test Procedure

The signal generator is set to the following settings:

Setting	Value
Frequency	162 000 000 Hz
Level	−80 dBm
Level attenuation mode ¹	MAX
Mod. scheme	FSK
Deviation	9600 Hz ($h = 1$)
Bit rate	9600 b/s
List	DLIST1 (packet), DLIST2 (training, 13 B)

C.3.5 Results

The results are shown in the individual test subsections, for each section in Chapter 10.

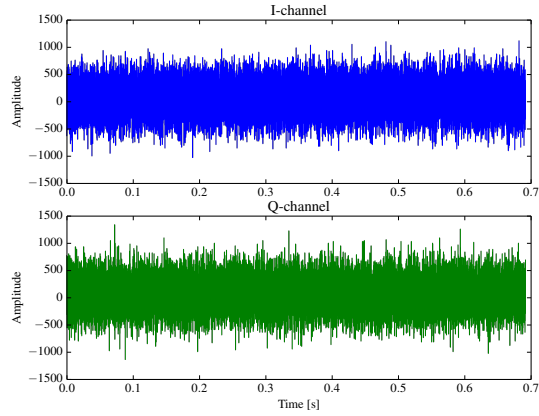
Furthermore a test of the idle output from the signal generator is made. The idle output is when RF of the signal generator is on but no packets are send. The results of the test of the idle output from the signal generator is shown in Figure C.3.

C.3.6 Conclusion

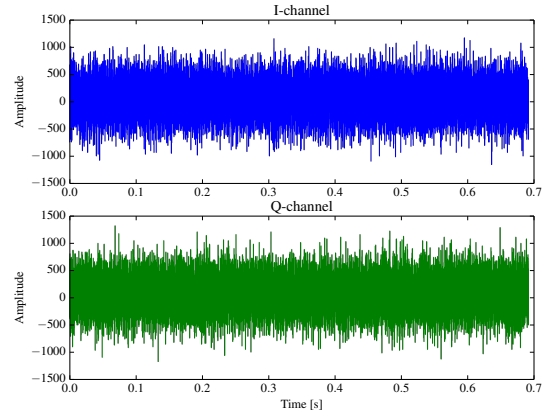
The conclusions are shown in the individual test subsections, for each section in Chapter 10.

As seen in Figure C.3 the sampled data is noise only, when RF is off. However during the idle output of the signal generator the data contains a specific peak in frequency. This can be proven to be the zero-frequency of the

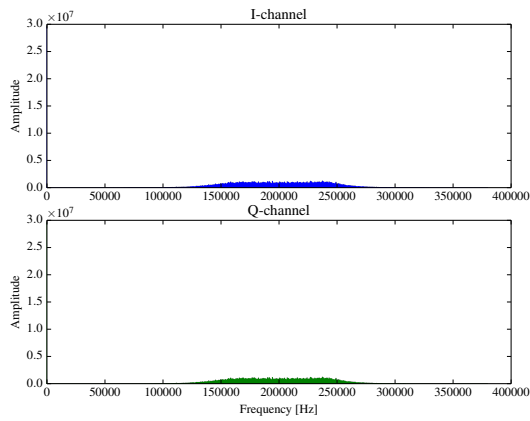
¹Changed from NORM to MAX. Further explanation in Section C.3.6.



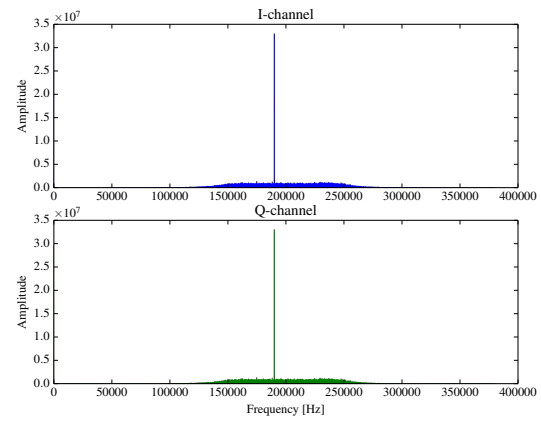
(a) Amplitude with RF off.



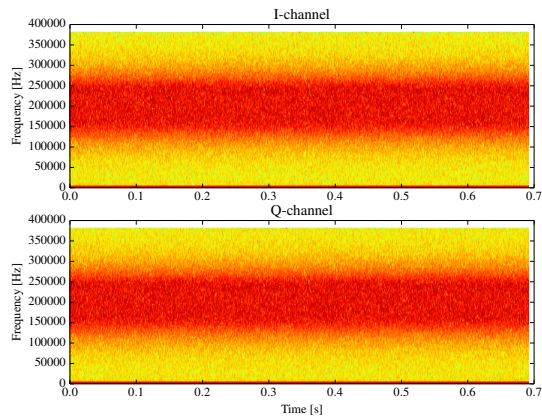
(b) Amplitude with RF on.



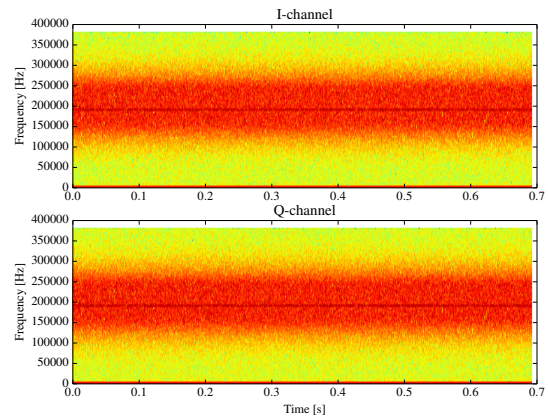
(c) FFT with RF off.



(d) FFT with RF on.



(e) Spectrogram with RF off.



(f) Spectrogram with RF on.

Figure C.3: In the left column the RF of the signal generator is off. In the right column the RF of the signal generator is on.

modulated signal even though no packets are transmitted. A mode called `Level attenuation mode`, which sets the level of attenuation during idle output, is changed from `NORM` to `MAX`. This effectively removes the idle output of the zero-frequency. All tests using the test as described in this section (Section C.3) uses a level attenuation mode of `MAX` except from the one shown in Figure C.3.

An implementation of the windowed FFTs as described in Section 3.3.1 on the Blackfin was during the study discarded due to bad performance. The test above does however show that it might have been due to an 'invisible' zero-frequency which at the time of implementation was unknown.

Bibliography

- [AAUa] AAUSATLAB. AAUSAT3. <http://www.space.aau.dk/ausat3/>.
- [AAUb] AAUSATLAB. Spacelink Format. <http://www.space.aau.dk/ausat3/index.php?n=Ham.HAMAndRadioInfo>.
- [Anaa] Analog Devices. *1 MSPS, 12-Bit, Simultaneous Sampling SAR ADC with PGA and Four Comparators AD7262*.
- [Anab] Analog Devices. *Blackfin Embedded Processor ADSP-BF534/ADSP-BF536/ADSP-BF537*.
- [Anac] Analog Devices. *C/C++ Compiler and Library Manual for Blackfin Processors*.
- [Anad] Analog Devices. *High Performance, ISM Band, FSK/ASK Transceiver IC*.
- [Anae] Analog Devices. *High Performance, Narrow-Band, Transceiver IC*.
- [Bor] Mark Borgerding. Kiss FFT. <http://sourceforge.net/projects/kissfft/>.
- [GK07] W.S. Gan and S.M. Kuo. *Embedded Signal Processing with the Micro Signal Architecture*. Wiley, 2007.
- [GPR] GPREDICT. GPREDICT. <http://gpredict.oz9aec.net/>.
- [HM01] S.S. Haykin and M. Moshier. *Communication Systems*. John Wiley & Sons, Limited, 2001.
- [HM10] S.S. Haykin and M. Moshier. *Communication Systems*. John Wiley & Sons, Limited, 2010.
- [KLT06] S.M. Kuo, B.H. Lee, and W. Tian. *Real-Time Digital Signal Processing: Implementations and Applications*. Wiley, 2006.
- [Lat98] B.P. Lathi. *Modern Digital and Analog Communications Systems*. Oxford University Press, Incorporated, 1998.
- [LF07] I. Land and B.H. Fleury. *Digital Modulation 1—Lecture Notes*. Aalborg University, 2007.
- [Lyo12] Richard Lyons. *Understanding Digital Signal Processing*. Pearson, 2012.
- [McC] McCullough. uCLinux For Programmers. <http://www.linuxjournal.com/article/7221>.
- [OS10] A.V. Oppenheim and R.W. Schaffer. *Discrete-Time Signal Processing*. Pearson Education, Limited, 2010.
- [PM96] J.G. Proakis and D.G. Manolakis. *Digital signal processing: principles, algorithms, and applications*. Prentice-Hall International editions. Prentice Hall, 1996.
- [Rap02] T.S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall communications engineering and emerging technologies series. Prentice Hall PTR, 2002.
- [Ros09] S.M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Elsevier Science, 2009.
- [Sk101] B. Sklar. *Digital Communications: Fundamentals and Applications*. Prentice Hall Communications Engineering and Emerging Technologies Series. Prentice-Hall PTR, 2001.

- [Su02] K.L. Su. *Analog Filters*. Springer, 2002.
- [TBT13] M. Tausen, J.W. Bundgaard, and T. Trillingsgaard. DSP-based GMSK demodulator for satellite AIS receiver. Project report, Aalborg University, 2013.
- [Vin] Vince. Continuous angles in C++ (eq unwrap function in matlab). <http://stackoverflow.com/questions/15634400/continuous-angles-in-c-eq-unwrap-function-in-matlab>.

Appendix CD

- report.pdf
- code/matlab_filterdesign/chan_filt_coef.m
- code/matlab_gfsk_workbench.m
- code/c_gfsk_workbench.m
- code/bf_gfsk_workbench.m
- code/data/acceptancetesting
- code/data/finaltest
- data/signal_generator/recording
- data/signal_generator/dat2csv.c
- data/AAUSAT3recording.wav