# Implementation of a real-time voice encryption system

Markus Albert Brandau

**Master Thesis**
**Universitat Politècnica de Catalunya**
**EUETIT**

Autor: ***Markus Brandau***
Tutor: ***Ignasi Esquerra Llucià***

**Feb. 2008**

Fachhochschule Köln
University of Applied Sciences Cologne

07 **Fakultät für Informations-, Medien- und Elektrotechnik,**

**Studiengang Technische Informatik/**

**Information Engineering**

Institut für Nachrichtentechnik

Labor für Technische Akustik

# Master Thesis

Thema: Implementation of a real-time voice encryption system

| | |
|---|---|
| Student : | Markus Brandau |
| Matrikelnummer: | 11031393 |
| Referent : | Prof. Dr.- Ing. Pörschmann |
| Korreferent : | Prof. Dr.- Ing. Elders- Boll |

Abgabedatum : 13. Feb. 2008

Hiermit versichere ich, dass ich die Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

Markus Albert Brandau

# Abstract

In this master thesis, a voice encryption system was programmed as a real-time software application. The idea was based on a previous graduation thesis and is meant to show an example of the possibilities of signal processing for educational use. The application uses a frequency scrambling technique on an audio signal taken from the computer microphone input and plays it scrambled back to the speakers, or other way around to descramble the signal. The basic idea is to use a frequency channel decomposition through digital signal filtering and down-sampling. Reassembling these sub-bands in a different order, a scrambled voice signal is the result. In addition, the scrambling process is embedded into a “software framework” which allows the use of an audio data stream from and to a sound card through different APIs so a computer platform real-time application can be accomplished.

# Table of Contents

# List of abbreviations

| | |
|---|---|
| ADPCM | Adaptive Differential Pulse Code Modulation |
| AES | Advanced Encryption System |
| ALSA | Advanced Linux Sound Architecture |
| API | Application Programming Interface |
| ASIO | Audio Stream Input/Output |
| CELP | Code Excited Linear Prediction |
| CLAM | C++ Library for audio and music |
| CVSD | Continuously Variable Slope Delta-modulation |
| DES | Data Encryption Standard |
| DLL | Dynamic Link Library |
| DSP | Digital Signal Processors |
| FEAL | Fast Data Encryption Algorithm |
| FIR | Finite impulse Response |
| FSK | Frequency Shift Keying |
| FSK - FDM | Frequency Shift Keying-Frequency DivisionMultiplex |
| GPL | General Public License |
| GUI | Graphic User Interface |
| IDEA | International Data Encryption Algorithm |
| IIR | Infinite Impulse Response |
| LPC | Linear Predictive Coding |
| MELP | Mixed Excitation Linear Prediction |
| OS | Operation System |
| OSS | Open Sound System |
| PCM | Pulse Code Modulation |
| PD | Pure Data |
| PFC | Program Flow Chart |
| PGP | PGPfone Pretty good privacy |
| RC5 | Rivest’s Code 5 |
| RIFF | Resource Interchange File Format |
| Safer | Secure and Fast Encryption Routine |
| STE | Secure Terminal Equipment |
| STK | Synthesis ToolKit |
| STU | Secure Telephone Unit, |
| WAV | Waveform Audio Format |

# 1 Introduction to Voice Security

**The first chapter gives a little overview of the general purpose and the basics of this thesis. A closer look is taken on to the history of voice security as well as on to the basic techniques and the nature of the human voice itself.**

Security and privacy of data is an overall issue, this fact rules for all kind of data at any time but especially for those which are transmitted in some way. Communication systems are often seen as possible security leaks for transmitted data even though these systems employ data security techniques.

Voice encryption systems are used to guarantee end-to-end security for speech in real time communication systems such as GSM, VoIP, Telephone, analogue Radio. Figure 1 illustrates a possible leak in a kind of network as we use it every day.

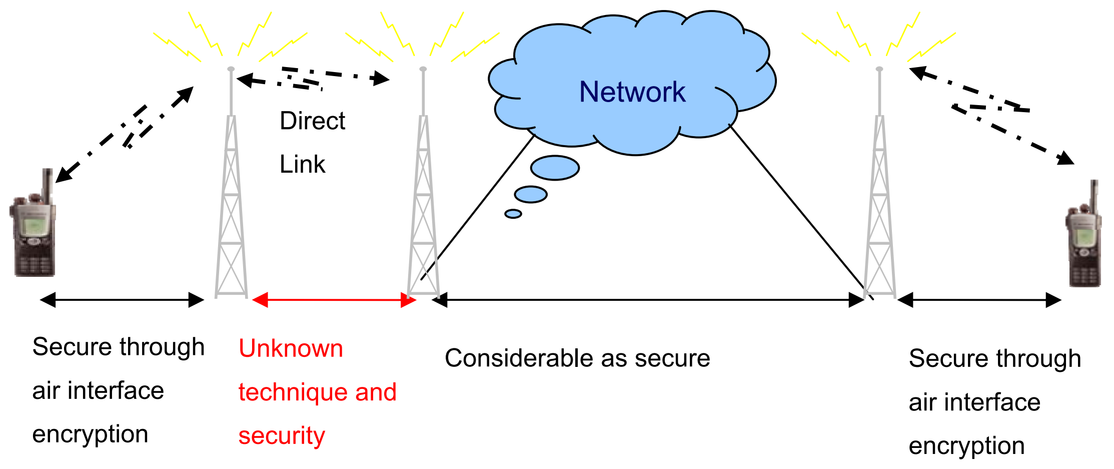

Network
Direct Link
Secure through air interface encryption
Unknown technique and security
Considerable as secure
Secure through air interface encryption

**Figure 1: Radio Network with unsecured sectors**

Surely there is quite a range of applications available on the market that solves this issue very good. Since this, often even standardised techniques are not meant to be challenged, the approach developed while this thesis can rather be seen as a demo for a lot of signal processing techniques (like channel filtering, up/down-sampling, real-time audio etc.) for academic purposes then as a serious guarantee for end- to end security.

## 1.1 History

The idea of encryption of messages is probably as old as the fact that messages have been sent between humans. A lot of different techniques have been developed over the centuries beginning with Caesar cipher all the way to modern standards like advanced encryption standard (AES). In fact, most of the encryption systems were meant to encrypt text messages; the development of encryption systems for voice is in turn a lot more difficult and mostly not satisfying achievable with analogue techniques. Due to that problem, much pioneering work for many digital capabilities was performed while inventing a system to provide secure voice communications. [9] So one could say, voice encryption was one of the earliest digital applications.

The first approaches for a voice encryption system were invented in the 1920th mainly at the Bell Labs between World War I and World War II and it of course had a strong military background. This system consisted out of an electronic circuit that could mix two signals into one, or alternately "subtract" two signals from each other. One of the signals was provided by a telephone microphone, and the other one by some kind of audio player. The audio signal was a 'noise' that was produced and recorded twice, as matching pair. The records were provided on large shellac phonographs, those were shipped to both ends of the telephone line where they were needed. The noise would then be played into the system, mixed with the signal of the telephone microphone and afterwards be sent through a cable or radio line. With the same system on the other side and the matching record, the noise could be subtracted back out of the signal, leaving the original voice signal. Eavesdroppers could only hear the noisy signal, unable to understand the voice inside. After a conversation the record had to be destroyed. Even though the technique worked in principle it was practically very difficult to achieve synchronization of the two records. Attainably the system was broken by the Germans.

In the 1940th a far more secure system again by Bell Labs was invented and used which was called Sigsaly (which was just a cover name and not an acronym). This 50 ton heavy, complex system that is shown in Figure 2 was the first secure voice encryption system.

**Figure 2: Sigsaly exhibit at the National Cryptology Museum. [9]**

The development of Sigsaly was a truly groundbreaking development; it contained pioneers work in not less than eight achievements:

- the first realization of enciphered telephony,
- the first quantized speech transmission,
- the first transmission of speech by Pulse Code Modulation (PCM),
- the first use of companded PCM,
- the first examples of multilevel Frequency Shift Keying (FSK),
- the first useful realization of speech bandwidth compression,
- the first use of FSK-FDM (Frequency Shift Keying-Frequency Division Multiplex) as a viable transmission method over a fading medium,
- the first use of a multilevel "eye pattern" to adjust the sampling intervals. [1]

These new technologies led not only to modern digital voice encryption but furthermore, they were important for modern digital signal processing in general. Figure 3 gives a short overview of the system, which is not to be explained further in this document, for more information to Sigsaly check reference [1][9].

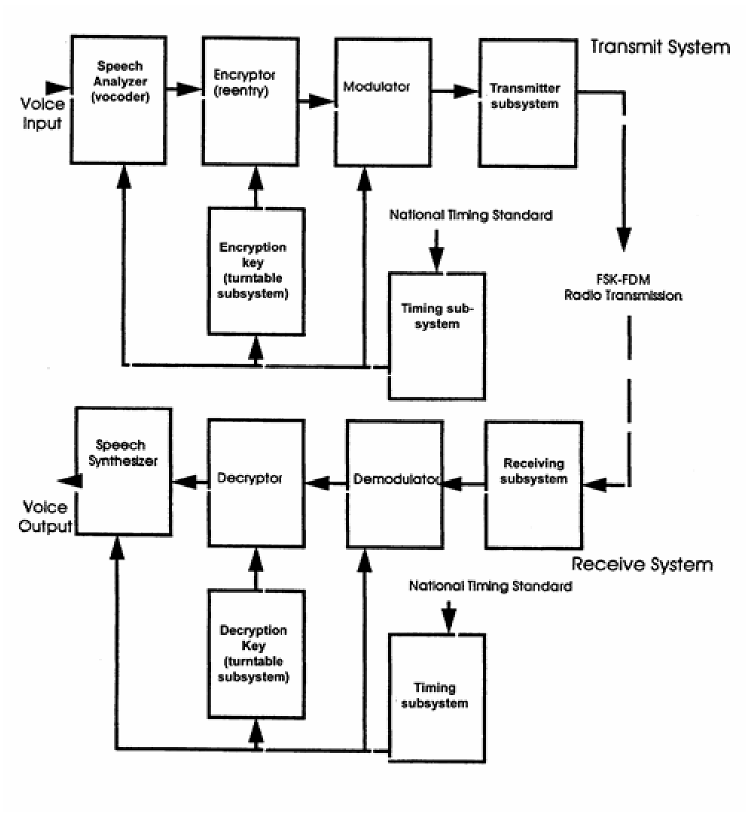


**Figure 3: Simplified overview of the Sigsaly [9]**

These systems were, among others, used for telephone conversations between Winston Churchill and Franklin D. Roosevelt while World War II.

No doubt that these two developments were big milestones in history of secure voice transmissions. In the following decades (beginning in the 1960th) a big range of different systems has been developed to protect the privacy of the voice. First analogue and then with the improvements of the digital signal processors (DSP), also digital techniques were invented leading into pure software applications, for example:

**Hardware Based Encryption Systems:**

- Digital Voice Protection (Motorola)
- STU III (Secure Telephone Unit, Generation III)

  (Mainly for the US government for telephony)
- STEs (Secure Terminal Equipment)

  (Mainly for the US government for high-speed data for video-conferencing, fax and voice applications)

**Software Based Encryption Systems:**

- PGPfone
- Nautilus
- Speak Freely

to just name a few. Besides that, there are an uncountable number of proprietary systems on the market.

## 1.2 The Human Voice

For the development and understanding of a voice scrambling system it is existentially to know some basic parameters of the voice itself. In fact the theory is quite more complex than illustrated in the following chapter.

The human voice is simply a sound or audio signal which is generated by a human being to communicate with one and another. Therefore, men use their vocal folds to modulate the air stream, coming from the lungs, into vibrations which make, with the rest of the human sound forming system (like: mouth hole, tong etc), a sound. This sound is treated by some different effects to make a variety of sounds like vowels or consonants.

Hence to the difference of the anatomic of every human the voice of two people never sound the same. The biggest difference is obviously recognised between the voices of men, women and children. This is due to the different fundamental frequencies (often also referred as *pitch* of the voice) these three groups usually have. These fundamental frequencies are mainly generated by the vocal cords which create through their opening and closing a periodic signal with the fundamental frequency $f_0$.

The male vocal cords are between 17 and 25 mm in length and their fundamental frequency are between 85 and 155 Hz, the ones of a woman are between 12.5 and 17.5 mm in length which obtains a fundamental frequency between 165 and 250 Hz. Kids have even smaller cords, their fundamental frequency is often about 440 Hz (Babies up to 500Hz).

The human speech consists out of vowels and consonants which both are important for understanding. Vowels are voiced sounds where the vocal cords vibrate while they are produced. Those ones have a very clear, narrow and harmonic spectrum that is at lower frequencies. Some consonants are unvoiced or voiceless sounds which have a wider and higher spectrum as Figure 4 shows the frequency distribution for one-octave bands.

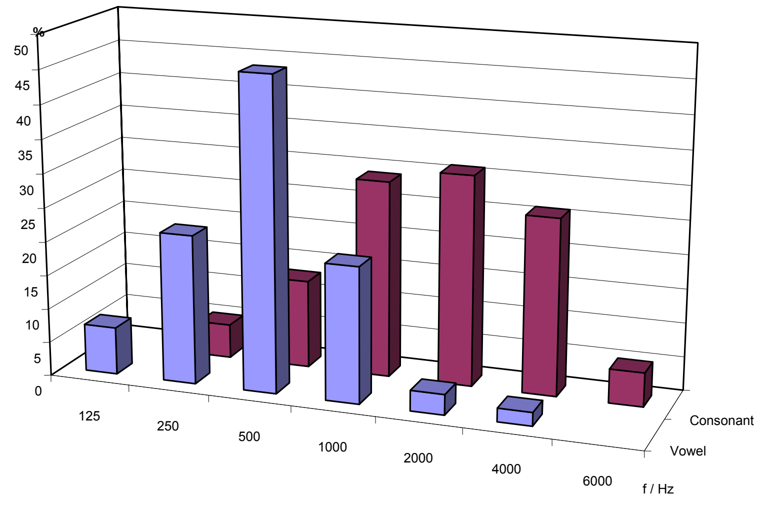


**Figure 4: Frequency distribution of the human voice**

However there is a kind of diagram that is often used for speech signals which is the spectrogram. In this one, the frequencies are plotted over the time and the amplitude is marked with the intensity of the colour like in Figure 5.

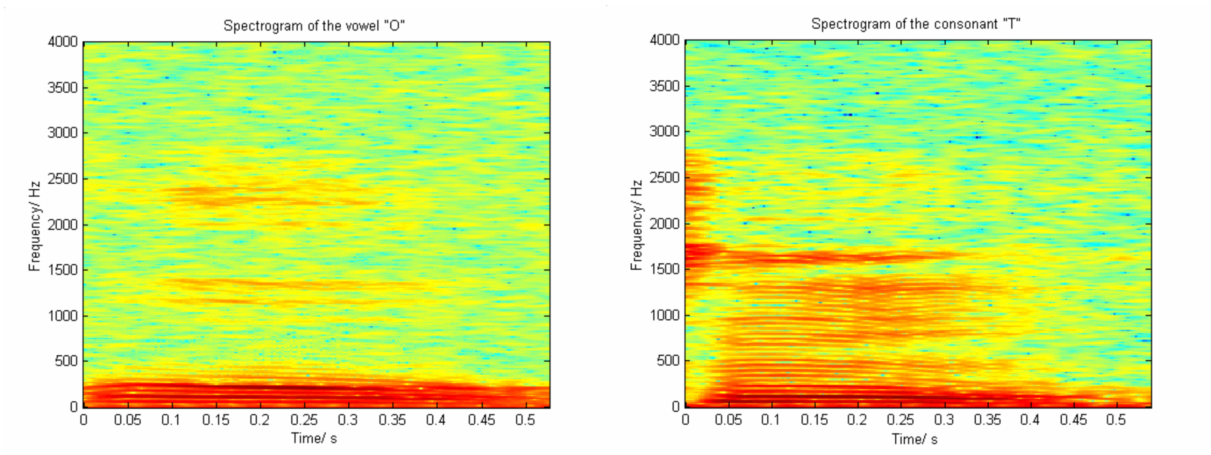


**Figure 5: Spectrogram of vowels and consonants**

As observed in the left spectrogram of Figure 5 the vowel has a predominant low spectrum. The right spectrogram is the one of a consonant followed by a vowel sound.

The sound of the human voice is of course a combination of the vowels and of the consonants frequencies which leads into a spectrogram like Figure 6 for a normal spoken sentence.

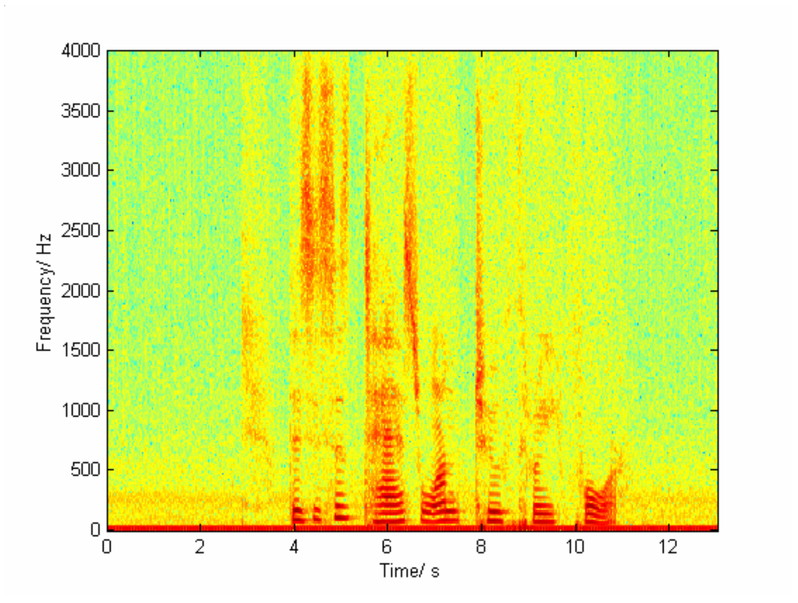


**Figure 6: Spectrogram of a human voice**

The complete spectrum of the human voice is individually about 1,3 – 2,5 octaves (where the ear has around 10 octaves). That makes the average range of voice about 80 Hz to 12 kHz. Where in this bandwidth are some sub bands more important than others because those are necessary for the ability of understanding and more significant for the difference of the voice.

For less bandwidth consumption the frequency used in telephony ranges from approximately 300 Hz to 3400 Hz, the fundamental frequency of most speech falls in this range, so that there are always harmonic series in the voice signal to give the listener the feeling of hearing the fundamental tone. With some guard bands added you can use a sample rate of 8 kHz. [10]

## 1.3 Techniques of Voice Security

An interesting question that may appear during the lecture of this document is the matter of:

**Is the developed application a scrambling or an encryption system?**

Definition:

> “A **scrambler** is a device that transposes or inverts signals or otherwise encodes a message at the transmitter to make the message unintelligible at a receiver not equipped with an appropriately set descrambling device. [13]
>
> Whereas **encryption** usually refers to operations carried out in the digital domain, scrambling usually refers to operations carried out in the analogue domain.” [14]

This definition from the Wikipedia Encyclopaedia leads me to the realisation that the developed system even though it is clearly digital its results are more of an analogue scrambling than an encryption. Even so a clear differentiation can not be made.

### 1.3.1 Scrambling

The ideas for techniques for voice Security are not quite new in general; there are two options which are, in literature, often referred to as analogue scrambling or digital encryption. Where in fact this is not always a valid characterisation, not only because of the in above mentioned, but further more because it is not always obvious how the scrambling techniques could be performed, with affordable strength, with only analogue recourses.

#### 1.3.1.1 Time domain scrambling

A common technique is to record voice data for some time and cut it into small frames. These frames can then be transmitted in a different time order depending on a secret code that provides a different not understandable signal since the fragments of the spoken words are not in the right order anymore. The longer the sampled data is and the smaller the frames become, the higher the scrambling rate can be which makes the system more secure. For example: 1 sec divided into 10 frames results into ca 3,6 e$^3$ theoretic possible scrambling options.

On the other side the signal still consists out of the same frequencies like before, which makes it very easy to recover basic information e.g. if there is a man or woman speaking. Also the recording of the data leads into a time delay which is clearly not wanted for a real-time system. Furthermore, a cryptanalysis is possible by performing a minimal distances search relating the differences at the beginnings and endings of the frame spectrums as described in reference [4].

#### 1.3.1.2 Frequency domain scrambling

The other even wider spread idea is to simply invert the frequencies of the voice. This will change high frequencies to low ones and other way around, turning the voice into something sounding like a familiar Disney World figure. This technique is also often called voice or speech inversion because the signal is inverted around the splitting frequency. Speech inversion is the most used method of speech scrambling. One of the problems of all the inversion techniques is that fundamental characteristics of the voice signal are not significantly changed which makes them always vulnerable. There are three types to differentiate:

**Base-band inversion**

Base-band inversion is often also called “phase inversion”, with this technique the spectrum is inverted at a single preset never changing frequency (split point). The descrambling is obviously very simple, if you take the scrambled input and invert it around the same frequency used to scramble you obtain the original again (see Figure 7).

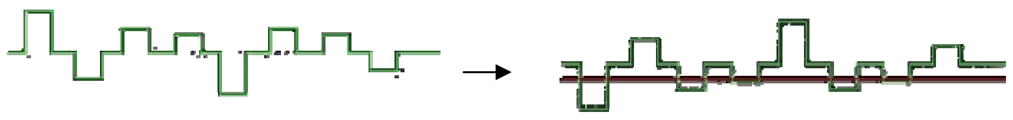


**Figure 7: Base Band scrambling with one split point [14]**

Because the split point never changes it is a very insecure technique. Motorola had for some time a product under the trademark "Secure Clear" on the market. [14]

Since the split points never changed, manufacturers had often a stock inversion points. In Table 1 some popular manufacturer communication systems are listed with their split point.

| Manufacturer | Carrier Frequency |
|---|---|
| Motorola & Selectone | 3.496KHz |
| MX-Com | 3.333KHz |
| MX-Com & Selectone | 3.339KHz |
| MX-Com & Selectone | 3.023KHz |
| Norcomm | 3.107KHz |
| Transcrypt & Selectone | 2.718KHz |
| Selectone | 3.729KHz |
| Selectone | 2.868KHz |
| Selectone | 2.718KHz |
| Selectone | 3.196KHz |
| Selectone | 2.632KHz |

**Table 1: Split points of popular Manufacturers [14]**

### Variable-band scrambling

Variable-band scrambling or “rolling phase inversion” is basically what was described in the history part as one of the first scrambling techniques. The signal inverted around a constantly varying frequency, making decryption possible only if you know how the steps for the frequency change are [14]. The time synchronisation is made by the repeated clicking sounds before the inverting frequency changes. That makes it clearly more secure than base band scrambling and may stop an amateur eavesdropper but it’s far away from being considered secure, as you can find the split frequency rapidly with a frequency look loop device.

**Split-band inversion**

Split-band inversion is an approach for more security in base band inversion. Instead of a single inversion of the complete band the spectrum is divided and base band inversion is processed to both parts before they are recombined. This can again be combined with rolling phase inversion and also the split point can be changed randomly to achieve more security. However, also this technique does not increase the security level significantly since it brings along the same vulnerabilities of insecurity as mentioned before. [14]

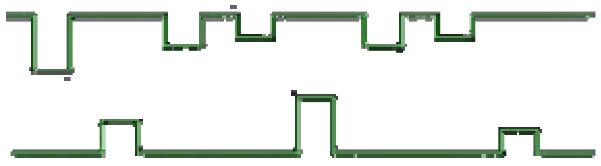


**Figure 8: Inversion of the divided signal out of Figure 7 [14]**

#### 1.3.1.3 Amplitude modification

Of course there is also the option to modify the amplitude of the signal but since it does not really change the signal and is hard to employ it is practically not used, with the exception of pause filling.

As we have seen, the different voice scrambling techniques are a quite insecure way of producing voice security if you use them isolated, but in combination they can bring up quite a notable level of scrambling. Another point to think about is that although “analogue scrambling” techniques have a lot of security lacks, they have one big advantage in common. The resulting scrambled signal is always still an audio signal which makes these techniques independent from the transmission system that they are desired for (assuming a voice transmission system).

### 1.3.2 Encryption

As outlaid before, encryption often refers to digital technologies, in fact, if you hear about data security and encryption in context with modern technologies, you barely talk about something else but digital encryption. “Digital encryption” can be seen as a much stronger method of protecting speech communications than “analogue scrambling”. The big advantage of digital encryption is that it does not matter what kind of signal is encrypted. That makes digital encryption quite powerful because you can create one standard to handle e.g. text, audio, video and every other kind of data. Certainly, digital encryption takes always the same start point, the analogue to digital conversation, however in voice encryption things are a bit different since there are two different ways to go after this.

#### 1.3.2.1 Digital Encryption Standards

Since IT security became a growing market in the past, there are a lot of different standards to choose from. Here is a list of some famous ones:

- DES (Data Encryption Standard)
- AES (Advanced Encryption Standard) / Rijndael
- FEAL (Fast Data Encryption Algorithm)
- IDEA (International Data Encryption Algorithm)
- Safer (Secure and Fast Encryption Routine)
- RC5 (Rivest’s Code 5) and RC6 (Rivest’s Code 6) [7]

The theory is always the same, a digital bit stream (e.g. a digitized voice) or block (e.g. digital recorded audio) is being processed through (by) an algorithm which consists mostly out of a number of mathematical trapdoor one-way function[1] that changes, and inserts bits in a way that the encrypted result contains as less readable information about the original as possible. Which means nothing else than: a good encryption will change the original signal so much that none of the original characteristics are preserved anymore [4]. The algorithm usually has same parameters, often referred as key K,

[1] In cryptography a mathematical function is called trapdoor one-way function, if a pre image of f(x) can be computed in polynomial-time with additional trapdoor information [3]

which ensure that the result of the encryption stays unreadable even though the right algorithm but the wrong key has been processed. Figure 9 gives a little impression of the process of the Data Encryption Standard (DES).

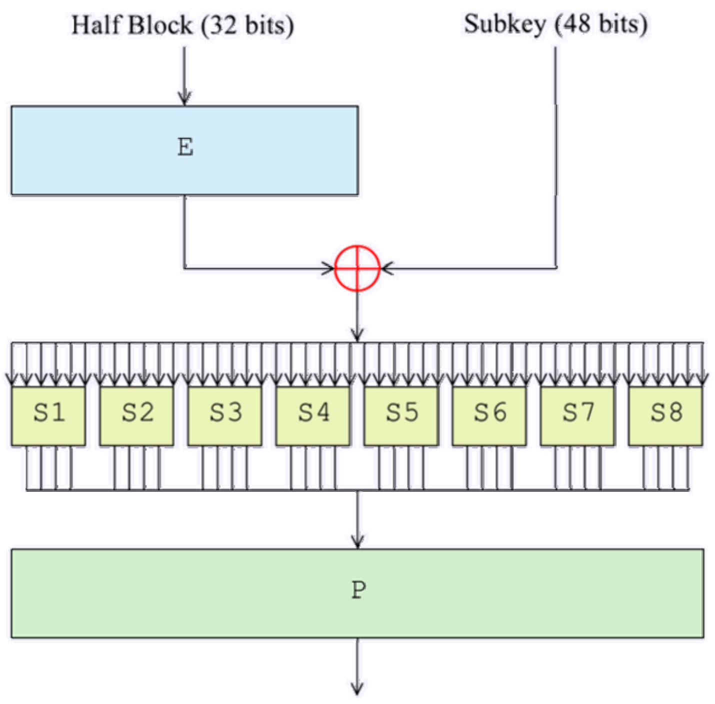


**Figure 9: Overview of Data Encryption Standard [3]**

### 1.3.2.2 Vocoder Encryption

In digital telecommunication systems, voice coders are an often used technique to reduce the required transmission bandwidth. A voice coder is a device which analyzes the sounds generated by the human voice, abstracts parameters from it which can be sent or stored and synthesised at the other end again. By doing this vocoder dramatically reduces the amount of information needed to store or transmit speech. Since the late 1970s, vocoders have been implemented using linear prediction which is a way of representing the spectral envelope, a technique leading into linear predictive coding (LPC). However assuming that a cryptographically algorithm is applied to the parameters before they are sent, the result is a clearly encrypted voice signal [15]. Several vocoders have been invented e.g.:

- LPC-10, FIPS Pub 137,
- Code Excited Linear Prediction, (CELP), Federal Standard 1016,
- Continuously Variable Slope Delta-modulation (CVSD),
- Mixed Excitation Linear Prediction (MELP),
- Adaptive Differential Pulse Code Modulation (ADPCM), former ITU-T G.721. [15]

Digital encryption has the big advantage that there are little possibilities for cryptanalysis compared to analogue scrambling. Except some advanced mathematical approaches, there are only brute force attacks available which are usually not in polynomial time[2] computable tasks.

But on the other side because encryption is a digital process, it can be critical to ensure that the encrypted data arrives correctly at the receiving end if the transmission channel is not sufficiently reliable. If the data is corrupted, it will not be decrypted correctly causing degradation in voice quality. Furthermore, if data are lost (or added), the encryptions may lose synchronisation and communication will be lost. [4]

[2] A polynomial-time algorithm is an algorithm whose worst-case running time function is of order $O(n^k)$ where n is the input size and $k \geq 0$ is a constant. [3]

# 2 Project Overview

**The following chapter will give you a brief overview of the different facets of the developed real-time voice scrambling system. It is meant to be an introduction to get an impression of the complete system that is going to be explained detailed in the following chapters.**

The idea for the system was based on the thesis of Pere Salvadó Lloveras [7] who developed a MatLab program that scrambles the frequencies of a voice, and offers a graphic user interface (GUI) to command and analyse the data. The disadvantage of the program was that, due to MatLab, it was slow and it could only scramble files and no data streams from the soundcard.

Based on that the requirements for this thesis were:

- programming the project of Pere Salvadó Lloveras in C,
- getting the audio directly from the sound card or from a wave file,
- operating in real-time,
- operating as a full duplex communication system,
- implementing the software for optional flexible use e.g. DSP or cross-platform usage.

The general idea is to obtain a system that works basically like Figure 10. Where the setup for a one way communication system is shown, for a full duplex communication is of course a second channel necessary.

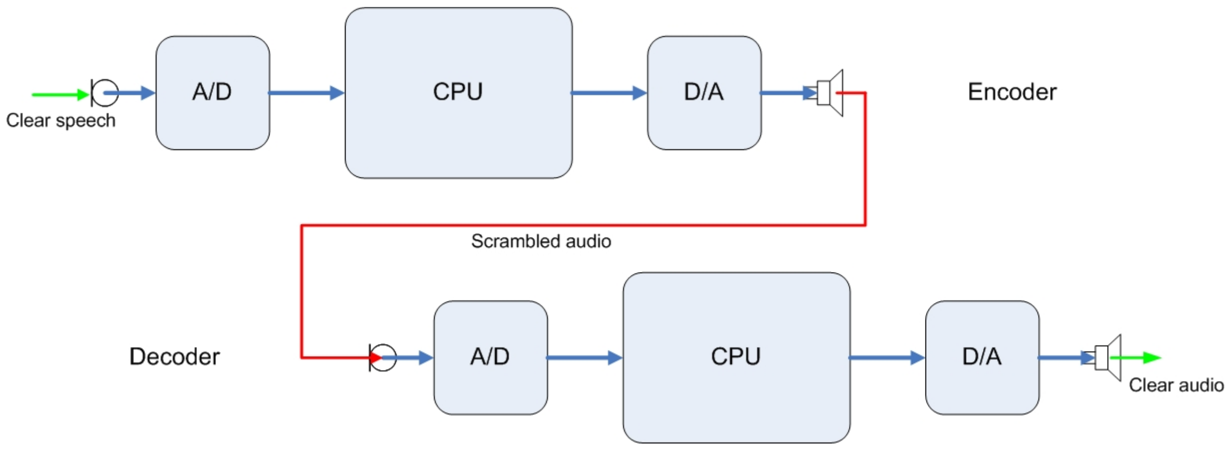


**Figure 10: Basic System Setup**

The basic scrambling code is generally designed to run in real time on any kind of processor. The scrambling code requires simply a digitalised audio stream and returns digital data which has to be converted back into analogue waveforms through a D/A converter. Since the software is primary supposed to be a computer application, which naturally makes it much easier to handle and to develop, the scrambling code has to be embedded into the computer environment which handles the audio data (see Figure 11).

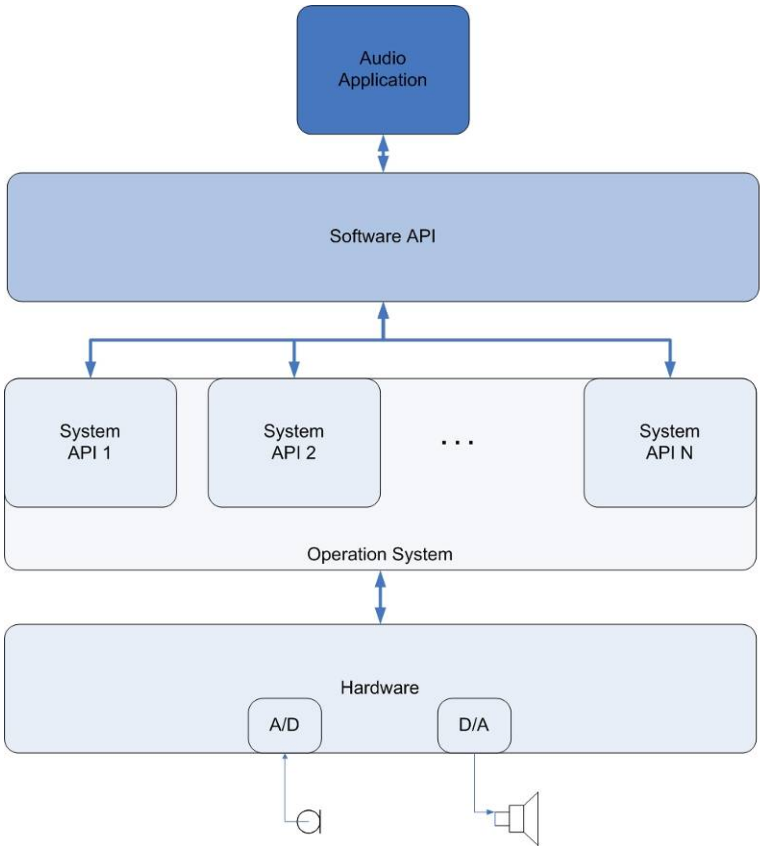


**Figure 11: Architecture of a normal computer**

The sound card gets and returns the audio data from and to the microphones and speakers through its AD/DA converter. The software access to the sound card is guided by varies application programming interfaces (API), which differs within and from platform to platform.

For a cross platform application, one that works one different platforms or OS, it is necessary to be able to use all these different API's. In this case an additional API a fully software orientated API is used to create one access point to interface a number of different system API's. The used API is PortAudio which will be described in chapter 5.3.

Additionally, the audio signal is also supposed to be recorded and played in a raw format including the scrambling process. That of course makes the development much easier.

The approach for the voice scrambling system, developed in this thesis, is quite different from the basic approaches described in chapter 1.3. The idea is a kind of frequency scrambling technique, but in this case, only to scramble sub-bands of the original frequency itself and not adding any extra signals or multiplexing the frequencies on to other signals. While most of the techniques, described earlier, can be strictly differentiated between analogue and digital signal operation, in this case the classification might be a bit confusing because the result of the scrambling process could appear like a analogue technique, but in fact the process itself is clearly digital and can only be achievable with digital signal processing. This is also the reason why I will refer to the process as scrambling instead of encryption.

Figure 12 gives a brief overview of the idea that is being explained detailed in the following chapters (see Appendix A for a bigger version).

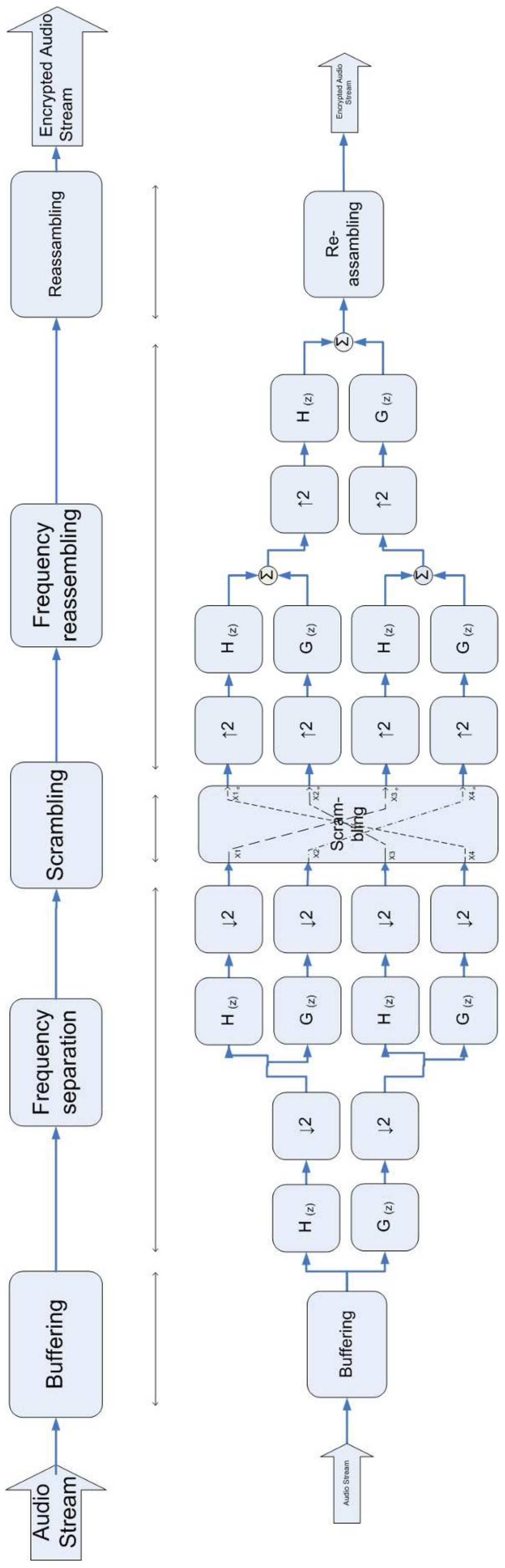


Figure 12: Overview of the voice scrambling system

Most important are the blocks in the middle where the frequencies of the voice signal are being **separated** through filters and down-sampled into sub-bands, **scrambled** and afterwards again **reassembled** passing through an up-sampling and filtering process again.

The other important point is the real-time processing. A system is commonly referred to be real-time if the total result of the process not only depends upon its logical correctness, but also upon the time in which it is performed. In a hard or immediate real-time system, the completion of an operation after a certain deadline is considered useless. A soft real-time system on the other hand will tolerate such lateness, and may respond with decreased service quality. In this case we have a soft real-time system and our deadline would be around 60 ms because the human ear / or brain will not notice a delay smaller than that. [23]

For the development of a real-time system the aspect of framing or **blocking** the incoming data is an existential task. Generally, real-time systems avoid computing the incoming data directly on the stream because the systems commonly do not have the ability for parallel processing and it can not be assured that there are now time differences between the incoming data and the process, i.e. when the continuing sampled data is not processed fast enough by the CPU.

Since the data is usually sampled at interrupt level, storing the incoming stream while the interrupt into blocks and processing those blocks at a time, makes a system stable and gives it time to process the data.

Unfortunately, this causes always a delay which is heavily depending on the amount of samples accumulated in one block and the sampling rate, but of course also on the process itself and on the system performance. On the other hand making the buffer as short as possible is also not a good idea because short buffer could cause easily a buffer overflow or would be rewritten before being processed, so a good balance of the block size and all the mentioned parameters has to be found.

# 3 The voice scrambling system

**The chapter "voice scrambling system" describes the system in detail. A closer look on each block maintained in the chapter before and at the corresponding basic signal theory is taken. Discovering what the reasons are, that the system works like it does.**

The signal that we use to visualise the process consists of a 1 kHz and 2,2 kHz sinusoid wave sampled at 8 kHz which is visible in the spectrogram and in the frequency plot in Figure 13.

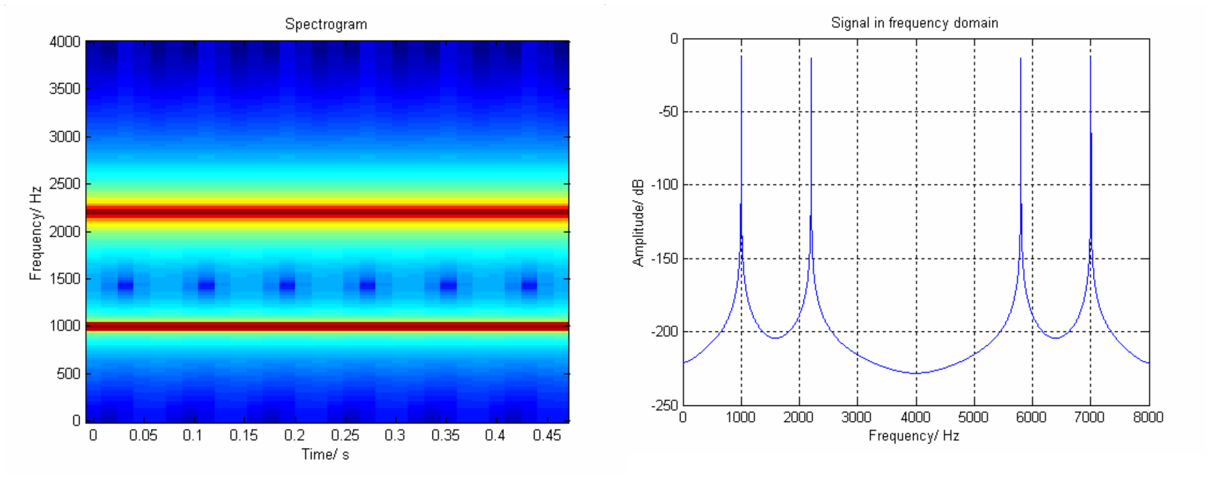


**Figure 13: Spectrogram and Frequency plot**

As commonly known we achieve the image frequencies in digital signal processing through the sampling process, which can be observed at 5,8 kHz and 7 kHz in Figure 13, which those are not important for this system so they will not be shown in the following graphs.

## 3.1 Encoder

The encoder is the device that transfers the clear voice signal into scrambled noise, while on the other side of the communication system there is a decoder located doing the inverse process, in fact for this system the encoder and the decoder do not differ a lot from each other.

### 3.1.1 Frequency separation

As mentioned before, the general idea is to scramble the frequencies of an audio or a speech signal. So the resulting signal sounds different from the original. As seen in chapter 1.2, the human voice consists of different frequencies. To scramble these frequencies, the first necessary thing to do is to cut the original signal into sub-bands which the in Figure 14 drawn process is used for.

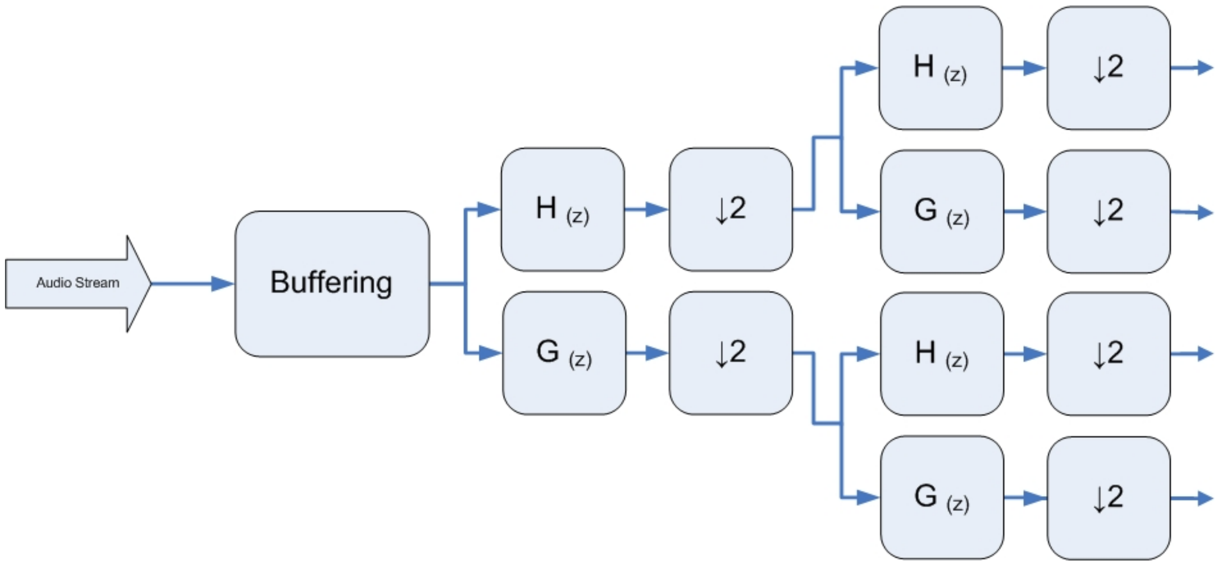


**Figure 14: Overview of the frequency separation process**

Generating sub-bands of frequencies can be easily achieved by filtering the signal into high-pass band (with filter $H_{(z)}$) and/ or a low-pass band (with filter $G_{(z)}$).

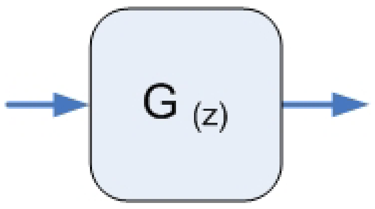


**Figure 15: Low-pass filter block**

The design of the filter will be described and discussed in chapter 5.4.3. Generally every kind of filter would be possible, in this case we use a Chebyshev II low-pass filter with $F_{stop} = 2{,}2kHz$ and a high-pass filter with $F_{stop} = 2{,}37kHz$ and a characteristic like in Figure 16.

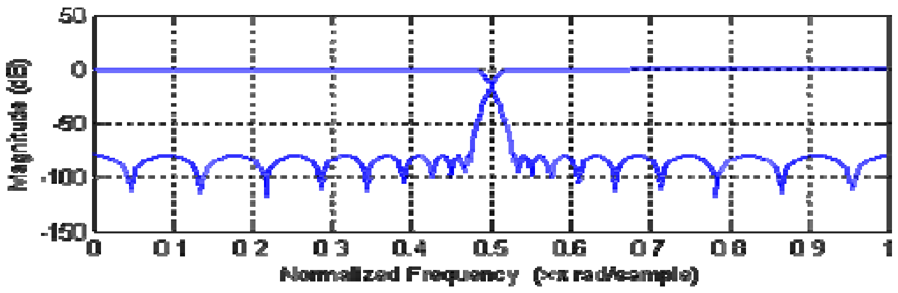


**Figure 16: Low-pass/ High-pass filter characteristic**

Filtering with this low-pass filter will obviously result in a signal which contains only the frequencies in the passing spectrum of the filter as in Figure 17. (The continually shape of the signal can be recognised from the discrete data in Figure 17 where only the first 50 samples of the signal are visualised)

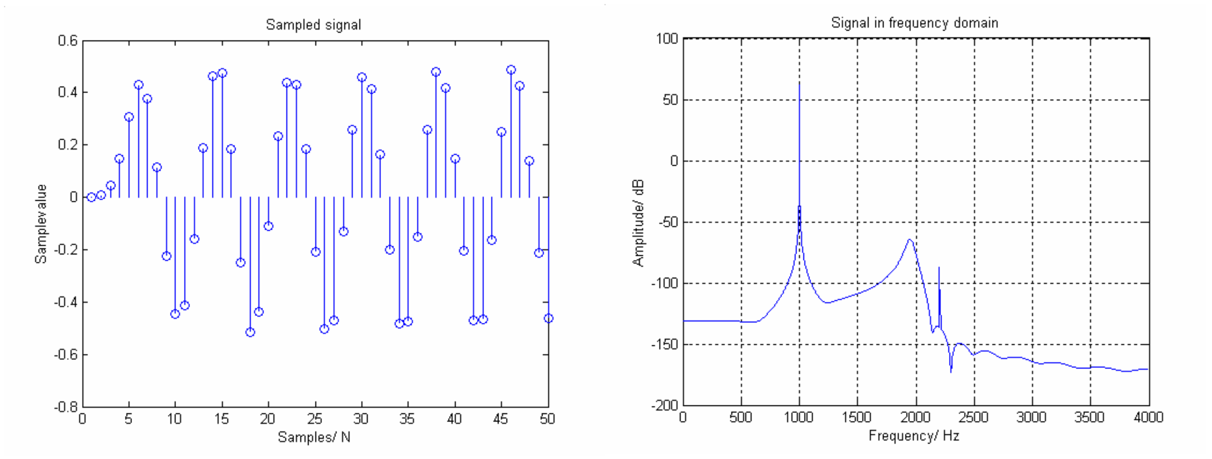


**Figure 17: Discrete- and frequency plot of filtered signal**

Since sub spectrums of one signal can not be reconstructed in any other than the original way, it is necessary to use some additional operations to obtain a signal which contains a part of the information of the original signal, but in turn has the ability to be scrambled in some way.

The manoeuvre of choice is, to down-sample the sub-band.

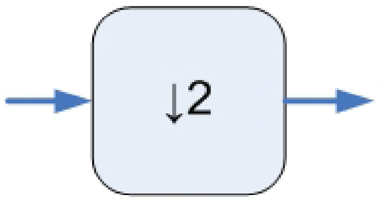


**Figure 18: Down-sample block**

The down-sampling process can be easily performed by taking every second sample out of the signal, which is not a problem in digital signal processing. Doing this, we reduce the size of the signal to its half but even more important from the signal's theoretical point of view, the frequency is shifted to its double; compare Figure 19:

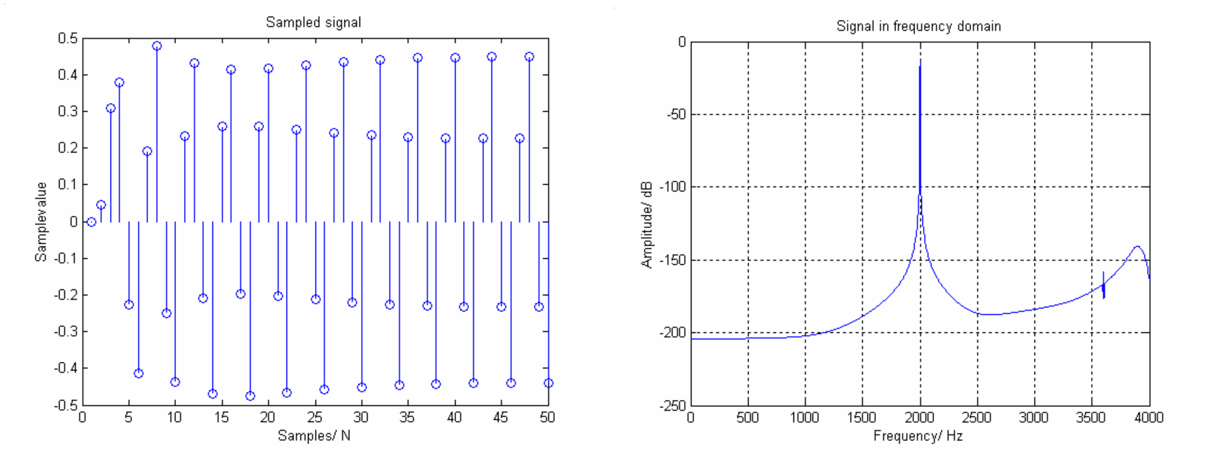


**Figure 19: Discrete- and frequency plot of down-sampled signal**

This hole cascade of operations like shown in Figure 14 must be done N times to obtain a variety of scrambling options. N should be high which again extends the processing time. In this system between 2 and 16 sub-bands are available (Figure 12 shows 4 sub-bands). The results are the sub-bands and with those are the scrambling objectives.

### 3.1.2 Scrambling

With the result of the sub-band generation we manage to have N items that can be scrambled. The scrambling process is the actual process of making the signal any different from the original.

To obtain a scrambling code it is necessary to generate N x N matrix which characterises the scrambling directions.

$$ScramblingMatrix = \begin{array}{cccc} \mathrm{LL} & \mathrm{LH} & \mathrm{HL} & \mathrm{HH} \end{array} \atop \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \begin{array}{c} LL \\ LH \\ HL \\ HH \end{array} \quad \textbf{Equation 1}$$

Where the rows in Equation 1 represent the path for the generation, and the columns represent the path for the reconstruction of the sub-bands, which means to where they are scrambled. Of course the matrix in Equation 1 is the identity and

$$InvMatrix = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \textbf{Equation 2}$$

Equation 2 is the entire inversion of the signal.

Theoretically there are:

$$x = N^N - 1 \quad \textbf{Equation 3}$$

possibilities for the scrambling result, but not all of the possible combinations make a good valid code. There are two main matters that have to be looked after.

- The generated matrix must have a minimum distance to the original one.

$$BadMatrix1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{Equation 4}$$

- The rows should not be in the same order like in the original matrix

$$BadMatrix2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{Equation 5}$$

Otherwise the scrambled spectrum is too similar to the original one and can be understood easily. To make up a unique matrix a user defined key is employed as a generator. Two different ways are possible:

- Calculating the minimum difference to the identity matrix and to the before calculated row. Whereas, a problem with deadlocks could occur.
- Providing a look up table with the best available combinations, this is addressed through the user defined key. This probably reduces the possible number of matrixes and with that the uniqueness.

After generating the matrix the scrambler has just to assure that the sub-bands are shifted to the desired rebuilding path.

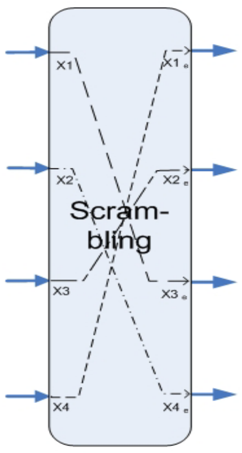


**Figure 20: Scrambling block**

The scrambling process has no direct impact on the signal processing but of course, it determines which path or which sequence of filters has to be used in the reconstruction branch.

### 3.1.3 Frequency reassembling

During this final action, the real frequency shifting is performed. Figure 21 illustrates that to rebuild the signal, similar operations as in the frequency separation are used, but for different signals since they have been scrambled before.

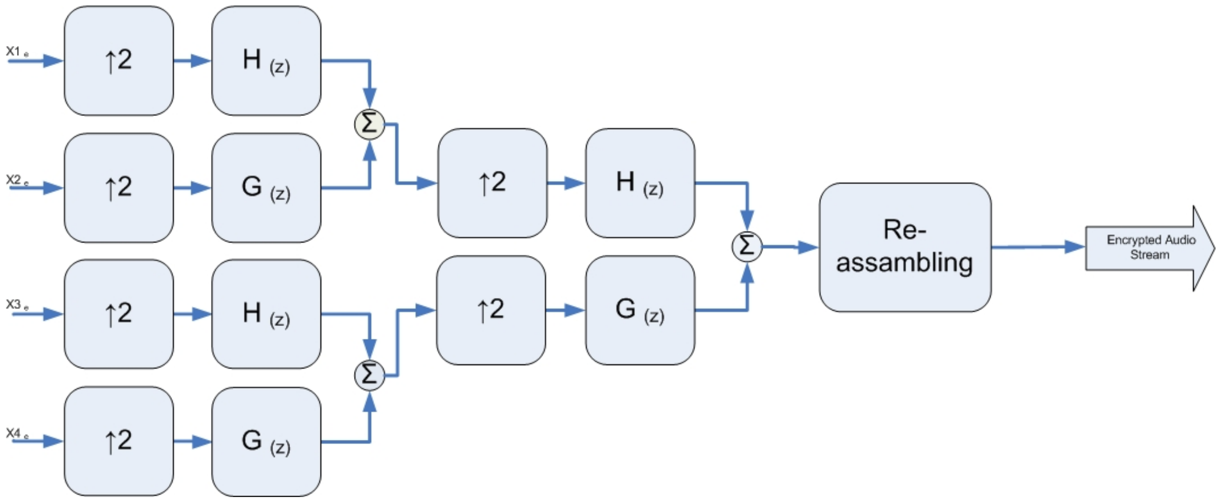


**Figure 21: Overview of the frequency reassembling process**

Although the graph looks quite similar, it has in terms of the signal theory a total different meaning. In this case we of course do not sample down again, but up.

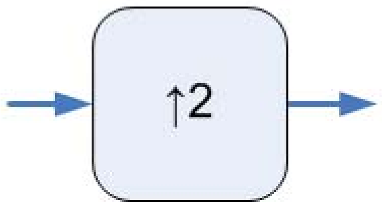


**Figure 22: Up-sample block**

Up-sampling is achieved as easy as down-sampling, simply by adding zeros after every sample. That causes the extension of the signal to its old length and, further more important, an image frequency as to be observed in Figure 23.

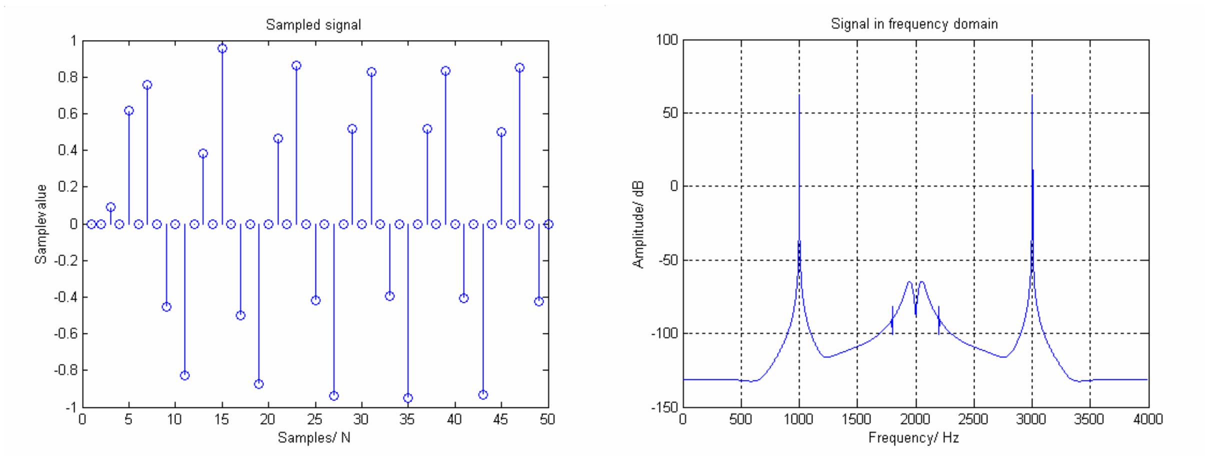


**Figure 23: Discrete- and frequency plot of up-sampled signal**

In D/A conversion one would now use a low pass filter at 2 kHz to reconstruct the original signal and to delete the image frequencies out of the spectrum. In this case we use exactly this to make a shift of the frequency. Nevertheless we need to reconstruct the signal because through the inserted zeros we have a signal full of sharp steps.

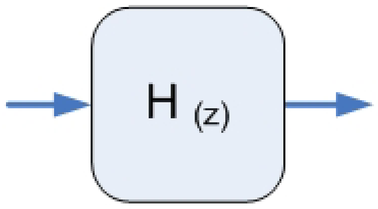


**Figure 24: High-pass filter block**

However, by high-pass filtering we obtain the mentioned smooth reconstruction and make **use of the image frequency instead of the original one** and obtaining an inverted shifted signal.

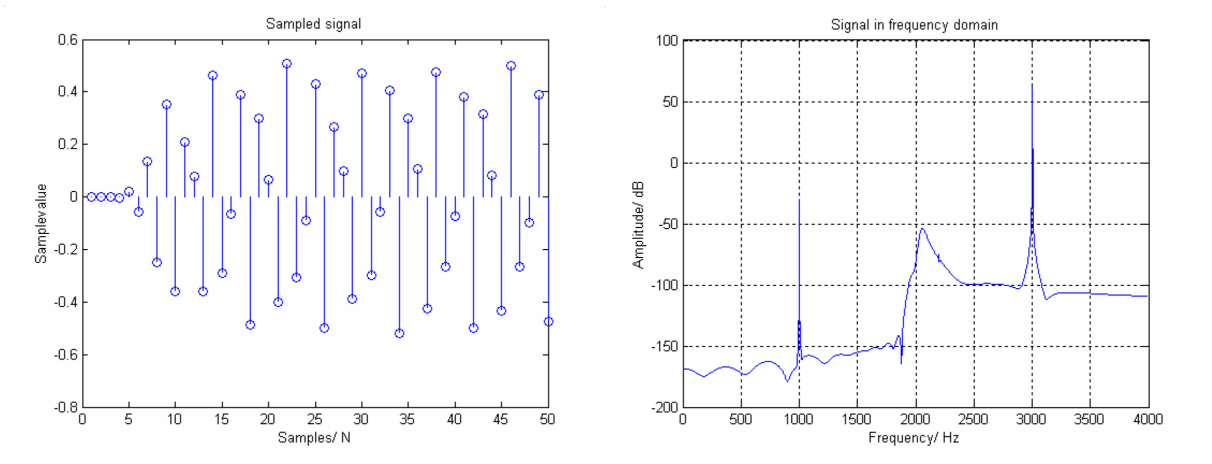


**Figure 25: Discrete- and frequency plot of filtered signal**

We do not have to worry about upper image frequencies (above 4 kHz not ins Figure 25) because they are going to be filtered out by the reconstruction filter at the very end at the D/A conversion.

Of course, one could argue that the down-sampling is an unnecessary step because one could directly set every second sample to zero instead of down-sampling it first, but there are two reasons why it is presented here with this step in between:

- it makes it easier to understand,
- depending on how you implement the scrambler, it may be easier and save memory if you work like that, i.e. if you implement strictly in blocks.

After all the signals are being mixed again always two at a time for the same number of times they have been cut into sub-bands.

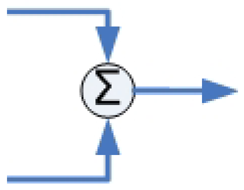


**Figure 26: Sum block**

After that is done, finally one encoded stream is being returned from the process.

With that we have the complete encoder explained. Figure 27 illustrates the complete encoding process as described before plus the already mentioned buffering. As seen in the description of the scrambler, there is not a very satisfying scrambling possibility with four sub-bands so it should not be used in practise but it takes good advantage for illustrating.

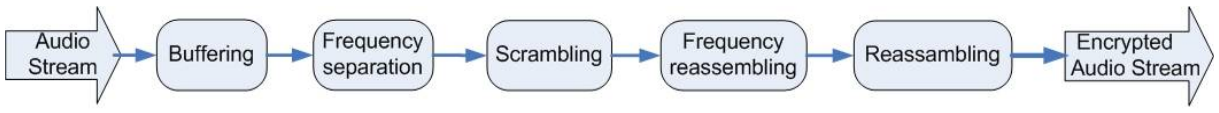


**Figure 27: Encoder/ Decoder**

The two main aspects of the encoder that have to be balanced are the:

- number of sub-bands,
- processing time,

each of those have a direct impact on one and the other, since a double number of sub-bands also creates three times the number of filter processes.

## 3.2 Decoder

The decoding process must have the same structure as the encoding process since we are talking about a symmetric scrambling technique.

The only difference is the scrambling part, where the matrix has to be inverted by transposing it as to be seen in Equation 6.

$$DecryptionMatrix = [EncryptionMatrix]' = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}' = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

**Equation 6**

# 4 Audio access

**This chapter gives a little introduction into the different possibilities of accessing audio data rather from a file or from an API. Special attention is put onto some software APIs that combine different system audio APIs.**

To develop a real time audio application it is necessary to obtain the audio data directly from the hardware respectively from the operation system (OS). Since there are a lot of different kind of hardware with their drivers and also a lot of different operation systems there are multifarious possibilities to access the audio data. These access points are commonly called application programming interface (API), those are to be used if the data can not be read from a file.

## 4.1 WAV Files

Beside the direct access of the audio data from the hardware it is also useful to be able to use recorded files, in fact this might not be necessary for real time systems but at least for development and testing it is an essential source. WAV is a short form for a Waveform audio format; it is a standard data format for storing audio data. The WAV format is a variant of the RIFF bit stream format method for storing data in “chunks”, it is the main format used on Windows systems for raw audio. To use WAV files it is necessary to read or write the WAV file header that is shown in Figure 28.

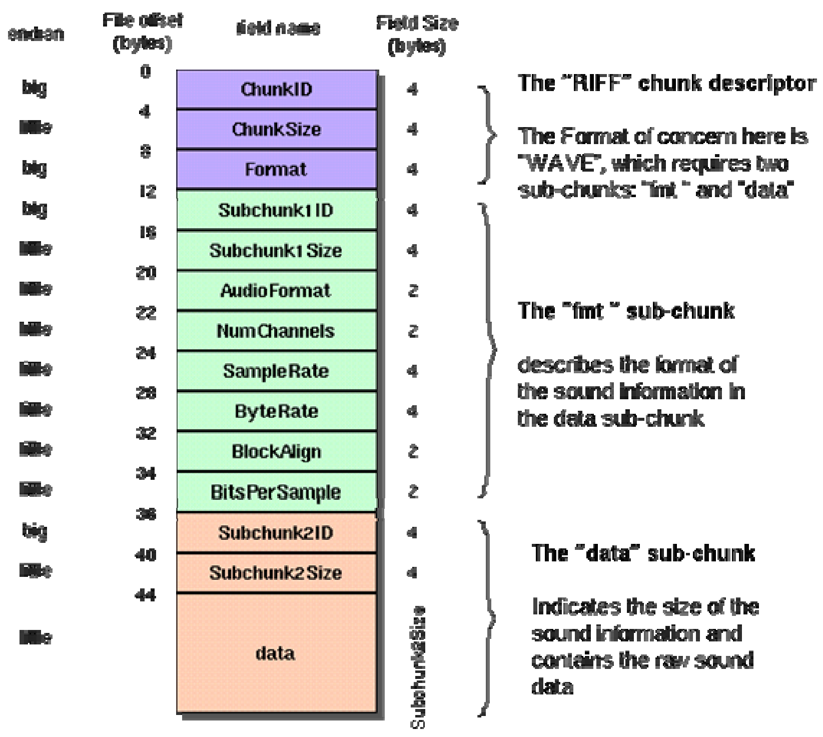

endian
File offset (bytes)
field name
Field Size (bytes)
big
little
ChunkID
ChunkSize
Format
Subchunk1 ID
Subchunk1 Size
AudioFormat
NumChannels
SampleRate
ByteRate
BlockAlign
BitsPerSample
Subchunk2ID
Subchunk2 Size
data
Subchunk2Size
The "RIFF" chunk descriptor
The Format of concern here is "WAVE", which requires two sub-chunks: "fmt " and "data"
The "fmt " sub-chunk
describes the format of the sound information in the data sub-chunk
The "data" sub-chunk
Indicates the size of the sound information and contains the raw sound data

Figure 28: The canonical WAVE file format [16]

A Wave file can be read like every other file. It only has to be taken care that the first 36 bytes are interpreted as the header, respectively are written as the header.

## 4.2 System Audio APIs

Due to the fact that every operation system works differently, the range of different audio API is big. Attainably manufactures of audio hardware and others sometimes even develop their own API which contributes to the number. These ones are the system APIs because those are somehow connected to the hardware or to the operation system. It is almost impossible to give a complete list of audio API. Nevertheless, here are some of the most important ones listed.

## Microsoft

Since Microsoft is probably still the most imported deliverer of operating system the main idea is explained on its example. Note that the basic function also rules for other ones.

Figure 29 shows the basic structure of a soundcard driver for the Windows operating systems. The middle area represents what is commonly called the "driver". That is not completely right since the upper parts of it are usually Dynamic Link Library (DLL), which are software modules that are part of the operation system or single software pieces which are loaded and used by the application on request. The audio software is furthermore accessing the API, which in turn accesses the driver and that one accesses the hardware. This means that the API is the interface between the audio application and the driver of the audio hardware.

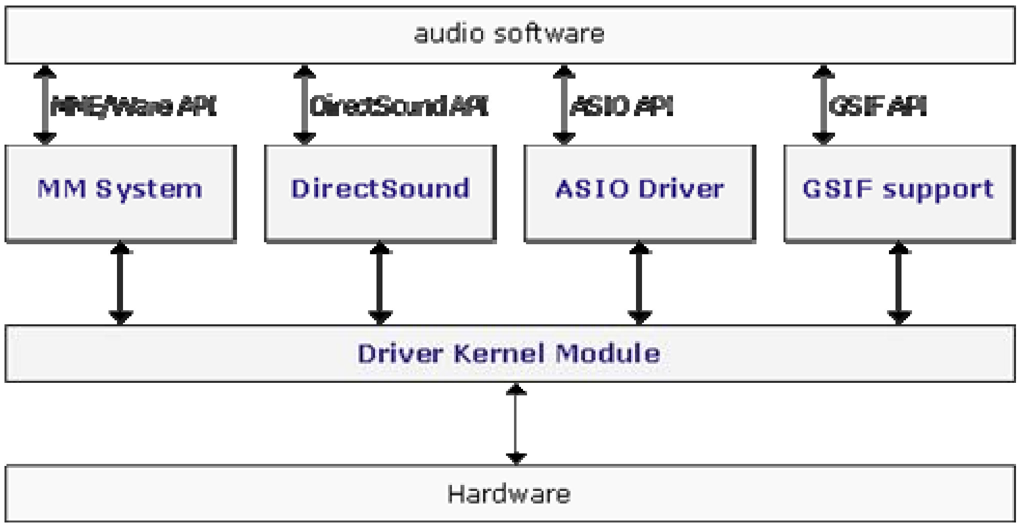


**Figure 29: Model of MS Windows audio APIs [17]**

- Windows MultiMedia Extensions (default for PCs)

The MultiMedia Extensions (MME) of the Microsoft Windows OS is probably the most used audio API. It was first integrated in the Windows 3.0 based "Windows with MultiMedia Extensions 1.0" (not very well known) but it contained in later releases as Windows 3.1 and 3.11 and is still, with small modifications, present in today’s versions. MultiMedia Extensions does not only specify the OS version but furthermore the API to access soundcards.[17]

- DirectSound

  DirectSound is part of the DirectX project which contains also DirectShow, DirectDraw; DirectSound 3D. It was developed by Microsoft to give especially the game programmers more flexibility in the development of applications. For that it provides a direct interface between applications and the sound card. The main features are:

  - passing audio data to the sound card,
  - volume control, recording and mixing sound,
  - adding effects to sound e.g. reverb, echo, flange,
  - using hardware controlled buffers for extra speed,
  - positioning sounds in 3D space,
  - capturing sounds from a microphone or other input, and
  - controlling capture effects during audio capture,
  - ability to play multi-channel sounds at high resolution,
  - allowing several applications to conveniently share access to the sound card at the same time.
  - It also provides the ability for games to modify a musical script in response to game events in real time, e.g. the beat of the music could quicken as the action heats up.

  If a driver is not optimized for DirectSound, Windows will automatically emulate DirectSound output using the MME devices.

- Core Audio APIs in Windows Vista

  Windows Vista has a new API which was not available in earlier versions of Windows. The core Audio combines the access point for the higher-level APIs such as DirectSound and waveXxx but can also be directly accessed. In Windows Vista, a new set of user-mode audio components provides client applications with improved audio capabilities. These capabilities include the following:

- low-latency, glitch-resilient audio streaming,
- improved reliability (many audio functions have moved from kernel mode to user mode),
- improved security (processing of protected audio content takes place in a secure, lower-privilege process),
- assignment of particular system-wide roles (console, multimedia, and communications) to individual audio devices,
- software abstraction of the audio endpoint devices (for example, speakers, headphones, and microphones) that the user manipulates directly.[18]

**Unix (Linux, FreeBSD, Solaris)**

- Open Sound System (OSS)

  The Open Sound System is the main sound API for UNIX like systems. It is available in at least 11 Unix OS and sometimes even part of the kernel. The main aim of OSS is to give access to the hardware driver which works for it as a collection of those. With that it is possible to write an audio application that works with any sound controller hardware.

- Linux ALSA

  Advanced Linux Sound Architecture is the evolution of OSS. Since Linux version 2.6 it replaces OSS by default, although a backwards-compatibility layer exists. The main innovations were:

  - automatic configuration of sound-card hardware,
  - possibility for multiple sound devices in a system,
  - low-latency.

- Linux Jack

  Linux Jack is mainly designed for professional audio applications, the main focuses are on:

  - synchronous execution of all clients,
  - low-latency operation.

  It is written for POSIX conformant operating systems such as GNU/Linux and Apple's OS X.

**Macintosh**

- Sound Manager for OS 7-9

  The Sound Manager is more or less the equivalent to the Windows MME but not so easy accessible. It controls the production, recording and manipulation of sounds on Macintosh or Apple computers.

- Core Audio for OS XBurk

  **Core Audio** is real API for audio propose on Apple's Mac OS X operating system. A big advantage is that it includes an implementation of the cross-platform Open AL library which we are going to talk about in chapter 4.3. Its primary goals are:

  - high-quality, superior audio experience for Macintosh users,
  - easier programming for developers.

**ASIO**

**Audio Stream Input/Output** (**ASIO**) is an API defined by Steinberg which is a major company for professional electronic audio devices. It connects the audio application directly to the Steinberg hardware, bypassing the operation system and with that saving latency allowing direct high speed communication with audio hardware. It is primary developed for Steinberg Hardware in Microsoft environment, but there are also some approaches with Linux Jack. Mac OS normally do not have the problem of high latency. Steinberg hardware also often brings along a very easy to use programming environment, which makes it first choice for professional musicians and audio engineers.

## 4.3 Software Audio API

The big amount of Audio APIs makes it quite difficult to develop a cross platform application because for that it is necessary to combine as many as possible API accesses to achieve an application that runs on any “home computer”. Fortunately, there are some software pieces that do exactly that. Those software APIs, because they connect software to software, work like an over layer on top of the different possible Audio APIs. There are different ones to choose from:

- **Pure Data**

  PD is a real-time graphical programming environment for audio, video, and graphical processing that comes as a data flow orientated programming language itself. Additionally it gives the user also the facilities of a visually development environment. It is mainly used to produce interactive multimedia applications etc. PD can be extended through new internal blocks but also through external classes written in C, C++, Ruby etc. Furthermore it is developed as cross-platform application; versions exist for Win32, IRIX, GNU/Linux, BSD, and MacOS X.

- **Synthesis ToolKit**

  STK is basically a set of open source audio signal processing and algorithmic synthesis classes written in the C++. It was developed by Perry Cook at Princeton University and Gary Scavone at CCRMA (Stanford University). It is supposed to serve as a cross-platform, real-time controlled, easy of use application also with an educational background. It contains both low-level synthesis and signal processing classes and higher-level instrument synthetisation classes.

- **CLAM**

  CLAM (C++ Library for audio and music) is a software framework for research and application development in the field of audio and music. It is fully object orientated C++ Library that was mainly developed by the Gruop de Recerca en Tecnologia Musical (MTG) de la Universitat Pompeu Fabra en Barcelona. It brings all the basic necessary tools for the analysis, synthesis and transformation of audio signals but furthermore, it serves with its GUI as a software framework for research use. [19]

- **OpenAL**

  The Open Audio Library is probably one of the most famous representatives of this class of software. It is a cross-platform 3D audio API and because it has big advantages in 3D it is very appropriate for use with gaming applications.

  OpenAL uses high level abstraction algorithms to simulate the feeling of surround and 3D sound to the listener. Unfortunately the recording direct access to the hardware is a bit more complex. [20]

- **PortAudio**

  PortAudio is another free, audio I/O library, it is meant to have cross platform abilities (including Windows, Macintosh (8, 9, X), Unix (OSS), SGI, and BeOS), and a very easy to use API for recording and/or playing. PortAudio is part of the PortMusic project which additionally contains PortMidi and PortSound Files. It is published under the General Public License (GPL), it is meant to help for a better exchange of audio application between developers on different platforms. PortAudio is a simple API and does not bring any functions like mixing, filtering or analysing.

Some of the facts of PortAudio are:

- data types: 16 and 32 bit integer and 32 bit floating,
- full duplex or half duplex,
- enumerating available devices and querying them for properties such as available sampling rates, number of supported channels and supported sample formats,
- cross platform (Microsoft Windows™, Macintosh™ ,Linux),
- platform neutral interface to real-time audio streaming services in the form of a 'C' language. [21]

Most of these APIs are licensed under the General Public License (GPL) (see Appendix B) so they are free to use within these regulations. For sure there are a lot more APIs like these, but for several reasons these were the ones to choose from for this project. The following table helped making the best choice.

| | CLAM | OpenAL | Pure Data | STK | PortAudio |
|---|---|---|---|---|---|
| Audio Output | + | ++ | + | ++ | ++ |
| Audio Input | + | - | + | ++ | ++ |
| Filter and effects included | ++ | + | ++ | + | - |
| Analysis tools included | ++ | - | ++ | + | - |
| Synthesis possible | + | + | + | + | - |
| 3D sounds possible | - | ++ | - | - | - |
| Real-time playback capability | + | + | + | ++ | ++ |
| Overhead | ↑ | = | ↑ | ↓ | ↓ |

**Table 2: Comparison of software APIs**

- not integrated; + integrated; ++ integrated and easy to handle

# 5 Project implementation

**Based on the theory and techniques outlaid in the previous chapters and on the MatLab code of Pere Salvadó Lloveras a real-time voice scrambling application was programmed. This chapter describes the actual implementation of the system and gives focus on the most important parts.**

## 5.1 Overview

Obviously the theory of the last chapters has to be simply converted into a programming language, but to achieve a duplex real-time encryption application it is also necessary to obtain the audio data directly from the hardware respectively from the operation system. Using one of the above mentioned software APIs makes this easier but of course not trivial since this API also requires controlling. For the duplex requirements some hardware configurations have to be done like described 5.2 and the source code has be organised in this case it consists out of two main sections:

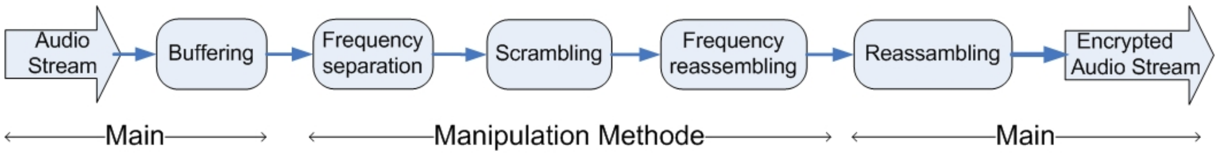


**Figure 30: Section of the source code**

the *main* loop with the audio control as described in 5.3. and the *manipulation* process as to be seen in 5.4. The actual sub-band generation with the frequency separation and down-sampling is part of the *manipulation* and performed in the *cut* method. Please note that because of space reasons the following are abstracted PFCs which do not show every detail of the real implementation but give you a good overview of the general idea. Also some functionality is left out to make it easier to understand.

## 5.2 Duplex communication

A duplex or to be more detailed a full duplex system is a system consisting out of two connected devices which can communicate with one another in both directions at the same time. Duplex systems are employed in nearly all communications networks. [22]

To achieve full duplex capability in the present system it is necessary to use two input and two output channel of the soundcard, which is not a problem for most hardware because they work stereo anyway. Nevertheless the software and the hardware have to be set up for that. The necessary software configurations are discussed in chapter 5.3 while Figure 31 shows how the hardware should be set up with two PCs.

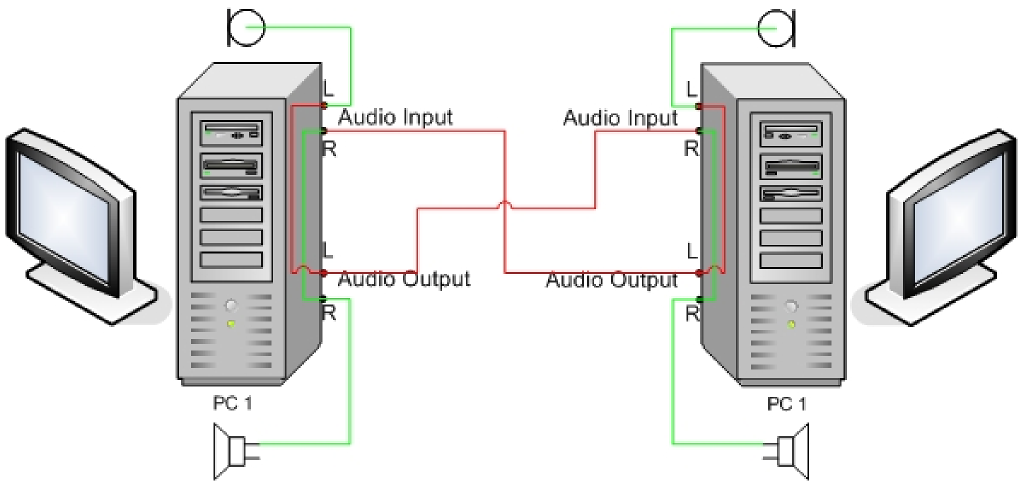


**Figure 31: Setup for duplex communication with two PCs**

## 5.3 Main method and use of PortAudio

As to be seen in Table 2 in chapter 4.3 there a lot of different reasons to choose one or the other software API. In this case the PortAudio Library was chosen for the access to the audio data because it brings the smallest overhead for this application and is easy to use. Nevertheless an existing application out of the thesis of Aitor Pérez Pellitero [8] was accessible. As mentioned before, PortAudio is a capsulation of different system audio APIs. To access the hardware you have to choose one path at a time as Figure 32 illustrates.

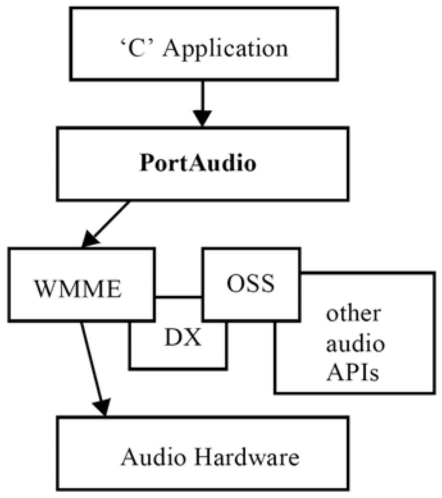


Figure 32: Use of PortAudio

Using PortAudio is pretty easy, the software comes as pure code, which has to be compiled as a dynamic link library (DLL). After doing so the header file has to be included and the DLL has to be linked to the project that desires to use audio data. The handling of PortAudio generates more or less the complete main method as visible in Figure 33.

To achieve a full duplex system the two channel (stereo) function of the soundcard is used. PortAudio presents these two channels like in a WAV file, the channel changes every other block. The first recorded block is the right channel the next one the left and so on. In Figure 33 this can be observed with the encoding and decoding process after each other. This of course alters the requirements of the processing speed of the encoding/ decoding functions because they have to work fast enough to run the process twice within the real-time time limits.

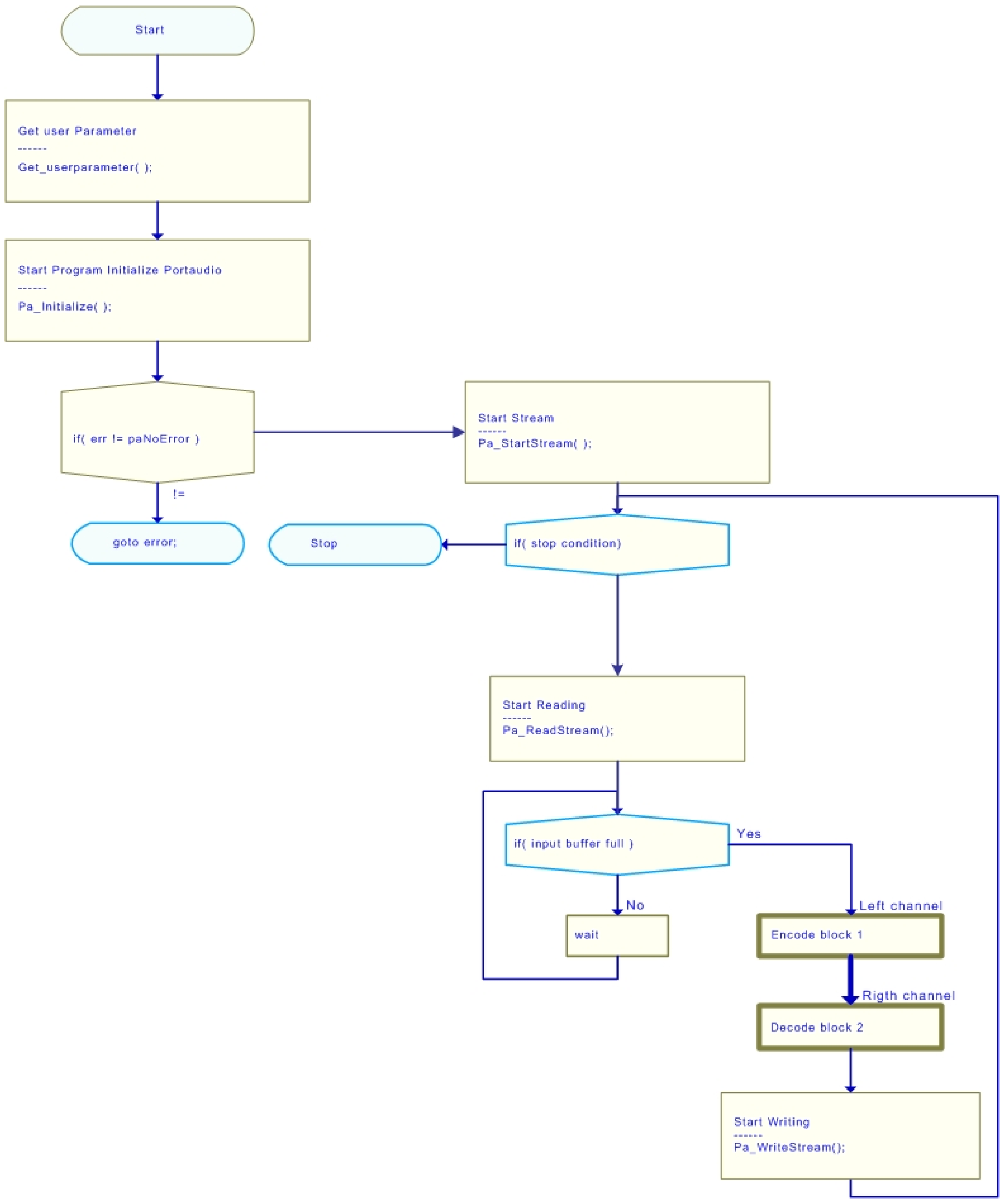


**Figure 33: Abstract PFC of main loop**

In the code itself there is only little to do to run PortAudio.

1. PortAudio has to be initialised.
2. The stream has to be managed (started and later stopped).
3. The manipulation of the audio data has to be performed.
4. The stream has to terminate.

A main part of the initialisation is in the `Pa_OpenDefaultStream` command since in here the basic parameter for the audio are set

```
Pa_OpenDefaultStream( &stream,
                       2,             /* input channels */
                       2,             /* output channels*/
                       paFloat32,     /* data type */
                       44100,         /* Sample Rate*/
                       256,           /* frames per buffer*/
                       Callback,      /* Name of callback function */
                       &data );       /*This is a pointer that will be
                                        passed to the callback function*/
```

[21]

However PortAudio delivers the audio data always in blocks of sizes that are defined like shown above. Furthermore there are two basic ways of using PortAudio. The first and older one is the Callback mode. The following code example gives a little overview of the main loop:

```
#include "stdio.h"
#include "portaudio.h"

int main(void)
{
PortAudioStream *stream;
Pa_Initialize();

Pa_OpenDefaultStream( &stream,
                        2, 2, paFloat32, 44100.0,
                        64, 0, Callback, NULL );

Pa_StartStream( stream );
Pa_Sleep( 10000 );
Pa_StopStream( stream );
Pa_CloseStream( stream );
Pa_Terminate();
return 0;
}
```

In the function `Pa_OpenDefaultStream()` the second to last parameter names the Callback function that is supposed to perform the audio manipulation. This function will be called by PortAudio when ever audio data is waiting to be manipulated or needed. A simple example of a Callback function is shown next:

```
static int Callback( void *inputBuffer, void *outputBuffer
                  unsigned long framesPerBuffer, PaTimestamp outTime, void
                  *userData )
{
float *out = (float *) outputBuffer;
float *in = (float *) inputBuffer;
float leftInput, rightInput;
unsigned int i;
if( inputBuffer == NULL ) return 0;

for( i=0; i<framesPerBuffer; i++ )
      {
      leftInput = *in++;
      rightInput = *in++;
      /* Manipulate */
      *out++ = leftInput  rightInput;
      *out++ = rightInput;
      }

return 0;
}
```

The other option (since version V19) is to use the blocking mode, this is also shown in Figure 33. In this case the process is performed like reading or writing from a file. The manipulation is simply performed after a block was read completely. See the following code:

```
#include "stdio.h"
#include "portaudio.h"

int main(void)
{
PortAudioStream *stream;
Pa_Initialize();
float *Block;

Pa_OpenDefaultStream( &stream,
                        2, 2, paFloat32, 44100.0,
                        64, 0, NULL, NULL );
Pa_StartStream( stream );

Pa_ReadStream( stream, Block, FRAMES_PER_BUFFER);
/* Manipulate */
Pa_WriteStream( stream, Block, FRAMES_PER_BUFFER);

Pa_Sleep( 10000 );
Pa_StopStream( stream );
Pa_CloseStream( stream );
Pa_Terminate();
return 0;
}
```

Both ways have their advantages. Callback: high performance, compatible with all system APIs. Block: easier to use, compatible with party projects.

But for this project the choice fell onto the blocking mode because the performance is high enough and it makes the code far easier to understand.

## 5.4 Manipulation

The manipulation method as shown in Figure 34 takes care of the actual signal processing and scrambling process.

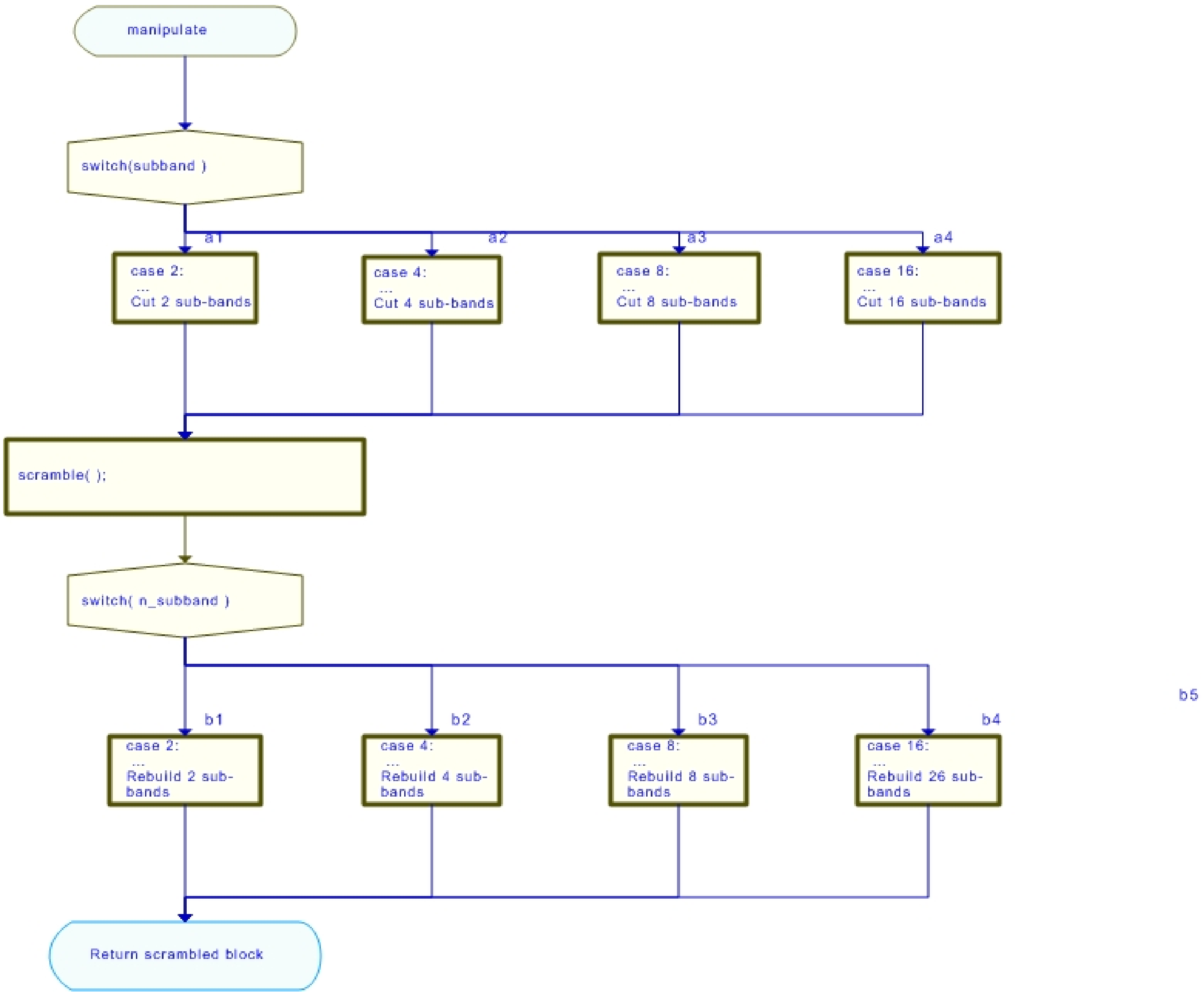

manipulate
switch(subband )
a1
a2
a3
a4
case 2: ... Cut 2 sub-bands
case 4: ... Cut 4 sub-bands
case 8: ... Cut 8 sub-bands
case 16: ... Cut 16 sub-bands
scramble( );
switch( n_subband )
b5
b1
b2
b3
b4
case 2: ... Rebuild 2 sub-bands
case 4: ... Rebuild 4 sub-bands
case 8: ... Rebuild 8 sub-bands
case 16: ... Rebuild 26 sub-bands
Return scrambled block

Figure 34: PFC of the manipulation method

For easier and reuse reasons it is divided into sub-routines, there are the cut, scramble and rebuild methods. The task that has to be performed by manipulation method is to call the sub-routines in the right order in the quantity that the user defined sub-bands require. The manipulate function looks like this:

```
int manipulate(const void *in, void *out,unsigned long length,
                    ScramblingParameters_t *ScramParameter)
```

The function works on a block of audio data that is referenced with the pointer `*in`, the result of the manipulation will be stored under the reference of `*out`, the parameter `length` inherits the length of a block and `*ScramParameter` is an enumeration for the user defined scrambling parameters.

### 5.4.1 Cut/ build method

The cut and build method use basically the same operations but in a different order. As in Figure 35 shown they consist out of a number of loops which process the up/ down-sampling and call the filter routines. The function can be called through:

```
int cut(SAMPLE_TYPE *data,unsigned int start,unsigned int end,
        SAMPLE_TYPE z[])
```

*Cut* separates the audio data in `*data` between `start` and `end` into a high-band and a low-band and memorises the down-sampled result back into same memory in `*data`. That way memory can be saved. `z[]` is an array that keeps the delay line of the filters (see 5.4.3 Filter).

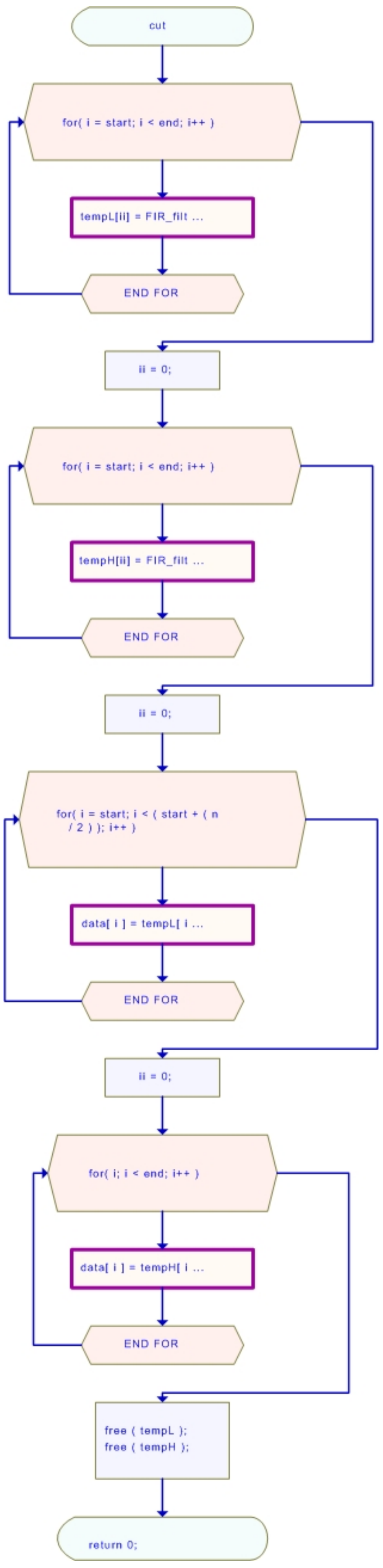


Figure 35: PFC of the cut method

### 5.4.2 Scrambling method

The Scrambling Method takes care of the generation of the scrambling matrix and also shifts the sub-bands so they get rebuild through the path the matrix defines.

```
int scramble(SAMPLE_TYPE *data,unsigned int length,int n_subbands,
             char mode,int code)
```

With the number of sub-bands in `n_subbands` and the length `length` it is easy to calculate the beginnings and endings of the sub-bands in `*data`, with the matrix, which is produced through a module operation with the user defined key in `code`, the position of the sub-bands can be shifted. `mode` determines if the scrambling is the encoding process or the decoding process.

### 5.4.3 Filter

One of the major parts of this project is the choice and the development of the filters. Theses processes have a direct impact on to the quality of the scrambled signal, on the quality losses of the resulting decoded signal and on the time performance.

That for the real-time scrambling system would need a filter that has:

+ a fast roll off for a clear sub-band cut,
+ a low order to be able to run in real-time,
+ linearity for less losses in the final signal,
+ stability.

As commonly known there are two ways of implementation filters in digital signal processing, finite impulse response (FIR) and infinite impulse response (IIR) which have different advantages to optimise the in Figure 36 shown problem.

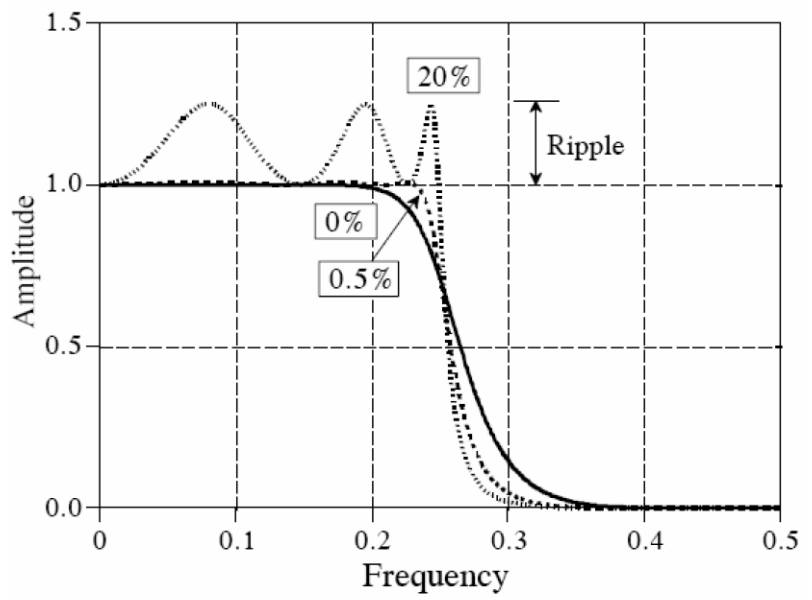


Figure 36: General features of a filter [5]

The most important features of FIR filters are:

+ inherently stable,
+ do not require feedback,
+ their linearity (phase change is proportional to the frequency change; a plus for crossover filters),
- high order,
- slow roll off.

IIR filter certainly do not have the advantage of the FIR filters, but do have some other important features:

+ fast roll off,
+ low order,
- can have ripple,
- need a feedback,
- might be unstable,
- non-linear phase.

To determine what kind of filter meets best the needs of the project, both where implemented in the following way.

FIR Low-pass filter:

- designed with a Kaiser window
- $F_s$=10 kHz; $F_{pass}$=2,4 kHz; $F_{stop}$= 3 kHz
- - 80 dB stopband ripple
- Order 84

FIR High-pass filter:

- designed with a Kaiser window
- $F_s$=10 kHz; $F_{pass}$=2 kHz; $F_{stop}$= 2,6 kHz
- - 80 dB stopband ripple
- Order 84

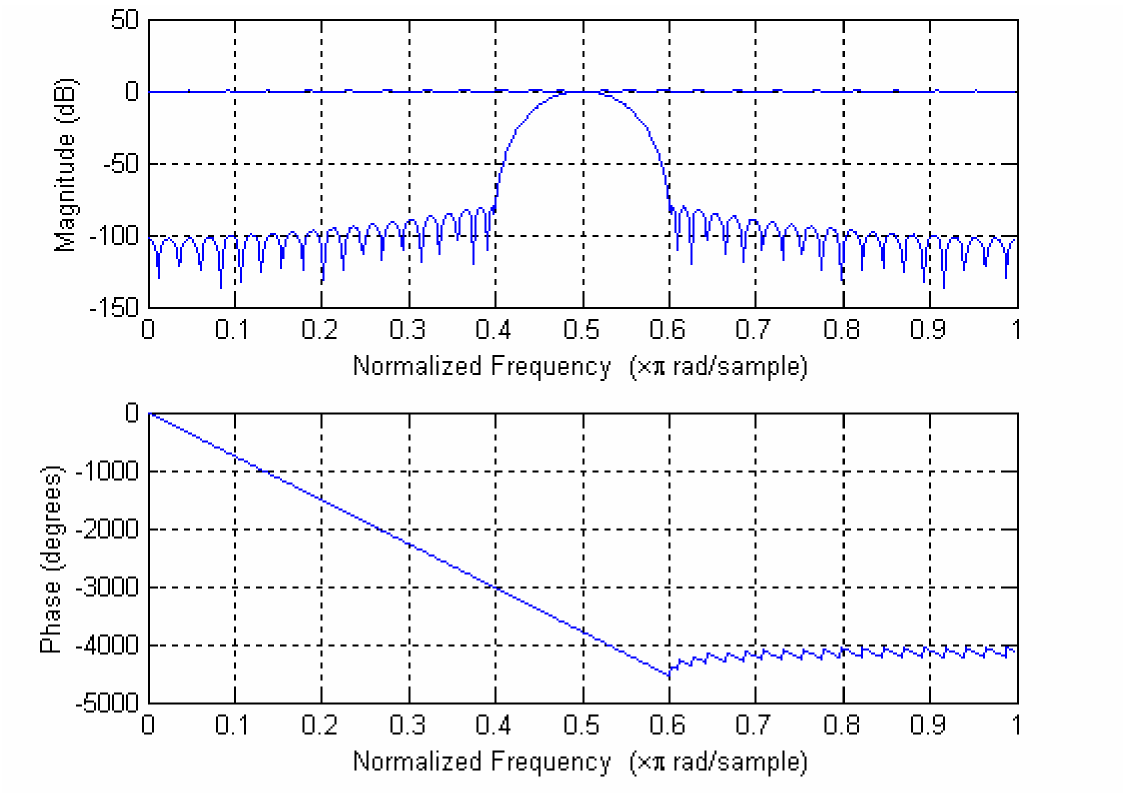


**Figure 37: Plot of FIR filter**

The source code is quit simple but the here presented code is only to be seen as a test version, because it has a lot of options for speed improvement.

```
for (k=NTAPS; k> 0; k--)
{
y += b[k] * z[k];
z[k-1] = z[k];
}
z[1] = x;
return y;
```

The result of filtering and reassembling for a test signal with 4 sub-bands with this filter is presented in Figure 38.

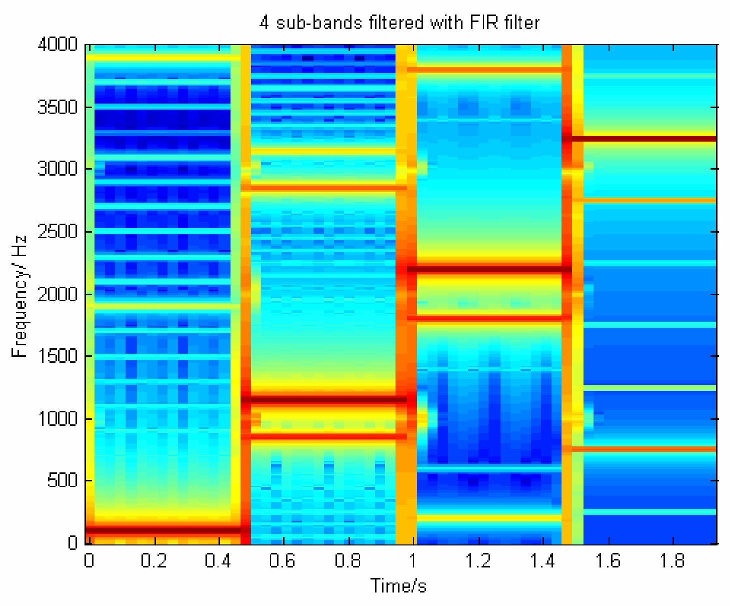


**Figure 38: Test signal filtered with 4 sub-bands and FIR filter**

The image frequencies are good to see, those are not filtered out due to the slow roll off a faster roll off would require a higher order which is not possible due to the real-time requirements.

IIR Low-pass filter:

- designed as Chebyshev II
- $F_s$=10 kHz; $F_{stop}$= 2.62 kHz
- - 80 dB stopband ripple
- Order 20

IIR High-pass filter:

- designed as Chebyshev II
- $F_s$=10 kHz; $F_{stop}$= 2.37 kHz
- - 80 dB stopband ripple
- Order 20

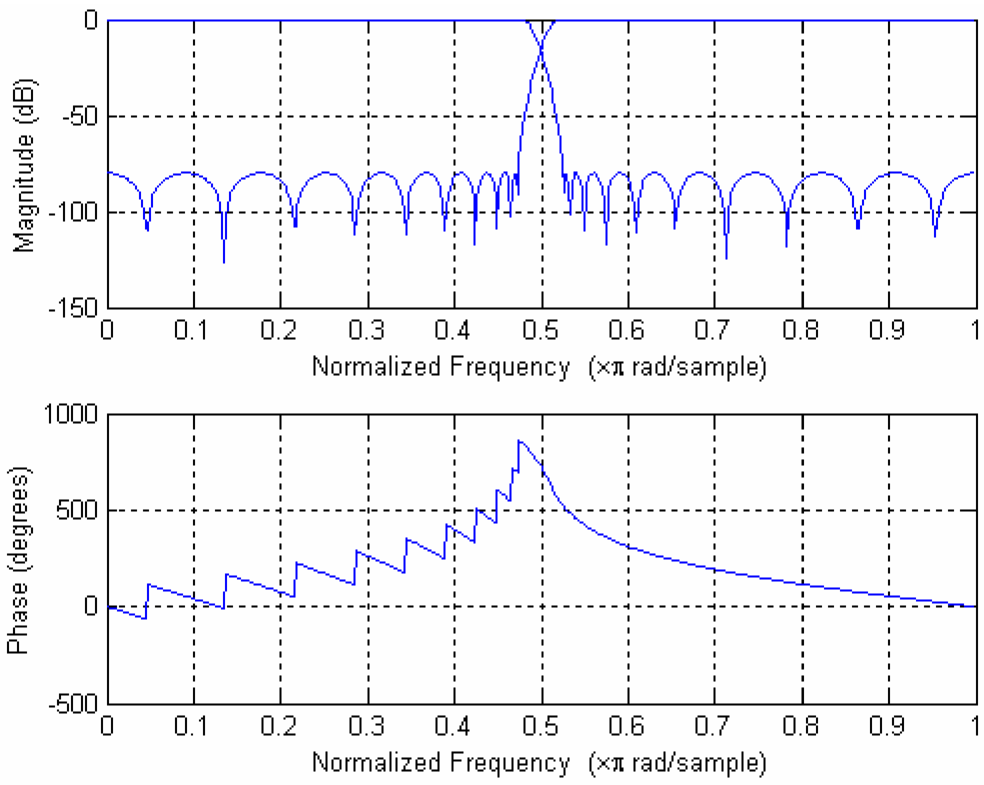


**Figure 39: Filter plot of IIR filter**

The test code for the IIR Filter is not far more complex than the FIR code but of course it requires the feedback. For this code also rules what was said for the FIR about the speed improvements.

```
for (k=NPOLES; k>0; k--)
{
y += w[k]*b[k]-v[k]*a[k];
w[k] = w[k-1];
v[k] = v[k-1];
}
w[1]=x;
v[1]=y;
return y;
```

The result of filtering and reassembling for a test signal with 4 sub-bands with this filter is presented in Figure 39.

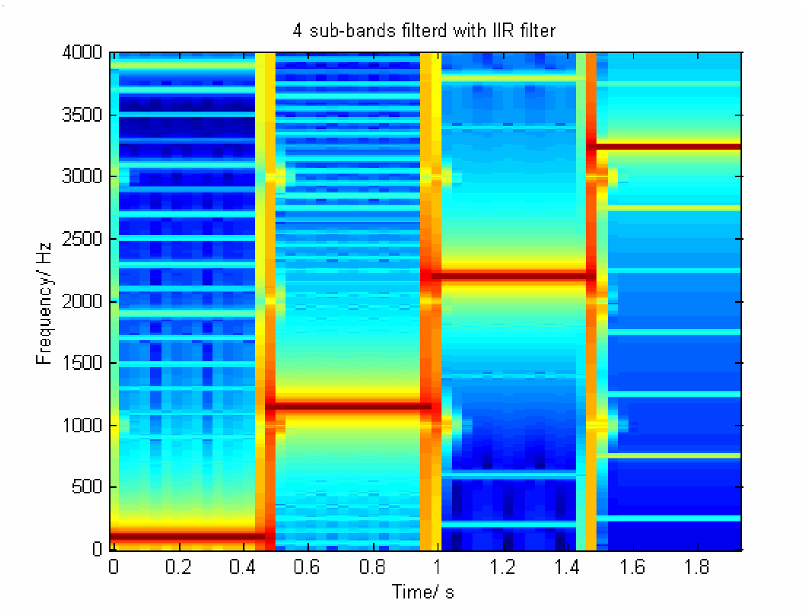


**Figure 40: Test signal filtered with 4 sub-bands and IIR filter**

Due to this outcome of the test signals and the smaller order of the IIR filter it was an obvious choice for the IIR filter. Also the frequencies disadvantages of the FIR filter would have probably not been dramatically relevant for a speech signal.

## Real-time audio and filters

When using filter in a real-time environment there is one problem occurring. The resulting signal is going to contain a periodic clipping noise.

This is because of the nature of the filters and their tendency to settlement. Supplementary digital filters have at least a delay line (FIR), some even a feedback line (IIR) which is at the initialisation of the filter certainly zero. Starting to filter a block a filter settlement at the beginning of every block is inserted like Figure 41 illustrates. In relation to the end of the previous block, this causes a crack and that sounds like the mentioned noise.

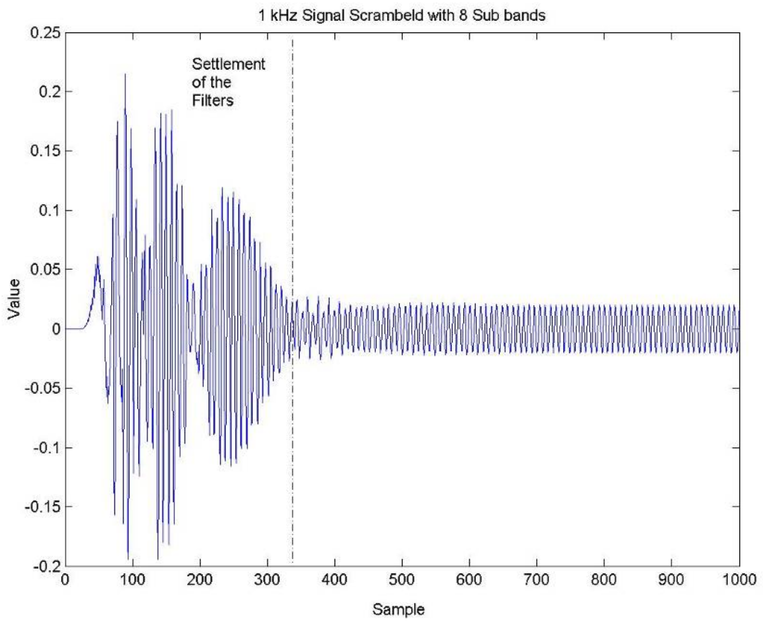


**Figure 41: Settlement of the filters**

To avoid this problem, one has to take care of the reassembling of the blocks as pointed out at the end of the process in Figure 42.

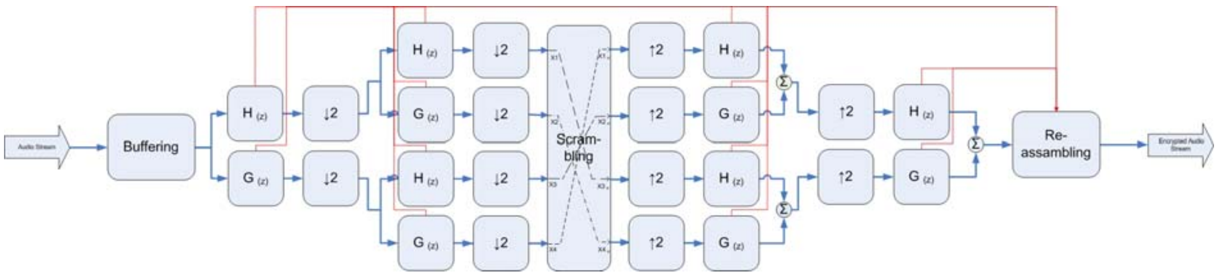


**Figure 42: Saving the delay lines of the filters**

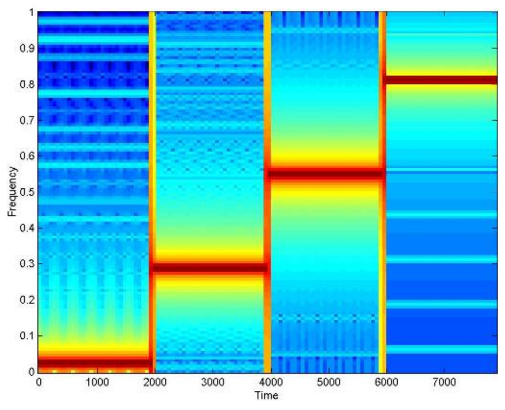
The solution is to save the values of the delay lines after every filtering, so they can be used as the filter initialisation of the next block, with that, a smooth crossover from one to the next block is guaranteed and the settlement takes only place during the first initialisation of the filter, where it does not bother. It is important to notice that the delay lines of every filter have to be saved. This means that i.e. for a 4 sub-band system 6 delay lines for the cutting path and 6 delay lines for reconstruction path have to be saved (when FIR filters are used). Otherwise, the filter for the reconstruction will be initialised by the result of the sub-band generating filters and other way around, which could result in a similar kind of problem.

# 6 Performance

## 6.1 Prove of scrambling

In chapter 3.1 we saw already some figures of the scrambling results. However this was of course only an example with two frequencies to get you familiar with the theoretic idea. In a real application there is of course, more than one frequency at a time.

The spectrograms in the next figures visualise the result of the encoder for a signal containing four single frequencies after each other.



Test signal of:

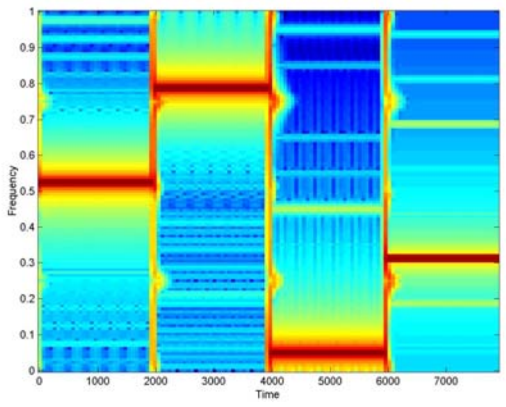
3,25 KHz

2,2 KHz

1,15 KHz

1 KHz

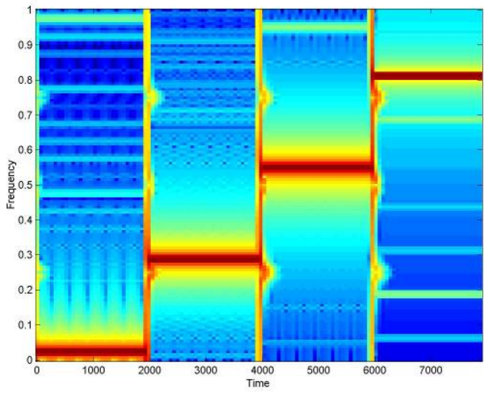
**Figure 43: Original test signal**



Signal scrambled with:

$$ScramMatrix = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

**Figure 44: Scrambled Signal**

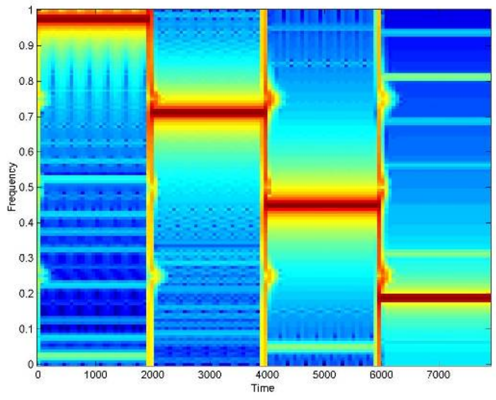


Signal decoded with:

$$DeScramMatrix = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

**Figure 45: Decoded signal**

Note that the transformation of the scrambling matrix in this case is the same matrix again. As mentioned before, the inversion of the signal is not the transposed identity matrix like above, but furthermore the one in Figure 46.



Signal scrambled with:

$$InvMatrix = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

**Figure 46: Inverted Signal**

Even though speech signals are far more complex, the scrambling system works quite the same way as the following figures prove for these matrixes.

$$\texttt{EncodeMatrix=} \begin{vmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix} \quad \texttt{DecodeMatrix=} \begin{vmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

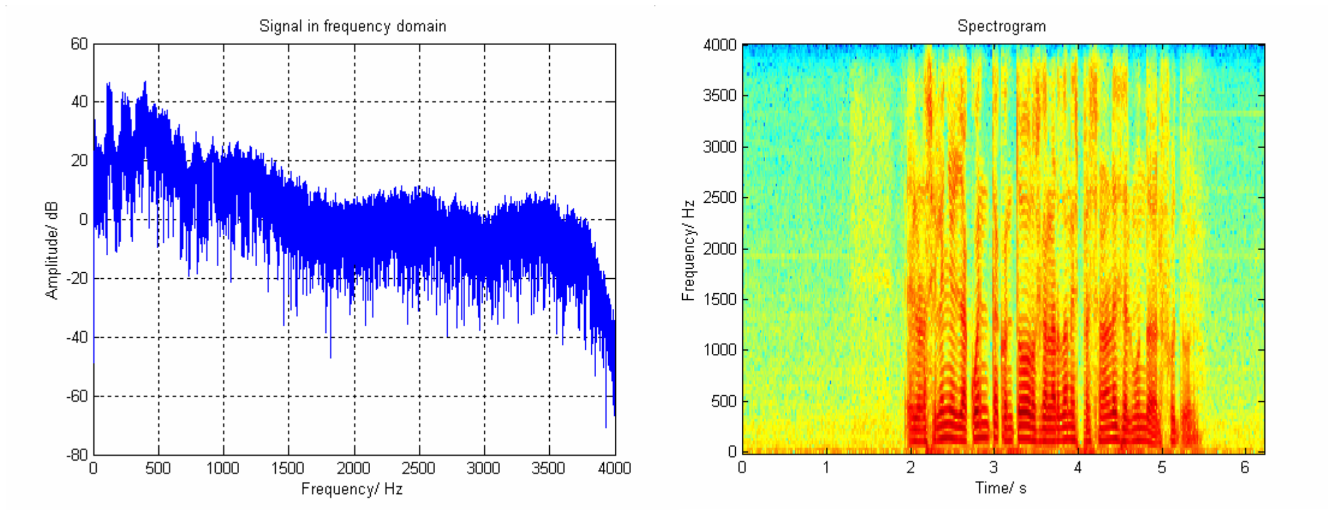


**Figure 47: Spectrum and spectrogram of original speech signal**

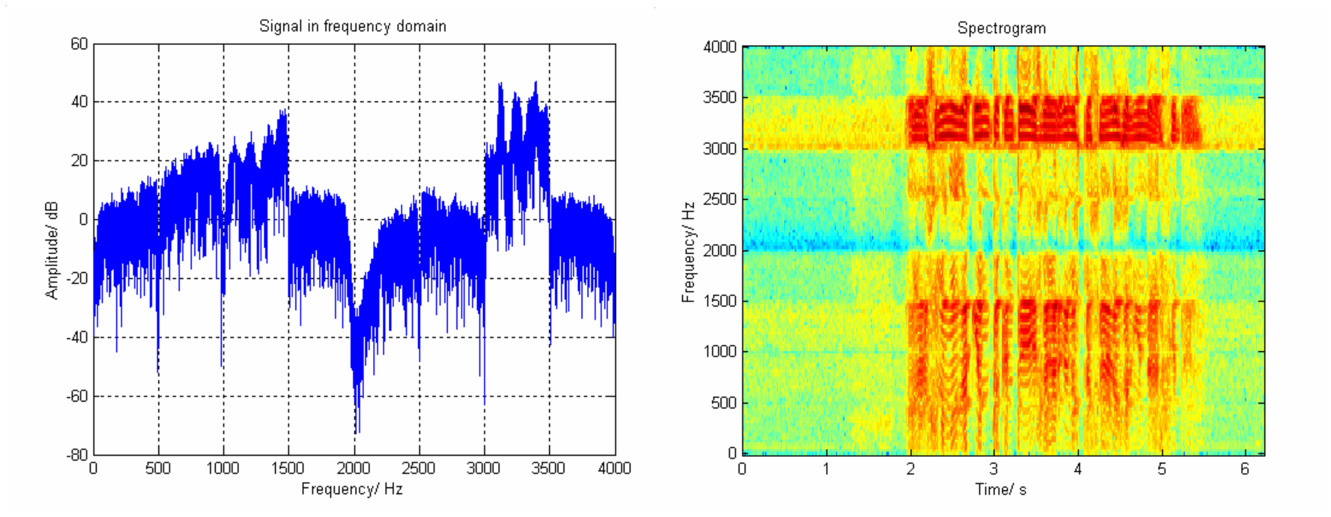


**Figure 48: Spectrum and spectrogram of scrambled speech signal**

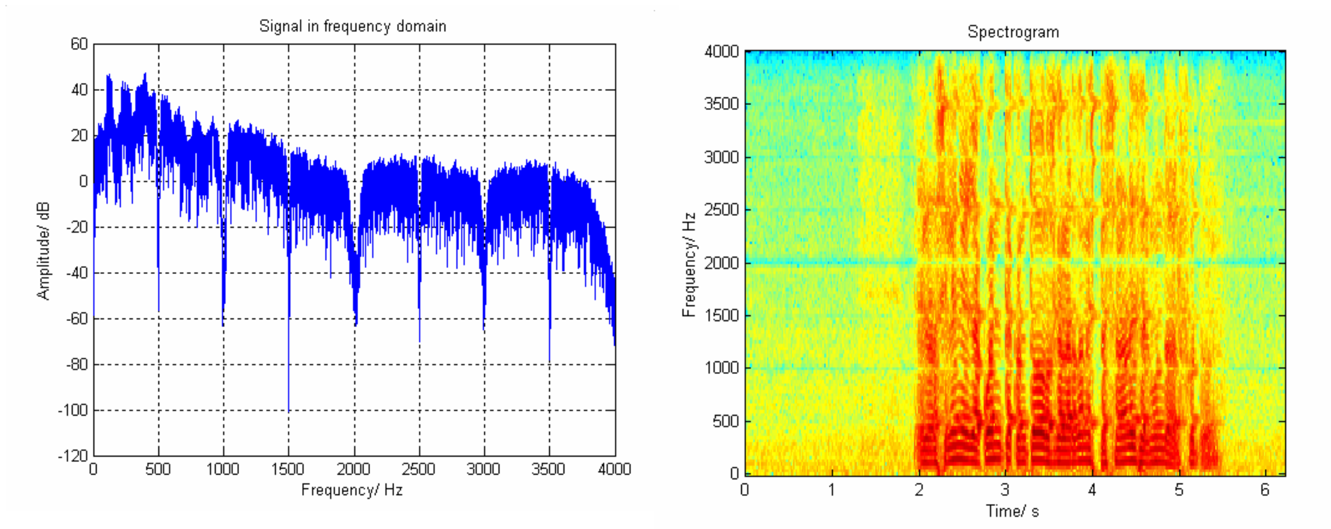


**Figure 49: Spectrum and spectrogram of decoded speech signal**

## 6.2 Time Performance

To determine that the application can perform in duplex and in real-time, two different setups are possible. For both some soldering work has to be done to connect the plugs of the sound card like it was shown in Figure 31 the other way which is easier for development is shown in Figure 50.

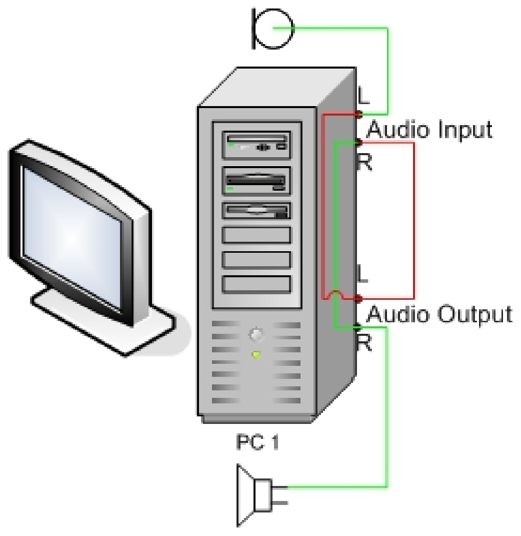


Figure 50: Real-time test with one PC

Of course the system can also be tested with just the encoding signal to one output channel and decoding to the other but for the final test the hardware should be set up once proper.

In both cases the system performed the scrambling process with a recognisable latency. This total latency is caused by:

- latency through hardware (very little),
- latency through overhead (OS, APIs etc),
- time for buffering one block,
- processing time.

Since there are little chances to improve on the hardware or on the OS side the other parameters and techniques should be chosen with care, that’s one of the reasons for PortAudio.

The main points to balance and to optimise are the block size and the process itself. The delay of the block size can be easily calculated through the sampling frequency and the number of samples per block. Table 3 shows the time measurement of the process for some block sizes.

| 1 Cannel | Block Size | |
|---|---|---|
| 1 | 256 | |
| Sub-bands | time for processing one Block/ ms | time for processing cut method for one Block/ ms |
| 4 | 8 | 3 |
| 8 | 10 | 4 |
| 16 | 29 | 12 |

| 1 Cannel | Block Size | |
|---|---|---|
| 1 | 512 | |
| Sub-bands | time for processing one Block/ ms | time for processing cut method for one Block/ ms |
| 4 | 50 | 20 |
| 8 | 70 | 29 |
| 16 | 100 | 40 |

| 1 Cannel | Block Size | |
|---|---|---|
| 1 | 1024 | |
| Sub-bands | time for processing one Block/ ms | time for processing cut method for one Block/ ms |
| 4 | 130 | 50 |
| 8 | 200 | 90 |
| 16 | 280 | 130 |

**Table 3: Time measurement of scrambling process**

The total delay is like mentioned before a combination of all three parameters and is difficult to determine because it changes from platform to platform. But for a gross calculation at 16 kHz sampling frequency can be said that:

| | |
|---|---|
| Block size 256 samples: | 25- 45 ms |
| Block size 512 samples: | 80- 130 ms |
| Block size 1024 samples: | 180- 350 ms |

where at a block size of 256 samples a danger of buffer overflows is present and at 1024 samples clearly a delay is recognisable. At 512 samples the delay is above the hard real-time requirements but since we talk about soft real-time and the speaker and listener are usually not directly next to each other the delay is acceptable.

## 6.3 Cryptanalysis

That the scrambling of the signal is not secure enough to guarantee voice security can be intuitively guessed already from hearing the result of the scrambling or from looking at Figure 48.

But also from the theoretical point of view this technique has some leakages which make it vulnerable even for cipher-text-only attacks. The main problem is that the intensity of the different frequencies stays the same even though they are shifted. With some knowledge of the human voice like it is presented in chapter 1.2 it is easy to recombine the sub-bands in the right way again. The histograms in Figure 51 show that the signal with highest intensity, which must be the lowest frequency according to chapter 1.2, is shifted only once.

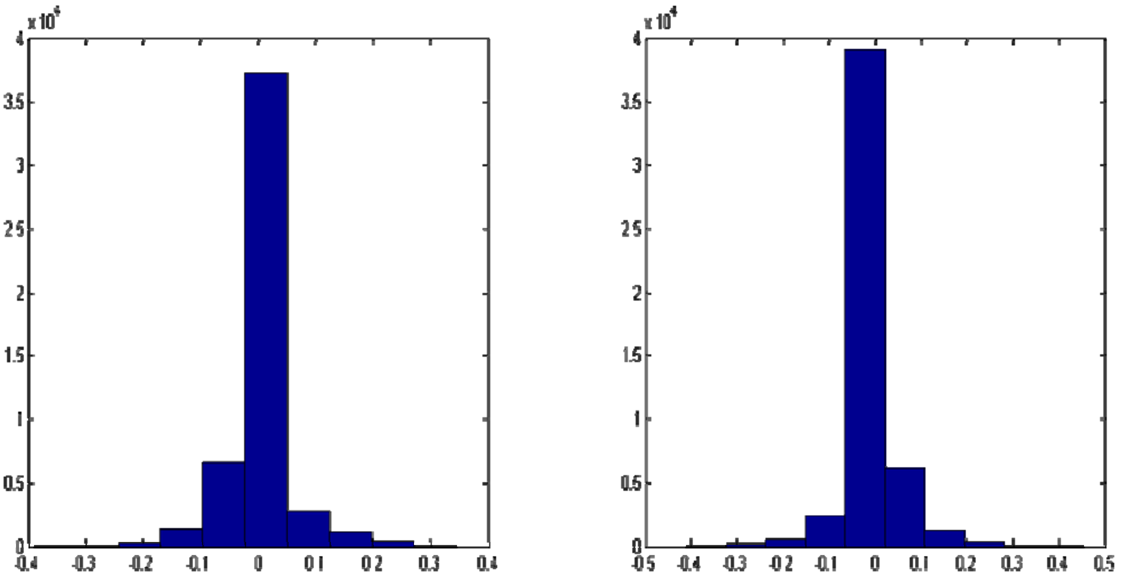


**Figure 51: Histogram of the original (l) and scrambled (r) signal**

Since the code never changes in our system the scrambling has to be considered as broken after once being analysed right for the rest of that communication. The only security this system can serve with is that the attacker might not be equipped with the same system which is not a recommendable security level under the Kerkhoff's Principle[3].

3

The security of an encryption algorithm should depend only on secret key(s), but not on the nondisclosure of the algorithm. [3]

# 7 Conclusion

As already mentioned in the introduction this system can not compete with modern high technology encryption standards and it is not meant to. Nevertheless, it is possible to achieve a voice scrambling system in real-time with only very basic digital signal processing operations. The human voice is a complex signal and furthermore is the human ear and brain very well trained to understand this signal and has even the ability to cover some losses, which makes it very hard to adulterate a signal with just scramble frequencies so it is not understandable anymore.

The direct access of the audio data on a computer platform is an often avoided field, but it turns out that with the right tools it is an easy to release task. The PortAudio API served an astonishing good service in terms of latency and easiness of use (after it was successfully complied).

As said before and seen in the cryptanalysis chapter the system can not guarantee voice security, but it is a very nice and practical example of the basics of signal processing and audio programming and that for might find a usage in an educational environment.

## 7.1 Future Work

There are quite some options to improve the system. At first it should be looked into the current system, there could be a lot of improvements using a library for the filters that could make them run faster and also perform better. With faster filters there would maybe even be a chance to use quadrate mirror filter (QMF) or other better filters that provide less losses. Furthermore, it would be interesting to have a GUI for commanding the system. Another interesting question could be to measure the performance through some other networks with voice coders.

To improve the voice security it would be an option to extend the project with other kinds of scrambling techniques like they were mentioned in the introduction. A floating split frequency or more than one split frequencies in different distances (in upper frequencies bigger, in low ones smaller) would certainly bring some effort. But also time domain scrambling or modulation of frequencies could be considered.

# 8 References

## Books/ Papers/ Scripts

*[1]* William R. Bennett, (Fellow IEEE); *"Secret Telephony as a Historical Example of Spread-Spectrum Communications,"*

IEEE Transactions on Communications, Vol. COM-31, No. 1, January 1983,

(Parts also available at: http://www.nsa.gov/publications/publi00019.cfm#6)

*[2]* B Goldburg, Z Oawson, S Sridharan; *"The Automated Cryptanalysis of Speech Scramblers"* Springer Verlag,1998

[3] Prof. Dr. Heiko Knospe; Script: *„Cryptography" Winter Term 2006/07*; FH-Cologne

*[4]* CES Communications Ltd; *„INTRODUCTION TO VOICE SECURITY"* http://www.cescomm.co.nz/

[5] Steven W. Smith; *"The Scientist and Engineer's Guide to Digital Signal Processing"* copyright ©1997-1998 by. www.DSPguide.com

[6] Roger J. Sutton; *"Secure Communications"*

Copyright 2002 by John Wiley & Sons, Ltd: ISBN: 9780471499046

*[7]* Pere Salvadó Lloveras; *"Encriptació de veu per mescla de subbandes"*

Escola Universitària d'Enginyeria Tècnica Industrial de Terrassa, UPC gen-2006 http://hdl.handle.net/2099.1/3861

[8] Aitor Pérez Pellitero; *"Mejora de un conversor de audio a MIDI e implementación en tiempo real"*

Escola Universitària d'Enginyeria Tècnica Industrial de Terrassa, UPC jun-2007

http://hdl.handle.net/2099.1/4368

## URLs

[9] http://www.nsa.gov/publications/publi00019.cfm#N4

[10] http://en.wikipedia.org/wiki/Voice_frequency

[11] http://www.nsa.gov/public/publi00007.cfm

[12] http://en.wikipedia.org/wiki/SIGSALY

[13] http://en.wikipedia.org/wiki/Scrambler

[14] http://seussbeta.tripod.com/crypt.html

[15] http://en.wikipedia.org/wiki/Vocoder

[16] http://ccrma.stanford.edu/courses/422/projects/WaveFormat

[17] http://www.staudio.de/kb/english/drivers/

[18] http://msdn2.microsoft.com/en-us/library/ms678518(VS.85).aspx

[19] http://clam.iua.upf.edu/

[20] http://www.openal.org/

[21] http://www.portaudio.com/

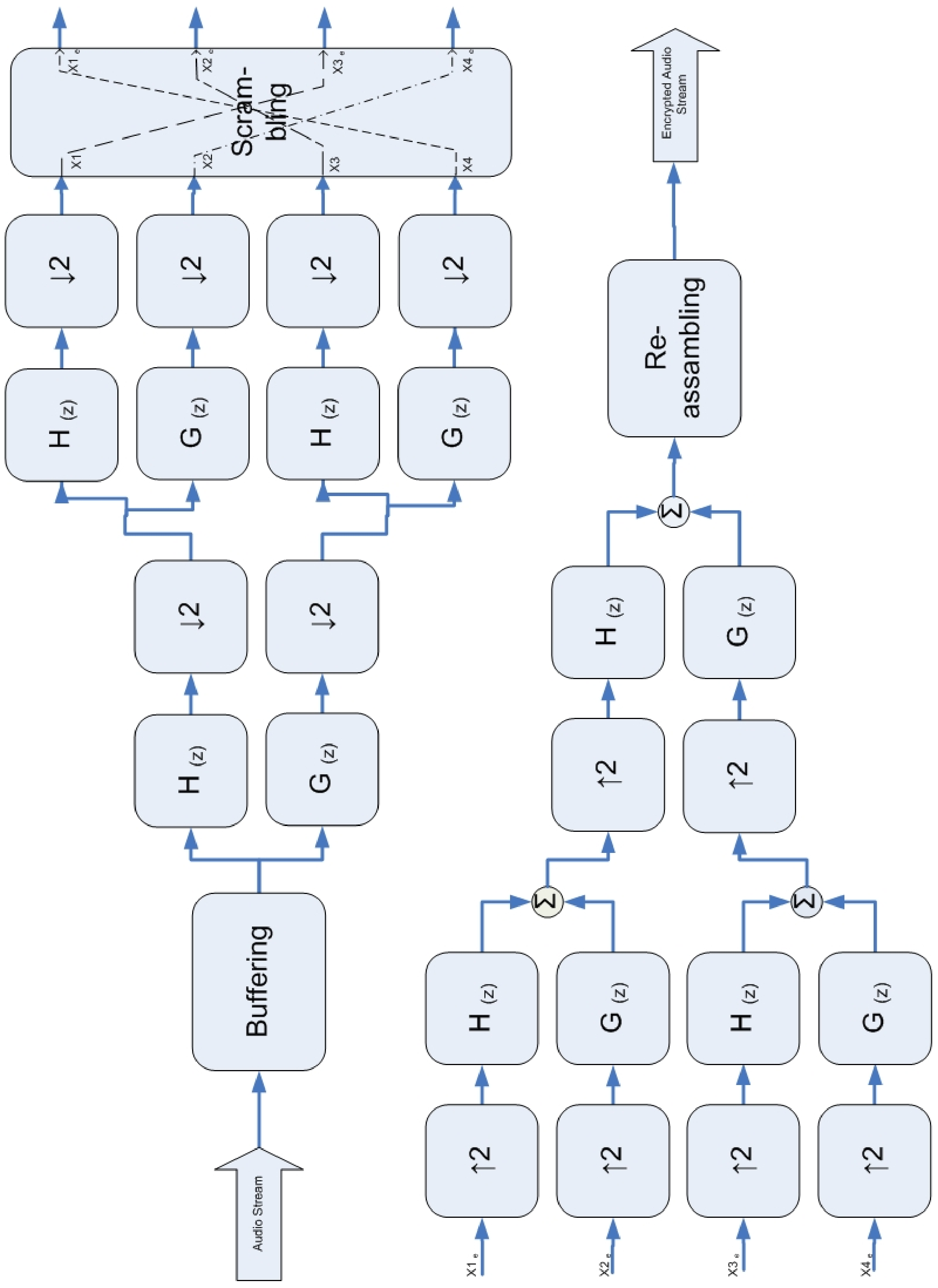
[22] http://en.wikipedia.org/wiki/Duplex_%28telecommunications%29

[23] http://en.wikipedia.org/wiki/Real_time

# List of figures

# List of tables

# Appendix A


Audio Stream
Buffering
H (z)
G (z)
↓2
Scram-bling
X1
X2
X3
X4
X1 e
X2 e
X3 e
X4 e
↑2
Σ
Re-assambling
Encrypted Audio Stream

# Appendix B

## GNU Licence

PortAudio is free to use software where even the source code is free and users are allowed to change the code. To ensure that it is used in the way the primary developer wanted it, it is published under the

> “The **GNU General Public License** (**GNU GPL** or simply **GPL**) is a widely used free software license, originally written by Richard Stallman for the GNU project. It is the license used by the Linux kernel. The GPL is the most popular and well-known example of the type of strong copy left license that requires derived works to be available under the same copy left. Under this philosophy, the GPL is said to grant the recipients of a computer program the rights of the free software definition and uses copy left to ensure the freedoms are preserved, even when the work is changed or added to. This is in distinction to permissive free software licences, of which the BSD licenses are the standard examples.
>
> The GNU Lesser General Public License (LGPL) is a modified, more permissive, version of the GPL, intended for some software libraries. There is also a GNU Free Documentation License, which was originally intended for use with documentation for GNU software, but has also been adopted for other uses, such as the Wikipedia project.”
>
> [http://en.wikipedia.org/wiki/GNU_General_Public_License]