Faculty of Electrical Engineering
Department of Applied Electronics and Telecommunications

Diploma Thesis

Communication system with M-ary chirp modulation

Author: Bc. Michael Křeček
Supervisor: Ing. Ivo Veřtát, Ph.D.

Pilsen 2018

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: **2017/2018**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Michael KŘEČEK**
Osobní číslo:	**E16N0067P**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Telekomunikační a multimediální systémy**
Název tématu:	**Komunikační systém s vícestavovou rozmítanou modulací**
Zadávající katedra:	**Katedra aplikované elektroniky a telekomunikací**

Zásady pro vypracování:

1. Realizujte komunikační systém s vícestavovou rozmítanou modulací pracující v reálném čase.
2. Navrhněte a otestujte vhodné metody demodulace s ohledem na výpočetní náročnost a realizaci v PC.

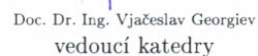
Rozsah kvalifikační práce:	**40 - 60 stran**
Forma zpracování diplomové práce:	**tištěná/elektronická**

Seznam odborné literatury:

Student si vhodnou literaturu vyhledá v dostupných pramenech podle doporučení vedoucího práce.

Vedoucí diplomové práce:	**Ing. Ivo Veřtát, Ph.D.** Katedra aplikované elektroniky a telekomunikací
Datum zadání diplomové práce:	**10. října 2017**
Termín odevzdání diplomové práce:	**24. května 2018**

Doc. Ing. Jiří Hammerbauer, Ph.D.
děkan



Doc. Dr. Ing. Vjačeslav Georgiev
vedoucí katedry

Abstrakt

Cílem této diplomové práce je experimentální softwarové realizace komunikačního systému pracujícím v reálném čase. Diplomová práce navazuje na [4], kde autor vytvořil v prostředí MATLAB softwarové řešení modulátoru a demodulátoru s využitím modulace vícestavových rozmítaných modulací. Tato práce poukazuje na problémy původní verze a popisuje následné řešení a vylepšení.
Značné úsilí bylo vyvinuto pro snížení výpočetního výkonu. To vedlo k novým neobvyklým možnostem demodulačního procesu, kdy se využila decimace a frakční Fourierova transformace.
Výsledkem této práce je funkční softwarový prototyp komunikačního systému se schopností volit energetickou nebo spektrální účinnost pro skupinu vícestavových rozmítaných modulací, který byl vytvořen v softwarovém prostředí MATLAB.

Klíčová slova

Vícestavové rozmítané modulace, rozprostřené spektrum, nízkoenergetický komunikační systém, zlomková Fourierova transformace

Abstract

Křeček, Michael. *Communication system with M-ary chirp modulation* [*Communication system with M-ary chirp modulation*]. Pilsen, 2018. Master thesis (in English). University of West Bohemia. Faculty of Electrical Engineering. Department of Applied Electronics and Telecommunications. Supervisor: Ivo Veřtát

---

The aim of this thesis is the experimental software implementation of a communication system working in real-time. Thesis follows up on [4], where author created software solution of the modulator and demodulator utilizing M-ary chirp modulation in a MATLAB environment. This paper points out to the issues of the original version and describes subsequent solutions and improvements. In particular, significant effort to decrease computation difficulty has been made. This lead to new unusual possibilities to demodulation process, where decimation and fractional Fourier Transform were utilized.

The result of this thesis is a functional software prototype of the communication system, with an ability to select energy or spectral efficiency over a group of the M-ary chip modulation, created in the MATLAB environment.

Keywords

M-ary chirp modulation, spread spectrum, low power communication system, Fractional Fourier transformation

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci, zpracovanou na závěr studia na Fakultě elektrotechnické Západočeské univerzity v Plzni.

Prohlašuji, že jsem svou závěrečnou práci vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 270 trestního zákona č. 40/2009 Sb.

Také prohlašuji, že veškerý software, použitý při řešení této diplomové práce, je legální.

V Plzni dne May 30, 2018

Bc. Michael Křeček

. . . .

Podpis

Acknowledgement

This work was created with the support of the student grant competition SGS-2015-002.

Contents

List of Figures **iv**

List of Symbols and Abbreviations **v**

1 Introduction **1**

1.1 Aims of the thesis . . . 2

2 Chirp Spread Spectrum **3**

2.1 M-ary Chirp Spread Spectrum . . . 5

2.1.1 Frequency Band Splitting . . . 5

2.1.2 Symbol Period Splitting Method . . . 6

3 Software Realization of Transmitting Side **8**

3.1 Software Implemntation of the Main Program Transmitter_main . . . 8

3.2 Enter Message Function . . . 11

3.3 Message Alignment Block Function . . . 11

3.4 Create Modulation Symbol Function . . . 13

3.5 Software Realization of Frequency Band Splitting Method . . . 17

3.6 Software Realization of Symbol Period Splitting Method . . . 19

3.7 Transmission of the Message . . . 21

4 Software Realization of Receiving Side **28**

4.1 Software Implemntation of the Main Program Receiver_main . . . 28

4.2 Setting Properties of the Receiver . . . 29

4.3 Interception of the Message . . . 32

4.3.1 Locate Synchronization Function . . . 33

4.3.2 Dem Chirp Function . . . 37

4.3.2.1 Decimation . . . 40

4.4 FrFT dem Function . . . 43

4.5 Translation of Received String . . . 45

5 Conclusion **48**

Literature and Sources **50**

Attachments **52**

A Used scripts and source code **52**

A.1 Main program Transmitter_main.m 52
A.2 Function enter_message.m 54
A.3 Function message_alignment_block.m 55
A.4 Function create_mod_symbol.m 56
A.5 Function chirpM_B_mod_time.m 56
A.6 Function chirpM_mod_time.m 57
A.7 Function symbol_mapping.m 58
A.8 Main program Receiver_main.m 58
A.9 Function locate_synchronization.m 61
A.10 Function dem_chirp.m 62
A.11 Function dem_FrFT.m 63
A.12 Function locate_aplha_values 64
A.13 Function frft.m 66
A.14 Function message_translation.m 67
A.15 Function BER_receiving_side 67

List of Figures

2.1 Modulation symbol representation in CSS system. 4
2.2 Frequency band splitting in time and time-frequency domain. 5
2.3 Time Duration Splitting in time and time-frequency domain. 6

3.1 Simplified flowchart of transmitter 10
3.2 Create modulation symbol function flowchart. 14
3.3 Band splitting method flowchart. 17
3.4 Symbol period splitting method flowchart. 19
3.5 Flowchart of the transmission loop. 22

4.1 Simplified flowchart of receiver. 30
4.2 Flowchart of the receiving loop. 34
4.3 Buffer data loading process. 37
4.4 BER simulation for 8-ary CSS and decimated versions, symbols created by a symbol period splitting technique. 42
4.5 BER simulation for 32-ary CSS and decimated versions, symbols created by a symbol period splitting technique. 42
4.6 BER simulation for 8-ary and 32-ary CSS-FrFT versions, symbols created by a symbol period splitting technique. 45

List of Symbols and Abbreviations

AFP .... Audio File Player
AFSK .... Audio Frequency-Shift Keying
AP .... Audio Player
ASCII .... American Standard Code for Information Interchange
α_K .... Frequency gradient for first half of symbol time duration
α'_K .... Frequency gradient for second half of symbol time duration
BER .... Bit Error Rate
DSP .... Design and Simulate Streaming Signal Processing
CSS .... Chirp Spread Spectrum
E .... Symbol Energy
$Eb/N0$ .... Energy per bit to noise power spectral density ratio
$f_{central}$ .... Central frequency
$fdown$ .... Down frequency
$fhigh$ .... High frequency
FrFT .... Fractional Fourier transform
Fs .... Sampling frequency
IOT .... Internet Of Things
k .... Chirp rate
K .... One Modulation State
M .... Number of modulation states
PAPR .... Peak to Average Power Ratio
PC .... Personal Computer
RF .... Radio Frequency
RMS .... Root Mean Square
T .... Symbol duration
Δ_f .... Frequency sweep
ω_c .... Angular frequency of carrier

Chapter 1

Introduction

At present, a radio link of the picosatellite generally utilizes the non-adaptive communication systems. Nevertheless, the signal quality received at the ground control center changes during the passage of the picosatellite. Therefore, when a power reserve in the link budget occurs, a fixed radio system cannot adapt communication to save power consumption or increase data rates. This method leads to ineffective use of the radio link budget. [1] Thus, adaptive communication system with an ability to select energy or spectral efficiency over a group of the modulations could solve the problem of the low data rate (a few kbit/s), which is currently limiting use of the picosatellites for science experiments. Selection of the modulation method for the adaptive communication system is a fundamental task. Chosen modulation method must have a constant envelope, which enables utilization of non-linear RF amplifiers with high energy efficiency. Other properties of the modulation method should be energy balance, resistance to narrowband interference and jamming. [2] Spread spectrum techniques satisfy those requirements. These techniques are modulation methods by which a signal generated with a specific bandwidth is purposely spread in the frequency domain, resulting in a signal with a bandwidth considerably larger than the bandwidth of the original signal. Chirp spread spectrum are potentially suitable for this application.

This thesis follows up on the diploma thesis High order chirp modulations [4]. An output of this work was created in a MATLAB environment. It is a software prototype of the modulator and demodulator with adjustable bandwidth, modulation number of states and the duration of the modulation symbol working in an acoustic band. Because solution of the predecessor contains many flaws, the main focus of this thesis was software optimization and creation of the adjustable communication system utilizing M-ary chirp modulation. There are two primary aims of this thesis. The first is to implement communication system with M-ary chirp modulation working in the real-time. The second aim is to design and test suitable demodulation methods with a consideration of a realization and computational difficulty for PC.

1.1 Aims of the thesis

A thesis is divided into three main chapters. In the first chapter, the chirp spread spectrum (CSS) is introduced. Likewise, techniques for creating a CSS are described. The next chapter presents software realization of transmitting side, where the concept of the main program `'Transmitter_main.m'` is revealed. Followed by a description of the functions belonging to`'Transmitter_main.m'`. To ensure, that transmitted message has defined properties under any given conditions, a data alignment and stuffing methods were implemented. The last chapter focuses on the software realization of receiving side, in the same manner as for the previous chapter. First of all, the main program `'Receiver_main.m'` is and its functions are introduced. Different approaches for demodulation process are implemented and measured with a consideration of the computational difficulty. A particular attention is paid to the comparison of demodulation duration between original and optimized versions. The result of this thesis is a functional software prototype of the communication system, with an ability to select energy or spectral efficiency over a group of the M-ary chip modulations, created in the MATLAB environment.

2

Chirp Spread Spectrum

The chirp spread spectrum (CSS) is a spread spectrum signalling technique in which a carrier is swept over a wide-band during a given time interval. Generally, in the spread spectrum method transmitted signal is spread over a wide frequency band that is much wider than the minimum bandwidth required for the information to be sent. Spread spectrum techniques are becoming increasingly popular, since their properties meet many needs for modern digital transmission systems. Among these advantages are energy balance, code division multiple access, multipath suppression, low probability of intercept, interference rejection and resistance to fading [3][2]. Energy balance is crucial for applications with limited power resources such as satellite technique and internet of things (IOT) devices. A lot of energy is lost in radio transmission due to RF power amplifiers with low energy efficiency. For modulation with high PAPR, these RF amplifiers must work in linear part of their characteristic curve. In order to utilize non-linear RF amplifiers with higher energy efficiency modulation with constant envelope is preferred. [1]

As mentioned above, CSS is type of spread spectrum modulation with the constant envelope that does not necessarily employ coding. Neverthless, this form has found its main application in radar because of its advantage in reducing powersignificantly. However, according to [6], M-ary CSS are more than suitable for multiple access systems or, in our case, for the single user system but with M-ary signal. Therefore, chirp modulation can be used for communication systems where power consumption is a critical factor. In the chirp systems, modulation symbols are represented by a frequency sweep of a carrier. The linear frequency sweep pattern is the most common one. Any pattern is suitable (e.g. exponential), but in this thesis linear sweep is utilized. In 2-ary CSS, an entire frequency band is reserved for both modulation symbols (0 and 1). From the negative and positive frequency sweep of the carrier, we are able to distinguish 1 from 0. The modulation of the symbol 1 is represented by a positive value of the frequency sweep, which is to say, frequency increase. On that basis, this means that symbol 0 is represented by a negative value of the frequency sweep, thus, frequency decrease. For both modulation symbols, the value of high and low frequency are identical. Likewise, the time interval for sweeping is the same for both symbols, as shown in figure 2.1. For better imagination a spectrogram,

of this CSS is displayed on the image on the right side. [4]

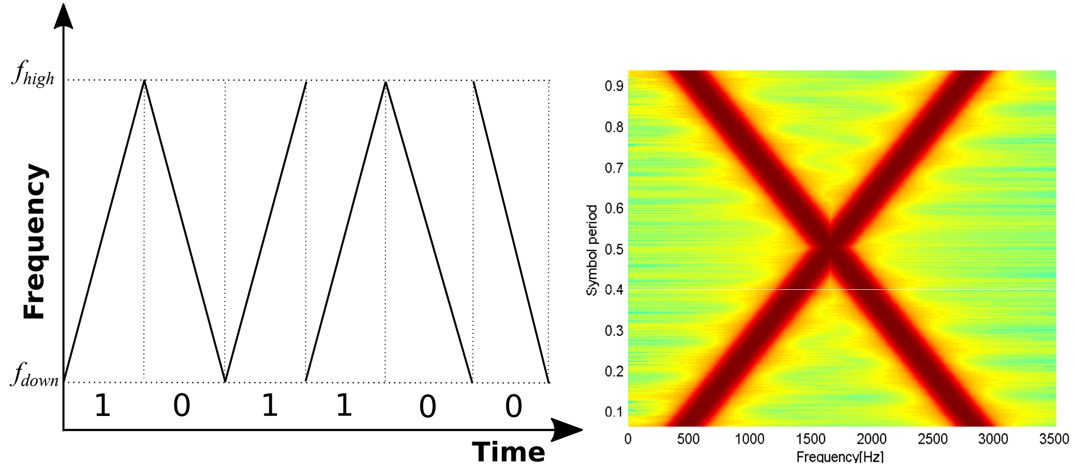


Figure 2.1: Modulation symbol representation in CSS system.

In linear CSS, symbols are described by the equations 2.1 and 2.2 shown underneath, where k is the rate of frequency change called chirp rate. The chirp initial and final frequencies are described as f_{down} and f_{high}. These frequencies are determining size of the bandwidth.

$$y_{Log1}(t) = sin[2\pi(f_{down}t + \frac{k}{2}t^2] \tag{2.1}$$

$$y_{Log0}(t) = sin[2\pi(f_{high}t - \frac{k}{2}t^2] \tag{2.2}$$

The value of the chirp rate k is described in equation 2.3, where $f_{high} - f_{down}$ is frequency sweep Δf and T represents time duration to sweep from f_{down} to f_{high}.

$$k = \frac{f_{high} - f_{down}}{T} \tag{2.3}$$

2.1 M-ary Chirp Spread Spectrum

The first method of creating 2-ary chirp modulation mentioned above, is used for low data rate transmissions. To avoid mutual interference, the signals should be orthogonal. To meet orthogonality criterium approximately, it isnecessary to use opposite polarity for the frequency sweep of the two signals or to separate the two signals in certain frequency space. With this in mind, M-ary chirp signals with different frequency band or diverse chirp rate are used to increase the data rate.[5]

2.1.1 Frequency Band Splitting

Frequency band splitting is one of the techniques on how to approach M-ary CSS. This method splits frequency band into subbands. The quantity of subbands correlates with the number of modulation states. Therefore for 4-ary CSS, 2 subbands are needed. For 8-ary modulation, each subband is split into half. This means, that 4 subbands are created for this type of modulation and so on. In each subband one symbol has positive chirp rate and one with negative chirp rate. In frequency band splitting method, chirp rate is constant for all symbols. Principle is shown in figure 2.2.

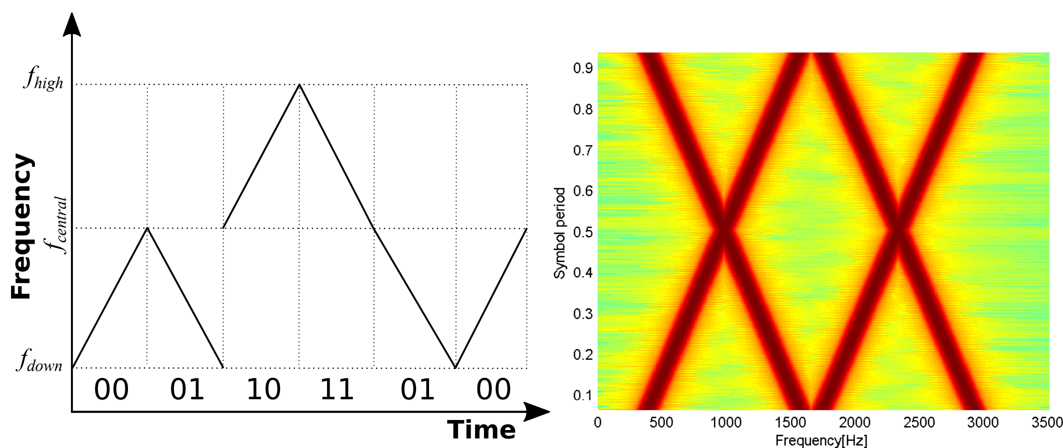


Figure 2.2: Frequency band splitting in time and time-frequency domain.

On the right side of figure 2.2 4-ary CSS spectrogram is displayed. On the left side, time domain is shown. In this image, symbols 0 and 2 are represented by a positive value of the chirp rate. Symbols 1 and 3 are represented by a negative value of chirp rate. For symbols 0 and 2, there is difference between final and initial frequency, which means that chirp rate is same for both. The only difference is in the placement of the symbols in the frequency band. The frequency band is split into half, where the central frequency $f_{central}$ lies. Symbol 0 corresponds to the constant increase of frequency from f_{down} to $f_{central}$. Symbol 2 represents same frequency increase but from $f_{central}$ to f_{high}. For symbols 1 and 2 the same principles are applied but with constant frequency decrease. Mathematical

description of modulation is formally identical to 2.1 and 2.2 equations. Chirp rate is characterized by equation 2.4.

$$k = \frac{f_{high} - f_{down}}{M/2}\frac{1}{T} \tag{2.4}$$

M means number of modulation states and $\frac{f_{high}-f_{down}}{M/2}$ represents bandwidth.

2.1.2 Symbol Period Splitting Method

The second method on how to create M-ary CSS is by utilizing various chirp rates. In this technique, time duration of modulation symbol is split into half. At this exact point, chirp rate is altered. This means, one modulation symbol is represented by two distinct chirp rates, that are changed in the middle of symbol time duration. All symbols include frequencies from whole frequency band. Therefore, each symbol has a unique pair of chirp rates with use of positive and negative chirp rates values. The principle of 4-ary CSS is shown in image 2.3.

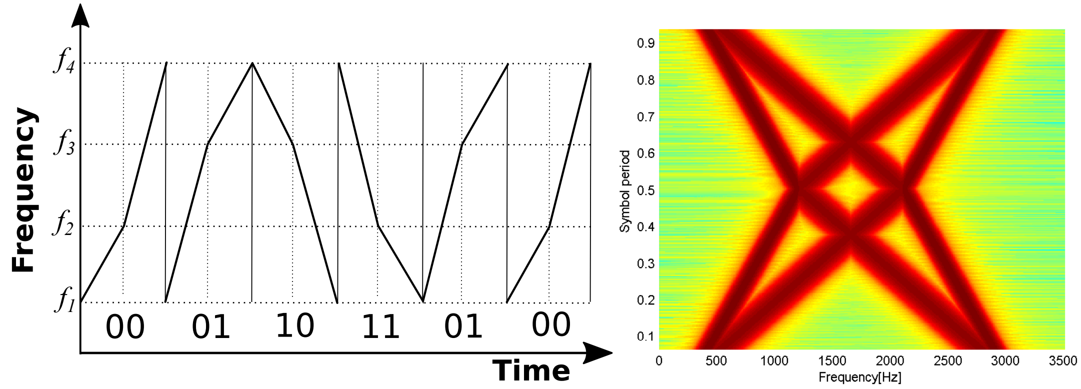


Figure 2.3: Time Duration Splitting in time and time-frequency domain.

In the spectrogram 2.3 displayed above, the change of chirp rate in middle of symbol duration is shown. 4-ary CSS is expressed by the equations bellow.

$$S_{K0}(t) = \sqrt{\frac{2E}{T}}cos(\omega_c - \pi\alpha'_K t^2) \tag{2.5}$$

$$S_{K1}(t) = \sqrt{\frac{2E}{T}}cos(\omega_c + \pi\alpha_K t^2) \tag{2.6}$$

$$S_{K2}(t) = \sqrt{\frac{2E}{T}}cos(\omega_c - 3\pi\alpha'_K t^2) \tag{2.7}$$

$$S_{K3}(t) = \sqrt{\frac{2E}{T}}cos(\omega_c + 3\pi\alpha_K t^2) \tag{2.8}$$

$$\alpha_K = \frac{M\Delta f}{T/2} \tag{2.9}$$

$$\alpha'_K = \frac{(M-1)-K}{T/2}\Delta f \tag{2.10}$$

The one of the modulation states represents K , E symbolizes symbol energy and T symbol duration. Angular frequency of carrier is ω_c, which is equal to $2\pi f_c$. Frequency sweep is stand for Δf and α_K with $\alpha`_K$ represent frequency gradient for first and second half of symbol time duration.

3

Software Realization of Transmitting Side

In this chapter, a software solution of M-ary chirp modulation is described. Both techniques for creating M-ary chirp modulation 2.1.1 and 2.1.2 can be used. Matlab environment was utilized in this project. As mentioned before, this thesis follows up on [4]. However, predecessor has committed many flaws in his program. Rather than describe them now, these flaws and their solutions will be presented as we approach them in the functions of the modulator. As a result, some of the imperfections required additional computation power. Since one of the key tasks of this thesis is reducing computation difficulty, code optimization was necessary.

3.1 Software Implemntation of the Main Program Transmitter_main

Predecessor created multiple Matlab functions for 8, 16, 32 and 64-ary chirp modulation. If a change of the modulation states or any other system parameters are requested, it is complicated or even impossible to do it. Because each function has its own unique folder and variables, this results in an inability to change properties of CSS inside of the program. For instance, in order to change from 8-ary to 16-ary modulation, it is necessary to check key variables `'fhigh', 'fdown', 'T', 'Fs'`, how many modulation symbols are loaded, and which type of chirp rate method is currently being used(2.1.1 or 2.1.2). This is confusing and in addition the risk of overlooking crucial variables could lead to modulator malfunction. The solution to this problem is to create one main file, which calls all requested functions and has global variables for easy change of any M-ary modulation and their properties. Another benefit of this solution is that adding another M-ary chirp modulation (e.g. 4, 128-ary) is not a problem.The main program is called `'Transmitter_main'`

Because `'Transmitter_main'` is quite complex flowchart 3.1 is introduced. However, this flowchart is purposely simplified to show just essential functions for easier understanding of the transmitter concept. These functions will be described more precisely later. As seen in 3.1 various functions are used in `'Transmitter_main'` Firstly global variables are loaded. Then an input dialog of the enter message function will pop up and ask the user to input a message, which is going to be transmitted. Afterwards, inputted characters are converted to the bit stream. Message alignment block is responsible for that inputted data string has defined properties under any given conditions. Subsequently, all necessary symbols are created and loaded. Lastly the transmitting loop will start transmission by sending synchronization symbol, followed by transmitting modulation symbols, which are representing data. In the end, termination of the transmission loop is done by sending termination symbol.

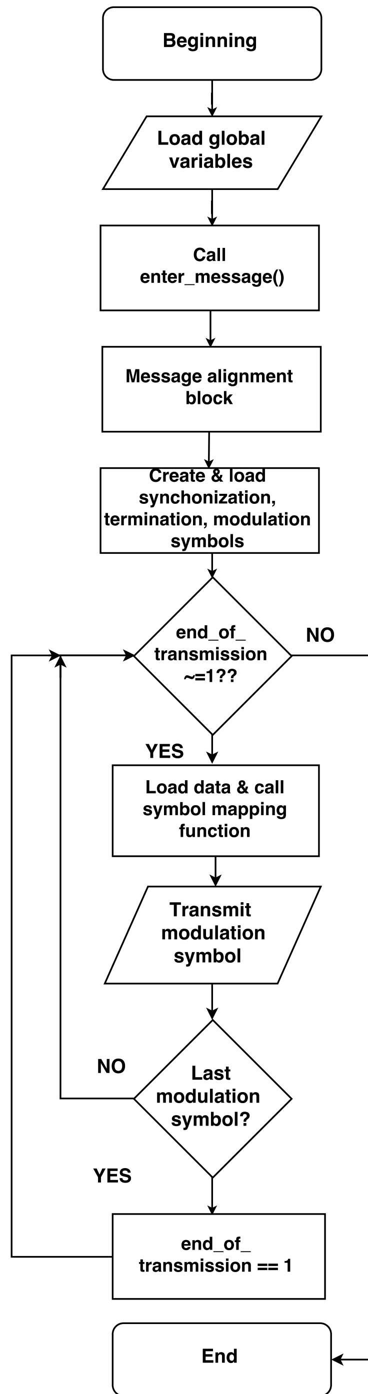


Figure 3.1: Simplified flowchart of transmitter

3.2 Enter Message Function

When global variables are listed `'enter_message'` function is called by the main program. This function requests the user to enter the message he is wishing to send. As a result, function returns a binary string, that represents inputted message which will be interpreted as data for transmitting. After entering the message to input dialog window, converting is necessary. Because `'inputdlg'` command returns a format cell. This data type is non-usable, so it is changed into char. Each char is represented by 8 bits for this reason length of the message is multiplied by 8. The last part of the code is shown below.

```
for i = 1:length(message_char)
  % converting chars to number
  ASCII_dec_msg = unicode2native(message_char(i), 'ISO-8859-2');
  % converting numbers which I get to bit expression
  ASCII_bin_string_msg = dec2bin(ASCII_dec_msg, 8);
    for i = 8
      (message (((i-1)*8) + j) = str2num(ASCII_bin_string_msg(j));
    end
end
```

Two for loops are used for last procedure of this function. In the first loop, chars are converted into the a number (vector of unicode) using the ISO-8859-2 character encoding scheme. ISO-8859-2 is informally referred to as ”Latin-2” and is intended for Central European languages. Decimal numbers are then converted to the vector representing a binary number. In some cases, less than 8 bits representation of decimal number can occur. For example, decimal 23 (represents symbol # in ISO-8859-2 encoding) is 010111 in binary, which means 5 bits representation. As a result, variable length of letters can happen which is inappropriate. To prevent this issue parameter, 8 in `'dec2bin'` is necessary. Due to this parameter, `'dec2bin'` produces a binary representation with at least 8 bits. Because `'dec2bin'` returns binary, but in string of chars form, a second loop is needed. This loop returns the required binary string.

3.3 Message Alignment Block Function

When the message is entered, the following step is to align it. This operation is crucial in terms of the synchronization. As mentioned above text char is always expressed in 8 bits. However, transmitting symbols are represented by number of states (e.g 5 bits for M=32). That means that transmitted message could be sent incomplete, leaving remainders of the bits. This results in a loss of data and instability of the synchronization process. It is necessary to ensure, that encoded message data size is defined at any given conditions.

'Message_alignment_block' functions guarantees it by bit stuffing. This function requires 3 input parameters 'message', 'number_of_bits' and 'operation'. The first parameter 'message' is output variable of binary string from function enter_message. The second parameter is the 'number_of_bits', which express how many bits are used for transmitting a single symbol. This parameter corresponds to global variable 'M' in relation to the number_of_bits = $log_2(M)$. As an example, when the global variable 'number_of_states' equals 16, 4 bits are necessary for transmitting a single symbol. The last parameter 'operation' is string compare command, where two options can be selected. The first is 'align' and the second is 'remove_stuff'. When operation 'remove_stuff' is chosen, 'message_alignment_block' is used by the receiver. This operation will be characterized in chapter 4.5.

The operation 'align' will be described in this chapter, because it is always used by 'Transmitter_main' to stuff message into the size, where there are no leftover bits. Firstly, the size of the message is increased by adding a suffix. The suffix is defined data string, that describes stuffing properties. These properties are important for a receiver. After demodulation of the captured data, the string properties will enable removing stuffing flawlessly. Firstly, the message taken from previous function 3.2 gains length by adding defined series shown below.

```
necessary_number_of_bits = length(message) +
separator_of_string_terminator + length(string_termination) +
length(description_number_of_additional_bits);
```

The 'separator_of_string_terminator' of the string terminator separates the message string from suffix. Value and length of the separator is 1. Another variable added to the message string is the length of 'string_termination'. String termination variable is equal to zeros(1, 2 * length_of_ASCII_character), where the size of ASCII character is 8 bits for ISO-8859-2 encoding. The last added variable is a description of addition bits. This variable is equal to zeros(1, number_of_bits), where the number of bits is a function input parameter representing how many bits will be for transmitting a single state. The length of this variable is modified whenever global variable 'M' is changed. For instance, 3 bits are needed for transmission of the single symbol when M=8. For M=16, 4 bits are required and so on. When the number of 'necessary_bits' is defined, the next step is to divide it by the number of bits and find out how many remain. This is done by mod command.

```
number_of_additional_bits=mod(necessary_number_of_bits,number_of_bits);
```

There are two options after this division is made. The first one is that number of remained bits is 0. In this case bit stuffing is not needed. Otherwise bit leftover can occur. The number of the bits leftovers fluctuates from 1 to number_of_bits -1. To find out how many bits are missing, the 'number of_bits' is subtracted from additional bits.

A variable 'missing_bits' is represented by a decadic value. This value symbolizes how many bits will be added to the string as stuffing. On a first basis, binary conversion is essential.

```
for i = 1:number_of_bits
  description_number_of_additional_bits(1, i) =
  bin2dec(bin_number_of_missing_bits(i));
end
```

Variable 'description_number_of_additional_bits' stores binary information on how many bits were added as stuffing. As a conclusion to the message string, suffix is added.

```
resulting_message = cat(2,message,separator_of_string_terminator,
ones(1, number_of_missing_bits), string_termination,
 description_number_of_additional_bits);
```

Variable 'separator_of_the_string' is a bit of the value 1. After separator ones are added. The number of the ones depends which value is stored in 'number of missing bits'. 16 zeroes are stored in the 'string_terminator'. It represents the double size of one ASCII character(8 bits). Lastly, binary value of stuffed bits is added. For better understanding let us assume, that two bits are going to be stuffed. Then resulting message will be as shown below, where each color symbolizes suffix variables mentioned above.

```
resulting_message = [(message)111000000000000000010]
```

3.4 Create Modulation Symbol Function

Afterwards, the desired message is entered and aligned. The next function called by main program is 'create_modulation_symbol'. This function's purpose is to convert the binary string of data into modulation symbols. These symbols are created as .wav files. Later on, they are used by the main program. The required input parameters are 'fhigh', 'fdown', 'T', 'Fs' and 'type'. All parameters are global variables taken from the main program. The flowchart is displayed in figure 3.2.

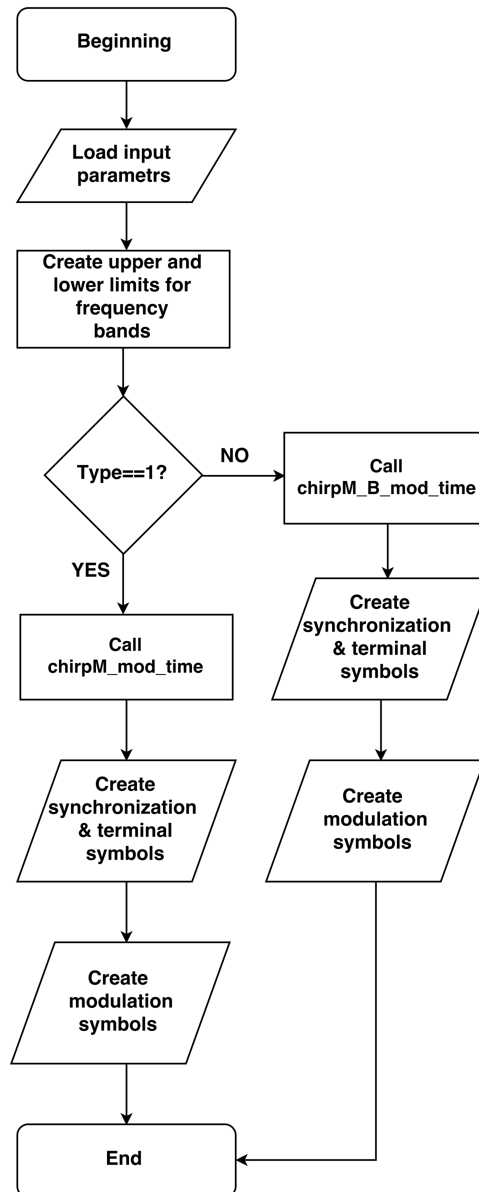


Figure 3.2: Create modulation symbol function flowchart.

Not all modulation symbol serves for data representation. The communication system needs to know when the data stream is starting and when it is ending. For this purpose, another two modulation symbols are made. They are called synchronization and termination symbols. First of all, the decision on where in the frequency band these symbols will be allocated has to be made. For a better system interception, the reliability of synchronization and termination symbols is designated by separated frequency band. This subband has the size of one tenth from assigned frequency band. Another one tenth of the assigned frequency band is reserved as the spacing between these two symbols and data

modulation symbols. Synchronization reliability is improved even more by that. Thus, eight tenths of the allocated frequency band are used for data modulation symbols.

```
lower_range_of_control_zone = fup - ((fup - fdown) / 10);
upper_range_of_mod_symbols = lower_range_of_control_zone -
((fup - fdown)/10);
```

The code lines above are describing frequency band allocating. This method decreases the band for data transmission, which is undesirable. Another approach could be to assign frequency band outside of desired bandwidth. But strong interference could occur considering an unawareness of transmission density by other systems. Since synchronization is crucial for a communication system, sacrificing 20% of assigned bandwidth for data transmission is the better option than the probability of total system failure.

After that function, it is important to decide which type of method will be used to calculate the chirp rate. If the global variable `'type'` is equal to 1, the frequency band splitting procedure will be applied. Otherwise, a symbol period splitting method will be utilized. These subfunctions will be described in great detail later on. For now, it will be enough to know that both functions return input variable `'y'`. This variable symbolizes the modulation symbol with unique chirp rate slopes for each method. However, the principle of the modulation symbols creation is identical for both. When the method is selected, in the first place synchronization and termination symbol are created by subfunctions `'chirpM_mod_time'` or `'chirpM_B_mod_time'`. These two symbols are 2-ary CSS, code sample shown below. Symbol 1 represents synchronization symbol. Symbol 2 stands for termination symbol. Due to the fact that both symbols are 2-ary CSS, there is no need to create them for higher M-ary CSS. As a result, both symbols are identical for any type of M-ary CSS. The main reason for creating these 2 symbols in 2-ary CSS is to not reduce a quantity of the modulation symbols representing data in any chosen M-ary CSS. Imagine if in 8-ary CSS where 2 symbols are used for synchronization and termination symbols. This would mean a 25% loss of capacity which is unacceptable. The last step after the creation of synchronization and termination symbols, is to write them into a .wav file.

```
Synchro_symbol = chirpM_mod_time(T,1,fhigh,lower_range_of_control_zone,
2,Fs);
Termination_symbol = chirpM_mod_time(T,2,fhigh,lower_range_of_control_
zone,2,Fs);
```

The second stage is the development of the modulation symbols, that are representing data. The predecessor had designed a code, which is inappropriate as shown below. If the change of M-ary CSS is requested, it is necessary to edit the code for this switch structure.

```
for i = 0:M-1
y = chirpM_mod_time(t,i,fup,fdown,M,Fs);
switch i
     case 0
           filename = 'Symbol_0.wav';
     case 1
           filename = 'Symbol_1.wav';
                      .
                      .
     case 63
           filename = 'Symbol_63.wav';
end;
audiowrite(filename,y,Fs);
end;
```

It was therefore essential to create a more sophisticated solution. As a result, string concatenation was used to solve this issue. String concatenation is the operation of joining character strings end-to-end. For example, the concatenation of 'data' and 'base' is the database. The code below is a for loop, where `'M'` global variable represents `'number_of_states'` for CSS. In first iteration loop calculates chirp rate for symbol number 1. Then string concatenation put together characters `'Symbol_1'`(because i=1) and .wav resulting in the creation of `'Symbol_1.wav'` string. Subsequently, chirp rate for symbol 1 is written to the audio file called Symbol_1.wav and so on for the next iteration.

```
for i=1:M
     y=chirpM_mod_time(T,i,upper_range_of_mod_symbols,fdown,M,Fs);
     %concation of strings
     filename = strcat('Symbol_', num2str(i), '.wav');
     wavwrite(y,Fs,filename);
end;
```

The resulting code optimization has made the code more arranged and decreased computation difficulty significantly. When parameter M was set to value 32 creation of the modulation symbol number 31 was requested for the original program. It took 1.56 seconds to create this modulation symbol. The same request was completed by optimized version with an outcome of 0.27 second. This results in a approximately five times faster creation of the modulation symbol comparing to the original program. For a larger value of `'M'`, even faster creation can be expected.

3.5 Software Realization of Frequency Band Splitting Method

As noticed above in image 3.4, `'create_modulation_symbol'` function is calling another subfunction, which returns a symbol with specific chirp rate slopes. If global variable `'type'` has value 1. Then `'create_modulation_symbol'` will call subfunction for calculation a unique chirp rates by frequency band splitting method 2.1.1.

Input parameters of this function are global variables `'fhigh'`, `'fdown'`, `'T'`, `'Fs'` and variable `'symbol'`. Parameter symbol is the decadic expression of the modulation symbol. Flowchart of band splitting method is shown in 3.3

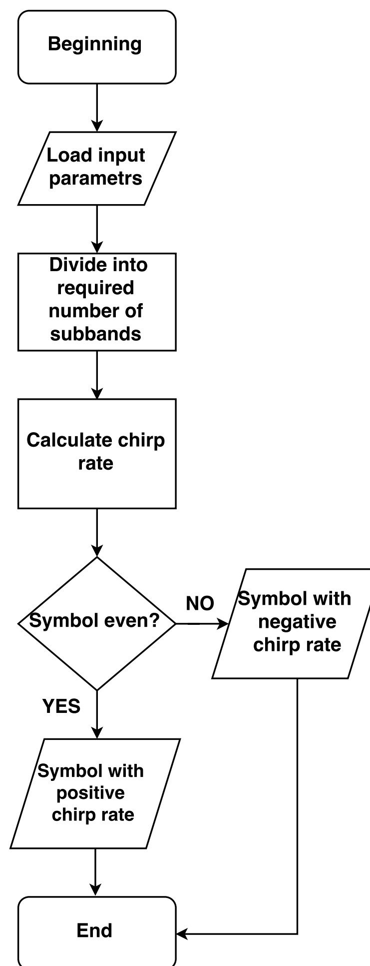


Figure 3.3: Band splitting method flowchart.

After input parameters are loaded size and number of subbands are defined. Subsequently chirp rate is calculated by equation 2.3. As seen, chirp rate is changing by symbol period but also with the number of states. Parameter M is divided by two because two

modulation symbols belong to each subband. Below, a code for calculating chirp rate is displayed.

```
subband_size = (fhigh-fdown)/(M/2);
chirprate = subband_size / T;
```

When chirp rate is computed, the next step is to decide which modulation symbol will be created. As mentioned before this is done by decadic expression stored in parameter symbol. For each decadic values 0, 1, 2, 3 corresponds a binary expression 00, 01, 10, 11. It is essential to distinguish which modulation symbol will have positive and negative chirp rate. This is done by dividing decadic values into 2 groups of even and odd symbols. In Matlab environment that could be accomplished by (x,2). Where mod(x,2) returns remainder after the division of 2. For odd numbers mod(x,2) returns 0 and for even 1. If the result will be 0, a symbol with positive chirp rate will be generated. A negative symbol will be created when modulus returns 1. Code implementation of modulus is shown beneath.

```
switch mod(symbol,2)
  case 0
    y = sin(2*pi.*((fdown+(subband_size*floor...
    (symbol/2))).*time+((chirprate/2).*(time.^2))));
  case 1
    y = sin(2*pi.*((fhigh-(subband_size*floor...
    ((M-symbol)/2))).*time + ((-chirprate/2).*(time.^2))));
end;
```

The division into the respective subbands is ensured by command floor(symbol/2). Floor command rounds down to next integer.

Code `((fdown+(subband_size*floor(symbol/2)))` determines subband value which will be added to initial frequency `'fdown'`. The whole code above decides in which subband symbol will be placed. Symbol 0 (00 in binary) starts from the frequency `'fdown'`, where floor returns 0 so nothing will be added to frequency `'fdown'`. Next symbol 2 (10) is shifted by one subband upwards because the result of the floor command is 1. The same procedure will be applied for even symbols. Odd symbols have a small adjustment in the code which will ensure the identical result from floor command as for even symbols. A detailed explanation is displayed in table 3.1.

symbol number(even)	floor(symbol/2)	symbol number(odd)	floor((M-symbol)/2)
0	0	1	0
2	1	3	1
4	2	5	2
6	3	7	3

Table 3.1: Subband division

3.6 Software Realization of Symbol Period Splitting Method

The software realization for the second method is more complicated than for the first one. The main reason is that for each modulation symbol two distinct chirp rates have to be created. Input parameters are identical as for frequency band splitting method. Therefore, global variables `'fhigh'`, `'fdown'`, `'T'`, `'Fs'` and variable `'symbol'` are requested as the input of this function. Flowchart of symbol period splitting function is shown in figure 3.4.

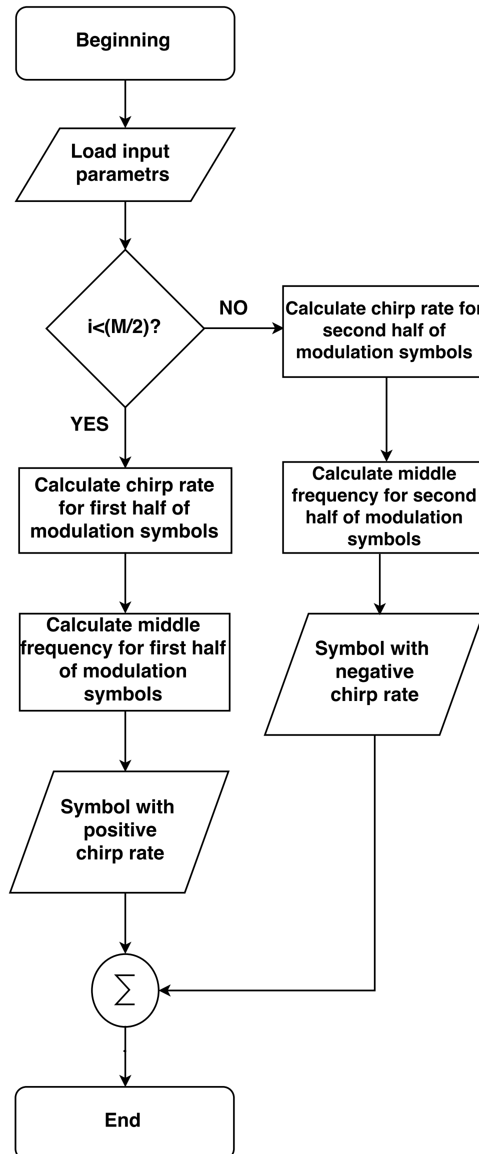


Figure 3.4: Symbol period splitting method flowchart.

From the description of this method in 2.1.2 there is a clear difference from the first method. For chirp rate calculation in the first method, it did not matter what the symbol is. The chirp rate was still the same, the only difference was in positive or negative sign depending on whether it was an even or odd symbol. The identical principle cannot be applied to this method. Each symbol has two different values of the chirp rate that jumps halfway in the middle of the modulation symbol period. Another difference is the distribution of the symbols to two halves, where the positive chirp rate is used for lower frequency half and for the second negative it utilized. In a case of 4-ary CSS modulation symbols, 0 (00) and 1 (01) are generated with the positive chirp rate. For the modulation symbols 2 (10) and 3 (11) are generated with the negative chirp rate.

```
for i=1:(M/2)
     chirprate1(i)=i*subband_size/(0.5*T);
     chirprate2(i)=((M/2)-i+1)*subband_size/(0.5*T);
end;
for i=(M/2)+1:M
     chirprate1(i)=-chirprate1(i-(M/2));
     chirprate2(i)=-chirprate2(i-(M/2));
end;
```

The program for generating two chirp rates for each symbol is shown above. In the first for loop both, chirp rates for first half of the modulation symbols duration are calculated. The second cycle calculates as well both chirp rates, but for the second half of the modulation symbols duration. It is seen that value of chirp rates for the second half of the modulation symbol is the negative value of the chirp rate from the first half.

Before the creation of the modulation symbol, the middle frequency needs to be calculated. It is the frequency at which chirp rate change is made. This frequency is located in the middle of the symbol duration. That means middle frequency name correlates with the symbol period, not with a bandwidth of the modulation Obviously, according to the current chirp rate, the middle frequency will change for each individual modulation symbols. The code for calculation of chirp rate is displayed below. Switch structure with fix command (rounds toward zero) ensures that for the first half of symbols fix command will return 0. This modulation symbols will be generated with positive chirp rate. When fix command returns 1, the second half of symbols with negative chirp rate will be generated.

```
switch fix((symbol-1)/(M/2))
  case 0
       y_first_half=sin(2*pi.*(fdown.*time+(chirprate1(symbol)/2)
       .*time.^2));
       %phase detection of last sample
```

```
        last_sample_phase=sin(2*pi.*(fdown*(T/2)+(chirprate1(symbol)/2)*
        (T/2)^2));
        y_second_half=sin(last_sample_phase+2*pi.*((freq_middle(symbol))
        .*time+(chirprate2(symbol)/2).*time.^2));
    case 1
        y_first_half=sin(2*pi.*(fhigh.*time+(chirprate1(symbol)/2)
        .*time.^2))
        last_sample_phase=sin(2*pi.*(fhigh*(T/2)+(chirprate1(symbol)/2)*
        (T/2)^2));
        y_second_half=sin(last_sample_phase+2*pi.*((freq_middle(symbol))
        .*time+(chirprate2(symbol)/2).*time.^2));
  end;
        %merging vectors of the first and the second half together
        y = cat(2, y_first_half, y_second_half);
```

Whatever which case is chosen, the procedure of generating positive or negative chirp rate for the symbol is the same. Two chirp rates are calculated for each modulation symbol as mentioned above. Between generation of both halves, phase of the last sample is calculated. For time axis is generated from 0 to `(T/2)-(1/Fs)`. Therefore `(T/2)` is technically the last sample of the time axis. The reason for creating this variable is to reduce possible phase jump in the bond between the first and the second halves. Lastly, both halves are concatenated into output variable `'y'`. At a closer examination, the chirp rate value for the first half of the modulation symbol's duration is a multiple of the symbol order. For the second half of the duration of the modulation symbol, the rate of frequency sweep is reversed.

3.7 Transmission of the Message

When the message is aligned and all symbols are created, the last step is to send it. This is done by transmission loop. Before the transmission loop is executed, it is required to set variables, that are crucial for synchronization and termination of the transmission. These parameters are an `'end_of_transmission'`, `'terminate_transmission'`, `'synchronization_counter'` and `'s'`. `'s'` refers to if the synchronization was set or not. All these variables have default value 0. The only exception is the parameter current character with default value equal to 1. `'current_character'` describes the order of the transmitting characters. The process of the transmission loop is shown in flowchart 3.5. However, flowchart is quite sophisticated. To make it easier to understand, the diagram was divided into three stages. For each stage a different color in the flowchart is chosen. Red color represents synchronization stage. Data sending phase is defined by green and the blue color describes termination stage.

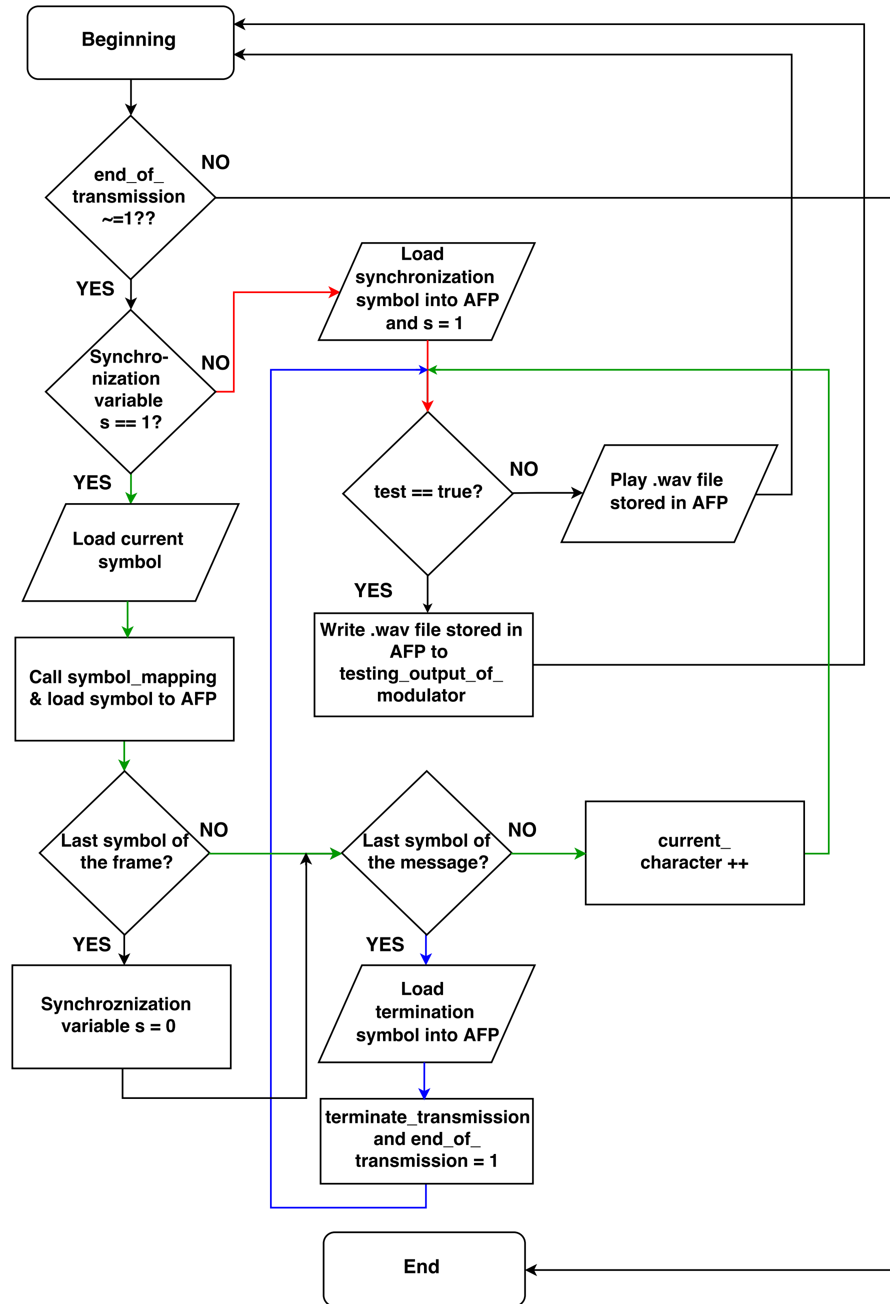


Figure 3.5: Flowchart of the transmission loop.

At the beginning of the transmission variable `'s'` is set to zero, which means that it is essential to send synchronization symbol. This symbol was already made by the function `'create_modulation_symbol'`. The next step is to allocate memory for audio file player(variable AFP) and then load synchronization symbol into this variable.

```
%allocation of the memory for loading symbols
AFP = zeros(length(wavread('Symbol_1.wav')), 1);
AFP = synchro_symbol;
```

After AFP is defined the variable `'s'` is set to 1. This means synchronization was sent, even if synchronization symbol will be sent in next few steps. The reason behind this is to save computational difficulty. So there is just one loop for sending modulation symbols, which does not recognize the difference between synchronization, termination and modulation symbol. Synchronization, termination and data sending are performed as a modulation symbol play. This embodiment is selected because amateur radio stations contain audio input and output. After synchronization symbol is loaded to AFP, it is required to set properties of an audio player which will play (transmit) the file stored in the AFP. The audio player is defined by Matlab toolbox dsp.AudioPlayer where key parameters are a sample rate which specifies the number of samples per second in the signal. The default value is 44100. This number is changed to the value which is stored in global variable `Fs`. The second parameter is queue duration, which specifies the duration of the signal, in seconds, that can be buffered during the simulation. The purpose of the queue is to control the trade-off between latency and data dropout. To minimize latency queue duration can be set to lower value. However, if the Matlab data throughput rate is lower than the device throughput rate, a buffer underrun occurs. This requires a device reading from the buffer to pause its processing while the buffer refills. This results in an unwanted data transmission delay. Underruns can be monitored by property `'OutputNumUnderrunSamples'` and by its value queue can be optimized. Another possible scenario is that Matlab data throughput rate is higher than the device throughput rate, a buffer overrun occurs. Buffer overrun causing overwriting of the buffer producing data dropouts. For this case, Matlab must wait before writing data to the queue. To minimize chances of the dropouts it is recommended that queue duration is at least as large as the frame size. This advice is respected, thus the queue is as long as the duration of the modulation symbol.

```
AP = dsp.AudioPlayer('SampleRate',Fs,'QueueDuration',T);
```

The last step in synchronization stage is to decide if synchronization symbol will be sent right away or attached to the testing file. If the global variable test is set to false synchronization symbol is transmitted (played) by dsp.AudioPlayer toolbox. Audio data

with properties of the dsp.AudioPlayer is played by calling step function.

When value true for the global variable test is chosen, synchronization symbol which is currently saved in variable `'AFP'` will be written to the vector `'testing_output_of_modulator'`. The main reasons to create testing vector is to analyze the whole modulation symbol sequence. Further advantage is reducing the time consumption for debugging received signal. For an example, transmitting 60 symbols, where each symbols has a duration 1 second. It will take 1 minute when the test is not enabled. But if the test is enabled the receiver almost instantly loads testing output file.

```
if test == true
    %add currently transmitted symbol to the test vector
    testing_output_of_modulator=cat(1,testing_output_of_modulator,AFP);
else
    %play .wav file stored in AFP variable
    step(AP, AFP);
end
```

Afterward, synchronization sequence ends because variable `'s'` is equal to 1. Therefore, the data sending phase(green in 3.5) begins. The initial current symbol is loaded from data string. Because current symbol is loaded in the binary string, firstly it has to be converted into decadic expression.

```
symbol_bin_to_string = num2str(string_sent(current_character:
(current_character + (number_of_bits - 1))));
%decadic number of binary string which represents symbol number
number_of_symbol = bin2dec(symbol_bin_to_string);
```

Then `'symbol_mapping'` function is called by main program. This function finds out which decadic expression stored in the variable number of symbol is corresponding to the modulation state, that is going to be transmitted. The state of the modulation is returned and displayed in Matlab console during real-time transmission. This allows an easier analysis and debugging of the transmission. Function `'symbol_mapping'` requests 3 input parameters. The first parameter, `'number_of_symbols'`, loads decadic value stored in the variable as described above. Then string compares command requests string, where two options can be chosen. The state is the one string that could be selected. When chosen, the function assign the value stored in the `'number_of_symbol'` to the table representing modulation state. Meaning decadic value stored in number of bits is equal to the modulation symbol, which will be transmitted. If for any reason current assignation is unsatisfying it is possible to make modification within the function quite easily. For example to the Gray code or any other method. The second select option for the string compare command is `'symbol'`. It is always used by the receiver. It will be described in

chapter 4, in sum when is selected symbol mapping function is doing reverse operation. The last necessary input parameter of the function is M.

```
number_of_state = symbol_mapping(number_of_symbol, 'state', M)
```

When mapping function returns, the number of the modulation symbol to be transmitted. The next step is to load this symbol to `'AFP'` variable. Firstly, all modulation symbols are allocated as array with the length of the `'M'`. Where each column represents a modulation symbol. This procedure is done just one time. The main reason is to decrease computation difficulty of the program. When global variables are inputted, it is already clear how many modulation symbols will be needed. Since the property of the modulation symbol is unchanged during transmission, there is no need to repeat this operation.

```
%loading of modulation symbols
  for =1:M
    %concatination of Symbol_+(i)+.wav
    filename = strcat('Symbol_', num2str(i), '.wav');
    %load array of the modulation symbols
    mod_symbols(:,i) = wavread(filename);
  end
```

Secondly, current modulation symbol is loaded to the variable `'AFP'` and ready to be transmitted. Then the `current_character` is tested by multiple conditions. The most important are the conditions verifying if `'current_symbol'` is the last modulation of the frame and of the whole message string. If any of these requirements is not met the current character will increase his value for next iteration and will be the transmitted one or saved to the testing output of the modulator. Incrementation of the `'current_character'` is done by adding a certain number of the bits. This value correlates with the selection how many states M-ary CSS will have. For instance, 8-ary CSS will increase current character by 3 bits in each iteration. One of the feature of this software application is an ability to send synchronization after certain given period by the value of the variable `'duration_of_data_frame'`.This is decided by a return value of the modulus command below, where `'synchronization_counter'` stores a value how many synchronization symbols were sent. This is for the case when message string is multiple times longer than the size of data frame variable. This variable is equal to: `(duration_of_data_frame * number_of_bits / T)`. To be capable to maintain functionality of the code among all possible variations of the global parameter `M`, `'number_of_bits'` variable is added into modulus command.

For a better understanding of modulus condition lets us have an example. If the size of data is 40, number of bits is 4, T equals 1 and 10 is stored in `'duration_of_data_frame'` variable. When transmission of this begins, that means synchronization was already sent, thus in `'synchronization_counter'` 1 is stored. The first `'current_symbol'` has value 1 (dividend) and the divisor is equal to 37. After 9 iterations the `'current_symbol'`

value $(4 \times 9 + 1)$ is equal to the divisor, resulting remainder 0 when modulus command is executed. Variable `'s'` is going to be set to 0 followed by transmission and counter incrementation of the synchronization. When 9 iterations are done current symbols is storing number 73 and division value is the same$(2 \times 40 + 1 - 8)$. Therefore next synchronization is sent and so on, until the end of message string. To ensure that modulus command works properly under any given conditions, `'current_character_synchro_fix'` and `'counter'` variables are necessary. The counter counts from 1 till reaches value inputted into the duration of the data frame. Then resets itself and rises value of `'synchronization_counter'` by 1. The `'current_character_synchro_fix'` ensures that initial value of this variable is 1 and in each iteration is increased by 1. In fact, `'synchro fix'` serves as decimator of the `'current_character'` for the `'counter'`. To secure function of the `'synchro fix'`, small adjustment has been made. The initial value of the `'current_symbol'` is 1 and then is incremented by 4 (for $M = 16$) in each iteration. But in code, `'current_character'` variable is increased before synchronization is executed. Meaning that for every first character after the synchronization `'synchro_fix'` variable is not calculated. This is done intentionally, because when synchronization is sent for every first character, modulus operation returns 0. Which would lead to false indication of the last character in the frame. This is the reason why there are 9 iterations instead of 10 even when the duration of data is set to 10.

```
AFP = mod_symbols(:, number_of_state);
%size of frame is exceeded => necessary to send synchronization again
if mod(current_character,(synchronization_counter*size_of_data_frame+1)
   -synchronization_counter*number_of_bits) == 0
%if s = 0 next symbol loaded into AFP will be Synchronization_symbol
s = 0;
end
%testing is current_character is last character of message
if (current_character + number_of_bits) <= length(string_sent))
  %next current_character for upcoming iteration
  current_character = (current_character + number_of_bits);
  current_character_synchro_fix=fix(current_character/number_of_bits);
  counter = mod(current_character_synchro_fix,duration_of_data_frame);
    if counter == 0
      synchronization_counter = synchronization_counter+1;
    end
else
terminate_transmission = 1;
end
```

When the last current character is about to be transmitted termination condition is fulfilled. At this moment termination sequence (blue in 3.5) has begun. When the if condition above is met variable `'terminate_transmission'` is set to 1. In the next iteration termination symbol will be loaded into `'AFP'` variable. Afterwards, transmitted or saved into the testing vector. Termination procedure is almost identical to the synchronization and data sending sequences. However, with the exception of changing the value of the variable `'end_of_transmission'` to 1. This will cause end of the transmission loop. When transmission loop is terminated the last operation is to create a .wav file from testing vector if test option was selected. Before an audio file is created, a portion of zeroes is added to the beginning of testing vector. The reason for putting these nulls into the sound file is because some audio players (e.g windows media player) have issues to play the .wav file without them.

```
testing_output_of_modulator=cat(1, zeros(Fs, 1), testing_output_of_
modulator);
wavwrite(testing_output_of_modulator, Fs, 'testing_output_of_
modulator.wav');
```

4

Software Realization of Receiving Side

In this chapter software solution of M-ary demodulation is described. This is the second part of the communication system. As mentioned in a chapter 3, the predecessor has made flaws in his program. This results in an unsatisfying performance of the demodulation program. Considering that reducing computation difficulty is the main task of this thesis, most of the predecessor's code had to be changed to achieve this goal. Detailed characterization will be discussed in sections below.

4.1 Software Implemntation of the Main Program Receiver_main

The forerunner has left plenty of imperfections in his program solution, which is managing data receiving process. The majority of flaws were related to the software solution of the transmitter referred in the 3.1. Among these code defects were separated folders for each M-ary CSS, an inability to change some properties of the receiver and inappropriate code structures. A consequence of these imperfections is low performance of the receiving side. In order to improve the performance, the same structure as for transmitter was followed. Therefore, the main program `Receiver_main` was created. As a result, there is just one file which calls all necessary function for receiving process. To understand concept of the `Receiver_main`, the flowchart 4.1 is shown. More detailed description of each essential block will be presented later on.

The receiver was designed in the same way as the transmitter. Firstly, the main program loads global variables, then the properties of the DSP systems toolbox are set. The DSP settings are essential for demodulation of the data string. Afterwards, all symbols are allocated. The principal reason for the symbol allocation operation is to reduce computation difficulty of the software solution. The core of the receiver is a demodulation loop. The function for finding synchronization symbol is called as the first

in this loop. When the synchronization symbol is discovered, function also locates the position of synchronization symbol in the buffer. Subsequently, `'dem_chirp'` function demodulates the data stored in the buffer and returns them as `received_state`. If `'dem_chirp'` evaluates the termination symbol as `received_state`, the -1 value will be written into the variable `'received_state'`. This will be followed by termination of the demodulation loop. Lastly, the received modulation states are translated into ASCII characters and displayed in the dialog window.

4.2 Setting Properties of the Receiver

To guarantee proper functionality of the receiver, it is necessary to set correctly the essential global variables. Some of them were inherited from the transmitter. The global variables `'fhigh', 'fdown', 'T', 'Fs', 'type' and 'duration_of_data_frame'` are among them. For the the `'Receiver_main'` extra global variables `'decimation_factor'` and `'demodulation_method'` were added. The receiver offers two methods how to demodulate signal. Global variable `'demodulation_method'` serves for the demodulation method selection. When `'corr'` is written in the variable cross-correlation method will be utilized by the receiver main program. If `'frft'` is stored in the `'demodulation_method'`, the fractional Fourier transform will be used as demodulation technique for the receiver. Both options will be described later, in their sections.

The other properties necessary to set are several Matlab DSP system toolboxes. The first DSP system toolbox is in charge of cross-correlation. The cross-correlation is a common technique used in signal processing which measures the similarity of two series as a function of the displacement of one relative to the other. Important feature of the CSS is an exceptional cross-correlation properties. This means, that the correlation among modulation symbols is very low. This feature is the main reason why CSS is implemented in the communication system. On the other hand, the disadvantage is that cross-correlation operation is vastly increasing the computational difficulty. The approaches to reduce the computational difficulty are introduced in this chapter.

In the receiver, cross-correlation is computed between data stored in the buffer and the modulation symbols. This process is managed by dsp.Crosscorrelator system object. The computation can be done in the time domain or in the frequency domain. The domain can be specified through the Method property. When 'Fastest' is selected dsp.Crosscorrelator computes the cross-correlation in the domain that minimizes the number of computations. This is the best solution to minimize the computational power introduced by the dsp.Crosscorrelator.

```
xcorr = dsp.Crosscorrelator('Method','Fastest');
```

As mentioned before, the communication system is using acoustic frequency band

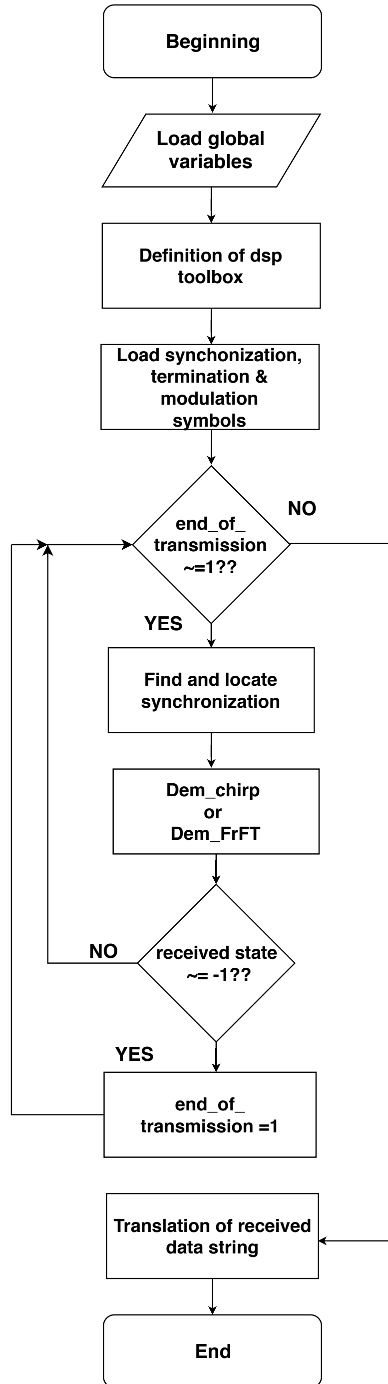


Figure 4.1: Simplified flowchart of receiver.

for transmitting. Therefore, a microphone is utilized as a receiving device. To ensure errorless functionality of the system microphone properties must be set precisely. A dsp.AudioRecorder system object defines and sets up your audio recorder object. As shown below, many properties has to be set. All set values are calculated from global variables. Resulting accurate set up of the microphone at any given conditions. This is a major advantage over the predecessor‘s version where any change of single variable (e.g. sample rate) could lead to complete failure of the microphone, resulting in the receiver malfunction. All the possible properties and their setting are described in [10] so only the most important settings are going to be mentioned. The crucial property to be configured is the buffer size, which specifies the size of the buffer that the audio recorder object uses to communicate with the audio device. To be able to tune this attribute, firstly the buffer size source must be set to 'Property'. Then buffer size is equal to the value stored in the variable `'length_of_symbol'`. Value of this variable is equal to $F\mathtt{s}\times T$.

```
Mic = dsp.AudioRecorder('SamplesPerFrame',length_of_symbol,
'BufferSizeSource','Property','BufferSize',length_of_symbol,
'QueueDuration',0,'NumChannels',1,'OutputDataType','double');
```

It is necessary to define the device which acquires audio data. When the option default is chosen, computer main recording device will be used. Another key feature to be set is the sample rate of the defined microphone. Since default value is set to 44100 it was required to change this feature to value stored in global variable `'Fs'`.

```
Mic.DeviceName = 'Default';
Mic.SampleRate = Fs;
```

Signal acquired by the microphone or from testing vector needs to be temporarily stored, until is processed by the software solution of the receiver. This momentary storage is called buffer and was mentioned above. The buffer is the last essential DSP system object defined in `'dsp.Buffer'`. The first fundamental feature to configure is the buffer length which specifies how many samples can be gathered in the buffer. Double size of symbol length was chosen as a length of the buffer for `'Receiver_main'`. The logic behind this choice will be explained in section 4.3.1. The second parameter defines how many samples will be taken from the buffer and processed by the receiver. Name of this property is overlap length. Overlap length is set to the value stored in `'length_of_symbol'`. With this setup of properties, the whole buffer is loaded by the receiver in 2 iterations since overlap is half of the buffer size. In comparison to predecessor‘s code, the ability to freely change any global variable without obligation to manually edit buffer properties to assure proper functionality of the system is another advantage of `'Receiver_main'` solution.

```
InBuff = dsp.Buffer('Length', 2*length_of_symbol,'OverlapLength',
length_of_symbol);
```

Lastly, synchronization, termination and modulation symbols are loaded for decreasing computation difficulty purpose. The same operation is done in the transmitter. Since transmitter and receiver were tested in one computer, receiver loads symbols created by the transmitter. However, if a communication system does not operate on the single pc. There is an issue of an inability to load all necessary symbols on the receiving side, because symbols (.wav files) do not exist in the folder assigned to the receiver. Two possible solutions can be considered. The first solution is to execute `'Transmitter_main` which will be created desired modulation symbols for the receiver. The second option is to create a function which will search for modulation symbols in the designated folder. If some modulation symbols are found, function will ask whether symbols should be erased or not. If the detected symbols have properties, that user wants to use, function will simply load those symbols. Due to time deficit and insignificant benefit compare to other functions in the current setup of the communication system the first solution was implemented. Even the second method is undoubtedly better for future implementation of the system on multiple workstations.

4.3 Interception of the Message

After all necessary parameters and modulation symbols are loaded, the receiving loop is initialized. The loop can be divided into two sections. The first sector is in charge of locating the synchronization symbol. Then the second part demodulates received data. Since the whole procedure is sophisticated, a flowchart is introduced in 4.2. Receiving loop begins with an initial condition which is controlling two circumstances logically conjuncted .The first condition inspects if the variable `'end_of_transmission'` is equal to 1 meaning that termination symbol was recognised. The second condition counts processing time of the locate synchronization function. When timer value is higher than the defined threshold the condition is evaluated as false resulting termination of receiving loop. The reason behind this is to terminate receiving process when no data are being transmitted. It is unnecessary to workload computer during that time. Because the transmitter had an option to create testing vector, it is logical that receiver supports this option. When in the variable `'test'` true is stored, a testing feature is enabled. Data are loaded from testing vector to the buffer by the step command. Afterwards, the iteration variable is increased. By help of this variable in the next cycle, the following data will be loaded from the position where the last data ended. In other words it is a shift register.

```
dataIn = step(InBuff, testing_output_of_modulator((((iteration-1) *
length_of_symbol) + 1):(iteration * length_of_symbol)));
```

```
iteration = iteration + 1;
```

If false is stored in the variable the test data will be loaded from the microphone to the buffer. Considering that the microphone works already as a shift register, because of its settings, there is no need to apply iteration variable. Both methods always store data of the length defined in `'length_of_symbol'`, ensuring compatibility with buffer.

```
dataIn = step(InBuff, step(Mic));
```

When data are already stored in the buffer it is necessary to find synchronization symbol and its location, in order to determine beginning of data the stream. For this purpose the `'locate_synchronization'` function was designed. This function will be explained detaily in the section 4.3.1. When `'locate_synchronization'` is called, it returns variables `'found'` and `'synchronization_location'`. When found variable stores value 1 (true) it means that synchronization was successfully discovered and its position in the buffer indicates `'synchronization_location'`. After synchronization is discovered the data demodulation can begin. Function `'dem_chirp'` is in charge of demodulation process. This function will be briefly introduced here and described later in 4.3.2. The main task of `'Dem_chirp'` is to demodulate signal in the buffer. Demodulation is done by correlation which assigns inputted data to the modulation state. This state is then returned by function to the main program. Next if condition is testing if termination symbol is detected. Till this event occurs demodulated modulation states are added to the vector of the received states. When the termination symbol is detected variable `'end_of_transmission'` changes its value to 1 and in the next cycle the receiving loop will be terminated. Leaving vector of the received states as an output.

4.3.1 Locate Synchronization Function

The crucial part of the receiver is to precisely locate the synchronization symbol. This is done by the `'locate_synchronization'` function with input parameters: `'dataIn'`, `'xcorr_synchr'`, `'synchro_symbol'`, `'Modsymbols'`, `'termination_symbol'`,

`'length_of_symbol'`. The signal loaded to buffer is stored in the variable `'dataIn'`. The number of modulation states is represented by `'M'` and `'xcorr_synchr'` parameter sets up the properties of `'dsp.Crosscorrelator'`. Definition of the modulation symbols size is stored in the `'length_of_symbol'`. The rest of the parameters are modulation symbols, which are utilized for correlation between them and data stored in the buffer. When `'locate_synchronization'` function is completed it returns the variables `'found'`, `'synchronization_location'` and `'correlation_location_of_synchronization'`.

The `'found'` variable stores information if synchronization was successfully detected or not. Default value of this variable is 0. This means, that the synchronization symbol was not identified. In case that synchronization symbol was revealed in the buffer `'found'`

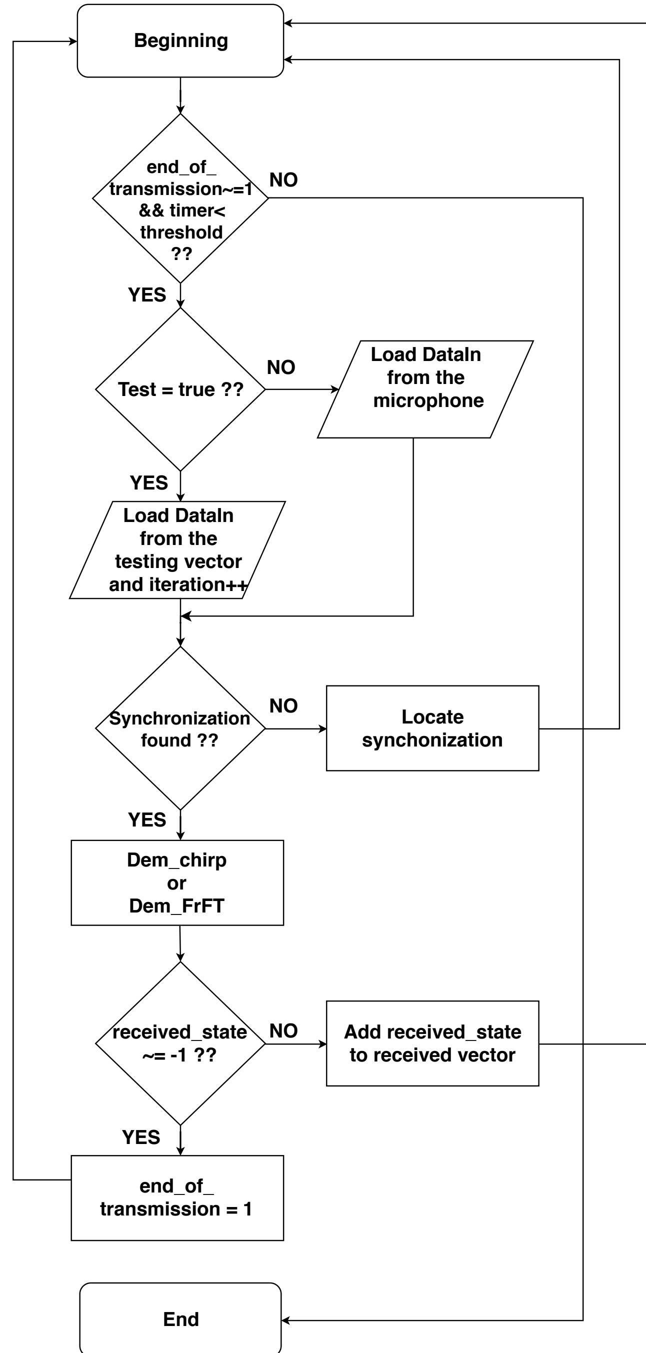


Figure 4.2: Flowchart of the receiving loop.

variable is set to 1. Position of synchronization symbol is indicated by the variables 'synchronization_location' and 'correlation_location_of_synchronization'.

Difference between them is that 'synchronization_location' shows where in the buffer synchronization symbol is located. On the other hand

'correlation_location_of_synchronization' marks the beginning of the synchronization symbol in the vector of cross-correlation.

Before 'synchronization_location' function is initialized, a normalization of the data is done. The main reason for this procedure is to set the received signal to the defined level at any given condition to ensure a correct evaluation. First of all, normalization level is set to value 0.7071. Then root mean square of the data stored in the buffer is computed. Afterwards, normalization level is divided by calculated rms value and result is stored in the variable 'normalization'. In the end each sample in the buffer is multiplied by a normalization factor, so its rms is equal to the normalization level.

After signal is normalized 'synchronization_location' function is called by the main program. In the first place function is testing if there is any data in the buffer. This is done by computing mean value of the buffer. If value is higher than 0.01 this means that some data are stored in the buffer thus, function will proceed. Otherwise 'synchronization_location' is termined, because buffer is empty. When data are stored in the buffer, firstly cross correlation of 'dataIn' between synchronization symbol, termination symbol, modulation symbol is done.

```
synchronization_correlation=step(xcorr_synchr, synchro_symbol, dataIn);
modulation_symbol_correlation=step(xcorr_synchr, Modsymbols, dataIn);
termination_symbol_correlation=step(xcorr_synchr, termination_symbol,
dataIn);
```

Then maximum value and its location in the cross correlation vector of "correlation_location_of_synchronization" is computed. The same procedure is executed for vectors 'modulation_symbol_correlation' and 'termination_symbol_correlation'.

Considering that length of the symbol is identical for all symbols, it is not required to calculate maximum value for entire correlation. Since position of maximum will be on location alike to the correlation_location_of_synchronization it is enough to calculate maximum in the range of 50 samples before and after the position of the correlation_location_of_synchronization'. Computing power is spared significantly by that.

```
[synchronization_level,correlation_location_of_synchronization]=max(abs
(synchronization_correlation));
modulation_symbol_level, ~] = max(abs(modulation_symbol_correlation
(((correlation_location_of_synchronization - 50):
```

```
(correlation_location_of_synchronization + 50)), :)));
[termination_symbol_level, ~] = max(abs(termination_symbol_correlation
(((correlation_location_of_synchronization - 50):
(correlation_location_of_synchronization + 50)), :)));
```

Subsequently, maximum values from each cross-correlation vectors are compared. If `'synchronization_level'` is higher than others maximums, a location of the synchronization is defined. To guarantee reliable synchronization detection, maximums values are multiplied by `'resolution_threshold'` variable. This ensures that only 'synchronization˙level' is far greater than the rest of maximums is able to meet the condition below. If the condition is met location of the synchronization symbol in the cross-correlation vector is defined. Because this vector and buffer have different sizes, the location of the synchronization symbol in the buffer must be defined.

Nevertheless, it is unknown if the whole synchronization symbol or just a portion of it was allocated in the buffer when cross correlation operation started. For successful location of synchronization it is essential to have whole synchronization symbol stored in the buffer. This issue is solved by setting up proper size of the buffer, which was mentioned in the chapter 4.2. The size of the buffer is two times longer than length of the symbol. This ensure that entire synchronization symbol will be in the buffer eventually. For better understanding figure 4.3 is shown below. Figure describes loading procedure of the buffer, which is done from the end of the buffer. Two extremes are displayed in the figure 4.3. In the left side entire synchronization symbol is successfully loaded on the second iteration. Therefore location of this symbol will be detected correctly. However, in the right side just half of the synchronization symbol is loaded in the buffer during the second iteration. At this point correlation of the synchronization symbol could be identified incorrectly. For this reason, `'locate_synchronization'` function is checking the location of synchronization symbol in the buffer. If the value stored in `'correlation_location_of_synchronization'` is higher than half of the buffer length, this means that incomplete symbol was loaded into to buffer. Thus, `'locate_synchronization'` will evaluate synchronization as not found. Therefore, main program routine will load new data to the buffer. This results in the shift of synchronization symbol. In this iteration whole synchronization symbol is stored in the buffer and its value of `'correlation_location_of_synchronization'` is lower than the half of the buffer length. When this condition is fulfilled, synchronization is successfully located, consequently "found" variable is set to 1 causing termination of the function. In the next iteration, the main program will not call `'locate_synchronization'`, again since the trigger condition is not met (found must be equal to 0). Instead, demodulation of data is done by the function `'dem_chirp'`.

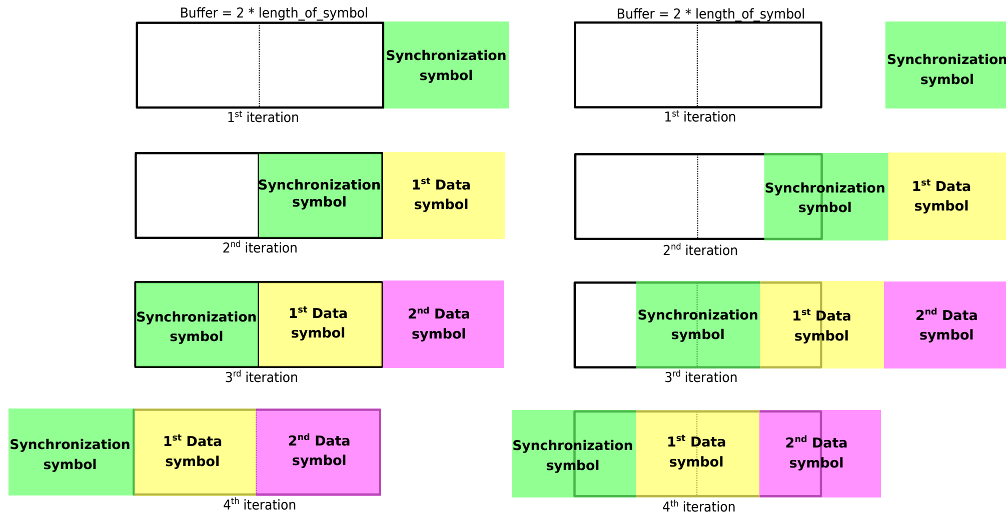


Figure 4.3: Buffer data loading process.

4.3.2 Dem Chirp Function

As once mentioned, CSS has great cross-correlation properties. Resulting very low relation among modulation symbols. This uniqueness of each symbol is responsible for low probability of detecting another symbol. Consequently robustness of communication system is increased. To be able to measure value of correlation between multiple modulation symbols correlation coefficients table 4.1 was introduced. Table indicates correlation values among modulation symbols for 8-ary CSS. Each column and row represent single modulation symbol, where correlation among single modulation symbol and the rest of them is calculated for each row. The main diagonal always has a value of 1, this means that symbols were auto-correlated. The second value shows correlation between first and the second modulation symbol and so on. From the results can be seen that demodulation probability of modulation symbol substitution with another one is very low. However, the modulation symbols neighbor to main diagonal are the most critical ones because of their higher correlation values compare to the rest. In chapter 3.7 assignation by Gray code was mentioned. Since table of correlation coefficients shows level of mutual correlation among modulation symbols. It is great tool for guiding assignation order for Gray code or any other encoding algorithm. Main reason to apply these algorithms is to lower probability of error and facilitate error correction.

When synchronization is successfully localized data demodulation can begin. Demodulation process is done by `'dem_chirp'` function which returns modulation state. This function calculates correlation which determines the similarity of the received signal with the modulation symbols and termination symbol. Since correlation procedure demands the most of computational power lots of effort is put in to reduce it. Value of the modulation state depends on the input parameters: `'dataIn'`, `'M'`, `'xcorr_demod'`, `'Modsymbols'`, `'termination_symbol'`, `'synchronization_location'`. Signal stored

1	0.0192	6.66e-08	1.94e-06	3.1e-07	7.5e-08	1.73e-07	2.50e-07
0.0192	1	-0.0384	-1.00e-07	0.0192	3.36e-08	1.56e-6	3.88e-07
6.66e-08	-0.0384	1	0.0192	-1.09e-07	1.53e-06	1.41e-05	3.99e-07
1.94e-08	-1.00e-07	0.0192	1	-0.0384	-5.28e-07	0.0192	1.35e-07
3.16e-07	0.0192	-1.09e-05	-0.0384	1	0.0192	-5.18e-07	2.00e-06
7.5e-08	3.33-08	1.53e-06	-5.58e-07	0.0192	1	-0.0384	-1.36-08
1.73-07	1.56e-06	1.41e-07	0.0192	-5.18e-07	-0.0384	1	0.0192
2.5e-07	3.88e-07	3.99e-07	1.35e-07	2.00-06	-1.36-08	0.0192	1

Table 4.1: Correlation coefficients for 8-ary CSS, symbols created by a symbol period splitting technique.

in the buffer refers to variable `'dataIn'`, `'M'` represents number of modulation states and `'xcorr_dem'` parameter performs setup of the `'dsp.Crosscorrelator'` for `'dem_chirp'` function. Synchronization location defines position of the synchronization symbol. Rest of the parameters are modulation symbols which are utilized for correlation between them and data stored in the buffer.

To lower computational load from `'dem_chirp'` function inputted data stored in the buffer are already normalized. Whenever `'dem_chirp'` is called primary, correlation between signal in the buffer (`'dataIn'`) and matrix of modulation symbols (`'Modsymbols'`) is done. Because each column of `'Modsymbols'` represents single modulation symbol computational difficulty increases with higher value of parameter `'M'`. After this process is completed position and value of correlation maximum for each column is found. Since all symbols have same length and location of synchronization is known maximum of correlation can be searched just on small area of buffer because, it is certain that next symbol will be near to location in the buffer as synchronization previously was. This eases computational load.

A vector with length equal to number of modulation states is stored in `'PomMax'`. In each column maximum correlation value for each modulation symbol is stored. In code below max command returns position of column(variable `'S'`) together with the highest value (`'AMP'`) in the `'PomMax'` vector.

```
PomXcorr = abs(step(xcorr_demod, dataIn, Modsymbols));
[PomMax,~]= max(PomXcorr((synchronizatin_location - 50):
(synchronizatin_location + 50), :));
```

Afterwards, similar procedure is done between `'dataIn'` and termination symbol to find out their correlation maximum. In the end variable `'AMP'` is compared with the result of correlation between `'dataIn'` and termination symbol multiplied by resolution threshold. This threshold is constant number (of value 3) which ensures that only symbol with very high correlation value will be returned by a function. If threshold condition is

fulfilled the position of the column to which 'AMP' variable belongs will be assigned to the variable state and returned as an output of the 'dem_chirp' function. When condition is not met two possible outcomes can occur. The first is that termination symbol was demodulated resulting end of the function. The second option is that level of 'AMP' variable was not significantly higher than the rest of modulation symbols. Possible cause of this event cold be dropout or interference thus function evaluates symbol as misinterpreted thus, value 0 will be written to variable state. In this situation 'dem_chirp' function returns 0 which will be later evaluated by symbol mapping function as incorrect input. This precaution helps prevent dropouts and interference mentioned above.

```
%AMP = max amplitude, S = column location of AMP
[AMP,S] = max(PomMax);
if AMP >= resolution_threshold * max_termination_symbol
     %modulation symbols sorting, symbol is determinated by location
     %of the maximum AMP
     if (S >= 1) && (S <= M))
       State = S;
end
else
     %termination symbol must be very clear
     if max_termination_symbol >= 20
     State = -1;
     else
     %HOLD => demodalator did not recognise any symbol
       State = 0;
     end
end
```

Since demodulation duration is essential parameter in the receiver, measurement of predecessor's code and optimized version introduced in this thesis was done. Both measurements were done on same computer(24 GB RAM & Intel©Xenon 2.5 GHz) with identical parameters: $f_{high} = 3000Hz$, $f_{down} = 300Hz$, T $= 1s$, F$s = 8000Hz$ for multiple M-arry CSS which are shown in table 4.5. As seen in the table below major improvement was accomplished. The last row is division between original and optimized is made to indicate how many times faster upgraded version is.

Modulation states	8	16	32	64
Duration of original version [ms]	682	1460	3000	6100
Duration of optimized version [ms]	56	80	128	223
Improvement over original [-]	12×	18×	23×	27×

Table 4.2: Demodulation duration before and after optimization

4.3.2.1 Decimation

Significant progress in reducing computational difficulty was made by optimizing predecessor‘s program. However, desire to diminish computational difficulty even more, so program can run in the real time on devices less powerfull than testing computer (24 GB RAM & Intel©Xenon 2.5 GHz). Considering that, cross-correlation computes each sample of modulation symbol with each sample of the data stored in the buffer, the idea of decimating cross-correlation process to reduce computational difficulty was brought. Before implementing decimation method, correlation coefficients of decimated modulation symbols are calculated. As a result cross-correlation properties of CSS are preserved, even when symbols are decimated. Tables 4.3 and 4.4 displays these coefficients for decimation factor 3 and 5. Although, values of the correlation coefficients are higher than for non decimated version of the 8-ary CSS, which was expected. Taking this factor into the consideration, decimated factors till factor 5 are suitable for communication system. Higher decimation factors (e.g. 7) are unsuitable for communication system, because of their high correlation coefficients values (up to 0.17).

1	0.0812	-0.0017	0.0492	0.0442	0.0199	0.0166	0.0267
0.0812	1	-0.0105	0.0333	0.0314	-0.0114	0.0222	0.0367
-0.0017	-0.0105	1	0.0585	-0.0101	0.0448	0.0179	-0.0130
0.0492	0.0333	0.0585	1	0.0853	-5.28e-04	0.0291	0.0303
0.0442	0.0314	-0.0101	0.0853	1	-0.0201	0.0219	0.0315
0.0199	-0.0114	0.0448	-5.58e-04	-0.0201	1	-0.0853	0.0067
0.0166	0.0222	0.0179	0.0291	0.0219	0.0853	1	0.0593
0.0267	0.0367	-0.0130	0.0303	0.0315	0.0067	0.0593	1

Table 4.3: Correlation coefficients for 8-ary CSS decimated by factor 3, symbols created by and symbol period splitting technique.

1	0.0345	-0.0129	-0.0102	0.0311	0.0062	-0.0047	0.0569
0.0345	1	0.0036	0.0387	0.0115	-0.0012	0.0444	-0.0150
-0.0129	0.0036	1	0.0249	0.0051	0.0463	-0.0120	-0.0034
-0.0102	0.0387	0.0249	1	0.0629	-0.0048	6.41e-04	0.0408
0.0311	0.0115	0.0051	0.0629	1	0.0141	0.0475	0.0039
0.0062	-0.0012	0.0463	-0.0048	-0.0141	1	0.0167	0.0339
-0.0047	0.0444	-0.0120	6.41e-04	0.0475	0.0167	1	0.0237
0.0569	-0.0150	-0.0034	0.0039	0.0039	0.0339	0.0237	1

Table 4.4: Correlation coefficients for 8-ary CSS decimated by factor 5, symbols created by a symbol period splitting technique.

Decimation technique ensures that, cross-correlating is calculated between every second, third or any n-th sample of the data. This feature, reduces computation time of the cross-correlation considerably. In the table 4.5 comparison of demodulation duration between decimated and non decimated versions is done.

Modulation states	8	16	32	64
Duration of original version [ms]	682	1460	3000	6100
Duration of optimized version [ms]	56	80	128	223
Duration of optimized version decimated by factor 2 [ms]	26	37	60	101
Duration of optimized version decimated by factor 3 [ms]	20	32	52	92
Duration of optimized version decimated by factor 5 [ms]	12	18	26	44
Improvement over original for decimation factor 5 [-]	56×	81×	115×	138×

Table 4.5: Demodulation duration comparison between decimated and non decimated versions, symbols created by a symbol period splitting technique.

The results show significant decrease of the demodulation process. However, a disadvantage of this improvement is a higher bit error rate of decimated versions over non decimated ones. Results of BER simulation for 8-ary CSS and 32-ary CSS are displayed below in figures 4.4 and 4.5. According to the simulation, BER performance of decimated CSS is worse than non decimated ones, as expected. Nevertheless, simulation shows that decimated modulation could be used for application, which could tolerate slightly higher BER.

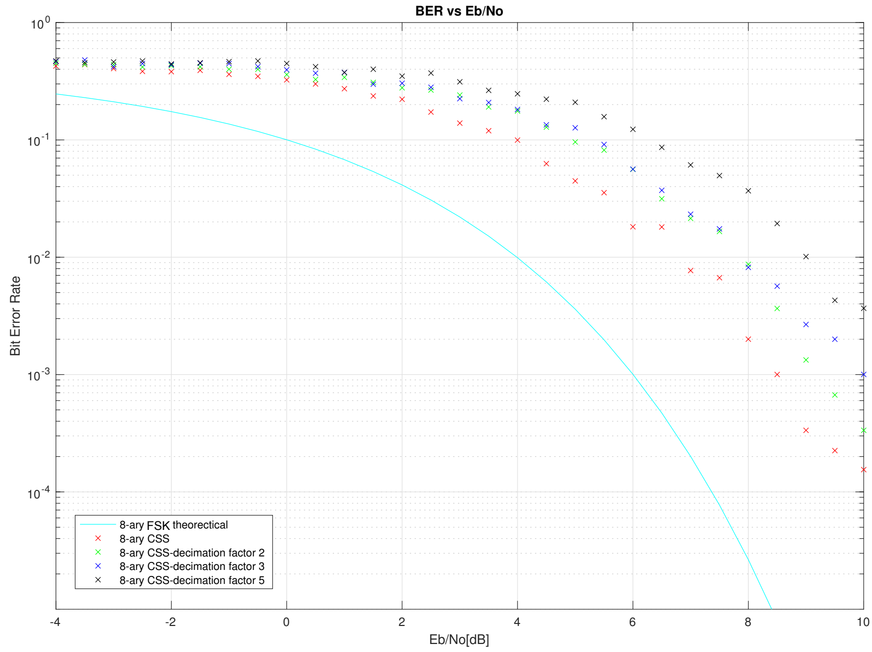


Figure 4.4: BER simulation for 8-ary CSS and decimated versions, symbols created by a symbol period splitting technique.

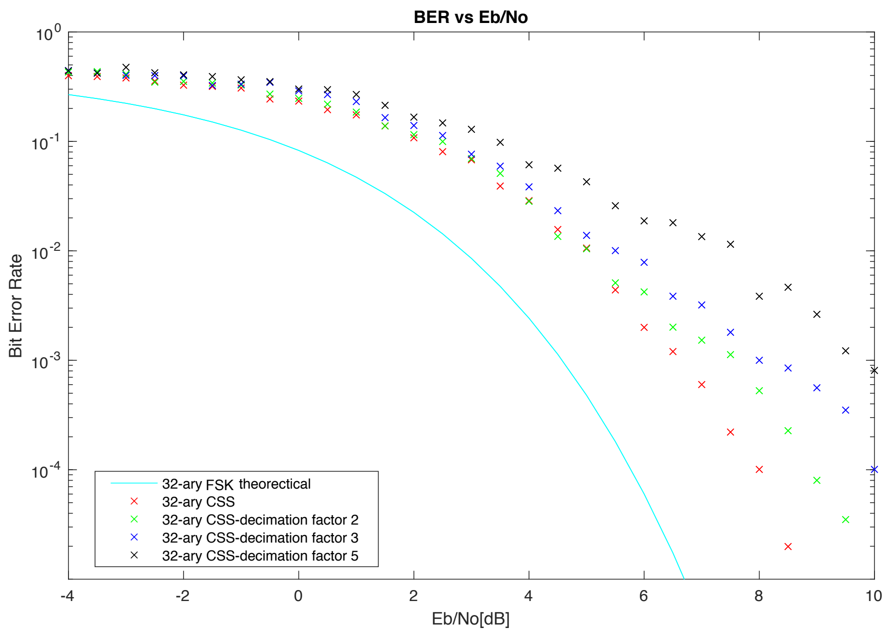


Figure 4.5: BER simulation for 32-ary CSS and decimated versions, symbols created by a symbol period splitting technique.

Implementation of the decimation procedure is made with consideration for simple and easily change of the decimation parameter. Therefore, `'decimation_factor'` was added as global variable. This solution allows easy access for setting up decimation parameter. Taking into account this modification, `'decimation_factor'` is added in as input parameter for functions `'dem_chirp'` and `locate_synchronization`. Value stored in the texttt'decimation˙factor' relates to, which n-th sample of the data will be calculated in the cross-correlation operation. For instance texttt'decimation˙factor' = 2 meaning that every second data sample will be cross-correlated. For non-decimated demodulation, texttt'decimation˙factor' must be equal to 1. Another usage of `'decimation_factor'` is shown bellow. The first line decimates modulation symbol by factor stored in the `'decimation_factor'`. The second row decimates data loaded into the buffer.

```
Modsymbols = Modsymbols(1:decimation_factor:end,:);
dataIn = dataIn(1:decimation_factor:end);
```

4.4 FrFT dem Function

Until now only demodulation method of the receiver was based on cross-correlation and its modifications. Function `'dem_FrFT'` is another approach to the demodulation process. This function is based on the fractional Fourier transformation (FrFT). FrFT is a linear transformation, which is a generalization of the Fourier transformation. The FrFT implements the so-called orderparameter α which acts as a ordinary Fourier transform operator. That is to say, the α-th order fractional Fourier transform represents the α-th power of the ordinary Fourier transform operator. For $\alpha = 0$, there will be no change after applying fractional Fourier transform. When $\alpha = \frac{\pi}{2}$ Fourier transform will be obtained. If α belongs into the interval $0 < \alpha < \frac{\pi}{2}$, FrFT transforms (rotates) a signal, either in the time domain or frequency domain into the domain between time and frequency: the time-frequency domain. [14] [15] The time-frequency domain was already mentioned in figure 2.3. In this picture (2.3), modulation symbol is represented by an unique combination of the two chirp rates. Since fractional Fourier transform has an ability to rotate in the time-frequency domain, a specific chirp rate can be detected. In other words, for each chirp rate a certain value of the α parameter causes, that FrFT will return sequence with strong maximum. This maximum indicates a particular chirp rate. Taking this into consideration, a demodulation of the signal is possible. Nevertheless, two chirp rates are necessary to detect a modulation symbol, thus only modulation symbols created by the symbol period splitting method 3.6 can be successfully demodulated.

For demodulation on the FrFT basis three function are used: `'locate_aplha_values'`, `'dem_FrFT'` and `'frft'`, which was taken from [16]. The fist step of the demodulation is to create vector of α parameter values: `'aplha_values'`. For any of these values, the fractional Fourier transform returns a sequence, with strong maximum, which represents

detected chirp rate. Since a single modulation symbol is represented by a combination of the two chirp rates, that are changed in the middle of the symbol duration. To calculate `'aplha_values'` only one half of the symbol can be used. Reason for this is that, for the linear chirp, the alpha values of the second half symbol duration are identical, but in the opposite order than the alpha values for the first half. The vector of `'aplha_values'` remains the same for whole demodulation procedure, so it is calculated only ones. The reason for separating this function from the rest is a relatively high computational difficulty, that would slow down the main demodulation function `'dem_FrFT'`. After `'aplha_values'` are created, function texttt'dem˙FrFT' can be called by the main program. Function `'dem_FrFT'` calls its subfunction `'frft'` which, performs the fractional Fourier transformation of given α values stored in the `'aplha_values'` vector. Then a maximum of FrFT output for each α value is stored in variable `'first_half_maximum'`. From this variable another we are able to locate position of maximum, which refers to discovered chirp rate of the first half.

```
for i = 1:length(alpha_values)
    first_half_of_max_vector = alpha_values(i);
    Frft_first_half = frft(data_first_half,alpha_vec
    (first_half_of_max_vector));
      [first_half_maximum(i),location(i)]= max(abs(Frft_first_half));
end
[first_half_mod_symbol_max,F]= max(first_half_maximum);
[second_half_mod_symbol_max,S]= max(second_half_maximum);
```

The identical procedure is done for the second half of the loaded data. In the end, location of discovered maximums `'F'` and `'S'` are compared. If the values are the same, this means symbol was successfully detected. Therefore, detected symbol is returned as output of the `'dem_FrFT'` function. If a values of the both symbol halves are not identical, `'dem_FrFT'` evaluate modulation symbol as unrecognized. To find out how big improvement was made by an implementation of the `'dem_FrFT'` function, a measurement with the same conditions as for decimation was done. Results are displayed in the table4.6. The progress was made, however not so significant as in decimation method case. The main issue is that function `'frft'` is not optimized for speed.

After determination of the `'dem_FrFT'` speed , testing of BER performance was done. The results of the simulation are shown in figure 4.6. As seen, BER outcome for 32-ary CSS is acceptable. However, result for 8-ary method is not satisfying. Because this implementation is new, further analysis is necessary to verify or disprove this result.

Modulation states	8	16	32	64
Duration of original version [ms]	682	1460	3000	6100
Duration of optimized FrFT version [ms]	261	520	1030	2012
Improvement over original [-]	2.6×	2.8×	2.9×	3×

Table 4.6: Demodulation duration before and after FrFT optimization, symbols created by a symbol period splitting technique.

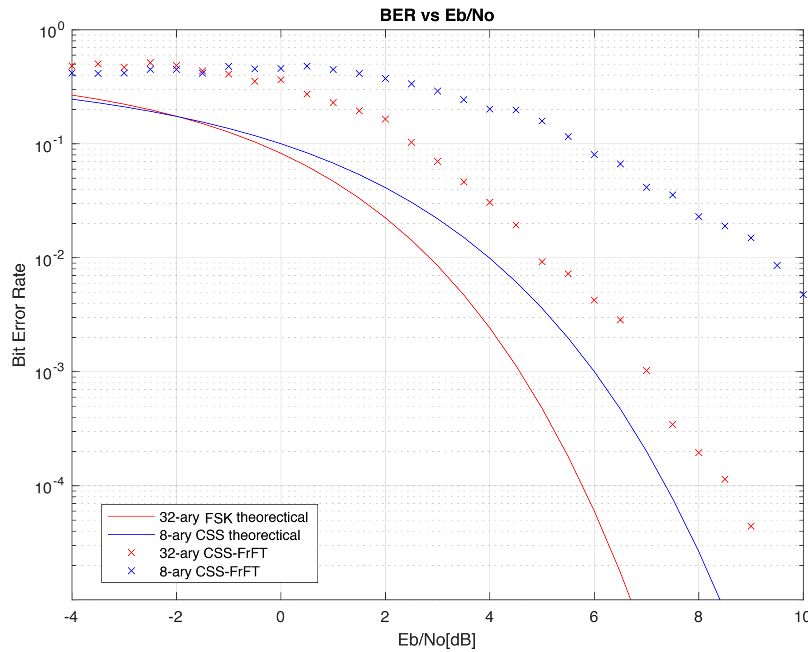


Figure 4.6: BER simulation for 8-ary and 32-ary CSS-FrFT versions, symbols created by a symbol period splitting technique.

4.5 Translation of Received String

The final section of the receiver is to translate received string of modulation states to message sent by transmitter. However, multiple operations need to be processed to achieve this goal. Firstly the `'symbol mapping'` function with parameter `'symbol'` is called to transform modulation states into modulation symbols. When `'symbol'` parameter is chosen the second mode of this function is accessed. In this mode function translates modulation state to modulation symbol according to the table, which was generated in the transmitter. Because a simple symbol assignation was used in the transmitter. The same method is utilized in the receiver to do reverse operation. Therefore, `'symbol mapping'` function with parameter symbol converts received string of the modulation states to modulation symbols. Afterwards, decadic expression of the modulation symbol

is converted to binary string. Code section in charge of this procedure is shown below.

```
for i = 1:length(recieved_string_of_state)
  recieved_symbol = symbol_mapping(recieved_string_of_state(1, i),
  'symbol', M);
  bin_string_of_symbol = dec2bin(recieved_symbol, number_of_bits);
    for j = 1:number_of_bits
    %if this is the first tranlated state
      ifrecieved_string == -1
        recieved_string = str2num(bin_string_of_symbol(j));
        else
          recieved_string = cat(2, recieved_string,
          str2num(bin_string_of_symbol(j)));
        end
    end
end
```

For correct translation, it is necessary to remove stuffing from captured binary string. The 'message alignment block manages this procedure when remove stuff parameter is chosen.

```
string_without_alignment = message_alignment_block
   (recieved_string, number_of_bits, 'remove_stuff');
```

Message alignment process goal is to locate position where message ended in the data string. This is achieved by doing exclusive disjunction operation between string termination and received string (message). In each cycle, the message is XORed with string termination and checked if a sum of this operation is equal to 0. If not `'message'` is shifted by one bit and XOR operation is repeated. Since string termination stores 16 zeroes sum result equal to 0 occurs only when message stores 16 zeroes as well. When this situation happens `'end_of_termination_string'` is successfully located. Afterwards, `'message alignment_block'` calculates how many bits were added as stuffing. Lastly, `'the_separator_of_data string'`, `'string_terminator'` and
`'description_number_of_additional_bits'` which were added to data string as stuffing are removed. The part of the message alignment code process is shown below.

```
for i = 1:length(message)
  if location_of_end_found == 0
    if (i + (length(string_termination) - 1)) <= length(message)
      difference = xor(message(i:(i + length(string_termination)-1)),
        if sum(difference) == 0
           location_of_end_found = i;
        end
```

```
            end
        end
    end
```

The final procedure to obtain text message is to translate bitstream without stuffing given by message alignment block. Message translation function manages this process. In code beneath according to the length of the ASCII character, 8 bits from bit stream are loaded. These 8 bits are converted to a decimal number. Afterward, the decimal number is transformed to Unicode representation (ASCII character) by the ISO-8859-2 encoding scheme. ASCII character is then concatenated to the message vector. This cycle repeats itself until the whole message is converted and the final dialog box with the translated message appears as the final result of the receiver.

```
while actual_bit < length(input_bit_stream)
    bin_string_ASCII = num2str(input_bit_stream(actual_bit:
    (actual_bit + (ASCII_character_length - 1))));
    dec_number_of_symbol = bin2dec(bin_string_ASCII);
    ASCII_char_symbol=native2unicode(dec_number_of_symbol,'ISO-8859-2');
    message = strcat(1, message, ASCII_char_symbol);
    actual_bit = actual_bit + ASCII_character_length;);
end
```

5

Conclusion

This thesis deals with a software implementation of the communication system. The goal was to create low speed transmission system with an ability to choose between energy or spectral efficiency over a group of the high order modulation. Communication system of these properties can be utilized in the technology of the picosatellites, IOT devices or any other applications, which is working with limited power resources. To meet criteria for a high power efficiency, a proper modulation method must be chosen. Methods with constant modulation envelope allows to use non-linear RF amplifiers, which are very efficient. Chirp spread spectrum modulation fulfills these requirements. This modulation method has good cross-correlation properties among modulation symbols. Another useful feature of the CSS is resistance to interference and jamming. Taking all these factors into a consideration, CSS was selected as the modulation method for communication system.

This paper follows up on the previous thesis High order chirp modulations [4]. The author created a software prototype of the modulator and demodulator using M-ary chip modulation. Because predecessor‘s solution involved many flaws, a optimization of the program was necessary. The biggest disadvantages of the original version were a low level of the parametrization and very high computational difficulty, especially for demodulation process. This leaded to inability to demodulate in the real-time, because processing times for a demodulation of the single symbol were units of second. Therefore, first primary task of this thesis was to implement communication system with M-ary chirp modulation working in the real-time. The second aim was to design and test suitable demodulation methods with a consideration of a computational difficulty for a PC.

In the first chapter, the CSS was introduced, as well the frequency band splitting and symbol period splitting methods, which are techniques for creating the modulation symbols. The second chapter was devoted to the realization of the main program `Transmitter_main`. To modify the properties of the transmitting signal as easily as possible, global variables were implemented. Therefore, transmitter features such as order of modulation, sample frequency, method of creating modulation symbols and many others can be controlled by managing several global variables. Transmitter contains various blocks, which are represented as functions. Input parameters of these function are mostly

global variables, thus a parameterization is improved.

The following chapter describes implementation of the main program `Receiver_main`. To improve parameterization, the global variables and the functions are applied in the same manner as for `Transmitter_main`. In the receiver the most of the computational power is demanded by the demodulation process. In the original version cross-correlation was the primary method for data demodulation. To reduce computational difficulty, multiple procedures were implemented. An inappropriate switch structure for sorting modulation symbols to data vector was used by the predecessor. With increasing order of the modulation, this solution was unacceptable. In optimized solution a single matrix of all modulation symbols serve for symbol sorting instead of switch structure. Another improvement was done by calculating cross-correlation just in the part of the buffer, where was symbol expected. In the end, a measurement between original and optimized versions was done. For the same input parameters, a demodulation of optimized version was on average 20 times faster than original version. Far lower duration times were achieved by a decimation technique, which was newly implemented. When 32-ary modulation and decimation factor equal to 5 were selected, decimated version was 115 times faster than original. However, when decimated versions were tested for a bit error rate, it was discovered that, with higher decimation factor, bit error rate increases. This was expected since decimation method reduces number of samples for cross-correlation computation. The results of the bit error rate show an increase in energy efficiency, as the number of modulation states grows.

The last demodulation method realized in this thesis, is based on the fractional Fourier transformation. An ability to distinguish an unique combination of two different chirp rates, which represents modulation symbol is indeed interesting. A functional demodulation prototype was created. A measurement proved that a demodulator based on FrFT is approximately three times faster than original version. The primary reason why demodulation times based on FrFT are not as impressive as other solutions is manly because function for computing FrFT is not optimized for speed. If this issue can be overcomed, a demodulation based on FrFT could found a use, since they have a potential, to gain better performance with applied time-frequency domain filtering as is introduce in [17] In current state of this thesis a bit error rates results shows, not so good properties as expected. Therefore, FrFT needs to be analyzed more profoundly.

Demodulation methods implemented in this thesis are able to decrease computational difficulty significantly. A drawback of these methods with decimation are slightly higher bit error rates in comparison to the non-decimated methods. M-ary CSS, they are a perspective alternative to the conventional modulation methods.

Bibliography

[1] VEŘTÁT, Ivo. *Efektivní komunikační systém pikosatelitů.* Plzeň, 2011. disertační práce (Ph.D.). ZÁPADOČESKÁ UNIVERZITA V PLZNI. Fakulta elektrotechnická

[2] S. Hengstler, D. P. Kasilingam and A. H. Costa. *A Novel Chirp Modulation Spread Spectrum Technique for Multiple Access.* IEEE Operation Center, c2002. ISBN 0-7803-7627-7.

[3] DIXON, Robert C. *Spread spectrum systems: with commercial applications. 3rd ed.* New York: Wiley, c1994. ISBN 0471593427.

[4] HOSEK,Jan. *Vícestavové rozmítané modulace.* Plzeň, 2016. diplomová práce. ZÁPADOČESKÁ UNIVERZITA V PLZNI. Fakulta elektrotechnická.

[5] J.Huang,Ch. He and Q. Zhang. *M-ary Chirp Spread Spectrum Modulationfor Underwater Acoustic Communication.* IEEE, c2004. ISBN 0-7803-9311-2.

[6] EL-KHAMY, S.E., S.E. SHAABAN a E.A. TABET. *Efficient multiple-access communications using multi-user chirp modulation signals. In: Proceedings of ISSSTA'95 International Symposium on Spread Spectrum Techniques and Applications.*[online]. IEEE, 1996, s. 1209-1213 [Retrieved 2018-03-05]. DOI: 10.1109/ISSSTA.1996.563498. ISBN 0-7803-3567-8. Available from : http://ieeexplore.ieee.org/document/563498/

[7] The MathWorks, Inc. *unicode2native.* [online]. United States:The MathWorks,Inc.©1994-2016 [Retrieved 2018-04-25]. Available from:https://www.mathworks.com/help/matlab/ref/unicode2native.html

[8] The MathWorks, Inc. *unicode2native.* [online]. United States:The MathWorks,Inc.©1994-2016 [Retrieved 2018-04-25]. Available from:https://www.mathworks.com/help/matlab/ref/dec2bin.html

[9] The MathWorks, Inc. *dsp.AudioPlayer System object.* [online]. United States:The MathWorks,Inc.©1994-2016 [Retrieved 2018-03-27]. Available from:https://www.mathworks.com/help/dsp/ref/dsp.audioplayer-system-object.html

[10] The MathWorks, Inc. *dsp.AudioRecorder System object.* [online]. United States:The MathWorks,Inc.©1994-2016 [Retrieved 2018-04-21]. Available from:https://www.mathworks.com/help/dsp/ref/dsp.audiorecorder-system-object.html

[11] The MathWorks, Inc. *dsp.Buffer System object.* [online]. United States:The MathWorks,Inc.©1994-2016 [Retrieved 2018-04-22]. Available from:https://www.mathworks.com/help/dsp/ref/dsp.buffer-system-object.html

[12] The MathWorks, Inc. *dsp.Crosscorrelator System object.* [online]. United States:The MathWorks,Inc.©1994-2016 [Retrieved 2018-05-07]. Available from:https://www.mathworks.com/help/dsp/ref/dsp.crosscorrelator-system-object.html?s˙tid=doc˙ta

[13] The MathWorks, Inc. *corrcoef.* [online]. United States:The MathWorks,Inc.©1994-2016 [Retrieved 2018-05-09]. Available fhttps://www.mathworks.com/help/matlab/ref/corrcoef.html

[14] Sejdic, Ervin & Djurovic, Igor & Stankovic. *Fractional Fourier transform as a signal processing tool: An overview of recent developments.* [online]. [Retrieved 2018-15-05]. DOI: 10.1016/j.sigpro.2010.10.008. ISBN 10.1016/j.sigpro.2010.10.008. Available from : http://linkinghub.elsevier.com/retrieve/pii/S0165168410003956

[15] ALMEIDA, L.B. *The fractional Fourier transform and time-frequency representations.* [online]. IEEE, 1996 42(11), 3084-3091[Retrieved 2018-05-19]. DOI: 10.1109/78.330368. ISBN 10.1109/78.330368. Available from : http://ieeexplore.ieee.org/document/330368/

[16] meng *frft.m.* [online]. United States: meng©2013 [Retrieved 2018-05-20]. Available from : https://www.mathworks.com/matlabcentral/fileexchange/41351-frft-m?focused=3784947&tab=function

[17] Wikipedia: the free encyclopedia *Fractional Fourier transform as a signal processing tool.* [online]. San Francisco (CA): Wikimedia Foundation[Retrieved 2018-05-30]. https://en.wikipedia.org/wiki/Fractional˙Fourier˙transform

.

Appendix A

Used scripts and source code

A.1 Main program Transmitter_main.m

```
1  close all;
2  clear all;
3  %true= symbols will be saved into waw. file
4  %false= transmitter will send all symbols
5  test = true;
6
7  %*******global variables *******
8  frequency_splitting=1;
9  time_splitting=2;
10
11 %setings of modulation parameters
12 T = 1;        %time interval for symbol in seconds
13 fhigh = 3000; %settings for bandwidth of symbols
14 fdown = 300;
15 M = 16;       %number of modulation states
16 Fs = 8000;    %symbol sampling rate
17 type = time_splitting; % choosen type of the modulation
18 duration_of_data_frame = 100; %how many symbols will be in 1 frame
19
20 %*****end of global parameters********
21
22 length_of_symbol = T * Fs;
23
24 number_of_bits = log2(M);
25
26 size_of_data_frame = duration_of_data_frame * number_of_bits / T;
27
28 %align size of data frame so synchronization doesnt fluctuate
29 size_of_data_frame = size_of_data_frame - mod(size_of_data_frame, number_of_bits);
30
31 message = enter_message();
32
33 %creation of  modulation symbol .wav file
34 create_mod_symbol(T, fhigh, fdown, M, Fs, type);
35
36 %load of synchronization & termination symbols
37 synchro_symbol = wavread('Synchonization_symbol.wav');
38
39 termination_symbol = wavread('Termination_symbol.wav');
40
41 %reallocation of memory for loading symbols
42 AFP=zeros(length(wavread('Symbol_1.wav')), 1);
43
44 %allocation of memory needed for symbols
45 mod_symbols=zeros(length_of_symbol,M);
46
47 %loading of modulation symbols
48 for i=1:M
49     %concatination of Symbol_+(i)+.wav
```

```
50      filename = strcat('Symbol_', num2str(i), '.wav');
51      %load field of modulation symbols
52      mod_symbols(:,i) = wavread(filename);
53  end
54
55  %variable which allows play audio file with defined parametrs for playing
56  AP=dsp.AudioPlayer('SampleRate',Fs,'QueueDuration',T);
57
58  %order of character in message which have to be sent right now
59  current_character = 1;
60
61  %for synchonization purpose
62  synchronization_counter = 1;
63  %synchronization not sent yet
64  s = 0;
65  %test of vector in which all transmitted symbols are compound together
66  testing_output_of_modulator = 0;
67
68  %run message_alignment_block to align message
69  message_alignment=message_alignment_block(message, number_of_bits,'align');
70
71  %possible for later use of interleaving or message coding
72  string_sent = message_alignment;
73
74  %creating variables for end of trannsmision
75  end_of_transmission = 0;
76  terminate_transmission = 0;
77
78  %till all character of message are sent
79  while end_of_transmission ~= 1
80      %if synchonization was not sent
81      if s == 0
82          %loading of syncho symbol into AFP variable
83          AFP = synchro_symbol;
84          %synchonization will be send after this
85          s = 1;
86
87      %if synchronization was sent
88      else
89          %till termanation symobol is not send
90          if terminate_transmission ~= 1
91              %load string from current_character + number of bits
92              symbol_bin_to_string = num2str(string_sent(current_character:(current_character + (number_of_bits - 1))));
93              %decadic number of binary string which represents symbol number
94              number_of_symbol = bin2dec(symbol_bin_to_string);
95              %run funcion symbol mapping
96              %find out character number of state corresponding to
97              %symbol which is about to be send
98              %state or symbol parameter
99              number_of_state = symbol_mapping(number_of_symbol, 'state', M)
100             %load number of state for sending
101             AFP = mod_symbols(:, number_of_state);
102                %if size of frame is exceeded => necessary to send
103                %synchronization again for a new frame
104                %synchronization_counter if more than 1 synchornization
105                %is needed to be send
106             if mod(current_character,(synchronization_counter*size_of_data_frame+1)-synchronization_counter*number_of_bits
107                 %if s = 0 next symbol loaded into AFP will be
108                 %Synchronization_symbol
109                 s = 0;
110              end
111                 %testing is current_character is last character of message
112             if (current_character + number_of_bits) <= length(string_sent)
113                 %next current_character for upcoming iteration
114                 current_character = (current_character + number_of_bits);
115                 %synchro fix necessary so that modulo in row 105 works fine
116                 %If current_char =1 =>fix(1/number_of_bits) = 0 which is
117                 %bad... to correct this >>
118                 %current_character_synchro actually compute current_char
119                 %after it was increased by next iteration(1+number_of_bits)
120                 %meaning for first time it does not calculate the first
121                 %current_character at all. But it starts from the second
122                 %however this doesnt bother us because it is corrected
```

```
123                 %in the counter variable (resets one symbol earlier)
124                 current_character_synchro_fix = fix(current_character/number_of_bits);
125                 %counter counts from 1 to duration_of_data_frame
126                 %then it resets to 0 and start again
127                 counter = mod(current_character_synchro_fix,duration_of_data_frame);
128                 %testing when counter resets => new frame => new synchro
129                 %symbol = counter of synchronization +1
130                if counter == 0
131                    synchronization_counter = synchronization_counter+1;
132                end
133             else
134                 terminate_transmission = 1;
135             end
136         else
137             %load terminatin symbol into AFP
138             AFP = termination_symbol;
139             end_of_transmission = 1;
140         end
141     end
142     %if is wanted created only 1 .wav file merged with all transmitted
143     %symbols for testing purposes
144     if test == true
145         %add curently transmitted symbol to test vector
146         testing_output_of_modulator = cat (1, testing_output_of_modulator, AFP);
147     else
148         %play .wav file stored in AFP variable
149         step(AP, AFP);
150     end
151 end
152
153 %if test== true => .wav file merged with all trasmitted symbols will be
154 %created
155 if test == true;
156     %adding part of 0 for reason that audio file will be played without
157     %issues for example in Media Player
158     testing_output_of_modulator=cat(1, zeros(Fs, 1), testing_output_of_modulator);
159     %creatation of testing .wav file
160     wavwrite(testing_output_of_modulator, Fs, 'testing_output_of_modulator.wav');
161 end
162 %wait till symbols are played
163 pause(AP.QueueDuration);
164 %close(AFR)
165 release(AP);
```

A.2 Function enter_message.m

```
1  %FUNCTION FOR ENTERING MESSAGE
2  %function in which message is inputed and sended by a program.
3  %called by transmitter_main
4  function message = enter_message()
5      %block for inserting the message
6      notification = {'Enter message for sending:'};
7      dialog_name = 'Message';
8      number_of_lines = 1;        %how many lines will dialog have
9
10     input_message = inputdlg(notification, dialog_name, number_of_lines);% set up a matlab window
11
12     message_char = char(input_message); %converting => because inputdlg returns format cell which is non usable
13
14     %message lenght is mutlplied by 8 => because I need to send bits individually
15     %chars are bits
16     message = (zeros(1, length(message_char) * 8));
17
18     for i = 1:length(message_char)
19         % converting chars to number
20         ASCII_dec_msg = unicode2native(message_char(i), 'ISO-8859-2');
21         % converting numbers which I get to bit expression
22         ASCII_bin_string_msg = dec2bin(ASCII_dec_msg, 8); % necessary put 8 otherwise just 7 bits words can occur
23                                                            %=> variable length of words -> inappropriately
24         for j = 1:8
```

```
25              %dec2bin return binary but in string of chars => necessary convert to numbers
26              message (((i-1)*8) + j) = str2num(ASCII_bin_string_msg(j));
27          end
28      end
29
30  end
```

A.3 Function message_alignment_block.m

```
1    %FUNCTION FOR MESSAGE ALINGMENT
2   function resulting_message = message_alignment_block( message,number_of_bits, operation )
3   %text characters always 8 bits. But transmitted symbols are express by
4   % a number of states for example 5 are 6 bits =>transmited message could
5   %apper with lefover of bits (1,2 or more)
6   %=> necessary to solve stuffing for whole number of states
7
8       length_of_ASCII_character = 8;
9       separator_of_string_terminator = 1;
10      string_termination = zeros(1, 2 * length_of_ASCII_character);
11      description_number_of_additional_bits = zeros(1, number_of_bits);
12
13      %block for transmitter to align message
14      %compares 2 strings if they are indentical strcmp returns 1 (true)
15     if strcmp(operation, 'align') == 1
16
17          necessary_number_of_bits = length(message) + separator_of_string_terminator + length(string_termination) + length(
18          %remainer after division
19           number_of_additional_bits = mod(necessary_number_of_bits, number_of_bits);
20
21          if number_of_additional_bits ~= 0
22              %how much bits are needed for stuffing
23              number_of_missing_bits = number_of_bits - number_of_additional_bits;
24          else
25              number_of_missing_bits = 0;
26          end
27          %contver number of missing bits to binar with at least number of
28          % bits that is stored in number_of_bits
29          bin_number_of_missing_bits = dec2bin(number_of_missing_bits, number_of_bits);
30
31          %bin_number_of_missing_bits return format char => type conversion
32          %is needed
33          for i = 1:number_of_bits
34              description_number_of_additional_bits(1, i) = bin2dec(bin_number_of_missing_bits(i));
35          end
36          %concatenate all arrays with dimesnion of 2 => returns vector
37          % with 0 remainer after dividing by length of number_of_bits
38          resulting_message = cat(2, message, separator_of_string_terminator, ones(1, number_of_missing_bits), string_termin
39
40     %block for reciever to remove stuffing
41     %this part doest work properly ..
42     elseif strcmp(operation, 'remove_stuff') == 1
43
44          location_of_end_found = 0;
45
46          for i = 1:length(message)
47              if location_of_end_found == 0
48                  if (i + (length(string_termination) - 1)) <= length(message)
49                      difference = xor(message(i:(i + length(string_termination)-1)), string_termination);
50                      if sum(difference) == 0
51                          location_of_end_found = i;
52                      end
53                  end
54              end
55          end
56
57          end_of_termination_string = location_of_end_found + (length(string_termination) - 1);
58
59          stuff_str_descriptrion = num2str(message((end_of_termination_string + 1):(end_of_termination_string + length(descr
60          number_of_stuffed = bin2dec(stuff_str_descriptrion);
61
```

```
62          resulting_message = message(1:((location_of_end_found - 1) - number_of_stuffed - separator_of_string_terminator));
63      else
64          errordlg('Wrong input of operation!! Choose align or remove_stuff');
65      end
66  end
```

A.4 Function create_mod_symbol.m

```
1  %FUNCTION FOR CREATING MODULATION SYMBOLS
2  function  create_mod_symbol( T,fhigh,fdown,M,Fs,type )
3  %T - %time interval for symbol in seconds
4  %symbol - actually modulated symbol
5  %fhigh - upper limit frequency
6  %fdown - lower limit frequency
7  %M - number of modulation states
8  %Fs - symbol rate
9  %type - % choosen type of the modulation
10         %1 for a frequency split method
11         %2 for time spliting method
12
13  %for better synchronization "catching"
14  %synchro and terminal symbols have own frequency band
15  lower_range_of_control_zone = fhigh - ((fhigh - fdown) / 10);
16  upper_range_of_mod_symbols = lower_range_of_control_zone - ((fhigh - fdown)/10);
17
18     %creation of synchro and terminal symbols
19     if type == 1
20         Synchro_symbol = chirpM_mod_time(T,1,fhigh,lower_range_of_control_zone,2,Fs);
21         Termination_symbol = chirpM_mod_time(T,2,fhigh,lower_range_of_control_zone,2,Fs);
22     elseif type == 2
23         Synchro_symbol=chirpM_B_mod_time(T,1,fhigh,lower_range_of_control_zone,2,Fs);
24         Termination_symbol=chirpM_B_mod_time(T,2,fhigh,lower_range_of_control_zone,2,Fs);
25     end
26     %write to  wav.file
27     wavwrite(Termination_symbol,Fs,'Termination_symbol.wav');
28     wavwrite(Synchro_symbol,Fs,'Synchonization_symbol.wav');
29
30     %creation of modulation symbols in different frequency band
31     for i=1:M
32         if type == 1
33             y=chirpM_mod_time(T,i,upper_range_of_mod_symbols,fdown,M,Fs);
34         elseif type == 2
35             y=chirpM_B_mod_time(T,i,upper_range_of_mod_symbols,fdown,M,Fs);
36         end
37         %concation of strings
38         filename = strcat('Symbol_', num2str(i), '.wav');
39         %audiowrite(filename,y,Fs); %write to  wav.souboru
40         wavwrite(y,Fs,filename);
41     end
42
43  end
```

A.5 Function chirpM_B_mod_time.m

```
1  %FUNCTION FOR CREATING MODULATION SYMBOLS BY FREQUENCY BAND SPLITTING
2  function y=chirpM_B_mod_time(T,symbol,fhigh,fdown,M,Fs)
3  %T - time interval for symbol in seconds
4  %symbol - actually modulated symbol
5  %fhigh - upper limit frequency
6  %fdown - lower limit frequency
7  %M - number of modulation states
8  %Fs - symbol rate
9
10     %generation of time axis
11     %in the end T - (1/Fs) because I am generating from 0
12     %=> necessary to remove one sample
13     time=(0:1/Fs:T-(1/Fs));
```

```
14
15     %size of one sub band
16     subband_size = (fhigh-fdown)/(M/2);
17
18     chirprate = subband_size / T;
19
20     %for even =0
21     %for odd =1
22     switch mod(symbol,2)
23         case 0
24             y=sin(2*pi.*((fdown+(subband_size*floor...
25                 (symbol/2))).*time+((chirprate/2).*(time.^2))));
26         case 1
27             y=sin(2*pi.*((fhigh-(subband_size*floor...
28              ((M-symbol)/2))).*time + ((-chirprate/2).*(time.^2))));
29     end;
30
31 end
```

A.6 Function chirpM_mod_time.m

```
1 %FUNCTION FOR CREATING MODULATION SYMBOLS BY TIME DURATION SPLITTING
2 function y=chirpM_mod_time(T,symbol,fhigh,fdown,M,Fs)
3 %T - time interval for symbol in seconds
4 %symbol - actually modulated symbol
5 %fhigh - upper limit frequency
6 %fdown - lower limit frequency
7 %M - number of modulation states
8 %Fs - symbol rate
9
10     %geration to T/2 because vector will be used for each half of
11     %time interval
12     %generation of time axis
13     time=(0:1/Fs:(T/2)-(1/Fs));
14
15     %size of one sub band
16     subband_size = (fhigh-fdown)/((M/2)+1);
17
18     %alocating required memory
19     chirprate1=zeros(M);
20     chirprate2=zeros(M);
21     freq_middle=zeros(M);
22
23
24     for i=1:(M/2)
25         chirprate1(i)=i*subband_size/(0.5*T);
26         chirprate2(i)=((M/2)-i+1)*subband_size/(0.5*T);
27     end;
28     for i=(M/2)+1:M
29         chirprate1(i)=-chirprate1(i-(M/2));
30         chirprate2(i)=-chirprate2(i-(M/2));
31     end;
32     for i=1:(M/2)
33         %frequency middle is where chirp rate is changing
34         freq_middle(i) = fdown + i * subband_size;
35
36     end;
37     for i=(M/2)+1:M
38         freq_middle(i) = freq_middle((M)-i+1);
39     end;
40     switch fix((symbol-1)/(M/2))
41     %fix rounds down => first half of symbols are case 0
42     %second half is case 1 => division of inclination
43       case 0
44           y_first_half=sin(2*pi.*(fdown.*time+(chirprate1(symbol)/2).*...
45         time.^2));
46
47           %phase detection of last sample for appropriate bonding
48           last_sample_phase=sin(2*pi.*(fdown*(T/2)+(chirprate1(symbol)/2)*(T/2)^2));
49           y_second_half=sin(last_sample_phase + 2*pi.*((freq_middle(symbol)).*time+(chirprate2...
```

```
50          (symbol)/2).*time.^2));
51
52        case 1
53            y_first_half=sin(2*pi.*(fhigh.*time+(chirprate1(symbol)/2).*...
54          time.^2));
55
56            last_sample_phase=sin(2*pi.*(fhigh*(T/2)+(chirprate1(symbol)/2)*(T/2)^2));
57            y_second_half=sin(last_sample_phase + 2*pi.*((freq_middle(symbol)).*time+(chirprate2...
58          (symbol)/2).*time.^2));
59      end;
60
61      %merging vectors of first and second half together
62      y=cat(2, y_first_half, y_second_half);
63
64  end
```

A.7 Function symbol_mapping.m

```
1  %FUNCTION FOR SYMBOL MAPPING AND ASSIGNATION
2      %Table with translation of which nummber expression in bits
3      %means state or symbol
4      %later on this is used for tranlsation to ASCII character
5
6      function output = symbol_mapping(input, find_out, number_of_states)
7
8      output = -1;
9      %number_of_states+1 because i need one more symbol if demodulator
10     %did not recognised the symbol
11     table=zeros(1, number_of_states+1);
12
13     %here just assignation
14     %but Gray code assignation can be done as well for example
15     for i = 1:number_of_states
16         % to be able to transmitt terminator symbol 0
17         %creation of table
18         table (1, i) = i -1;
19     end
20
21     if strcmp(find_out, 'state') == 1
22         for i = 1:number_of_states
23             if table(1, i) == input
24                 output = i;
25             end
26         end
27
28         if output == -1;
29             errordlg('Problem with translation of the symbol');
30         end
31     elseif strcmp(find_out, 'symbol') == 1
32         % symbolize NaN
33         if input == 2 * number_of_states
34             output = input;
35         else
36             output = table (1, input);
37         end
38     else
39         errordlg('Wrong input parameter for mapping');
40     end
41
42 end
```

A.8 Main program Receiver_main.m

```
1   close all;
2  clear all;
3
4  %true = demodulation of vector testing_output_of_modulator
```

```
5 %false = demodulate signal captured by microphone
6 test = true;
7
8 %global variable for easy change
9 frequency_splitting=1;
10 time_splitting=2;
11
12 %setings of modulation parameters
13 T = 1;          %time interval for symbol in seconds
14 fhigh = 3000;   %settings for bandwidth of symbols
15 fdown = 300;
16 M = 32;         %number of modulation states
17 Fs = 8000;      %symbol sampling rate
18 type = time_splitting;
19 duration_of_data_frame = 100; %how many symbols will be in 1 frame
20 decimation_factor = 1;   %always choose odd number as the decimation factor
21 demodulation_method = 'xcorr';
22
23 length_of_symbol = T * Fs;
24
25 number_of_bits = log2(M);
26
27 size_of_data_frame = duration_of_data_frame * number_of_bits / T;
28
29 %align size of data frame so synchronization doesnt fluctuate
30 size_of_data_frame = size_of_data_frame - mod(size_of_data_frame, number_of_bits);
31
32 %tady to chce blok k nalezeni modulacnich symbolu, pokud neexistujou, tak je vytvori,
33 %pokud nejaky existujou tak se zepta jestli je ma vymazat,kdyby byly pro jinej pocet stavu
34 %nebo pro jinej typ modulace; nebo je jen necha a proste je nacte - pokud
35 %vime ze posledni pouzita modulace je stejna jako kterou chcem pouzit ted
36
37 %zatim byly vysilac i prijimac pousteny na stejnem pocitaci ve stejne slozce,
38 %proto prijimac pouze nacita stejne symboly vytvorene uz vysilacem
39
40 %load of synchronization & termination symbols
41 synchro_symbol = wavread('Synchonization_symbol.wav');
42 synchro_symbol =synchro_symbol(1:decimation_factor:end);
43 termination_symbol = wavread('Termination_symbol.wav');
44 termination_symbol = termination_symbol(1:decimation_factor:end);
45 %loading of modulation symbols
46 Modsymbols=zeros(length_of_symbol,M);  %modulation symbols
47
48 %loading of modulation symbols
49 for i=1:M
50     %concatination of Symbol_+(i)+.wav
51     filename = strcat('Symbol_', num2str(i), '.wav');
52     %load field of modulation symbols
53     Modsymbols(:,i) = wavread(filename);
54 end
55 Modsymbols = Modsymbols(1:decimation_factor:end,:);
56 %loading of testing vector testing_output_of_modulator
57 testing_output_of_modulator = wavread('testing_output_of_modulator.wav');
58
59 %laod locations of max for FRFT demodulator
60 load('alpha_values.mat');
61 %and a factor for FRFT
62 alpha_vec= (0.51:0.01:1.49);
63
64 %adding zeroes to testing vector for reason that syncho symbol
65 %would't start at intiger multiple of buffer
66 % testing funcionality of synchro_symbol position determination
67 %so 8K zeroes from transsmiter +2K zeroes = 10K zeroes
68 testing_output_of_modulator = cat(1, zeros(2000, 1), testing_output_of_modulator);
69
70 %variable calls dsp.Crosscorrelator function
71 %correlation determines if any symbol is in received signal
72 xcorr = dsp.Crosscorrelator('Method','Fastest');
73
74 %variable calling recording with set parameters
75 Mic=dsp.AudioRecorder('SamplesPerFrame',length_of_symbol,'BufferSizeSource','Property','BufferSize',length_of_symbol,'Queu
76 %specify the device which to acquire audio data, 'Default' => computer
77 %starndard input device
```

```
78 Mic.DeviceName='Default';
79 Mic.SampleRate=Fs;
80
81 %variable where data are stored in buffer which were acquired by mic
82 %buffer is 2*length_of_symbol long and overlaping by lenght of
83 %length_of_symbol => buff load one symbol in single iteration
84 InBuff = dsp.Buffer('Length', 2 * length_of_symbol, 'OverlapLength',length_of_symbol);
85
86 %in the begining synchronization is not recieved
87 s = 0;
88 current_character = 0;
89 %serves for stepping of testing vector for demodulation when test=true
90 iteration = 1;
91
92
93 %start of countdown till when synchonization must be acquired
94 tic;
95 duration_of_synchronization = toc;
96
97 recieved_string_of_state = -1;
98 transmission_found = 0;
99 end_of_transmission = 0;
100
101 %if transmitter is transmitting synchronization has to appear in the frame
102 %if not transmitting is not happening or reciever is not able to capture
103 %synchronization => if message‘s max size of buffer is exceeded process
104 %will be terminated
105
106 while (duration_of_synchronization < (duration_of_data_frame + 2 * T)) && (end_of_transmission ~= 1)
107     if test == false
108         %record one lenght of buffer from microphone
109         dataIn = step(InBuff, step(Mic));
110         dataIn = dataIn(1:decimation_factor:end);
111     else
112         %load length of buffer from test signal
113         dataIn = step(InBuff, testing_output_of_modulator((((iteration-1) * length_of_symbol) + 1) : (iteration * length_o
114         %decimation of dataIn for
115         dataIn = dataIn(1:decimation_factor:end);
116         iteration = iteration + 1;
117     end
118
119     %if synchronization was not found yet
120     if s == 0
121         %returns the maximum values in vector synchronizatin_location
122         [found, synchronizatin_location,correlation_location_of_synchronization] = locate_synchronization(dataIn, xcorr, s
123         if found == true
124             release(xcorr)
125             %testing if synchronization location was found properly
126             x = step(xcorr,dataIn,synchro_symbol);
127             plot(x)
128             s = 1;
129             transmission_found = 1;
130             tic;
131         end
132     else
133         %demodulation of recieved state
134        % tic
135         if strcmp(demodulation_method,'xcorr') == 1
136                 recieved_state = dem_chirp(dataIn, M, xcorr, Modsymbols, termination_symbol, correlation_location_of_synch
137         elseif strcmp(demodulation_method,'frft') == 1
138                 [recieved_state] = dem_FrFT(dataIn,M,alpha_vec, alpha_values, synchronization_location, correlation_locati
139         else
140                 errordlg('Wrong input of demodulation_method, Choose 'xcorr' or 'frft');
141         end
142         %counter of the recieved symbols
143         current_character = current_character + 1;
144       %  toc
145         if recieved_state ~= -1
146             %if this is the first recieved symbol
147             if recieved_string_of_state == -1;
148                 recieved_string_of_state = recieved_state;
149             else
150                 %some symbol was recieved already, add next one to the end
```

```
151                 recieved_string_of_state = cat(2, recieved_string_of_state, recieved_state);
152             end
153         else
154             end_of_transmission = 1;
155         end
156         %if size of frame is exceeded look for synchonization again
157         if mod(current_character*number_of_bits, size_of_data_frame) == 0
158             s = 0;
159             current_character = 0;
160             found = 0;
161         end
162     end
163 
164     %duration_of_synchronization = toc;
165 
166 end
167 
168 %message will be listed only of transmission was found
169 if transmission_found == 1
170 
171     recieved_string = -1;
172 
173     %translation recived state numbers (0:M-1) to the symbols
174     for i = 1:length(recieved_string_of_state)
175         recieved_symbol = symbol_mapping(recieved_string_of_state(1, i), 'symbol', M);
176         bin_string_of_symbol = dec2bin(recieved_symbol, number_of_bits);
177 
178         for j = 1:number_of_bits
179             %if this is the first tranlated state
180             if recieved_string == -1
181                 recieved_string = str2num(bin_string_of_symbol(j));
182             else
183                 recieved_string = cat(2, recieved_string, str2num(bin_string_of_symbol(j)));
184             end
185         end
186     end
187 
188     string_without_alignment = message_alignment_block(recieved_string, number_of_bits, 'remove_stuff');
189 
190     tranlated_message = message_translation(string_without_alignment);
191 
192     msgbox(tranlated_message, 'Recieved message');
193 
194 else
195     errordlg('Transmission not found');
196 end
197 
198 release(xcorr);
199 
200 % close(xcorr);
201 release(Mic);
202 
```

A.9 Function locate_synchronization.m

```
1  %FUNCTION FOR LOCALIZATION OF THE SYNCHRONIZATION SYMBOL AND ITS POSITION IN THE BUFFER
2  function [ found, synchronizatin_location,correlation_location_of_synchronization] = locate_synchronization( dataIn, xcorr
3 
4      found = false;
5      synchronizatin_location = 0;
6      resolution_threshold = 3;
7      correlation_location_of_synchronization = 0;
8 
9      mean_input_value = mean(abs(dataIn))
10     lenght_of_buffer = 2 * length_of_symbol;
11 
12     %if input sample are all 0 => mean value =0
13     %for this case SNR = 0/noise =>SNR=inf
14     %so it would evaulate that synchronization is found => that is bad
15     %this means SNR cant be used..
```

```
16      %however if mean=0 if>0.0001 will secure that for all 0 correlation
17      %will not be calculated
18      if mean_input_value > 0.0001;
19          release(xcorr_synchr);
20          synchronization_correlation = step(xcorr_synchr, synchro_symbol, dataIn);
21          release(xcorr_synchr);
22           modulation_symbol_correlation = step(xcorr_synchr, Modsymbols, dataIn);
23          release(xcorr_synchr);
24          termination_symbol_correlation = step(xcorr_synchr, termination_symbol, dataIn);
25
26          %location of synchonization in correlation output
27          %synchro level = max value
28          %corr_location = location of max in synchronization_correlation
29          [synchronization_level,correlation_location_of_synchronization] = max(abs(synchronization_correlation));
30          %searching for modulation symbol in correlation output
31          %in location of symbol begining +- 50 samples
32          [modulation_symbol_level, ~] = max(abs(modulation_symbol_correlation(((correlation_location_of_synchronization - 5
33          modulation_symbol_maximum = max(modulation_symbol_level);
34          [termination_symbol_level, ~] = max(abs(termination_symbol_correlation(((correlation_location_of_synchronization -
35
36          if (synchronization_level > resolution_threshold * modulation_symbol_maximum) && (synchronization_level > resoluti
37
38              %(( length_of_symbol + lenght_of_buffer - 1)/2) for decimation
39              %of 2
40              synchronizatin_location = - correlation_location_of_synchronization + (( length_of_symbol + lenght_of_buffer -
41
42              if ((correlation_location_of_synchronization > 0) && (correlation_location_of_synchronization < length_of_symb
43
44                  if synchronization_level > 20
45                      found = true;
46                  end
47              end
48          end
49
50      end
51
52  end
```

A.10 Function dem_chirp.m

```
1   %Function for demodulation of the signal
2   function [ Symbol, max_termination_symbol, AMP,PomMax,mean_PomMax,I,PomXcorr ] = dem_chirp(dataIn, M, xcorr_demod, Modsymb
3
4       resolution_threshold = 1;
5       release(xcorr_demod);
6       symbol_begining = synchronization_location - correlation_location_of_synchronization;
7       %correlation of the input data with all symbols
8       PomXcorr = abs(step(xcorr_demod, dataIn(correlation_location_of_synchronization : correlation_location_of_synchronizat
9
10      %(synchronization_location - 50):(synchronization_location + 50)
11      [PomMax,I]= max(PomXcorr((symbol_begining - 50):(symbol_begining + 50), :));
12          %jenze problem - kdyz bude synchronizace objevena u zacatku tak mi
13          %to hlasilo ze nemuze hledat maximum o 50 vzorku pred
14
15      %AMP = max amplitude, S = column location of AMP
16      %=> each column is for individual modulation symbol
17      %=> S = modulation symbol with highest correlation
18      [AMP,S] = max(PomMax);
19      mean_PomMax = mean(PomMax);
20      release(xcorr_demod);
21      termination_symbol_correlation = abs(step(xcorr_demod, dataIn, termination_symbol));
22      [max_termination_symbol,~] = max(termination_symbol_correlation((synchronization_location - 50):(synchronization_locat
23
24      if ((AMP >= resolution_threshold * max_termination_symbol) &&(AMP >= resolution_threshold * mean_PomMax))
25          %modulation symbols sorting, symbol is determinated by location
26          %of the maximum AMP
27          if ((S >= 1) && (S <= M))
28                  Symbol = S;
29          end
30      else
```

```
31         %termination symbol must be very clear
32         if max_termination_symbol >= 50
33             Symbol = -1;
34         else
35             %HOLD symbol 0 because in this case signal is noise or
36             %interruption could occur => demodalator did not recoignase any
37             %symbol so wait until reciver call dem_chirp again
38             Symbol = M+1;
39         end
40     end
41 end
```

A.11 Function dem_FrFT.m

```
1 function [ recieved_state,symbol_first_half,symbol_second_half] = dem_FrFT(dataIn,M,alpha_vec, alpha_values,synchronizatio
2
3 data = dataIn(correlation_location_of_synchronization : correlation_location_of_synchronization + (length_of_symbol/decima
4 % %the first half of data
5 data_first_half = data(1:(length_of_symbol/2));
6 %the second half of the data
7 data_second_half = data((length_of_symbol/2)+1:end);
8
9
10     for i = 1:length(alpha_values)
11         first_half_of_max_vector = alpha_values(i);
12         Frft_first_half = frft(data_first_half,alpha_vec(first_half_of_max_vector));
13         [first_half_maximum(i),location(i)]= max(abs(Frft_first_half));
14 %                 figure(5)
15 %         plot(real(Frft_first_half))
16 %         title('Symbol 17')
17 %
18     end
19
20    % F = location of this maximums
21   [first_half_mod_symbol_max,F]= max(first_half_maximum);
22
23   location_of_first_halfmax = location(F);
24
25    %for detection of the Location_of_max_first_half in the alpha vec%%%%%%
26
27    %distinguish symbols 1-16 &17-32
28    %because symbol 1 & 17 have the same symbol position
29    if location_of_first_halfmax < (length_of_symbol/4)
30        %first half => symbols 1-16
31        symbol_first_half= F;
32    else
33        %second half => symbols 17-32
34         symbol_first_half= (M/2)+F;
35
36
37    end
38 %second half maximum**************
39
40     for ii = 1:length((alpha_values))
41         second_half_of_max_vector = fliplr(alpha_values);
42           second_half_position = second_half_of_max_vector(ii);
43         Frft_second_half = frft(data_second_half,alpha_vec(second_half_position));
44          [second_half_maximum(ii),location_second_half(ii)]= max(abs(Frft_second_half));
45
46     end
47     %S =second half location
48     [second_half_mod_symbol_max,S]= max(second_half_maximum);
49     %     toc
50 location_of_second_halfmax = location(S);
51
52 %for detection of the Location_of_max_first_half in the alpha vec%%%%%%
53
54    %distinguish symbols 1-16 &17-32
55    %because symbol 1 & 17 have the same symbol position
56    if location_of_second_halfmax < (length_of_symbol/4)
```

```
57          %first half => symbols 1-16
58          symbol_second_half = S;
59      else
60          %second half => symbols 17-32
61           symbol_second_half = (M/2)+S;
62
63      end
64
65  if  symbol_first_half == symbol_second_half
66        recieved_state = symbol_first_half;
67  else
68
69      recieved_state = M+1;
70  end
71
72  end
73
74
```

A.12 Function locate_aplha_values

```
1  %FUNCTION FOR LOCALIZATION OF ALPHA VALUES
2  function [alpha_values,auxiliary_vec ] = locate_alpha_values( alpha_vec,M,length_of_symbol )
3
4  %load modulation symbols
5  for i=1:M
6      %concatination of Symbol_+(i)+.wav
7      filename = strcat('Symbol_', num2str(i), '.wav');
8      %load field of modulation symbols
9      Modsymbols(:,i) = wavread(filename);
10 end
11
12 Modsymbols_first_half = Modsymbols(1:half_of_length_of_symbol,:);
13 Modsymbols_second_half = Modsymbols(half_of_length_of_symbol:end,:);
14
15 for j=1:M
16     %find max value of frft for element from a vector
17     for n = 1:length(alpha_vec)
18         Faf_first_half = frft(Modsymbols_first_half(:,j),alpha_vec(n));
19         %largest element and its location
20         [max_first_half(n),location(n)] = max(abs(Faf_first_half));
21 %          figure(5)
22 %          plot(real(Faf_first_half))
23     end
24    %since identical maximuns are found, sorting is done
25    %Mag = magnitude, Loc = location of this maximums
26   [Mag,Loc]= sort(max_first_half,'descend')
27   %the first maximum is desired one, this if commnad assure it
28   if Loc(1)< Loc(2)
29       MaX_first_half = Mag(1)
30       F(j)= Loc(1)
31   else
32       MaX_first_half = Mag(2)
33       F(j)= Loc(2)
34   end
35
36    %for detection of the Location_of_max_first_half in the alpha vec%%%%%%
37     symbol_position = location(F);
38    %auxiliary vector for distinguish symbols 1-16 &17-32
39    %because symbol 1 & 17 have the same symbol position
40    if symbol_position(j) < (half_of_length_of_symbol/2)
41        %first half => symbols 1-16
42        auxiliary_vec(j)= 1;
43    else
44        %second half => symbols 17-32
45         auxiliary_vec(j)= -1;
46
47     end
48 end
49 alpha_values = F;
```

```
50 auxiliary_vec;
51 %cat just for control 1-16 has value 1 , 17-32 value -1
52 % auxiliary_vec1= auxiliary_vec(1:16);
53 % auxiliary_vec2= auxiliary_vec(17:32);
54 % cat(1,auxiliary_vec1,auxiliary_vec2);
55
56 save('alpha_values.mat','alpha_values', '-mat' );
57 save('auxiliary_vec.mat','auxiliary_vec', '-mat' );
58
59
60
61 %%%%%%%%%%%%second half%%%%%%%%%%%%%%%%%%%%%%%%
62 %just for control because values are same as for first half but flipped
63 % for jj=1:M
64 %
65 %     for nn = 1:length(a)
66 %         Faf_second_half = frft(Modsymbols_second_half(:,jj),a(nn));
67 %        [max_second_half(nn),location2(nn)] = max(abs(Faf_second_half));
68 %     end
69 %      [Mag2,Loc2]= sort(max_second_half,'descend')
70 %     if Loc2(1)< Loc2(2)
71 %       MaX_second_half = Mag2(1)
72 %       S(jj)= Loc2(1)
73 %     else
74 %       MaX_second_half = Mag2(2)
75 %       S(jj)= Loc2(2)
76 %     end
77 %
78 %      symbol_position2 = location2(S);
79 %
80 %        if symbol_position2(jj) < (2000)
81 %        %first half => symbols 1-16
82 %        auxiliary_vec2(jj)= 1;
83 %         else
84 %        %second half => symbols 17-32
85 %         auxiliary_vec2(jj)= -1;
86 %
87 %         end
88 %
89 % end
90
91 % Location_of_max_second_half = S;
92 % auxiliary_vec2;
93 % %cat just for control 1-16 has value 1 , 17-32 value -1
94 % auxiliary_vec11= auxiliary_vec2(1:16)
95 % auxiliary_vec22= auxiliary_vec2(17:32)
96 % bbbb=cat(1,auxiliary_vec11,auxiliary_vec22)
97
98 %***DEMODUALTION TESTING*******
99
100 % Location_of_max_second_half_flipped = fliplr(Location_of_max_second_half)
101 % mod_symbol_table = cat(1,Location_of_max_first_half,Location_of_max_second_half_flipped)
102 %load('Location_of_max_first_half.mat');
103
104 % %the first half maximum
105 % %for j=1:M
106 % tic
107 %     for i = 1:length(Location_of_max_first_half)
108 %         first_half_of_max_vector = Location_of_max_first_half(i);
109 %         Faf_first_half = frft(Modsymbols_first_half(:,5),alpha_vec(first_half_of_max_vector));
110 %         first_half_maximum(i) = max(abs(Faf_first_half));
111 %     end
112 % %     mod_symbol_max = zeros(1,M);
113 %     [first_half_mod_symbol_max(j),location_of_first_halfmax] = max(first_half_maximum)
114 %     toc
115 % %end
116 %
117 % %second half maximum
118 % %for j=1:M
119 %     for ii = 1:length(Location_of_max_first_half)
120 %         second_half_of_max_vector = fliplr(Location_of_max_first_half);
121 %          second_half_position = second_half_of_max_vector(ii);
122 %         Faf_second_half = frft(Modsymbols_second_half(:,6),alpha_vec(second_half_position));
```

```
123 %         second_half_maximum(ii) = max(abs(Faf_second_half));
124 %     end
125 % %     mod_symbol_max = zeros(1,M);
126 %     [second_half_mod_symbol_max(j),location_of_second_half_max] = max(second_half_maximum)
127 % %     toc
128 % %end
129 end
```

A.13 Function frft.m

```
1  function Faf = frft(f, a)
2  %meng\copyright 2013
3  % The fast Fractional Fourier Transform
4  % input: f = samples of the signal
5  %        a = fractional power
6  % output: Faf = fast Fractional Fourier transform
7
8  error(nargchk(2, 2, nargin));
9
10 f = f(:);
11 N = length(f);
12 shft = rem((0:N-1)+fix(N/2),N)+1;
13 sN = sqrt(N);
14 a = mod(a,4);
15
16 % do special cases
17 if (a==0), Faf = f; return; end;
18 if (a==2), Faf = flipud(f); return; end;
19 if (a==1), Faf(shft,1) = fft(f(shft))/sN; return; end
20 if (a==3), Faf(shft,1) = ifft(f(shft))*sN; return; end
21
22 % reduce to interval 0.5 < a < 1.5
23 if (a>2.0), a = a-2; f = flipud(f); end
24 if (a>1.5), a = a-1; f(shft,1) = fft(f(shft))/sN; end
25 if (a<0.5), a = a+1; f(shft,1) = ifft(f(shft))*sN; end
26
27 % the general case for 0.5 < a < 1.5
28 alpha = a*pi/2;
29 tana2 = tan(alpha/2);
30 sina = sin(alpha);
31 f = [zeros(N-1,1) ; interp(f) ; zeros(N-1,1)];
32
33 % chirp premultiplication
34 chrp = exp(-i*pi/N*tana2/4*(-2*N+2:2*N-2)'.^2);
35 f = chrp.*f;
36
37 % chirp convolution
38 c = pi/N/sina/4;
39 Faf = fconv(exp(i*c*(-(4*N-4):4*N-4)'.^2),f);
40 Faf = Faf(4*N-3:8*N-7)*sqrt(c/pi);
41
42 % chirp post multiplication
43 Faf = chrp.*Faf;
44
45 % normalizing constant
46 Faf = exp(-i*(1-a)*pi/4)*Faf(N:2:end-N+1);
47
48 %%%%%%%%%%%%%%%%%%%%%%%%%
49 function xint=interp(x)
50 % sinc interpolation
51
52 N = length(x);
53 y = zeros(2*N-1,1);
54 y(1:2:2*N-1) = x;
55 xint = fconv(y(1:2*N-1), sinc([-(2*N-3):(2*N-3)]'/2));
56 xint = xint(2*N-2:end-2*N+3);
57
58 %%%%%%%%%%%%%%%%%%%%%%%%%
59 function z = fconv(x,y)
60 % convolution by fft
```

```
61
62 N = length([x(:);y(:)])-1;
63 P = 2^nextpow2(N);
64 z = ifft( fft(x,P) .* fft(y,P));
65 z = z(1:N);
```

A.14 Function message_translation.m

```
1      ASCII_character_length = 8;
2
3      acutal_bit = 1;
4
5      message = '';
6
7      while acutal_bit < length(input_bit_stream)
8          bin_string_ASCII = num2str(input_bit_stream(acutal_bit:(acutal_bit + (ASCII_character_length - 1))));
9          dec_number_of_symbol = bin2dec(bin_string_ASCII);
10         ASCII_char_symbol = native2unicode(dec_number_of_symbol, 'ISO-8859-2');
11         message = strcat(1, message, ASCII_char_symbol);
12         acutal_bit = acutal_bit + ASCII_character_length;
13     end
14
15 end
```

A.15 Function BER_receiving_side

```
1 function [ output_args ] = BER_receiving_side(  )
2 %UNTITLEDs2 Summary of this function goes here
3 %   Detailed explanation goes here
4 close all;
5 clear all;
6
7 %true = demodulation of vector testing_output_of_modulator
8 %false = demodulate signal captured by microphone
9 test = true;
10
11 %global variable for easy change
12 frequency_splitting=1;
13 time_splitting=3;
14
15 %setings of modulation parameters
16 T = 1;          %time interval for symbol in seconds
17 fhigh = 3000;   %settings for bandwidth of symbols
18 fdown = 300;
19 M = 32;         %number of modulation states
20 Fs = 8000;      %symbol sampling rate
21 type = time_splitting;
22 duration_of_data_frame = 10000000; %how many symbols will be in 1 frame
23 decimation_factor = 1;   %always choose odd number as the decimation factor
24
25 length_of_symbol = T * Fs;
26
27 number_of_bits = log2(M);
28
29 size_of_data_frame = duration_of_data_frame * number_of_bits / T;
30
31 %align size of data frame so synchronization doesnt fluctuate
32 size_of_data_frame = size_of_data_frame - mod(size_of_data_frame, number_of_bits);
33
34 %load of synchronization & termination symbols
35 synchro_symbol = wavread('Synchonization_symbol.wav');
36 synchro_symbol = synchro_symbol(1:decimation_factor:end).*8;
37 termination_symbol = wavread('Termination_symbol.wav');
38 termination_symbol = termination_symbol(1:decimation_factor:end)./8;
39 %loading of modulation symbols
40 Modsymbols=zeros(length_of_symbol,M);  %modulation symbols
41
```

```
42  [sequence,Bin_sequence, BER_testing_output_of_modulator ] = BER_transmitting_side();
43 %loading of modulation symbols
44 % sequence= load('sequence.mat')
45
46 for i=1:M
47     %concatination of Symbol_+(i)+.wav
48     filename = strcat('Symbol_', num2str(i), '.wav');
49     %load field of modulation symbols
50     Modsymbols(:,i) = wavread(filename);
51 end
52 Modsymbols = Modsymbols(1:decimation_factor:end,:);
53 %correlation coefficients
54 coef= corrcoef(Modsymbols);
55 %laod locations of max for FRFT demodulator
56 load('alpha_values.mat');
57 %and a factor for FRFT
58 alpha_vec= (0.51:0.01:1.49);
59 %Modsymbols_first_half = Modsymbols(1:4000,:);
60
61 %loading of testing vector testing_output_of_modulator
62 BER_testing_output_of_modulator = wavread('BER_testing_output_of_modulator.wav');
63
64 %adding zeroes to testing vector for reason that syncho symbol
65 %would`t start at intiger multiple of buffer
66 % testing funcionality of synchro_symbol position determination
67 %so 8K zeroes from transsmiter +2K zeroes = 10K zeroes
68 shift = 500;
69 BER_testing_output_of_modulator = cat(1, zeros(shift, 1), BER_testing_output_of_modulator);
70
71 %variable calls dsp.Crosscorrelator function
72 %correlation determines if any symbol is in received signal
73 xcorr = dsp.Crosscorrelator('Method','Fastest');
74
75 %variable calling recording with set parameters
76 Mic=dsp.AudioRecorder('SamplesPerFrame',length_of_symbol,'BufferSizeSource','Property','BufferSize',length_of_symbol,'Queu
77 %specify the device which to acquire audio data, 'Default' => computer
78 %starndard input device
79 Mic.DeviceName='Default';
80 Mic.SampleRate=Fs;
81
82 %variable where data are stored in buffer which were acquired by mic
83 %buffer is 2*length_of_symbol long and overlaping by lenght of
84 %length_of_symbol => buff load one symbol in single iteration
85 InBuff = dsp.Buffer('Length', 2 * length_of_symbol, 'OverlapLength',length_of_symbol);
86
87 %in the begining synchronization is not recieved
88 s = 0;
89 current_character = 0;
90 %serves for stepping of testing vector for demodulation when test=true
91 iteration = 1;
92
93
94 %start of countdown till when synchonization must be acquired
95 tic;
96 duration_of_synchronization = toc;
97
98 recieved_string_of_state = -1;
99 transmission_found = 0;
100 end_of_transmission = 0;
101
102 %if transmitter is transmitting synchronization has to appear in the frame
103 %if not transmitting is not happening or reciever is not able to capture
104 %synchronization => if message`s max size of buffer is exceeded process
105 %will be terminated
106 xxcounter= 0;
107 symbol_counter = 0;
108 EbNoVec = -4:0.5:10;
109 % ((1:length(EbNoVec))= BER;
110 % ((1:length(EbNoVec))= SER;
111 for n = 1:length(EbNoVec)
112 %     %converting En/Bo to SNR
113     EbNo = EbNoVec(n)
114
```

```
115     % snrdB = EbNoVec(n)+10*log10(number_of_bits)-(10*log10((Fs/decimation_factor)/2));
116      %  snrdB = -18
117
118        snrdB = EbNoVec(n)+10*log10(number_of_bits);
119      BER(n) = 0;
120      SER(n) = 0;
121      Error_counter = 0;
122      bit_error_counter = 0;
123      symbol_counter = 0;
124      end_counter = 0;
125
126      rms_orig = rms(BER_testing_output_of_modulator)
127      AWGN_BER_testing_output_of_modulator = awgn(BER_testing_output_of_modulator,snrdB,'measured');
128      rms_awgn = rms(AWGN_BER_testing_output_of_modulator)
129      normalization = rms_orig/rms_awgn;
130      AWGN_BER_normilazed = AWGN_BER_testing_output_of_modulator.*normalization;
131      rms_normalized = rms(AWGN_BER_normilazed)
132
133  %while SER < 100 %&& numBits < 1e10
134      %(duration_of_synchronization < (duration_of_data_frame + 2 * T)) && (end_of_transmission ~= 1)
135
136     while (SER < 100) %& (end_counter < (n)*13)
137      if test == false
138          %record one lenght of buffer from microphone
139          dataIn = step(InBuff, step(Mic));
140          dataIn = dataIn(1:decimation_factor:end);
141      else
142          %load length of buffer from test signal
143          dataIn = step(InBuff, AWGN_BER_normilazed((((iteration-1) * length_of_symbol) + 1) : (iteration * length_of_symbol
144  %          dataIn_buffer_rms = rms(dataIn);
145          %decimation of dataIn for
146          dataIn = dataIn(1:decimation_factor:end);
147          iteration = iteration + 1;
148      end
149
150      %if synchronization was not found yet
151      if s == 0
152  %          Modsymbols = Modsymbols(1:decimation_factor:end,:)./1;
153  %          %returns the maximum values in vector synchronizatin_location
154  %                [found, synchronizatin_location,correlation_location_of_synchronization] = locate_synchronization(dataIn, :
155  %          Modsymbols = Modsymbols(1:decimation_factor:end,:).*1;
156  found = 1;
157  synchronization_location = 16500;
158  correlation_location_of_synchronization = 7499;
159  release(xcorr);
160
161  %             plot(x)
162          if found == true
163              release(xcorr)
164              s = 1;
165              transmission_found = 1;
166              tic;
167          end
168      else
169
170      end
171          %demodulation of recieved state
172  %           tic;
173     %      [recieved_state, max_termination_symbol, AMP,PomMax,mean_PomMax,I, PomXcorr] = dem_chirp(dataIn, M, xcorr, Modsym
174  %tic
175   [recieved_state,symbol_first_half,symbol_second_half] = dem_FrFT(dataIn,M,alpha_vec, alpha_values, synchronization_locati
176    %        toc
177          %counter of the recieved symbols
178       release(xcorr);
179  %  x = step(xcorr,dataIn,Modsymbols(1,:));
180  %  plot(x)
181          current_character = current_character + 1;
182          symbol_counter = symbol_counter  + 1;
183          current_input = sequence(current_character);
184
185              if current_character == 9999
186
187                  xxcounter= xxcounter +1
```

```
188                 current_character = 0;
189             iteration = 2;
190             end_counter = end_counter+1
191
192
193         %this for converts char string to bin double
194
195 %         current_char_in_sequence = sequence(i);
196          %fixed number of bits
197 %          sequence_string = dec2bin(current_char_in_sequence, number_of_bits);%fixed number of bits
198 %              for j = 1:number_of_bits
199 %                   %dec2bin return binary but in string of chars => necessary convert to numbers
200 %                     bin_input (j) = str2num(sequence_string(j));
201 %              end
202
203 %         toc;
204         if recieved_state ~= -1
205             %if this is the first recieved symbol
206             if recieved_string_of_state == -1;
207                 recieved_string_of_state = recieved_state;
208                 recieved_symbol = symbol_mapping(recieved_string_of_state, 'symbol', M);
209             else
210                 %some symbol was recieved already, add next one to the end
211                 recieved_symbol = symbol_mapping(recieved_state, 'symbol', M);
212                 recieved_string_of_state = cat(2, recieved_string_of_state, recieved_state);
213                 bin_string_of_symbol = dec2bin(recieved_symbol, number_of_bits);
214                 %this loop convers char from dec2bin to double string
215 %                    for j = 1:number_of_bits
216 %                        %dec2bin return binary but in string of chars => necessary convert to numbers
217 %                        bin_output (j) = str2num(bin_string_of_symbol(j));
218 %                    end
219 %current_input = 2
220                    comparsion= [current_input, recieved_symbol];
221
222                 if xor(current_input~=recieved_symbol,recieved_symbol==0)
223                 %biterror of input and output signal
224
225                 bin_input = (zeros(1,number_of_bits ));
226
227                 %this loop convers char from dec2bin to double string
228                 current_input_bin = dec2bin(current_input, number_of_bits);%fixed number of bits
229              for j = 1:number_of_bits
230                    %dec2bin return binary but in string of chars => necessary convert to numbers
231                      bin_input (j) = str2num(current_input_bin(j));
232              end
233
234                 bin_output = (zeros(1,number_of_bits ));
235
236                 current_output = dec2bin(recieved_symbol, number_of_bits);
237                 %this loop convers char from dec2bin to double string
238                   for j = 1:number_of_bits
239                       %dec2bin return binary but in string of chars => necessary convert to numbers
240                       bin_output (j) = str2num(current_output(j));
241                   end
242
243
244                 nErrors = biterr(bin_input,bin_output);
245
246                 BER(n)= BER(n)+ nErrors;
247
248                 SER(n) = SER(n)+ 1
249
250                 Error_counter = Error_counter + nErrors;
251
252
253                 end
254         %else
255 %             end_of_transmission = 1;
256         end
257         %if size of frame is exceeded look for synchonization again
258 %         if mod(current_character*number_of_bits, size_of_data_frame) == 0
259 %             s = 0;
260 %             current_character = 0;
```

```
261 %             found = 0;
262 %         end
263     end
264
265     %duration_of_synchronization = toc;
266     end
267
268    end
269 end
270
271     %Esitmated the BER
272     BER_estimated(n) = Error_counter/((number_of_bits)*(symbol_counter));
273     %Theorectical BER
274     BER_theory = berawgn(EbNoVec,'fsk', M, 'coherent');
275     BER(n) = Error_counter/((number_of_bits)*(symbol_counter));
276     BER_8 = zeros(1,length(EbNoVec));
277     BER_8 = cat(2,BER, BER_8)
278     save('BER_32_FrFT.mat','BER_8', '-mat' );
279     %Symbol Error Rate
280     SER(n)=SER(n)/(symbol_counter);
281     figure(2)
282     semilogy(EbNo,BER(n),'b*');
283     hold on
284     semilogy(EbNoVec,BER_theory,'red')
285     grid
286     title('BER vs Eb/No')
287     axis([-4 10 10^-6 1]);
288     xlabel('Eb/No[dB]')
289     ylabel('Bit Error Rate')
290     legend('BER','Theorectical BER')
291
292 end
```