Steganography in MP3

Tomer Shay Aviad Seady Lee Zaid

June 20, 2022

Contents

Introduction

What is MP3?

MP3 (formally MPEG-1 Audio Layer III or MPEG-2 Audio Layer III) is a popular coding format for digital audio, including sound and music files of all kinds. It uses a variety of data compression techniques that allows a large reduction in file size when compared to uncompressed audio – to put things in perspective, for an uncompressed music disc, to get an average sound quality, about 44,000 audio samples per second are needed, with each sample taking up 32 bits – about 1.4 million bits per second!

Though general compression algorithms (such as ZIP) may lesser these massive sizes for audio files, they are not nearly effective enough, especially when one needs to quickly transfer files through the web. Here, the magic of MP3 shines, as it uses a psychoacoustic model that helps get rid of unnecessary info in the data (using lossy compression), in addition to more general compression techniques, to great results – In practice, with sound qualities that are considered "good" ($128 - 130\,{}^{\mathrm{KB}}/_{2}$), an MP3 file takes only about 10%-20% of the original file size. As an example, a 4-minute song will generally take up about 40 MB of space with normal sound encoding, but only about 4MB after being encoded with MP3. For that reason, MP3 is still one of the most popular file formats for sound files to this day.

What is steganography?

Steganography is defined as the practice of concealing a message within another message or a physical object. In computing contexts, a computer file, message, image, or video is concealed within another file, message, image, or video.

The first recorded use of the term was in 1499 by Johannes Trithemius in his Steganographia, a treatise on cryptography and steganography, disguised as a book on magic. Generally, the hidden messages appear to be (or to be part of) something else: images, articles, shopping lists, or some other cover text. For example, the hidden message may be in invisible ink between the visible lines of a private letter.

Whereas cryptography is the practice of protecting the contents of a message alone, steganography is concerned with concealing the fact that a secret message is being sent and its contents. The advantage of steganography over cryptography alone is that the intended secret message does not attract attention to itself as an object of scrutiny. Plainly visible encrypted messages, no matter how unbreakable they are, arouse interest and may in themselves be incriminating in countries in which encryption is illegal.

Steganography includes the concealment of information within computer files. In digital steganography, electronic communications may include steganographic coding inside of a transport layer, such as a document file, image file, program, or protocol. Media files are ideal for steganographic transmission because of their large size. For example, a sender might start with an innocuous image file and adjust the color of every hundredth pixel to correspond to a letter in the alphabet. The change is so subtle that someone who is not specifically looking for it is unlikely to notice the change.

How can steganography be used in MP3 files?

With information security being such a prevalent topic among computer scientists in recent years, many of those who research security and ways to attack it turn their gaze to MP3 as a unique file format for hiding information that passes through the web.

As discussed, media files are ideal for information hiding due to their large sizes, and MP3 files, while not as large as other sound formats, usually take up good amounts of storage. Thus, even big changes on the encoded sound may be subtle enough for the human ear to notice. This makes MP3 files great candidates for steganography of large volumes – for example, discreetly extracting log files out of a server in the guise of normal music files. However, the act of hiding information in such files must be taken with care and expertise. It is not enough to append information onto the audio file – the output file containing the hidden info must be of the same size and format as the original one, as well as indistinguishable to the senses from it (in our case, hearing) as to not arouse suspicion.

What is the aim of our project?

With the guidance of the Matzov unit, we started this project with the mission of researching the MP3 format and ways of using steganography within MP3 files.

The IDF's army network is a massive network, with thousands of files entering and exiting the network every day – including audio files. Of course, the network also contains incredibly sensitive information that must be protected. The information we have collected here and the steganography code system we built will be used as a basis for future security projects within the army's network – to make sure no unwanted info leaves the network, or malicious info enters it, in the guise of a regular MP3 file.

Part I of this research document is an overview of the theory behind the MP3 format, detailing how audio files are compressed and restored with MP3 encoding, as well as the mathematical and physiological ideas behind its compression. This information is crucial for understanding ways to hide info inside MP3 files. Part II is an overview of known steganographic methods for MP3 files, from simple methods for general audio files to state-of-the-art methods using tricks related to the structure of the file for hiding the information. Part III details the steganography system we have built as part of this project, which showcases the encoding and decoding processes for MP3, as well as MP3 steganography, for ease of understanding this written material in future use.

Part I
The Theory Behind MP3

Since the MPEG-1 Layer III is a complex audio compression method it may be quite complicated to get hold of all different components and to get a full overview of the technique. The information presented here contains all you need to know about the big picture of the inner workings of MP3 files. For better understanding of minor nuances, we recommend looking at the source code of individual components in our steganography system.

1 History

Uncompressed digital CD-quality audio signals consume a large amount of data and are therefore not suited for storage and transmission. The need to reduce this amount without any noticeable quality loss was stated in the late 80's by the International Organization for Standardization (ISO). A working group within the ISO referred to as the Moving Pictures Experts Group (MPEG), developed a standard that contained several techniques for both audio and video compression. The audio part of the standard included three modes with increasing complexity and performance. The third mode, called Layer III, manages to compress CD music from 1.4 Mbit/s to 128 kbit/s with almost no audible degradation. This technique, also known as MP3, has become very popular and is widely used in applications today.

2 Introduction To Data Compression

The theory of data compression was first formulated by Claud E. Shannon in 1949 when he released his paper: "A Mathematical Theory of Communication". He proved that there is a limit to how much you can compress data without losing any information. This means that when the compressed data is decompressed the bitstream will be identical to the original bitstream. This type of data compression is called lossless. This limit, the entropy rate, depends on the probabilities of certain bit sequences in the data. It is possible to compress data with a compression rate close to the entropy rate and mathematically impossible to do better. Note that entropy coding only applies to lossless compression.

Lossless compression is required when no data loss is acceptable, for example when compressing data programs or text documents. Three basic lossless compression techniques are described below.

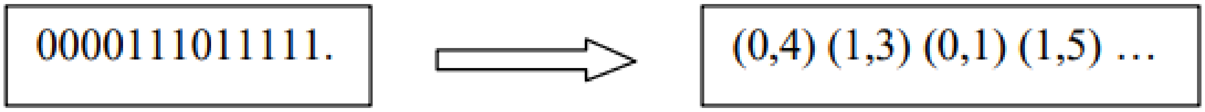
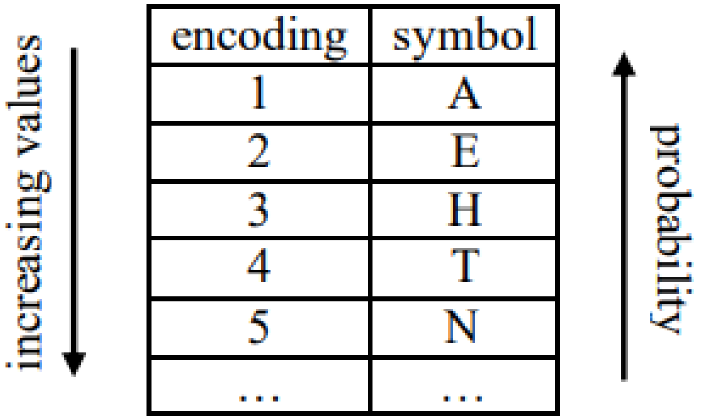


Figure 2.1: Runlength Encoding (RLE)

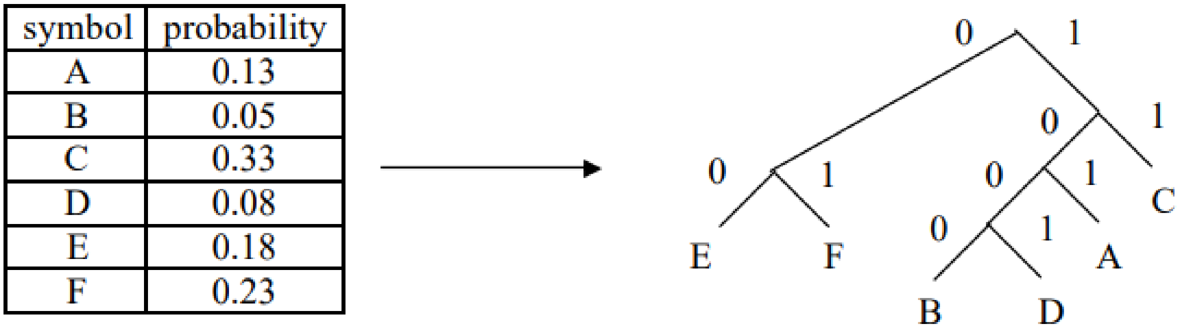
This method takes in a stream of input bits and outputs a list of tuples. Instead of using four bits for the first consecutive zeros the idea is to simply specify that there are four consecutive zeros next. This will only be efficient when the bitstreams are non random, i.e. when there are a lot of consecutive bits.



encoding	symbol
1	A
2	E
3	H
4	T
5	N
...	...

Figure 2.2: Move To Front Encoding (MTF)

This is a technique that is ideal for sequences with the property that the occurrence of a character indicates it is more likely to occur immediately afterwards. A table as the one shown above is used. The initial table is built up by the positions of the symbols about to be compressed. So if the data starts with symbols 'AEHTN...', the N will initially be encoded with 5. The next procedure will move N to the top of the table. Assuming the following symbol to be N it will now be represented by 1, which is a shorter value. This is the root of Entropy coding; more frequent symbols should be coded with a smaller value. It's worth mentioning that RLE and MTF are often used as subprocedures in other methods.



symbol	probability
A	0.13
B	0.05
C	0.33
D	0.08
E	0.18
F	0.23

Figure 2.3: Huffman Coding

The entropy concept is also applied to Huffman hence common symbols will be represented with shorter codes. The probability of the symbols appearing in the data has to be determined prior to compression. A binary tree is constructed with respect to the probability of each symbol. The coding for a certain symbol is the sequence from the root to the leaf containing that symbol. A greedy algorithm for building the optimal tree:

1. Find the two symbols with the lowest probability.
2. Create a new symbol by merging the two and adding their respective probability. It has to be how to treat symbols with an equal probability.
3. Repeat steps 1 and 2 until all symbols are included.

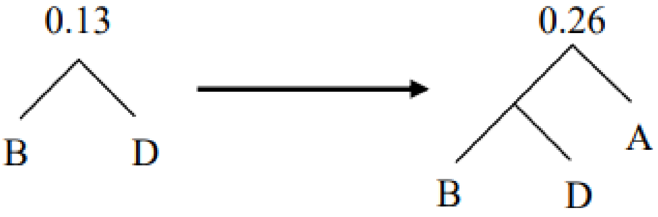


Figure 2.4: A step in the greedy Huffman algorithm

When decoding the probability table must first be retrieved. To know when each representation of a symbol ends simply follow the tree from the root until a symbol is found. This is possible since no encoding is a subset of another (prefix coding).

Lossy compression uses inexact approximations and partial data discarding to represent the content. An example method is shown below.

Quantization. Quantization is the process of mapping input values from a large set (often a continuous set) to output values in a (countable) smaller set, often with a finite number of elements. Rounding and truncation are typical examples of quantization processes. Quantization is involved to some degree in nearly all digital signal processing, as the process of representing a signal in digital form ordinarily involves rounding.

As an example, let's say we have a list of values that we want to quantize, with the first one being 12,592. We choose a *quantization* step of 100 for all our values. Thus, we divide the value by 100 and round up, getting 126. This has the effect of compressing the data, as 12,592 requires 14 bits to hold, while 126 only requires 7. When reconstructing the data, we will multiply 126 by 100 and get 126,000 – an error of 8 from the original value, a minor change overall. However, smaller compressed values could also be achieved, if one is ready to sacrifice accuracy. We can choose a *scalefactor* of 0.1 for our value. The quantized value will be the original value multiplied by the *scalefactor* and then divided by the quantization step and rounded, so in our case, $^{1259}/_{100} = 13$. We now have a value that is represented by only 4 bits – however in the reconstruction, we will receive a much larger quantization error. Depending on the context, this steep decrease in data size might be worth the information lost.

3 Background

3.1 Psychoacoustics and Perceptual Coding

Psychoacoustics is a research branch that aims to understand how the ear and brain interact as various sounds enter the ear.

Humans are constantly exposed to an extreme quantity of radiation. These waves are within a frequency spectrum consisting of millions of different frequencies. Only a small fraction of all waves are perceptible by our sense organs; the light we see and the sound we hear. Infrared and ultraviolet light are examples of light waves we cannot perceive. Regarding our hearing, most humans can not sense frequencies below 20 Hz nor above 20 kHz. This bandwidth tends to narrow as we age. A middle aged man will not hear much above 16 kHz. Frequencies ranging from 2 kHz to 4 kHz are easiest to perceive, they are detectable at a relatively low volume. As the frequencies changes towards the ends of the audible bandwidth, the volume must also be increased for us to detect them (see figure 3.1). That is why we usually set the equalizer on our stereo in a certain symmetric way. As we are more sensitive to midrange frequencies, these are reduced, whereas the high and low frequencies are increased. This makes the music more comfortable to listen to since we become equal sensitive to all frequencies.

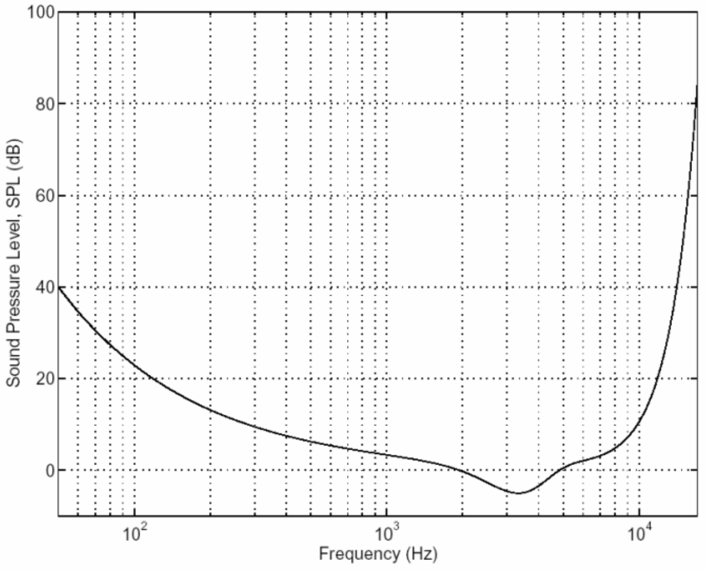


Figure 3.1: The absolute threshold of hearing

As our brain cannot process all the data available to our five senses at a given time, it can be considered as a mental filter of the data reaching us. A perceptual audio codec is a codec that takes advantage of this human characteristic. While playing a CD it is impossible to percept all data reaching your ears, so there is no point in storing the part of the music that will be inaudible. The process that makes certain samples inaudible is called *masking*. There are two masking effects that the perceptual codec needs to be aware of; *simultaneous* masking and temporal masking.

Experiments have shown that the human ear has 24 frequency bands. Frequencies in these so called critical bands are harder to distinguish by the human ear. Suppose there is a dominant tonal component present in an audio signal. The dominant noise will introduce a masking threshold that will mask out frequencies in the same critical band (see figure 3.2). This frequency-domain phenomenon is known as simultaneous masking, which has been observed within critical bands.

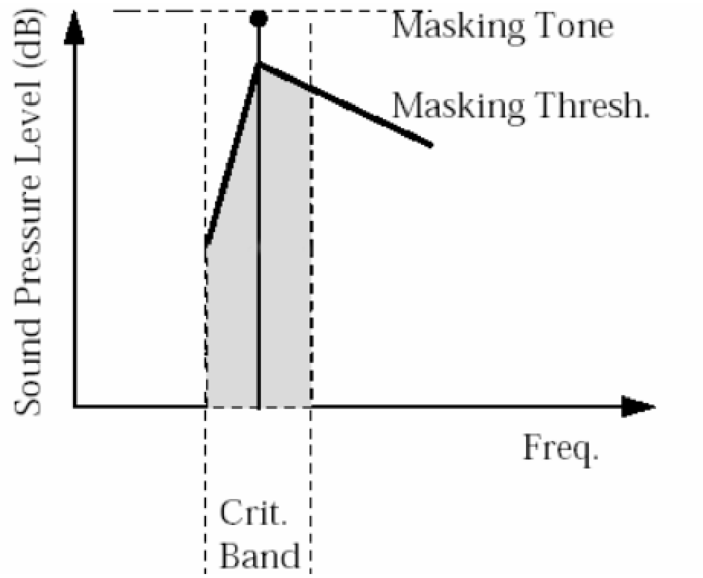


Figure 3.2: Simultaneous masking

Temporal masking occurs in the time-domain. A stronger tonal component (masker) will mask a weaker one (maskee) if they appear within a small interval of time. The masking threshold will mask weaker signals pre and post to the masker. Premasking usually lasts about 50 ms while postmasking will last from 50 ms to 300 ms, depending on the strength and duration of the masker as shown in figure 3.3.

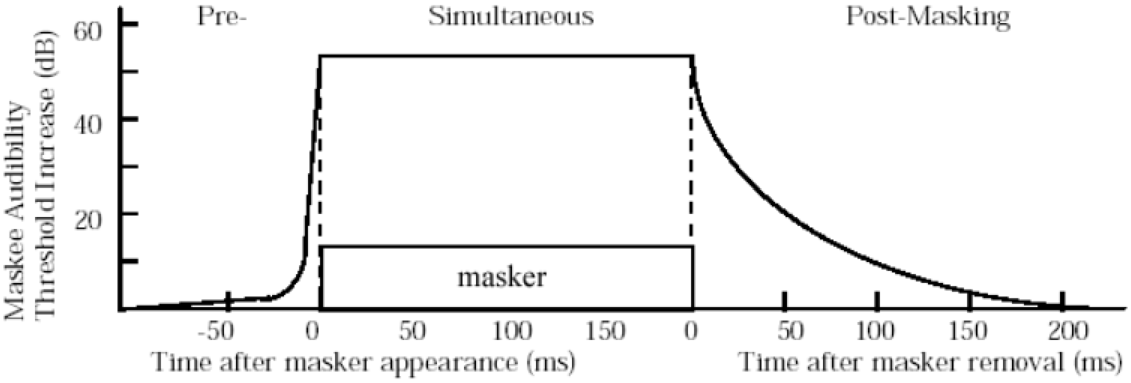


Figure 3.3: Temporal masking

3.2 PCM

Pulse Code Modulation is a standard format for storing or transmitting uncompressed digital audio. CDs and DATs are some examples of media that adapts the PCM format. There are two variables for PCM; sample rate [Hz] and bitrate [Bit]. The sample rate describes how many samples per second the recording consists of. A high sample rate implies that higher frequencies will be included. The bitrate describes how big the digital word is that will hold the sample value. A higher bitrate gives a better audio resolution and lower noise since the sample can be determined more exactly using more bits. CD audio is 44, 100 Hz and 16 Bit. A crude way of compressing audio would be to simply record at a lower sample rate or bitrate. Using a bitrate of 8 bits instead of 16 bits will reduce the amount of data to only 50% but the quality loss in doing this is unacceptable.

4 An Overview of the MPEG-1 Layer III Standard

4.1 The MPEG-1 Standard

The International Organization for Standardization (ISO) is an international federation that aims to facilitate the international exchange of goods and services by publishing international standards. Working within ISO, the Moving Picture Experts Group was assigned to initiate the development of a common standard for coding/compressing a representation of moving pictures, audio and their combination. This standard had to be generic, meaning that any decoder using the standard had to be capable of decoding a bitstream generated by a random encoder using the same standard. Furthermore, trying to preserve both the video and audio quality was obviously very essential.

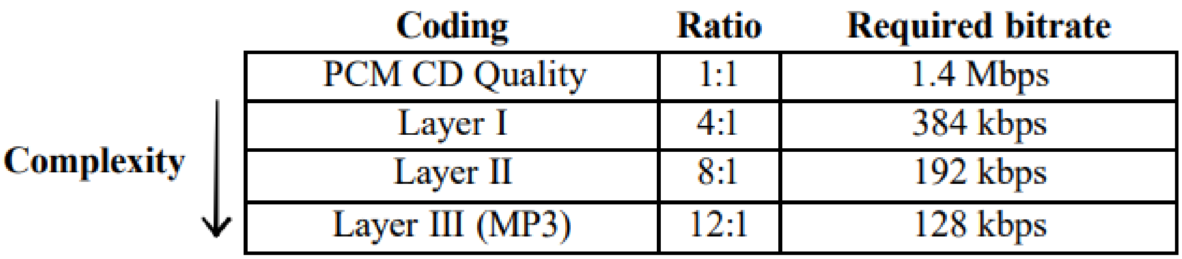
The development began in 1988 and was finalized in 1992 given the name MPEG-1. The standard consisted of three different parts:

- An audio part
- A video part
- A System part

The system part was a description of how to transmit multiple audio and video signals on a single distribution media. Using the MPEG-1 standard it was possible to transmit video and associated audio at a bitrate of between $1 - 2\,{}^{\mathrm{Mbit}}/_{\mathrm{s}}$.

For the audio part there were three levels of compression and complexity defined; Layer I, Layer II and Layer III. Increased complexity requires less transmission bandwidth since the compression scheme becomes more effective. Table 4.1 below gives the transmission rates needed from each layer to transmit CD quality audio.

The third layer compresses the original PCM audio file by a factor of 12 without any noticeable quality loss, making this layer the most efficient and complex layer of the three. The MPEG-1 Layer III standard is normally referred to as MP3. What is quite easy to misunderstand at this point is that the primary developers of the MP3 algorithm were not the MPEG but the Fraunhofer Institute, who began their work in 1987 together with



	Coding	Ratio	Required bitrate
	PCM CD Quality	1:1	1.4 Mbps
Complexity ↓	Layer I	4:1	384 kbps
	Layer II	8:1	192 kbps
	Layer III (MP3)	12:1	128 kbps

Table 4.1: Bitrates required to transmit a CD quality stereo signal

the German University of Erlangen. ISO then codified the work into the MPEG1 Layer III standard. This is usually the way standards are created.

Nevertheless, the work continued and MPEG-2 was finalized in 1994, introducing a lot of new video coding concepts. The main application area for MPEG-2 was digital television. The audio part of MPEG-2 consisted of two extensions to MPEG-1 audio:

- Multichannel audio encoding, including the 5.1 configuration. (Backward compatible).
- Coding at lower sample frequencies (see 4.5).

More standards (MPEG-4, MPEG-7) have been developed since then but this paper will only mention the two first phases of this research.

4.2 The Idea Behind MPEG Data Reduction

Since MP3 is a perceptual codec it takes advantage of the human system to filter unnecessary information. Perceptual coding is a lossy process and therefore it is not possible to regain this information when decompressing. This is fully acceptable since the filtered audio data cannot be perceptible to us anyway. There is no point in dealing with inaudible sounds.

Each human critical band is approximated by scalefactor bands. For every scalefactor band a masking threshold is calculated. Depending on the threshold the scalefactor bands are scaled with a suited scalefactor to reduce quantization noise caused by a later quantization of the frequency lines contained in each band.

But merely lossless compression will not be efficient enough. For further compression the Layer III part of the MPEG-1 standard applies Huffman Coding. As the codec is rather complex there are additional steps to trim the compression. For a more detailed description on the encoding algorithm, see section 5.

4.3 Freedom of Implementation

The MP3 specification (ISO 11172-3) defines how the encoded/decoded bitstream should be structured/interpreted. The output of an encoder developed according to this specification will be recognizable to any MP3 decoder and vice versa. This is of course necessary for it to be a standard specification. But the specification does not exactly specify the steps of how to encode an uncompressed stream to a coded bitstream. This means that the encoders can function quite differently and still produce a compliant to the standard. It is up to the developer to decide how to implement certain parts of the encoder. For instance, it is not specified how to deal with the frequencies over 16 kHz. Since it is quite hard to detect audio signals in that spectrum a developer might choose to discard these frequencies, which will leave bits available to encode more audible signals.

Two important aspects when developing an encoder are speed and quality. Unfortunately, the implementations given by the standard do not always apply the most efficient algorithms. This leads to huge differences in the operating speed of various encoders. The quality of the output may also vary depending on the encoder.

Regarding the decoding, all transformations needed to produce the PCM samples are defined. However, details for some parts are missing and the emphasis lies on the interpretation of the encoded bitstream, without using the most efficient algorithms in some cases.

4.4 Bitrate

The bitrate is a user option that has to be set prior to encoding. It will inform the encoder of the amount of data allowed to be stored for every second of uncompressed audio. This gives the user an opportunity to choose the quality of the encoded stream. The Layer III standard defines bitrates from 8 kbit/s up to 320 kbit/s, default is usually 128 kbit/s. A higher bitrate implies that the samples will be measured more precisely giving an improved audio resolution.

Note that a stereo file with a certain bitrate divides the bitrate between the two channels, allocating a larger portion of the bitrate to the channel which for the moment is more complex.

The standard specifies two different types of bitrates; Constant Bitrate (CBR) and Variable Bitrate (VBR). When encoding using CBR (usually default) every part of a song is encoded with the same amount of bits. But most songs will vary in complexity. Some parts might use a lot of different instruments and effects while other parts are more simply composed. CBR encoding causes the complex parts of a song, which require more bits, to be encoded using the same amount of bits as the simple parts, which require less bits. VBR is a solution to this problem allowing the bitrate to vary depending on the dynamics of the signal. As you will see in section 5, the encoded stream is divided into several frames. Using VBR makes it possible for the encoder to encode frames using different bitrates. The quality is set using a threshold specified by the user to inform the encoder of the maximum bitrate allowed. Unfortunately there are some drawbacks of using VBR. Firstly, VBR might cause timing difficulties for some decoders, i.e. the MP3 player might display incorrect timing information or none at all. Secondly, CBR is often required for broadcasting, which initially was an important purpose of the MP3 format.

4.5 Sampling Frequency

The audio resolution is mainly depending on the sampling frequency, which can be defined as the number of times per second the signal is stored. A high bitrate will give a better precision of a sampled value whereas a high sampling frequency gives the ability to store more values, which in turn gives a broader frequency spectrum. MPEG-1 defines audio compression at 32 kHz, 44.1 kHz and 48 kHz.

4.6 Channel Modes

There are four different channel modes defined:

- Single Channel (Mono)
- Dual Channel (channels are encoded independently of each other)
- Stereo
- Joint Stereo

Note. Dual channel files are made of two independent mono channels. Each one uses exactly half the bitrate of the file. Most decoders output them as stereo, but it might not always be the case. One example of use would be some speech in two different languages carried in the same bitstream, and then an appropriate decoder would decode only the chosen language.

4.6.1 Joint Stereo

The Joint Stereo mode considers the redundancy between left and right channels to optimize coding. There are two techniques here; *middle/side stereo* (MS stereo) and *Intensity Stereo*. MS stereo is useful when two channels are highly correlated. The left and right channels are transmitted as the sum and difference of the two channels, respectively. Since the two channels are reasonably alike most of the time the sum signal will contain more information than the difference signal. This enables a more efficiently compressing compared to transmitting the two channels independently. MS stereo is a lossless encoding.

In intensity stereo mode the upper frequency subbands are encoded into a single summed signal with corresponding intensity positions for the scalefactor bands encoded. In this mode the stereo information is contained within the intensity positions because only a single channel is transmitted. Unfortunately stereo inconsistencies will appear for this model since audio restricted to one channel will be present in both channels. The inconsistencies will not be conceivable by the human ear if they are kept small.

Some encodings might use a combination of these two methods.

5 Encoding

The MP3 encoding process is very complex. Here we will describe the general encoding algorithm and the theory behind it – for better understanding of the technicalities, see

Part III and our implementation of an MP3 encoder. An MP3 encoder receives raw sound data as input, in the form of PCM. It uses a combination of a psychoacoustic model and data compression methods to compress this data, and outputs a MP3 file (see section 7 on page 25 for the structure of MP3 files).

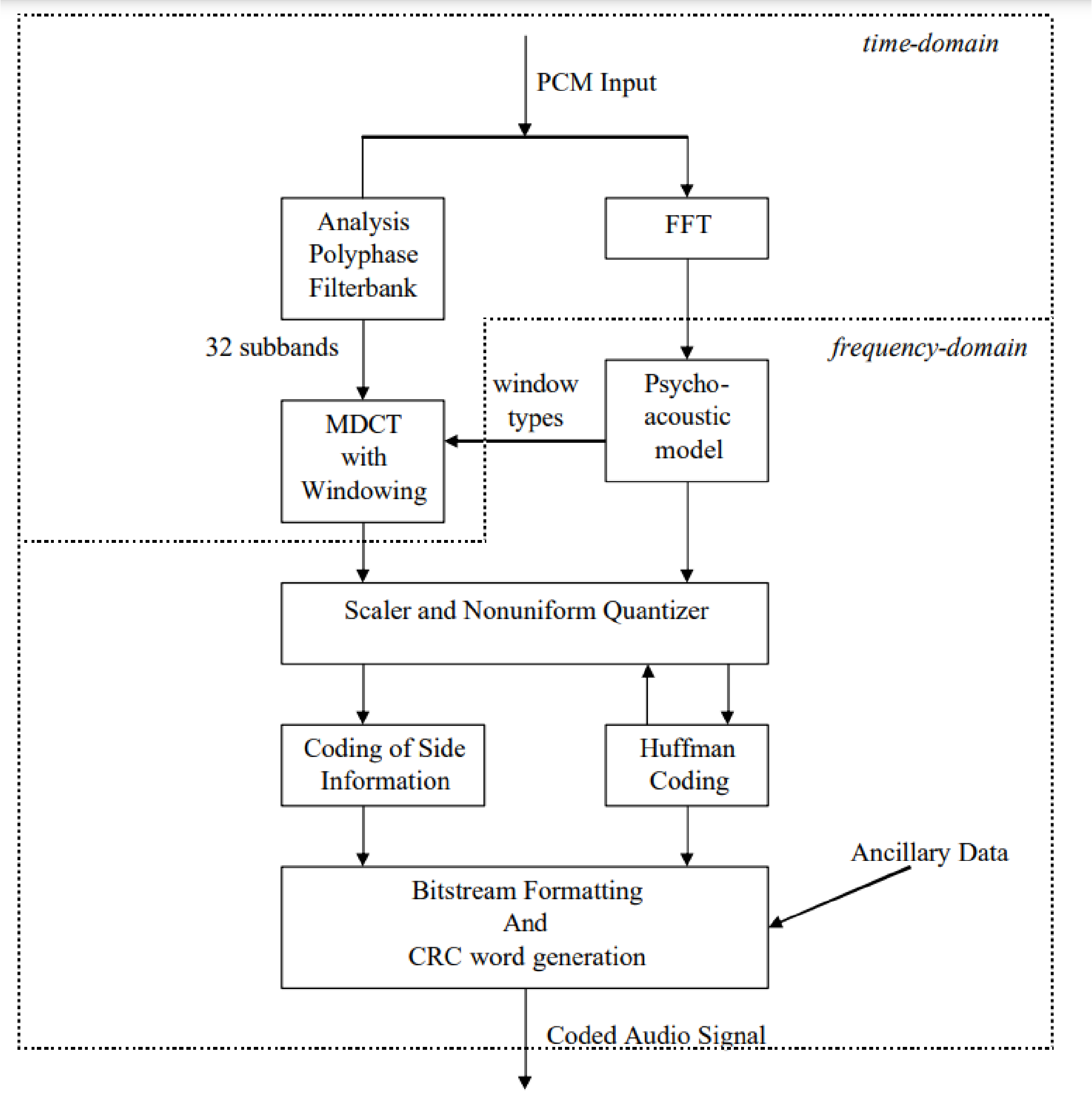


Figure 5.1: MPEG-I Layer III Encoding Scheme for one Frame

5.1 Analysis Polyphase Filterbank

The first step is to divide the PCM into frames of 1152 PCM samples. Each of these sample sequences are then filtered into 32 equally spaced frequency subbands depending on the Nyquist frequency of the PCM signal (the highest frequency that it is possible to reconstruct from a digital signal). If the sample frequency of the PCM signal is 44.1 kHz the Nyquist frequency will be 22.05 kHz. Each subband will be approximately $^{22050}/_{32} = 689\,\text{Hz}$ wide.

The lowest subband will have a range from $0-689\,\text{Hz}$, the next subband $689-1378\,\text{Hz}$, etc. Every sample (might) contain signal components from $0-22.05\,\text{kHz}$ that will be filtered into the appropriate subband. This means that the number of samples has increased by a factor 32 since every subband now stores a sub-spectra of the sample. For example, having filtered 100 samples increases the number of samples to 3200. The 100 samples in every subband will then be decimated by a factor 32, hence only every thirty-second sample is retained. The number of samples are now reduced from 3200 back to 100. But note that there has been a data reduction since a sample in a subband does not include the whole frequency spectra since it has been filtered.

Since it is not possible to construct bandpass filters with a perfectly square frequency response, some aliasing will be introduced by the decimation. Aliasing refers to the distortion or artifact that results when a signal reconstructed from samples is different from the original continuous signal.

5.2 Modified discrete cosine transform (MDCT)

By applying a modified discrete cosine transform to each time frame of subband samples, the 32 subbands will be split into 18 finer subbands creating a *granule* with a total of 576 frequency lines. Each granule is treated as independent of the other – this has the added benefit of backwards compatibility for MPEG layer II decoders. The computation for the transformation is done as followed:

$$X_k = \sum_{n=0}^{2N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2} + \frac{N}{2}\right)\left(k + \frac{1}{2}\right)\right]$$

However, prior to the MDCT, each subband signal has to be windowed. Windowing is done to reduce artefacts caused by the edges of the time-limited signal segment. There are four different window types defined in the MPEG standard (see figure 5.2). Depending on the degree of stationarity the psychoacoustic model determines which window type to apply and forwards the information to this block.

If the psychoacoustic model decides that the subband signal at the present time frame shows little difference from the previous time frame, then the long window type is applied, which will enhance the spectral resolution given by the MDCT. Alternatively, if the subband signal shows considerable difference from the previous time frame, then the short windows is applied. This type of window consists of three short overlapped windows and will improve the time resolution given by the MDCT. A higher time resolution is necessary in order to control time artifacts, for instance pre-echoes. In order to obtain a better adaptation when window transitions are required, two windows referred to as start windows and stop windows are defined.

A long window becomes a start window if it is immediately followed by a short window. Similarly, a long window becomes a stop window if it is immediately preceded by a short window. The start and stop windows are skewed to account for the steeper sides of the adjacent short window.

The aliasing introduced by the polyphase filter bank is now removed to reduce the amount of information that needs to be transmitted. This is achieved using a series of

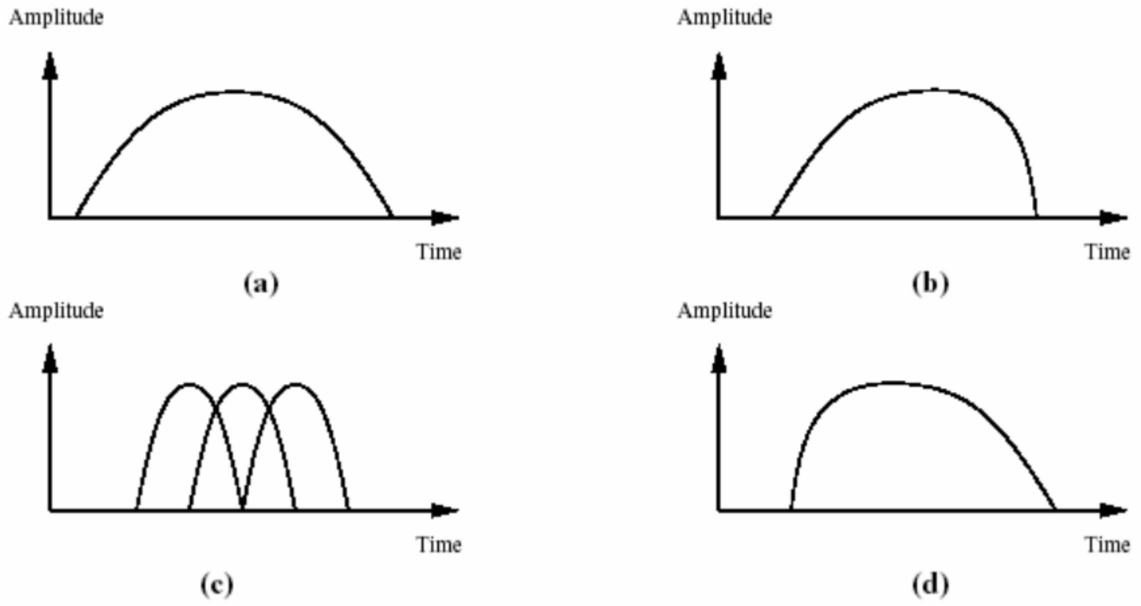


Figure 5.2: Window Types. (a) long window, (b) start window, (c) short windows, (d) stop window

butterfly computations that add weighted, mirrored versions of adjacent subbands to each other (see section 6: Alias Reduction).

5.3 FFT (Fast Fourier Transform)

Simultaneously as the signal is processed by the polyphase filterbank it is also transformed to the *frequency domain* by a Fast Fourier Transform. A signal in the *time domain* represents change over time, while a signal in the *frequency domain* represents the division of the signal into frequency bands over a range of frequencies. The main reason for this transformation is that the conversion from time domain to frequency domain simplifies mathematical analysis of the signal, allowing the psychoacoustic model to work its magic.

Both a 1024 and a 256 point FFT are performed on 1152 PCM samples at the time to give higher frequency resolution and information on the spectral changes over time. For more information on the FFT algorithm, see bibliography.

5.4 Psychoacoustic Model

This block retrieves the input data from the FFT output. Since the samples are in the frequency domain they can be applied to a set of algorithms. These algorithms will model the human sound perception and hence they can provide information about which parts of the audio signals that are audible and which parts are not. This information is useful to decide which window types the MDCT should apply and also to provide the Nonuniform Quantization block with information on how to quantize the frequency lines.

To know which window type to send to the MDCT block, the two presently FFT spectra and the two previous spectra are compared. If certain differences are found a change to short windows requested. As soon as the differences fades away the MDCT block will be informed to change back to long (normal) windows (see figure 5.3).

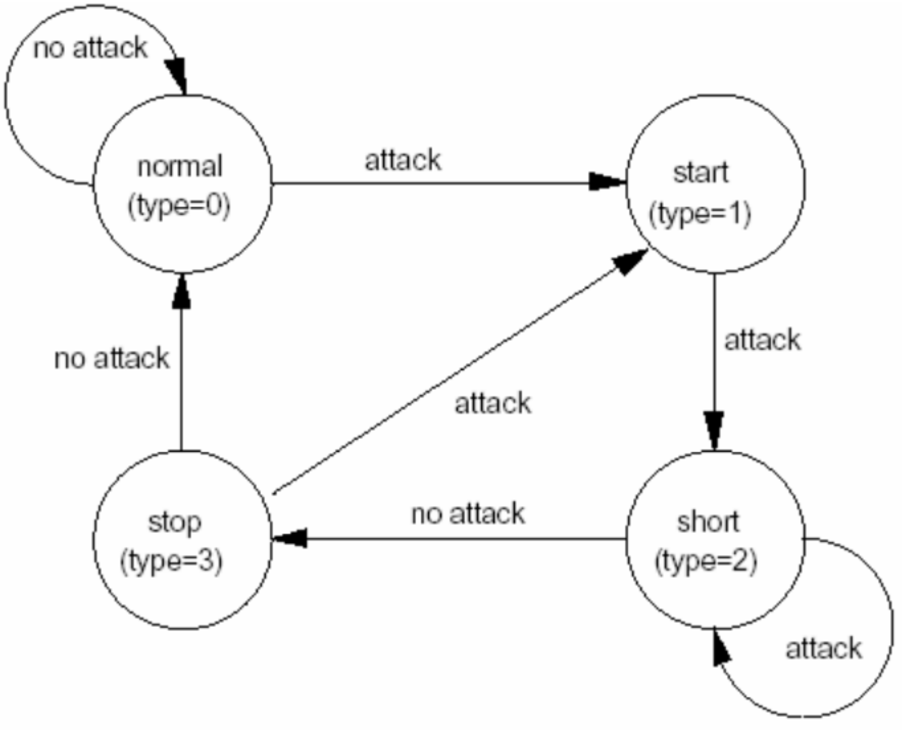


Figure 5.3: Window switching decision

The Psychoacoustic Model also analyzes the FFT spectrum to detect dominant tonal components, and for each critical band, masking thresholds are calculated. Frequency components below this threshold are masked out. Recall that the scalefactor bands are roughly equivalent to the critical bands of human hearing. The thresholds limits for each scalefactor band are used by the quantization block to keep the quantization noise below these limits.

5.5 Nonuniform Quantization

In these two blocks the scaling, quantization and Huffman coding are applied to 576 spectral values at a time. This is done iteratively in two nested loops, a *distortion control loop* (outer loop) and a *rate control loop* (inner loop).

Rate control loop. The rate loop does the quantization of the frequency domain samples and thus also determines the required quantization step size. Furthermore the subdivision of the `big_values` into regions, the Huffman table selection decision for each region and the calculation of the boundaries between the regions take place here.

To begin with the samples are quantized with an increasing step size until the quantized values can be coded using one of the available Huffman code tables. A larger step size leads to smaller quantized values. Then the overall Huffman coded bit sum is calculated and compared with the number of bits available. If the calculated bit sum exceeds the number of bits available the quantization step size is further increased and the entire procedure is repeated until the available bits are sufficient.

The nonlinearity is achieved raising each sample to the power of $^3/_4$.

Distortion control loop. This loop controls the quantization noise which is produced by the quantization of the frequency domain lines within the rate control loop. The aim is to keep the quantization noise below the masking threshold (allowed noise given by the psychoacoustic model) for each scalefactor band.

To shape the quantization noise scalefactors are applied to the frequency lines within each scalefactor band. The scalefactors of all scalefactor bands and the quantization step size are then saved before the rate control loop is called. After the inner loop the quantization noise is calculated. This is repeated until there is no more scalefactor band with more noise than allowed by the threshold. The values of the scalefactors belonging to bands that are too noisy are increased for each iteration loop. Finally the noise caused by the quantization will not be audible by a human and the loop will exit.

There are still situations where both loops can go on forever depending on the calculated threshold. To avoid this there are several conditions in the distortion control loop that can be checked to stop the iterations more early.

Loops input:

- Vector of the magnitudes of the 576 spectral values.
- The allowed distortion of the scalefactor bands.
- The number of scalefactor bands.
- Bits available for the Huffman coding and the coding of the scalefactors.
- The number of bits in addition to the average number of bits, as demanded by the value of the psychoacoustic entropy for the granule.

Loops output:

- Vector of 576 quantized values.
- The scalefactors.
- Quantizer step size information.
- Number of unused bit available for later use.
- Preflag (loops preemphasis on/off)
- Huffman code related side information (see part 6: Side Information)

- big_values (number of pairs of Huffman coded values, excluding "count1").
- count1table_select (Huffman code table of absolute values <= 1 at the upper end of the spectrum).
- table_select (Huffman code table of regions).
- region0_count and region1_count (used to calculate boundaries between regions).
- part2_3_length.

5.6 Huffman Encoding

The quantized values are Huffman coded. Each division of the frequency spectrum can be coded using different tables. The tables are chosen from 32 tables set in the standard; the chosen table for each region is the one that its encoding will be the most effective given that region's values (meaning the least amount of bits will be used in the encoding).

The Huffman coding is one of the major reasons why the MPEG1 Layer III retains a high quality at low bitrates.

5.7 Coding of Side Information

All parameters generated by the encoder are collected to enable the decoder to reproduce the audio signal. These are the parameters that reside in the side information part of the frame. See section 7 for more information.

5.8 Bitstream Formatting

In this final block the defined bitstream is generated (see section 7). The frame header, side information, CRC, Huffman coded frequency lines etc are put together to form frames. Each one of these frames represents 1152 encoded PCM samples.

6 Decoding

The MP3 decoding process is very complex. Here we will describe the general decoding algorithm and the theory behind it – for better understanding of the technicalities, see part 3 and our implementation of an MP3 decoder.

An MP3 encoder receives an MP3 file, and reverses the compression done by the encoder to output PCM data that audio players can play. Frames outputted by the decoder will not be identical to those inputted into the encoder, as MP3 does use lossy compression methods – however, due to the psychoacoustic model, the data should sound the same to the human ear.

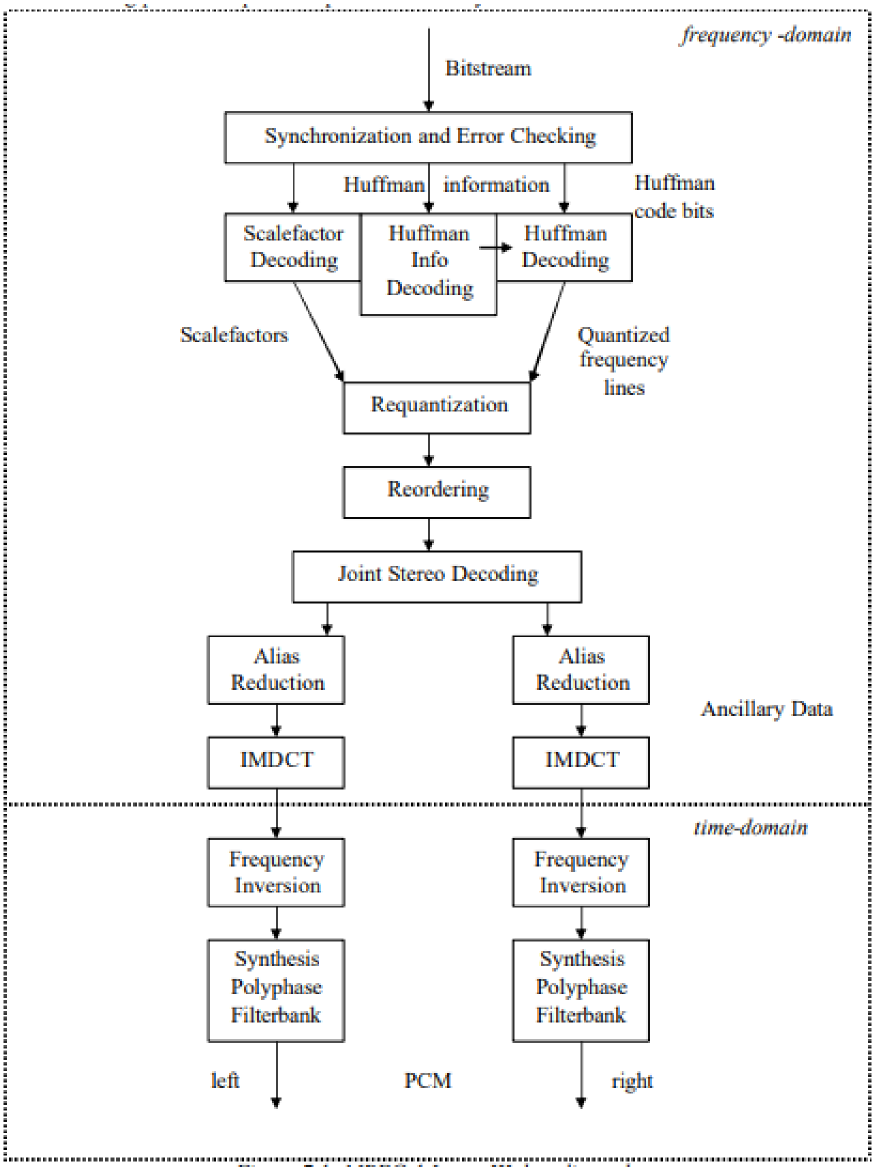


Figure 6.1: MPEG-I Layer III Decoding Scheme for one Frame

6.1 Sync and Error Checking

This block receives the incoming bitstream. Every frame within the stream must be identified by searching for the synchronization word (see section 7). It is not possible for the following blocks to extract the correct information needed if no frames are located.

6.2 Huffman Decoding & Huffman Info Decoding

Since Huffman coding is a variable length coding method, a single codeword in the middle of the Huffman code bits cannot be identified. The decoding must start where the codeword starts. This information is given by the Huffman info decoding block. The purpose of this block is to provide all necessary parameters by the Huffman decoding block to perform a correct decoding.

Moreover, the Huffman info decoder block must insure that 576 frequency lines are generated regardless of how many frequency lines are described in the Huffman code bits. When fewer than 576 frequency lines appear the Huffman info decoding block must initiate a zero padding to compensate for the lack of data.

6.3 Scalefactor decoding

This block decodes the coded scalefactors, i.e. the first part of the main data. The scalefactor info needed to do this is fetched from the side information. The decoded scalefactors are later used when requantizing.

6.4 Requantizer

Here the `global_gain`, `scalefactor_scale`, *preflag fields* in the side information contributes to restoring the frequency lines as they were generated by the MDCT block in the encoder. The decoded scaled and quantized frequency lines output from the Huffman decoder block are requantized using the scalefactors reconstructed in the Scalefactor decoding block together with some or all fields mentioned. Two equations are used depending on the window used. Both these equations are raised to the power of 4/3, which is the inverse power used in the quantizer.

For short blocks, the equation used is:

$$\mathtt{xr}\left[i\right] = \operatorname{sign}\left(\mathtt{is}\left[i\right]\right) \cdot \left|\mathtt{is}\left[i\right]\right|^{\frac{4}{3}} \cdot 2^{A} \cdot 2^{B}$$

with A and B being defined as:

$$\begin{aligned} A &= \frac{1}{4} \cdot \left(\mathtt{global\_gain}\left[\mathtt{gr}\right] - 210 - 8 \cdot \mathtt{subblock\_gain}\left[\mathtt{window}\right]\left[\mathtt{gr}\right]\right) \\ B &= -\left(\mathtt{scalefac\_multiplier} \cdot \mathtt{scalefac\_s}\left[\mathtt{gr}\right]\left[\mathtt{ch}\right]\left[\mathtt{sfb}\right]\left[\mathtt{window}\right]\right) \end{aligned}$$

For long blocks, the equation used is:

$$\mathtt{xr}\left[i\right] = \operatorname{sign}\left(\mathtt{is}\left[i\right]\right) \cdot \left|\mathtt{is}\left[i\right]\right|^{\frac{4}{3}} \cdot 2^{C} \cdot 2^{D}$$

with C and D defined as:

$$\begin{aligned} C &= \frac{1}{4} \cdot \left(\mathtt{global\_gain}\left[\mathtt{gr}\right] - 210\right) \\ D &= -\left(\mathtt{scalefac\_multiplier} \cdot \begin{pmatrix} \mathtt{scalefac\_s}\left[\mathtt{gr}\right]\left[\mathtt{ch}\right]\left[\mathtt{sfb}\right]\left[\mathtt{window}\right] \\ + \mathtt{preflag}\left[\mathtt{gr}\right] \cdot \mathtt{pretab}\left[\mathtt{sfb}\right] \end{pmatrix}\right) \end{aligned}$$

6.5 Reordering

The frequency lines generated by the Requantization block are not always ordered in the same way. In the MDCT block the use of long windows prior to the transformation, would generate frequency lines ordered first by subband and then by frequency. Using short windows instead, would generate frequency lines ordered first by subband, then by window and at last by frequency. In order to increase the efficiency of the Huffman coding the frequency lines for the short windows case were reordered into subbands first, then frequency and at last by window, since the samples close in frequency are more likely to have similar values.

The reordering block will search for short windows in each of the 36 subbands. If short windows are found, they are reordered.

6.6 Stereo Decoding

The purpose of the Stereo Processing block is to perform the necessary processing to convert the encoded stereo signal into separate left/right stereo signals. The method used for encoding the stereo signal can be read from the mode and `mode_extension` in the header of each frame. If Mid/Side Stereo is used, the following equations are used:

$$\begin{aligned}\text{Left channel } L(i) &= \frac{1}{\sqrt{2}} \cdot [M(i) + S(i)] \\ \text{Right channel } R(i) &= \frac{1}{\sqrt{2}} \cdot [M(i) - S(i)]\end{aligned}$$

with M representing middle signal, and S representing side signal.

If Intensity Stereo is used, a variable $\texttt{is\_pos}\,(\texttt{sb})$ will be transmitted for the right channel instead of scalefactors.

6.7 Alias Reduction

In the MDCT block within the encoder it was described that an alias reduction was applied. In order to obtain a correct reconstruction of the audio signal in the algorithms to come the aliasing artifacts must be added to the signal again. The alias reconstruction calculation consists of eight butterfly calculations for each subband, as illustrated in figure 6.2. Aliasing is only applied to granules using short blocks.

The coefficients for the butterfly calculations are calculated using the values from table 6.1 and substituting them in the following formulas:

$$\begin{aligned}\texttt{cs}\,(i) &= \frac{1}{\sqrt{(1 + (c(i)))^2}} \\ \texttt{ca}\,(i) &= \frac{c(i)}{\sqrt{(1 + (c(i)))^2}}\end{aligned}$$

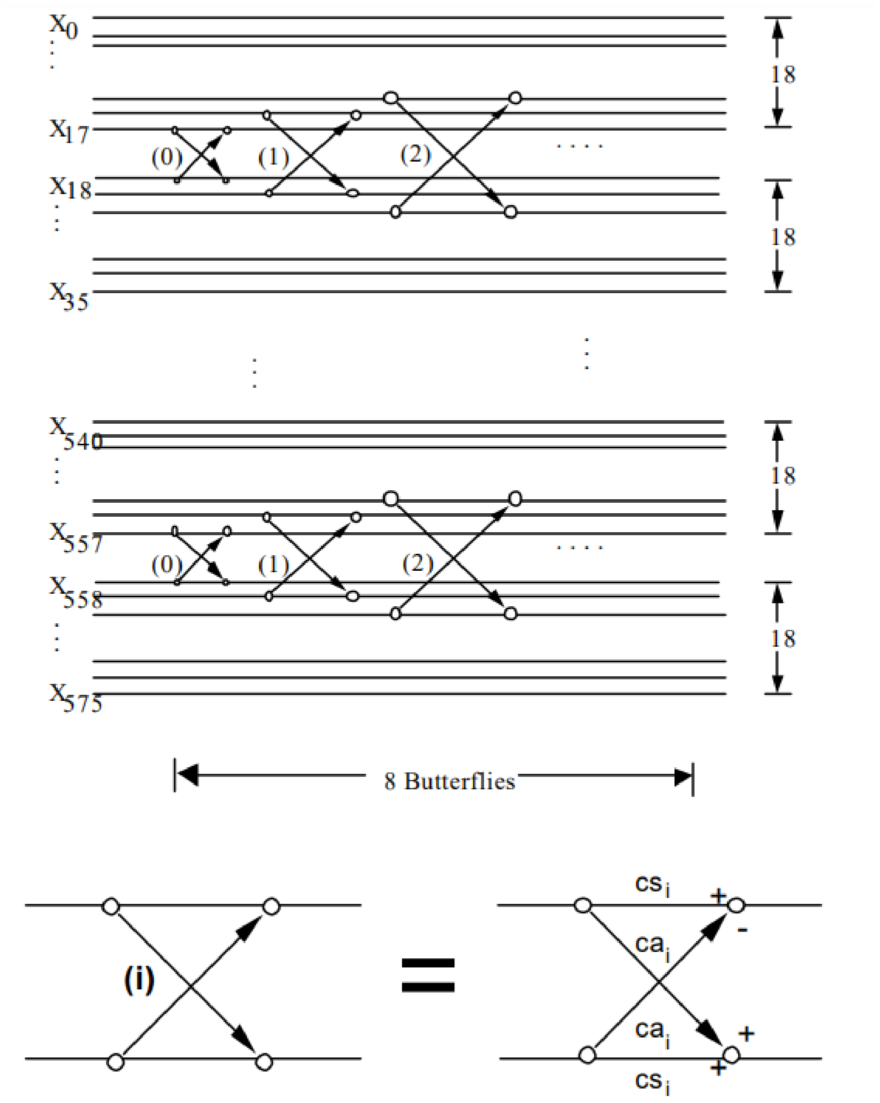


Figure 6.2: Alias Reduction Butterflies

6.8 Inverse Modified Discrete Cosine Transform (IMDCT)

This reverses the MDCT transformation from the encoding process. The frequency lines from the Alias reduction block are mapped to 32 Polyphase filter subbands. The IMDC will output 18 time domain samples for each of the 32 subbands.

The equation for IMDCT is as follows:

$$x(i) = \sum_{k=0}^{(n/2)-1} X(k) \cos\left(\frac{\pi}{2n}\left(2i + 1 + \frac{n}{2}\right)(2k+1)\right)$$

6.9 Frequency Inversion

In order to compensate for frequency inversions in the synthesis polyphase filter bank, every odd time sample of every odd subband is multiplied with -1.

i	0	1	2	3	4	5	6	7
$c(i)$	-0.6	-0.535	-0.33	-0.185	-0.095	-0.041	-00142	-0.0037

Table 6.1: Coefficients for Butterfly Calculations

6.10 Synthesis Polyphase Filterbank

The synthesis Polyphase filterbank transforms the 32 subbands of 18 time domain samples in each granule to 18 blocks of 32 PCM samples, which is the final decoding result.

7 The Structure of an MP3 File

All MP3 files are divided into smaller fragments called frames. Each frame stores 1152 audio samples and lasts for 26 ms. This means that the frame rate will be around 38 fps. In addition a frame is subdivided into two granules each containing 576 samples. Since the bitrate determines the size of each sample, increasing the bitrate will also increase the size of the frame. The size is also depended on the sampling frequency according to following formula:

$$\frac{144\cdot\texttt{bitrate}}{\texttt{samplefrequency}}+\texttt{Padding}\,[\texttt{bytes}]\,.$$

Padding refers to a special bit allocated in the beginning of the frame. It is used in some frames to exactly satisfy the bitrate requirements. If the padding bit is set the frame is padded with 1 byte. Note that the frame size is an integer. Ex: $144\cdot{}^{128000}/_{44100}=417$.

7.1 Frame Layout

A frame consists of four parts; header, side information, main data and ancillary data. A frame might contain CRC, a protection method for sensitive header information, if a corresponding flag in the header is set.

Header	CRC	Side Information	Main Data	Ancillary Data

Table 7.1: MP3 Frame Layout

7.1.1 Frame Header

The frame header is always 32 bits (4 bytes) long and contains a synchronization word together with a description of the frame. The synchronization word found in the beginning of each frame enables MP3 receivers to lock onto the signal at any point in the stream. This allows MP3 decoders to find the beginning of each new frame. It also makes it possible to broadcast any MP3 file. A receiver tuning in at any point of the broadcast just has to search for the synchronization word and then start playing. A problem here is that spurious synchronization words might appear in other parts of the frame. A decoder should instead check for valid syncwords in two consecutive frames, or check for valid data in the side information, which could be more difficult.

Below is figure 7.1 illustrating the frame header, as well as table 7.2 showing which of the header's bits contains which part.

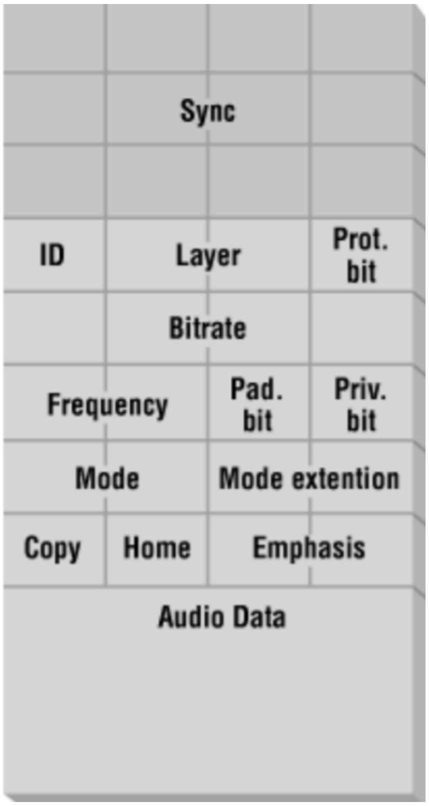


Figure 7.1: MP3 Frame Header

Content	Amount of Bits	Bits In Header
Sync Word	12	1 – 12
ID	1∗	13
Layer	2	14 – 15
Protection Bit	1	16
Bitrate	4	17 – 20
Frequency	2	21 – 22
Padding Bit	1	23
Private Bit	1	24
Mode	2	25 – 26
Mode Extension	2	27 – 28
Copyright Bit	1	29
Home	1	30
Emphasis	2	31 – 32

Table 7.2: MP3 Frame Header fields

Sync. This is the synchronization word described above. All 12 bits must be set, i.e. '1111 1111 1111'.

Id. Specifies the MPEG version. A set bit (1) means that the frame is encoded with the MPEG-1 standard, if not (0) MPEG-2 is used.

Note. Some add-on standards only use 11 bits for the sync word in order to dedicate 2 bits for the id. In that case, the following table applies:

00	MPEG-2.5 (Later extension of MPEG-2)
01	Reserved
10	MPEG-2
11	MPEG-1

Table 7.3: Bit values when using two id bits

Layer. Represents the MPEG layer of the file.

00	Reserved
01	Layer III
10	Layer II
11	Layer I

Table 7.4: Definition of layer bits

Protection Bit. If the protection bit is set, the CRC field will be used.

Bitrate. These 4 bits tell the decoder in what bitrate the frame is encoded. This value will be the same for all frames if the stream is encoded using CBR, but might change if VBR is used. The following table contains defined bit values.

Bits	**MPEG-1, Layer I**	**MPEG-1, Layer II**	**MPEG-1, Layer III**	**MPEG-2, Layer I**	**MPEG-2, Layer II**	**MPEG-2, Layer III**
0000	-	-	-	-	-	-
0001	32	32	32	32	32	8
0010	64	48	40	64	48	16
0011	96	56	48	96	56	24
0100	128	64	56	128	64	32
0101	160	80	64	160	80	64
0110	192	96	80	192	96	80
0111	224	112	96	224	112	56
1000	256	128	112	256	128	64
1001	288	160	128	288	160	128
1010	320	192	160	320	192	160
1011	352	224	192	352	224	112
1100	384	256	224	384	256	128
1101	416	320	256	416	320	256
1110	448	384	320	448	384	320
1111	-	-	-	-	-	-

Table 7.5: Bitrate Definitions

Bits	**MPEG-1**	**MPEG-2**	**MPEG-2.5**
00	44, 100 Hz	22, 050 Hz	11, 025 Hz
01	48, 000 Hz	24, 000 Hz	12, 000 Hz
10	32, 000 Hz	16, 000 Hz	8, 000 Hz
11	-	-	-

Table 7.6: Definition of accepted sampling frequencies

Frequency. 2 bits that give the sampling frequency.

Padding Bit. An encoded stream with bitrate 128 kbits/s and sampling frequency of 44100 Hz will create frames of size 417 bytes. To exactly fit the bitrate some of these frames will have to be 418 bytes. These frames set the padding bit.

Private Bit. One bit for application-specific triggers.

Mode. Specifies what channel mode is used.

00	Stereo
01	Joint Stereo
10	Dual Channel
11	Single Channel

Table 7.7: Channel Modes and respective bit values

Mode Extension. These 2 bits are only usable in joint stereo mode and they specify which methods to use. The joint stereo mode can be changed from one frame to another, or even switched on or off. To interpret the mode extension bits the encoder needs the information in the following table.

Bits	**Intensity Stereo**	**MS Stereo**
00	Off	Off
01	On	Off
10	Off	On
11	On	On

Table 7.8: Definition of mode extension bits

Copyright Bit. If this bit is set it means that it is illegal to copy the contents.

Home. The original bit indicates, if it is set, that the frame is located on its original media.

Emphasis. The emphasis indication is used to tell the decoder that the file must be de-emphasized, i.e. the decoder must 're-equalize' the sound after a noise suppression, such as ones used by Dolby products. It is rarely used.

00	None
01	50/15ms
10	Reserved
11	CCITT J.17

Table 7.9: Noise Suppression Model

7.1.2 CRC

If exists, the CRC (Cyclic Redundancy Check) field is 16 bytes long and comes right after the frame header ends. This field will only exist if the protection bit in the header is set and makes it possible to check the most sensitive data for transmission errors. Sensitive data is defined by the standard to be bit 16 to 31 in both the header and the side information. If these values are incorrect they will corrupt the whole frame whereas an error in the main data only distorts a part of the frame. A corrupted frame can either be muted or replaced by the previous frame. CRC itself is a type of checksum error-detecting code based on polynomial division. For additional information, see bibliography.

7.1.3 Side Information

The side information part of the frame consists of information needed to decode the main data. The size depends on the encoded channel mode. If it is a single channel bitstream the size will be 17 bytes, if not, 32 bytes are allocated. The different parts of the side information are presented in table 7.10.

main_data_begin	private_bits	scfsi	Side_info gr.0	Side_info gr.1

Table 7.10: Frame Side Information

Table 7.11 contains the length of each field in the side information. Note that these lengths change based on whether the frame's channel mode is set to Mono or Stereo. All tables below assume Mono channel, if Stereo is set they are replicated for each channel (separate values for each channel).

$*$ Starting from part_2_3_length and onward, each granule contains its own field of these types. Thus each granule contains said amount of bits for each channel. For example, on Single Channel, each granule will contain 12 bits for the part_2_3_length field.
$**$ See window_switching below.
$***$ See tabel_select below.

Content	**Amount of Bits (Single Channel)**	**Amount of Bits (Dual Channel)**
main_data_begin	9	9
private_bits	5	3
scfsi	4	8
part_2_3_length*	12	24
big_values	9	18
global_gain	8	16
scalefac_compress	4	8
window_switching	1	2
block_type**	2	4
mixed_block_flag**	1	2
table_select***	$2 \cdot 5$ or $3 \cdot 5$	$2 \cdot 2 \cdot 5$ or $2 \cdot 3 \cdot 5$
subblock_gain**	9	18
region0_count	4	8
region1_count	3	6
preflag	1	2
scalefac_scale	1	2
count1table_select	1	2

Table 7.11: Side Information field sizes

main_data_begin. A pointer for the start of this frame's main data. Using the layer III format there is a technique called the bit reservoir which enables the leftover free space in the main data area of a frame to be used by consecutive frames. To be able to find where the main data of a certain frame begins the decoder has to read the main_data_begin value. The value is as a negative offset from the first byte of the synchronization word. Since it is 9 bits long it can point $(2^9-1) \cdot 8 = 4088$ bits. This means that data for one frame can be found in several previous frames. Note that static parts of a frame like the header, which is always 32 bytes, are not included in the offset. If main_data_begin $= 0$ the main data starts directly after the side information.

Private Bits. Bits for private use, these will not be used in the future by ISO.

scfsi. The ScaleFactor Selection Information determines weather the same scalefactors are transferred for both granules or not. Here the scalefactor bands are divided into 4 groups according to table 7.12. 4 bits per channel are transmitted, one for each scalefactor band.

Group	**Scalefactor Bands**
0	0,1,2,3,4,5
1	6,7,8,9,10
2	11,12,13,14,15
3	16,17,18,19,20

Table 7.12: Scalefactor Groups

If a bit belonging to a scalefactor band is zero the scalefactors for that particular band are transmitted for each granule. A set bit indicates that the scalefactors for granule0 are also valid for granule1. This means that the scalefactors only need to be transmitted in granule0, the gained bits can be used for the Huffman coding.

If short windows are used (`block_type` = 10) in any granule/channel, the scalefactors are always sent for each granule for that channel.

Side info for each granule. The last two parts of a frame have the same anatomy and consist of several subparts. These two parts store particular information for each granule respectively.

part_2_3_length. States the number of bits allocated in the main data part of the frame for scalefactors (part2) and Huffman encoded data (part3). This field can be used to calculate the location of the next granule and the ancillary information (if used).

big_values. The 576 frequency lines of each granule are not coded with the same Huffman code table. These frequencies range from zero to the Nyquist frequency and are divided into five regions (see figure 7.2).

The purpose of this partitioning is to allow different Huffman tables to different parts of the spectrum in order to enhance the performance of the Huffman encoder. Partitioning is done according to the maximum quantized values. This is done with the assumption that values at higher frequencies are expected to have lower amplitudes or do not need to be coded at all.

The rzero region represents the highest frequencies and contains pairs of quantized values equal to zero. In the count1 region quadruples of quantized values equal to -1, 0 or 1 reside. Finally, the `big_values` region contains pairs of values in representing the region of the spectrum which extends down to zero. The maximum absolute value in this range is constrained to 8191. The `big_values` field indicates the size of the `big_values` partition hence the maximum value is 288.

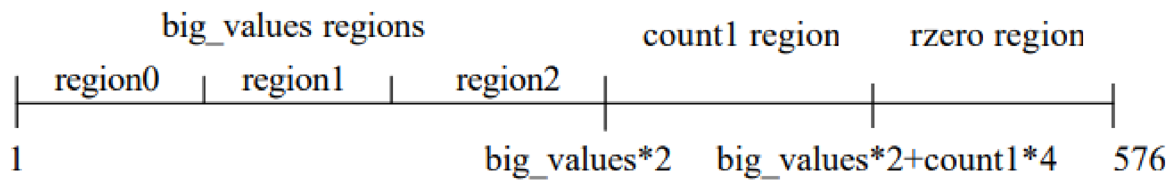


Figure 7.2: Regions of the frequency spectrum

global_gain. Specifies the quantization step size, this is needed in the requantization block of the decoder.

scalefac_compress. Determines the number of bits used for the transmission of scalefactors. A granule can be divided into 12 or 21 scalefactor bands. If long windows are used (`block_type` $= \{0, 1, 3\}$) the granule will be partitioned into 21 scalefactor bands. Using short windows (`block_type` $= 2$) will partition the granule into 12 scalefactor bands. The scale factors are then further divided into two groups, $0 - 10$, $11 - 20$ for long windows and $0 - 6$, $7 - 11$ for short windows.

The `scalefac_compress` variable is an index to a defined table (see table 7.13). `slen1` and `slen2` gives the number of bits assigned to the first and second group of scalefactor bands respectively.

window_switching_flag. Indicates that another window than the normal is used (see subsection 6.2). `block_type`, `mixed_block_flag` and `subblock_gain` are only used if `windows_switching_flag` is set. Also when `windows_switching_flag` is set all remaining values not being in `region0` are contained in `region1`, thus `region2` is not used.

block_type. This field is only used when `windows_switching_flag` is set and indicates the type of window used for the particular granule (see table 7.14, all values but 3 are long windows). The value 00 is forbidden since `block_type` is only used when other than normal windows are used.

mixed_block_flag. This field is only used when `windows_switching_flag` is set.

The `mixed_block_flag` indicates that different types of windows are used in the lower and higher frequencies. If `mixed_block_flag` is set the two lowest subbands (see subsection 6.1) are transformed using a normal window and the remaining 30 subbands are transformed using the window specified by the `block_type` variable.

table_select. There are 32 possible Huffman code tables available in the standard. The value of this field gives the Huffman table to use when decoding and its size is 5 bits, i.e. 32 different values, for each region, granule and channel. The `table_select` only specifies the tables to use when decoding the `big_values` partition. The table specified is dependent on the local statistics of the signal and by the maximum quantization allowed to quantize the 576 frequency lines in the granule.

saclefac_compress	slen1	slen2
0	0	0
1	0	1
2	0	2
3	0	3
4	3	0
5	1	1
6	1	2
7	1	3
8	2	1
9	2	2
10	2	3
11	3	1
12	3	2
13	3	3
14	4	2
15	4	3

Table 7.13: scalefac_compress values

As stated above, when the `windows_switching_flag` is set `region2` is empty so only two regions are coded. This implies that in mono mode $5 \cdot 2 \cdot 1 = 10$ bits are needed and in stereo mode $5 \cdot 2 \cdot 2 = 20$ bits are needed if `windows_switching_flag` $= 1$. Using all regions (`windows_switching_flag` $= 0$) the bits needed will be $5 \cdot 3 \cdot 1 = 15$ and $5 \cdot 3 \cdot 2 = 30$ respectively.

subblock_gain. This field is only used when `windows_switching_flag` is set and when `block_type` $= 2$, although it is transmitted independently of `block_type`. This 3 bit variable indicates the gain offset from `global_gain` for each short block.

region0_count, region1_count. `region0_count` and `region1_count` contain one less than the number of scalefactor bands in `region0` and `region1` respectively. The region boundaries are adjusted to the partitioning of the frequency spectrum into scalefactor bands. If short windows are used the number of each windows is counted. For instance if `region0` $= 8$ there are $9/3 = 3$ scalefactor bands in `region0`.

preflag. This is a shortcut for additional high frequency amplification of the quantized values. If `preflag` is set, the values of a defined table (7.15) are added to the scalefactors.

block_type	**window type**
00	forbidden
01	start
10	3 short windows
11	end

Table 7.14: block_type values

If `block_type` = 2, i.e short blocks, `preflag` is never used.

scalefac_scale	0	1	2	3	4	5	6	7	8	9	10
pretab	0	0	0	0	0	0	0	0	0	0	0
scalefac_scale	12	12	13	14	15	16	17	18	19	20	
pretab	1	1	1	1	2	2	3	3	3	2	

Table 7.15: Preflag table

scalefac_scale. The scalefactors are logarithmically quantized with a step size of 2 or $\sqrt{2}$ according to table 7.16.

scalefac_scale	**step size**
0	$\sqrt{2}$
1	2

Table 7.16: Quantization step size applied to scalefactors

count1tabel_select. Two possible Huffman code tables are available for the count1 region. This field specifies which table to apply.

7.1.4 Main Data

The main data part of the frame consists of scalefactors, Huffman coded bits and ancillary data.

scalefactors. The purpose of scalefactors is to reduce the quantization noise. If the samples in a particular scalefactor band are scaled the right way, the quantization noise will be completely masked. One scalefactor for each scalefactor band is transmitted. The scfsi field determines if the scalefactors are shared between the granules or not. The actual bits allocated for each scalefactor depends on the `scalefac_compress` field.

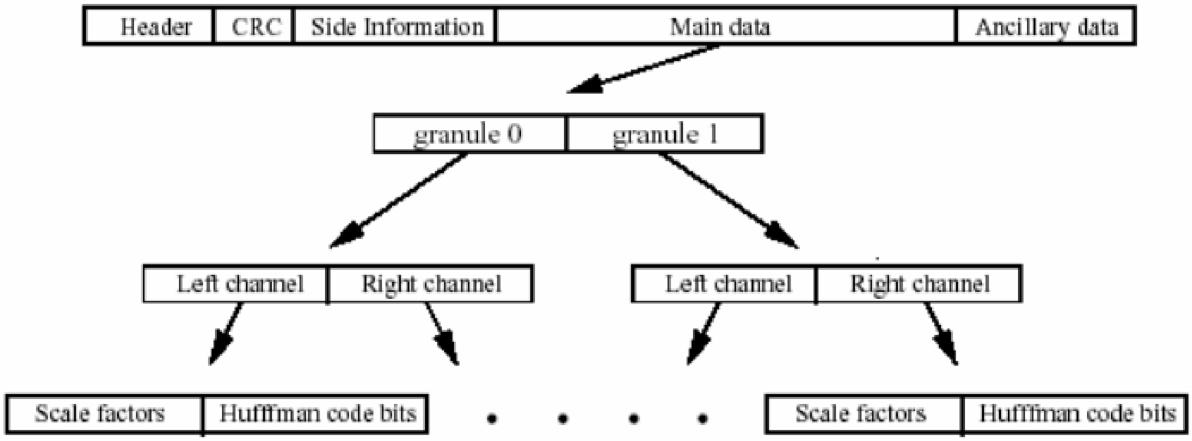


Figure 7.3: Organization of main data to granules and channels

The subdivision of the spectrum into scalefactor bands is fixed for every window length and sampling frequency and stored in tables in the encoder and decoder.

Huffman code bits. This part of the frame contains the Huffman code bits. Information on how to decode these is found in the side information. For the three subregions in the `big_values` region always pairs are encoded. For instance, the Huffman table number 7 (see 7.4) may be applied to `big_values`. x and y are the pair of values to be coded, hlen specifies the length of the Huffman code for x and y and hcode is the actual Huffman code representing x and y. In the case of `count1` values, they are encoded in quadruples, v, w, x, y. Only tables A and B support coding in this region. The `rzero` region is not Huffman coded but run length coded since all values are zero.

Depending on whether long or short blocks are used, the order of the Huffman data differs. If long blocks are used, the Huffman encoded data is ordered in terms of increasing frequency.

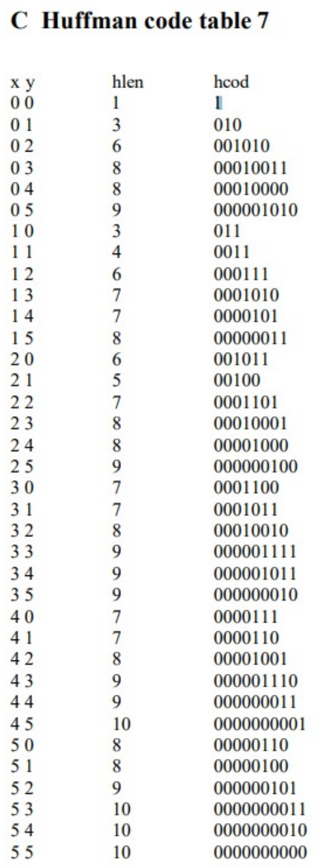
7.1.5 Ancillary Data

The ancillary data is optional and the number of bits available is not explicitly given. The ancillary data is located after the Huffman code bits and ranges to where the next frame's `main_data_begin` points to. It can contain additional data added by the encoder.

7.2 Metadata

7.2.1 ID3

Although the MP3 format did manage to compress audio files very effectively without any noticeable quality degradation, there was no possibility to store textual information. For this purpose a fixed-size 128-byte tag at the end of the file was introduced. The tag was called ID3 and contained fields for title (30 bytes), artist (30 bytes), album (30 bytes), year (4 bytes), comment (30 bytes) and genre (1 byte). The byte value in the genre field



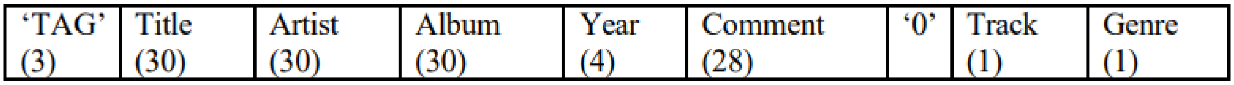
C Huffman code table 7

x y	hlen	hcod
0 0	1	1
0 1	3	010
0 2	6	001010
0 3	8	00010011
0 4	8	00010000
0 5	9	000001010
1 0	3	011
1 1	4	0011
1 2	6	000111
1 3	7	0001010
1 4	7	0000101
1 5	8	00000011
2 0	6	001011
2 1	5	00100
2 2	7	0001101
2 3	8	00010001
2 4	8	00001000
2 5	9	000000100
3 0	7	0001100
3 1	7	0001011
3 2	8	00010010
3 3	9	000001111
3 4	9	000001011
3 5	9	000000010
4 0	7	0000111
4 1	7	0000110
4 2	8	00001001
4 3	9	000001110
4 4	9	000000011
4 5	10	0000000001
5 0	8	00000110
5 1	8	00000100
5 2	9	000000101
5 3	10	0000000011
5 4	10	0000000010
5 5	10	0000000000

Figure 7.4: Example of a Huffman code table

corresponds to a value in a predefined list. An MP3 file using the ID3 tag should write 'TAG' at the end of the ordinary encoding. Counting these 3 bytes needed to write 'TAG' the ID3 tag will be 128 bytes. Unfilled space in each field should be filled with the binary value 0.

The comment field was later reduced by two bytes in order to include a one byte track field telling the decoder which track number on the CD this music came from. The byte left over should be a binary 0 and written between the comment field and the track field. This variant of the ID3 tag was named ID3v1.1 (see figure 7.5).



'TAG' (3)	Title (30)	Artist (30)	Album (30)	Year (4)	Comment (28)	'0'	Track (1)	Genre (1)

Figure 7.5: ID3v1.1

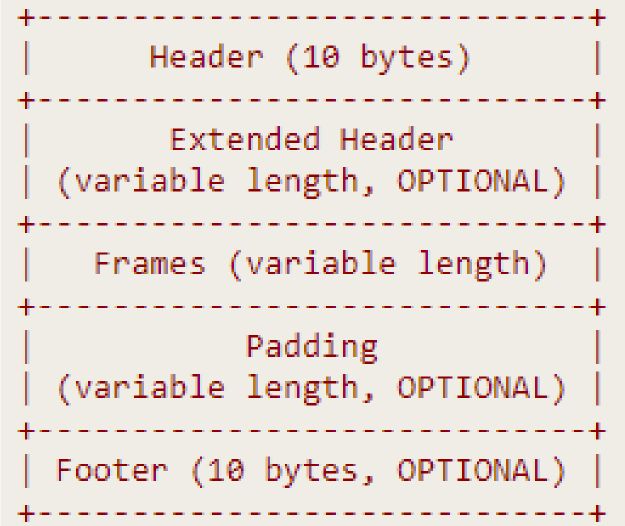
7.2.2 ID3v2

Unfortunately ID3 was not a clever way to store textual information. It only supported a few fields of information and these were limited to 30 characters. An additional drawback was that since the tag was put at the end the information could not be retrieved when streaming the file. With respect to these drawbacks a second more complex version was released, ID3v2.

If this tag is used 'ID3' will be written at the very beginning of the file. ID3v2 introduced a lot of new fields and mostly every field can store information of any length since this tag is dynamic in size. This is achieved by dividing the metadata into frames, and just like the actual MP3 bitstream, each new frame starts with a sync word. 83 types of frames are defined for ID3v2, and applications can define additional ones for specific uses. There are frames for simple textual metadata (song name, artist name, etc.) as well as ones for album pictures, tempo information, legal and copyright information, lyrics, relevant links and more.

The tag is located in the beginning of the file to facilitate streaming. The current informal standard is ID3v2.4.

The ID3v2 contains five parts – a header, an extended header (optional), frames, padding, and footer (optional). See figure 7.6.



```
+-----------------------------+
|      Header (10 bytes)      |
+-----------------------------+
|       Extended Header       |
| (variable length, OPTIONAL) |
+-----------------------------+
|   Frames (variable length)  |
+-----------------------------+
|           Padding           |
| (variable length, OPTIONAL) |
+-----------------------------+
| Footer (10 bytes, OPTIONAL) |
+-----------------------------+
```

Figure 7.6: ID3v2

ID3v2 Header. The first part of an ID3v2 tag is always a header. It's length is 10 bytes, and it can be recognized by the following pattern: $49 44 33 *yy yy xx zz zz zz zz*

- The first 3 bytes represent the characters 'ID3', thus telling the decoder that ID3v2 is in use.
- The next 2 bytes (y) represent the current format version. For example, 00 04.
- The sixth byte (x) represents flags for the tag. Currently, four flags can be used for ID3v2, which are represented by the first 4 bits of the byte: %abcd0000. Flag a is set if unsynchronization should be applied on all frames (see ID3v2 Frame below).

Flag b is set if an extended header can be found after this header. Flag c indicates an experimental version of the ID3 format. Flag d is set if a footer can be found at the end of the tag. Note that all other bits in this byte must be unset – otherwise the tag may become unreadable to a decoder that does not know the flags purposes.

- The last 4 bytes (z) represent the tag size, which is the sum of frame sizes, padding and extended header (does not include the header or the footer). If a footer exists, this number shall be (total tag size – 20) bytes, otherwise it will be (total tag size – 10) bytes.

ID3v2 Extended Header. The extended header contains fields that can grant additional insight on the tag's structure. It is, however, not required for correct parsing of the tag, and thus optional.

The first 4 bytes represent the extended header's size, which is not fixed. The fifth byte represents the amount of set extended flags.

The sixth byte represents the extended flags themselves. The rest of the bytes in the extended header (starting from the 7th byte and onto the length found in the first four bytes) represent information related to these flags. This information is found in the same order as the set flags (i.e. info for flag b will be found before info for flag c). Each such information segment starts with one byte representing the segment's length, then the information itself in that amount of bytes. Flags with no additional information use byte $00 as their information length. As per said in the normal header, undefined flags cannot be set, and they do not have information related to them.

The byte can be defined as %0bcd0000.

Flag b is set if the tag is an update of a metadata tag previously found in the file or stream. Unique frames in this tag shall therefore override the corresponding frames in the old tag. This flag has no additional information attached to it (length byte $00).

Flag c is set if CRC is applied on the tag (see 5.1.2 for additional info). The CRC is calculated on all the data between the header and footer as indicated by the header's tag length field, minus the extended header. Note that this includes the padding (if there is any), but excludes the footer. The CRC is stored as this flag's additional information, as an 35 bit synchsafe integer, leaving the upper four bits always zeroed. Thus, the data length byte will be $05, and additional info will be 5 * %0xxxxxxx.

Flag d is set if some restrictions are imposed on the tag. For some applications it might be desired to restrict a tag in more ways than imposed by the ID3v2 specification. Note that the presence of these restrictions does not affect how the tag is decoded, merely how it was restricted before encoding. If this flag is set, it's data length byte is $01, and the single byte of additional information represents which restrictions are imposed on the tag. This byte can be defined as %ppqrrstt. The following tables show the meaning for various values of these bits:

00	No more than 128 frames and 1 MB total tag size.
01	No more than 64 frames and 128 KB total tag size.
10	No more than 32 frames and 40 KB total tag size.
11	No more than 32 frames and 4 KB total tag size.

Table 7.17: p - Tag size restrictions

0	No restrictions.
1	Strings are only encoded with ISO-8859-1 or UTF-8.

Table 7.18: q - Text encoding restrictions

00	No restrictions.
01	No string is longer than 1024 characters.
10	No string is longer than 128 characters.
11	No string is longer than 30 characters.

Table 7.19: r - Text field size restrictions*

*Note that nothing is said about how many bytes is used to represent those characters, since it is encoding dependent. If a text frame consists of more than one string, the sum of the strings is restricted as stated.

0	No restrictions.
1	Images are encoded only with PNG or JPEG.

Table 7.20: s - Image encoding restrictions

00	No restrictions.
01	All images are 256×256 pixels or smaller.
10	All images are 64×64 pixels or smaller.
11	All images are exactly 64×64 pixels, unless required otherwise.

Table 7.21: t - Image size restriction

Padding. It is optional to include padding after the final frame (at the end of the ID3 tag), making the size of all the frames together smaller than the size given in the tag header. A possible purpose of this padding is to allow for adding a few additional frames or enlarge existing frames within the tag without having to rewrite the entire file. The value of the padding bytes must be $00. A tag must not have any padding between the frames or between the tag header and the frames.

Furthermore it must not have any padding when a tag footer is added to the tag.

ID3v2 Footer. To speed up the process of locating an ID3v2 tag when searching from the end of a file, a footer can be added to the tag. It is required to add a footer to an appended tag, i.e. a tag located after all audio data. The footer is a copy of the header, however the first three bytes are "3DI" for the ease of backwards searching.

ID3v2 Frame. Each frame in the tag includes a header, followed by one or more fields containing the frame's data. A frame header is made of 10 bytes.

- The first 4 bytes represent the frame's ID, and they can be capital letters A-Z and digits 0-9. IDs starting with X,Y,Z are used for experimental frames and are open for personal uses. All other IDs are used for specific frames used in the format, or are saved for future use.

- The next 4 bytes represent the frame's final size, excluding the header (total frame size − 10).

- The last 2 bytes represent frame flags. The first byte is status flags and can be represented as %0abc0000. Flag a is set if the frame should be discarded on tag alteration. Flag b is set if the frame should be discarded on file alteration. Flag c is set if the contents of the frame are read-only, and changing them might break something (for example, a signature). The second byte is format flags and can be represented as %0h00kmnp. Flag h is set if this frame belongs in a group with other frames – if set, a group identifier byte is added to the frame. Flag k is set if the frame is compressed (compression method is zlib deflate method) – if set, a data length indicator byte must be included. Flag m is set if the frame is encoded – if set, a byte representing the encoding method is included. Flag n is set if unsynchronisation is used – a process whose only purpose is to make the ID3v2 tag as compatible as possible with existing software and hardware. There is no use in 'unsynchronising' tags if the file is only to be processed only by ID3v2 aware software and hardware (See id3v2 structure in bibliography for additional information about the workings of this process). Flag p is set if a data length indicator has been added to the frame. The data length indicator is the value one would write as the 'Frame length' if all of the frame format flags were zeroed, represented as a 32 bit synchsafe integer.

An ID3v2 tag has to contain at least one frame. A frame must have a size of at least one byte, excluding the header. There does not exist a specific order in which frames should appear. Defaultly, strings are encoded using ISO 8859, but other standards (like UTF-8) can be used if the corresponding flag is set. Other than strings, common fields in frame may include timestamps or used language. Specific frames might contain fields specific to their uses (for example, an image frame might contain a URL). Some common types of frames set in the standard include comment frames (id COMM), album title (id TALB), and track number (id TRCK). For a full list of known frames, see bibliography.

8 Conclusions

As assumed, the audio part of the ISO MPEG-1 standard is very complex. It contains several subprocedures to achieve the optimal compression. These include the psychoacoustic model, which determines non perceptible signals, the filterbanks and cosine transforms, which effectively handles the mapping between the frequency- and the time-domains, scaling and quantization of the sample values and finally the Huffman coding. Both lossy and lossless compression has to be combined in the process. Neither of the two alone will be able to reduce the data to meet the compression demands.

Part II
Steganography in MP3

9 Introduction to Steganography

While cryptography is the art of securing private communication data, in a way that third parties may not be allowed to observe it, *steganography* is the art of hiding such data in plain sight, concealing a private message in, for example, another message, an image, audio, or some other seemingly innocent object. In essence, it is the art (and science) of communicating in a way which hides the existence of communication.

The word comes from Greek *steganographia*, which combines the words steganós, meaning “covered or concealed”, and *-graphia* meaning “writing”. And indeed, The first recorded uses of steganography can be traced back to 440 BC in Greece, when Herodotus mentions two examples in his *Histories*. Histiaeus sent a message to his vassal, Aristagoras, by shaving the head of his most trusted servant, “marking” the message onto his scalp, then sending him on his way once his hair had regrown, with the instruction, “When thou art come to Miletus, bid Aristagoras shave thy head, and look thereon.” Additionally, Demaratus sent a warning about a forthcoming attack to Greece by writing it directly on the wooden backing of a wax tablet before applying its beeswax surface. Wax tablets were in common use then as reusable writing surfaces, sometimes used for shorthand.

However, technology has highly evolved since those times. When speaking in computing contexts, steganography methods are used to hide crucial data in binary files. And with the rapid growth in use of digital data communication, it is becoming more and more essential to secure data transferred in networks against steganography methods that might, for example, steal sensitive information, or instill malicious code into the network.

The following figure demonstrates the workings of a general steganography system:

1. The sender writes a *cover message*, which is non-secret.

2. A *stego-message* is produced by hiding a *secret message* embedded on the *cover message* by using a *stego-key*. A stego-key might be a pre-agreed upon protocol or algorithm between the sender and the receiver.

3. The sender sends the *stego-message* over the insecure channel to the receiver.

4. On receiving the *stego-message* at the receiver’s end, the intended receiver extracts the secret embedded message from the *stego-message* by using the pre-agreed upon *stego-key*.

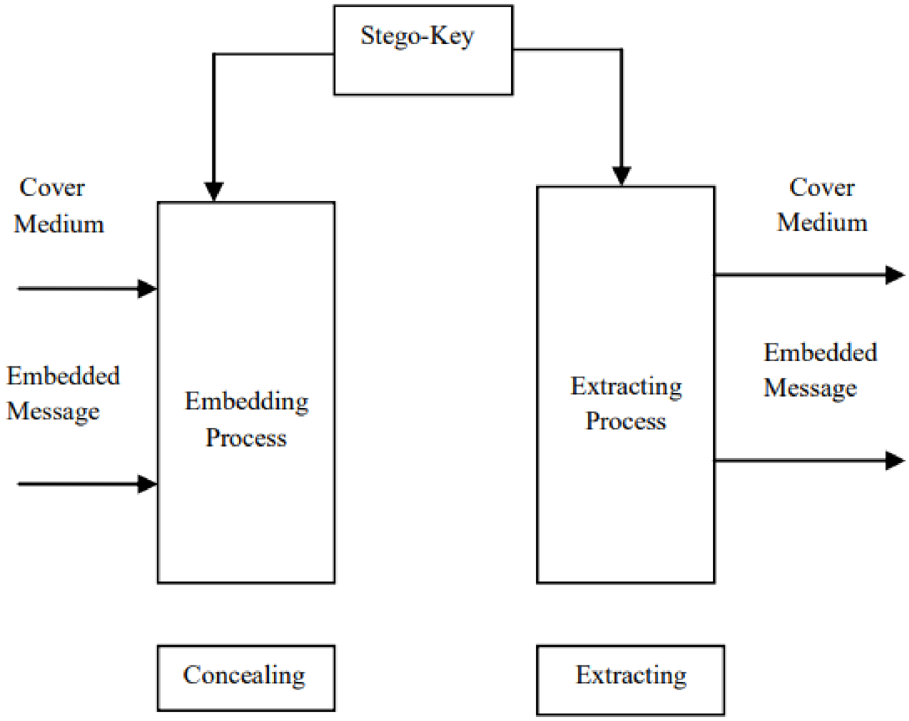


Figure 9.1: The Steganographic System

10 Types of Steganography

10.1 Steganography In Text

Perhaps the most common type of file sent and received in communications, text files are also notoriously difficult to use steganography with. This is because, when thinking of steganographic methods for a file format, one would like to try and use parts of the file that might be redundant – meaning that adding a hidden message in those parts will change little to none of the file's anatomy and readability by humans. Text files contain little redundant data, as most changes with them correspond to actually changing the text within, which is problematic from a steganographic standpoint. Another con is the ease at which text based Steganography can be altered by unwanted parties by just changing the text itself, or reformatting the text into some other form (from .TXT to .PDF, etc).

10.2 Steganography In Images

Image steganography is very effective, efficient, and can serve a variety of purposes, including authentication, concealing of messages, etc. Steganographic methods in images usually include an altering of the image's pixels, in a way that only has small visual effects on the image itself.

10.3 Steganography In Video

Steganography in videos basically deals with hiding of information in each frame of video. This might combine steganographic methods of both video and audio, or use specific breaches in the video format.

10.4 Steganography In Audio

In audio steganography systems, secret messages are embedded into the digitized audio signal which results in altering the binary sequence of corresponding audio files. These techniques usually exploit the human auditory system, making changes in the audio signal that the naked ear cannot distinguish between. Audio based steganography has more potential to conceal information, as audio files are larger than images, and even a small change in amplitude can store large amounts of information.

11 Audio Steganography

11.1 Overview

The main goal of Steganography is to communicate securely in a completely undetectable manner and to avoid drawing suspicion to the transmission of a hidden data. This fact makes audio a great candidate for the job – if one can communicate a message using a cover of an innocent sounding audio, bystanders may not even notice the existence of such a message in the signal.

Audio Steganography is focused on hiding secret information in an innocent cover audio file or signal in a securely and strongly fashioned manner. Communication security and robustness are vital for transmitting important information to authorized entities, while denying access to unpermitted ones. By embedding secret information using an audio signal as a cover medium, the very existence of secret information is hidden away during communication. This is a serious and vital issue in some applications such as battlefield communications and banking transactions. In computer-based audio steganography systems, secret messages are embedded in *digital sound*. The secret message is embedded by slightly altering the binary sequence of a sound file. Existing audio Steganography software can embed messages in WAV, AU, and even MP3 sound files.

The basic model of audio steganography consists of a *carrier* (audio file), a *message* and a *password*. The carrier is also known as a cover-file, which conceals the secret information. A message is the data that the sender wishes to remain confidential. A message can be plain text, image, audio or any type of file. Password is known as a *stego-key*, which ensures that only the recipient who knows the corresponding decoding key will be able to extract the message from a cover-file. The cover-file with the secret information is known as a *stego-file*.

The information hiding process in digital sound consists of the following two steps:

1. Identification of redundant bits in the cover-file. Redundant bits are those bits that can be modified without corrupting the quality or destroying the integrity of the cover-file.

2. To embed the secret message in the cover file, the redundant bits in the cover file are replaced by the bits of the secret information.

11.2 Encoding Secret Information in Audio

Encoding secret messages in audio is one of the most challenging techniques to use when dealing with steganography. This is because the *human auditory system* (HAS) has such a dynamic range that it can listen over. To put this in perspective, the HAS perceives over a range of power greater than one million to one and a range of frequencies greater than one thousand to one, making it extremely difficult to add or remove data from the original audio structure without this being perceivable. The only weak spot in the HAS can be found in psychoacoustic effects (see part I, subsection 3.1). Exploiting such effects is one common way to encode messages in audio without being detected.

11.3 Advantages of Audio Steganography

- Audio based steganography has the potential to hide more information (also known as *capacity*) than other mediums:
  - Audio files are generally larger than images or text files.
  - Slight changes in amplitude can store vast amounts of information.
- The flexibility of audio steganography makes it potentially very powerful:
  - The methods that are discussed in section 12 provide users with a large amount of choice and makes the technology more accessible to everyone. A party that wishes to communicate can rank the importance of factors such as data transmission rate, bandwidth, robustness, or noise audibility, and select the method that best fits their specifications.
  - For example, two individuals who just want to send the occasional secret message back and forth might use the LSB coding method (12.3) that is easily implemented. On the other hand, a large corporation wishing to protect its intellectual property from pirating might consider a more sophisticated method such as phase coding (12.5), spread spectrum (12.6) or echo hiding (12.7).
- One aspect that makes audio based steganography attractive is the ease at which these methods can be combined with existing cryptography technologies, in respect to other steganographic mediums. Thus a secret message can be encrypted, as well as hidden altogether, to maximize security.
- Security advantages:

  - Many attacks that are possible against the more commonly used image steganography (such as geometrical distortion, spatial scaling, etc.) cannot be implemented against audio steganography schemes. Consequently, embedding information in audio seems more secure due to less *steganalysis* techniques for attacking audio.
  - Many of the methods described make statistical analysis difficult – this increases robustness.
  - As emphasis placed on the areas of copyright protection and privacy protection in audio increase, steganography will continue to grow in importance as a protection mechanism.
  - Audio steganography in particular addresses key issues brought by the MP3 format, P2P software, and the need for a secure broadcasting scheme that can maintain the secrecy of the transmitted information, even when passing through insecure channels.

11.4 Disadvantages of Audio Steganography

- Embedding additional information into audio sequences is a more tedious task than doing that with images, due to the dynamic supremacy of the HAS over the human visual system.
- Robustness: hidden information in audio samples could be easily manipulated or destroyed, if a miscreant comes to know the information is hidden this way.
- On a commercial level, audio steganography will usually be easily recognized, meaning it is limited in size in terms of large scale data capacity.
- Compressing an audio file with lossy compression (like in MP3 files) may result in loss of the hidden message if not handled carefully, as it will change the whole structure of the file. Also, lossy compression methods that use the HAS to their advantage by removing all frequencies that cannot be heard, also remove the possibility of taking advantage of those frequencies for steganographic methods.

11.5 Applications

There are many applications for information hiding in today's world. However, while information hiding can be used in ethical ways, there are some ways that digital data hiding could be misused.

- Secret communication: commonly, steganography can be used in order to hide data and information in confidential communication. As an example, it can be a trusted way for those who want to have an undisclosed conversion, and especially in countries where cryptography is banned or otherwise untrustable.
- Secure storage: steganography is not only useful during communication, but also as a secure method for storing confidential data. Examples include medical records

and drug prescriptions of patients – information that could lead to awful damage if fallen into the wrong hands.

- Covert communication: some people or organizations have the need for a cover communication scheme for their operations. For example, the military may use audio steganography in radio communications for sending battle plans to the front lines that should not be compromised. In these situations, it is in best interest to also apply cryptography to achieve more security.
- Fingerprinting: the originators or recipients of a specific copy of an audio file could be traced with fingerprint watermarking. For example, before distributing copies of multimedia products to recipients, they can be watermarked using steganography by way of serial numbers.
- Copyright protection: In the copyright protection, a watermark which contains the information of the owner is embedded into the host audio. The watermark is supposed to be robust and enables the owner to prove their ownership in case it is needed. Also, with fragile watermarking, watermarks are used to verify if the host signals are tampered with. Furthermore, in the copy control application, watermarks control access policy or limit to a certain copy.

11.6 Requirements of the Efficient Steganography Technique

According to IFPI (International Federation of the Phonographic Industry), audio Steganography algorithms should meet certain requirements. The most significant requirements are perceptibility, reliability, capacity, and speed performance.

11.6.1 Perceptibility

Perhaps the most important feature of an efficient steganography technique is that the stego-signal should not lose the quality of the original signal. The measurement unit for this is known as the *signal-to-noise ratio* (SNR), and by IFPI standards should be maintained greater than 20DB in order for the technique to be considered efficient. In addition, the technique should make the modified signal not perceivable by the human ear.

11.6.2 Reliability

Reliability covers the features like the robustness of the stego-signal against attacks and signal processing techniques. The signal should be made in a way that they provide high robustness against attacks. In addition, watermark detection rates should be high under any types of attacks in the situations of proving ownership.

11.6.3 Capacity

The efficient audio steganography technique should be able to carry more information in terms of volume, while also not degrading the quality of the original signal. It is also important to know if the hidden information is completely distributed over the host signal, because it is possible that near the extraction process only a part of the signal is available. Hence, capacity is also a primary concern in real-time streaming situations.

11.6.4 Speed Performance

Speed of embedding is one of the main criteria for efficient steganography techniques. The speed of embedding of watermark is especially important in real time applications where the embedding is done on continuous signals. Thus, Both the embedding and the extraction process need to be made as fast as possible with greater efficiency.

11.7 Problems and Attacks On Audio Steganography

As discussed above, both robustness against attack and inaudibility are important requirements from an efficient steganography method. There is a tradeoff between the two requirements, however – usually, methods that have good inaudibility are not very robust, and vice versa. With that in mind, one can find the minimal gap between these requirements by way of testing the steganography algorithm with signal processing attacks. Every application has its specific requirements, and can choose high robustness compensating with the quality of the stego-signal, or vice versa. It is also good to understand such attacks from a malicious viewpoint, as a possible way to disrupt or even destroy information hidden with steganography.

Some of the common types of processes an audio signal can undergo while transmitted through an attacker medium are presented here.

- Dynamics: changing properties of the signal, by way of amplitude modification or attenuation. Limiting, expansion and compression are modifications of non-linear type that are more powerful, and thus prove higher robustness. For MP3 specifically, modifications such as re-quantization should prove an interesting challenge.

- Filtering: filtering is a common practice, which is used to amplify or attenuate some part of the signal. Basic low pass and high pass filters can be used to achieve these types of attacks.

- Ambience: in some situations, a signal can be delayed by ambience sound, or an original signal can be recorded from the source with ambience sound in order to bypass copyright protection. Those situations could be simulated in real life to test the stego-signal's robustness against such attacks.

- Conversion and lossy compression: audio generation is done at a particular sampling frequency and bit rate; However, an attack on the signal could possibly take the signal through different compression or conversion techniques that would change those parameters. Examples of attacks include converting WAV audio to MP3 or AAC (Advanced Audio Codec) and vice versa, or reducing sampling rates from 128KBps to 64 or 48KBps.

- Noise: It is important to make sure the stego-signal is robust against added noise in order to simulate channel noise and other factors that may affect the signal. Commonly, the signal is tested with *additive white Gaussian noise*.
- Time stretch and pitch shift: these attacks change either the length of the signal without changing its pitch, or vice versa. These are known as de-synchronization attacks, and are quite common in data transmission. Another such attack is applying jittering on the signal.

12 Steganography Methods In MPEG-II Layer III

Here, we summarize many of the common steganography methods used to hide data in audio signals, and specifically in MP3 files. These are ordered from the most simple and naive to the most robust, state-of-the-art methods.

12.1 Using Metadata

The first and simplest method for hiding information in MP3 files (and in general file formats as well) is using metadata to contain the message. In the case of MP3, this means using ID3v2 frames (the comment frame, for example, might be ideal for this), or, additionally, the ancillary data in the audio frames.

This method is plain and simple, as the sender can add the secret message as-is into the metadata, and the receiver can extract it from there. This will not make any change to the audio itself. However, this method is as easy to defend from as it is to use, as a secure MP3 decoder can simply remove all metadata from the file, rendering this practice useless.

12.2 Using Redundant Bits

With this method, a sender might translate their message into a binary format, then hide it by replacing “redundant” bits in the file with that message – meaning, bits that are not normally used by the format. In the example of MP3, this might include the private bits in the side info, padding bits, or bits that are rarely used, such as Emphasis. The receiver will then read those specific bits and build the message again.

As with the metadata method, a secure decoder might choose to “ignore” redundant bits by zeroing out such unsafe bits, thus overriding any possible message hidden in them. Additionally, this method might cause problems in the decoding process of some decoders, depending on their implementation.

12.2.1 In-Frame Hiding

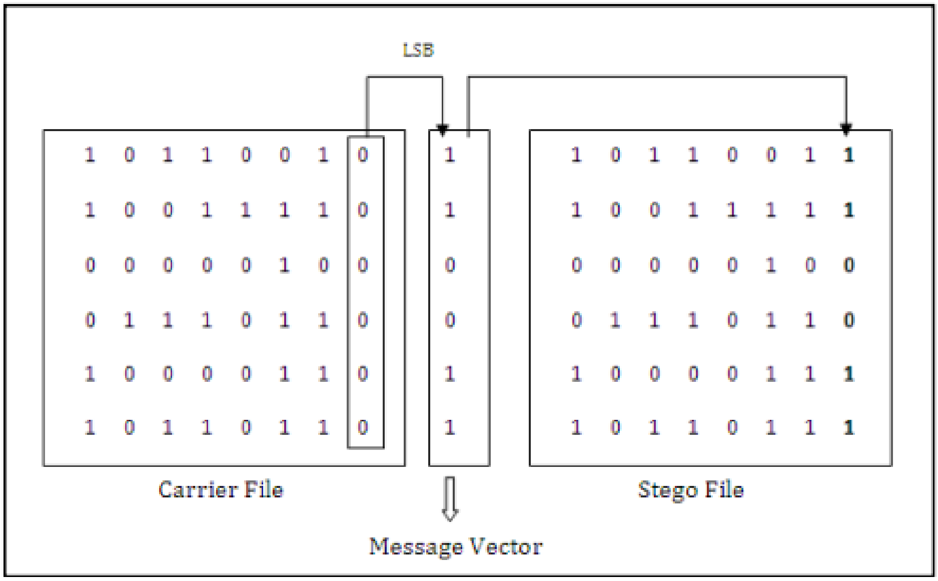
It is important to mention that message hiding can be done by creating actual MP3 frames in the file, after the compression process is done – each frame is only a couple of milliseconds of sounds, thus a small amount of frames containing “random” bits (actually our message bits) will probably not be noticed by the human ear.

We will create an amount of frames relevant to hide our message, each with a “default” frame header, so as to not break any decoder that may try to parse them. There are several approaches to this – the message frames can be appended before all sound frames or between them, and can either replace some of the original frames, or be added on top of them.

This can be a very powerful technique for small messages – it will be hard to distinguish message frames in a normal file, and there is no easy way for a decoder to clean such frames from a file without possibly damaging it. The downside is clear, however; The longer the hidden data is, the more probable this method will inevitably create hearable noise in the audio.

12.3 Least Significant Bit (LSB) Coding

LSB coding is a simple, fast, and popular steganography method that can be used for many file types. The least significant bit of a binary string is the bit to the right of it, that denotes if the number represented is even or odd. In MP3, the idea is to change the LSB of each sample in the encoding process, to instead correspond to a single bit of the secret message. For example, say we want to hide the message ‘HEY’ in our file, we will write it in binary, then take each of the samples LSB and replace it with the corresponding bit from the message (before the Huffman encoding, of course). The receiver will then extract those bits from the decoded file and build up the message.



1 0 1 1 0 0 1 0	1	1 0 1 1 0 0 1 1
1 0 0 1 1 1 1 0	1	1 0 0 1 1 1 1 1
0 0 0 0 0 1 0 0	0	0 0 0 0 0 1 0 0
0 1 1 1 0 1 1 0	0	0 1 1 1 0 1 1 0
1 0 0 0 0 1 1 0	1	1 0 0 0 0 1 1 1
1 0 1 1 0 1 1 0	1	1 0 1 1 0 1 1 1

Figure 12.1: example of LSB coding

The idea here is that by changing only the LSB, we only ever add or remove 1 from each sample. This is only a small change to the signal, and thus it has very little impact on the audio. However, it is again a method that is easy to counteract, as a secure system can zero out all LSBs, again not having much impact on the sound itself but overriding any potential hidden message. It is also susceptible to data loss due to channel noise and re-sampling.

12.3.1 Genetic Algorithm Approach

One possible way of increasing the robustness of LSB coding is not necessarily using the least significant bits. Instead, bits in "deeper" layers can be found, in a way that changing them will cause minimal noise alteration. A possible method for finding such bits is using a genetic algorithm (GA) that will learn which bits in different sound types may affect or not affect that sound. As an example, it is possible that some bits in an audio signal that represents, say, drumming, have little effect on the actual sound itself, and thus can be flipped safely to represent the stego message's bits.

A genetic algorithm that is able to learn the locations of such bits is only theoretical at this moment, and no full experiments were created to test this idea yet. There are other problems to consider with this approach – how can the receiver get the location of the chosen bits from the sender, as an example. Additionally, genetic algorithms are an entire subject of their own that requires much background to understand, and thus we will not expand on this method in this document. However, the genetic approach may be an interesting challenge for the future, and can definitely increase the robustness of redundant-bit style steganography.

12.4 Parity Coding

This method takes inspiration from parity bits, a simple form of error detecting code. The sender will:

1. Group the samples (before huffman encoding) into regions of equal size (that is decided as part of the stego-key).

2. Calculate the parity bit for each region. The parity bit is 0 if the number of total set bits (bit = 1) in the region is odd, and 1 if that number is even.

3. Compare the parity bit to the corresponding bit in the message. If they are equal, continue to the next region. Otherwise, flip the LSB of that region – thus, increasing or decreasing the number of set bits by one, changing it from odd to even or vice versa.

The receiver needs to simply calculate the parity bit for each region. Because of step 3, we can be assured that each parity bit will be corresponding to a bit in the secret message, and thus the receiver can easily build the message back.

Parity coding is a simple and robust steganographic method, and it is less likely to be detected than other methods. However, as it also uses LSB bits, it is easily defended against just like the actual LSB method.

12.5 Phase Coding

A phase in sound refers to the momentary state of a signal wave. The human auditory system cannot easily recognize a phase shift in an audio signal. The phase coding method exploits this fact by encoding the secret message bits as phase shifts in the phase spectrum of a digital signal, achieving an inaudible encoding in terms of signal-to-noise ratio. A Discrete Fourier Transformation (DFT) to transform the signal's phase.

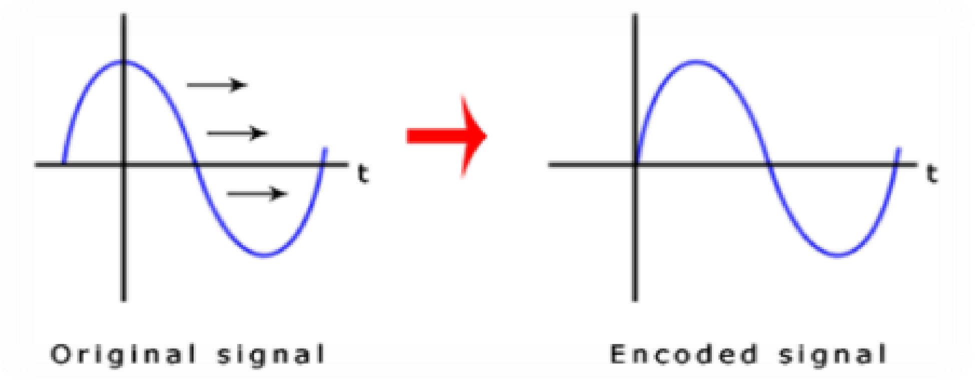


Figure 12.2: Phase coding

The following steps take place in order to apply phase encoding onto a signal:

1. The original signal is broken up into smaller segments whose lengths are equal $(S = s_1 s_2 s_3 \dots s_n, |s_i| = \ell)$.

2. Compute magnitude A_i and phase i for each segment using FFT, where $A_i = |\text{FFT}(S_i)|$ and $\phi_i = \text{phase-angle}(\text{FFT}(S_i))$.

3. Compute phase differences $\Delta\phi_i = \phi_i - \phi_{i-1}$ for $i \in \{1, \dots, n\}$.

4. To encode the binary data d of length m $(m < \ell)$ assign:

$$\phi_{\text{data}}[i] = \begin{cases} \frac{\pi}{2} & d_i = 0 \\ -\frac{\pi}{2} & d_i = 1 \end{cases}$$

5. Replace following elements of first phase sequence, ϕ_1, with $\phi_{\text{data}}[i]$ for $i \in \{1, \dots, m\}$ as: $\phi'_\ell[{}^{L}/_{2} - m + i] = \phi_{\text{data}}[i]$.

6. To maintain the odd symmetry which is a property of DFT, repeat the process as: $\phi'_1[{}^{L}/_{2} + 1 + i] = \phi_{\text{data}}[m + 1 - i]$.

7. To maintain phase differences, sequentially reassign $\phi'_i = \phi_{i-1} + \Delta\phi_i$ for each $i \in [2, \dots, n]$.

8. Reconstruct the signal using inverse FFT applied to each segment $A_i\text{exp}(j\phi'_i)$ (where j is an imaginary unit) and joining all segments together.

A receiver can then use inverse DFT to extract the phase from the signal and reconstruct the message, using the assignment rules in step 4.

One disadvantage associated with phase coding is a low data transmission rate due to the fact that the secret message is encoded in the first signal segment only. This might be addressed by increasing the length of the signal segment. However, this would change phase relations between each frequency component of the segment more drastically, making the encoding easier to detect. As a result, this method is useful when only a small amount of data needs to be concealed.

12.6 Spread Spectrum

In a normal communication channel, it is often desirable to concentrate the information in as narrow a region of the frequency spectrum as possible in order to conserve available bandwidth and to reduce power. The basic spread spectrum technique, on the other hand, is designed to encode a stream of information by spreading the encoded data across as much of the frequency spectrum as possible. This allows the signal's reception, even if there is interference on some frequencies.

While there are many variations of spread spectrum communication, the main technique used for steganographic encoding is *Direct Sequence Spread Spectrum* (DSSS). The DSSS method spreads the signal by multiplying it by a *chip* – a maximal length pseudo-random sequence modulated at a known rate (usually the sampling rate of the file). The idea is to spread band-limited "white noise" across the signal, like an audio recording of static. However, this noise-like signal is used to exactly reconstruct the original data at the receiving end. This process, known as despreading, is mathematically a correlation of the transmitted spreading sequence with the spreading sequence that the receiver already knows the sender is using.

In DSSS, a key is needed to encode the information and the same key is needed to decode it. The key is pseudorandom noise that ideally has flat frequency response over the frequency range, i.e., white noise. The key is applied to the coded information to modulate the sequence into a spread spectrum sequence.

The DSSS method is as follows: The code is multiplied by the carrier wave and the pseudo-random noise sequence, which has a wide frequency spectrum. As a consequence, the spectrum of the data is spread over the available band. Then, the spread data sequence is attenuated and added to the original file as additive random noise. DSSS employs bi-phase shift keying since the phase of the signal alternates each time the modulated code alternates (see figure 12.4). For decoding, phase values $\varphi 0$ and $\varphi 0 + \pi$ are interpreted as a 0 or a "1," which is a coded binary string.

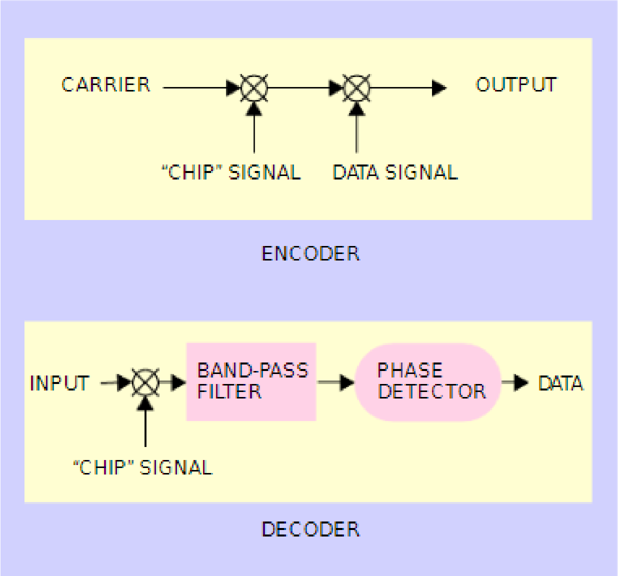


Figure 12.3: Spread Spectrum Encoding

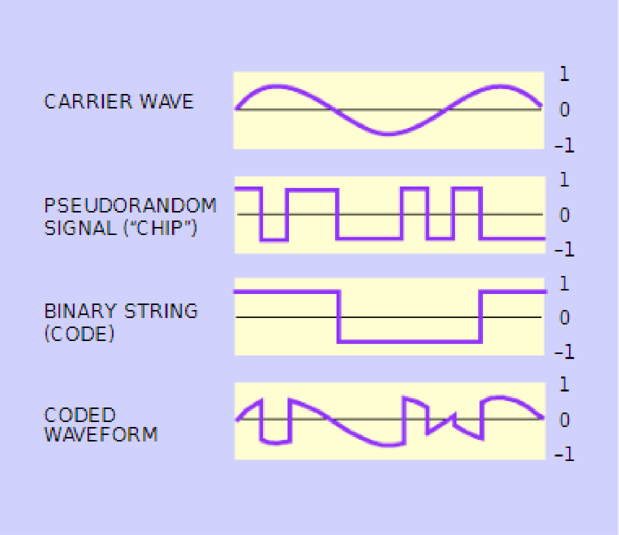


Figure 12.4: Synthesized spread spectrum information encoded by the direct sequence method

Though additional noise created by a spread spectrum is, as implied, spread across the carrier signal, this white noise might be noticeable by human ears and thus compromise the steganography.

12.7 Echo Hiding

Echo data hiding embeds data into a host audio signal by introducing an echo. The data is hidden by varying three parameters of the echo: initial amplitude,decay rate, and offset.

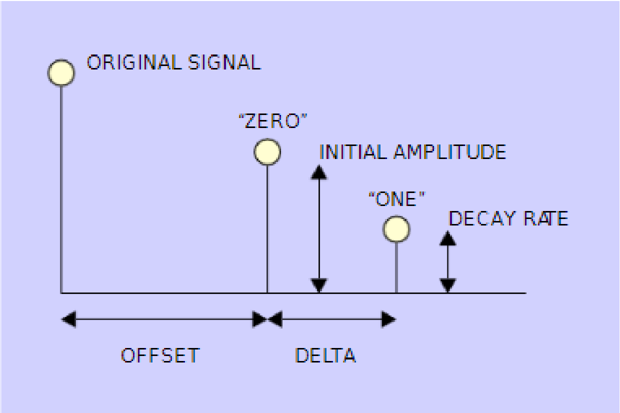


Figure 12.5: Adjustable parameters

As the offset (or delay) between the original and the echo decreases, the two signals blend. At a certain point, the human ear cannot distinguish between the two signals. The echo is perceived as added resonance. (This point is hard to determine exactly. It depends on the quality of the original recording, the type of sound being echoed, and the listener. In general, it seems that this fusion occurs around 1/1000 of a second for most sounds and most listeners.)

The encoder uses two delay times, one to represent a binary one (offset) and another to represent a zero (offset + delta). Both delay times are below the threshold at which the human ear can resolve the echo. In addition to decreasing the delay time, we can also ensure that the information is not perceivable by setting the initial amplitude and the decay rate below the audible threshold of the human ear.

The encoding process can be represented as a system that has one of two possible system functions. In the time domain, the system functions are discrete time exponentials differing only in the delay between impulses.

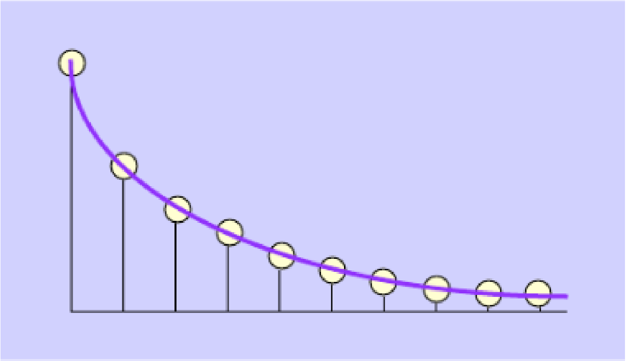


Figure 12.6: Discrete time exponential

For simplicity, we chose an example with only two impulses (one to copy the original signal and one to create an echo). Increasing the number of impulses is what increases the number of echoes. We let the kernel shown in 12.7 (A) represent the system function for encoding a binary one and we use the system function defined in 12.7 (B) to encode azero. Processing a signal through either kernel will result in an encoded signal, as shown in figure 12.8.

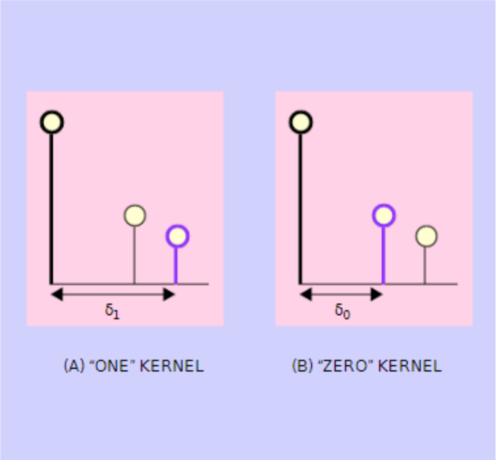


Figure 12.7: Zero and One Kernels

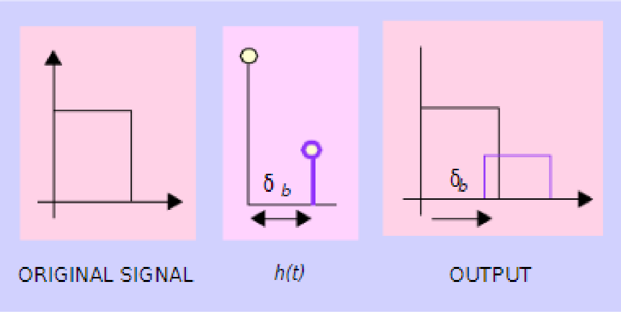


Figure 12.8: Echoing example

In figure 12.9, the example signal has been divided into seven equal portions.

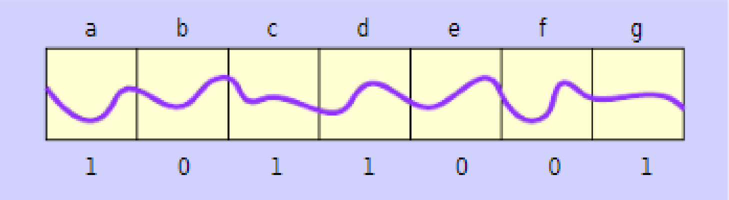


Figure 12.9: signal division

We want portions a, c, d, and g to contain a one. Therefore, we use the "one" kernel (12.7 A) as the system function for each of these portions. Each portion is individually convolved with the system function. The zeros encoded into sections b, e, and f are encoded in a similar manner using the "zero" kernel (12.7 B). Once each section has been individually convolved with the appropriate system function, the results are recombined. To achieve a less noticeable mix, we create a "one" echo signal by echoing the original signal using the "one" kernel. The "zero" kernel is used to create the "zero" echo signal. The resulting signals are shown in figure 12.10.

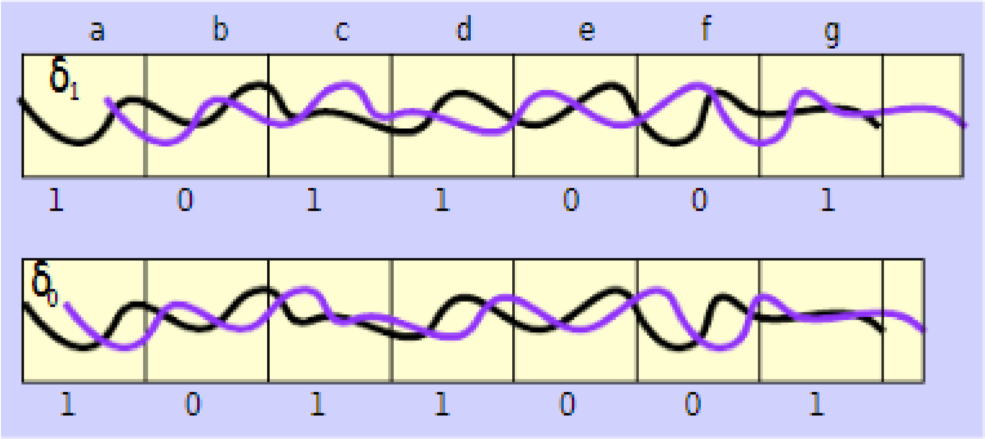


Figure 12.10: echoed signals (echo in purple)

The "one" echo signal and the "zero" echo signal contain only ones and zeros, respectively. In order to combine the two signals, two mixer signals are created. The mixer signals are either one or zero depending on the bit we would like to hide in that portion of the original signal.

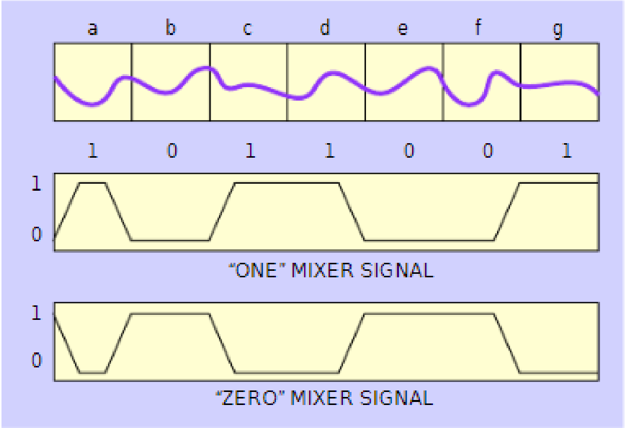


Figure 12.11: mixer signals

The "one" mixer signal is multiplied by the "one" echo signal while the "zero" mixer signal is multiplied by the "zero" echo signal. In other words, the echo signals are scaled by either 1 or 0 throughout the signal depending on what bit any particular portion is supposed to contain. Then the two results are added. Note that the "zero" mixer signal is the complement of the "one" mixer signal and that the transitions within each signal are ramps. The sum of the two mixer signals is always one. This gives us a smooth transition between portions encoded with different bits and prevents abrupt changes in the resonance ofthe final (mixed) signal.

All in all, we can view the encoding process as:

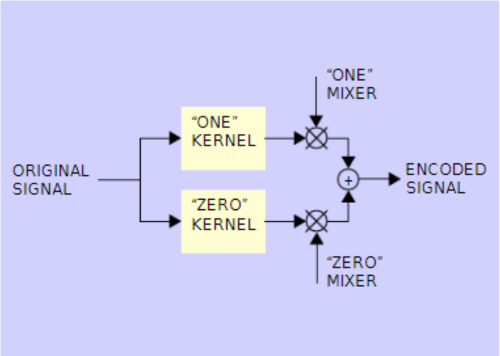


Figure 12.12: Encoding process

During the decoding process, Information is embedded into a signal by echoing the original signal with one of two delay kernels. A binary one is represented by an echo kernel with a ($\delta 1$) second delay. A binary zero is represented by a ($\delta 0$) second delay. Extraction of the embedded information involves the detection of spacing between the echoes. In order to do this, we examine the magnitude (at two locations) of the autocorrelation of the encoded signal's cepstrum: $F^{-1}\left(\ln_{\text{complex}}\left(F\left(x\right)\right)^2\right)$.

The following procedure is an example of the decoding process. We begin with a sample signal that is a series of impulses such that the impulses are separated by a set interval and have exponentially decaying amplitudes. The signal is zero elsewhere.

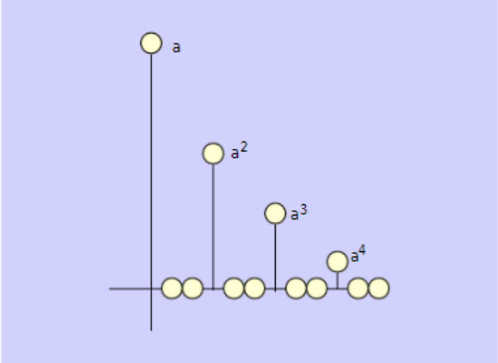


Figure 12.13: Example signal $x\left[n\right] = a^n u\left[n\right]; 0 < a < 1$

The next step is to find the cepstrum of the echoed version. The result of taking the cepstrum makes the spacing between the echo and the original signal a little clearer.

Unfortunately, the result of the cepstrum also duplicates the echo every (δ) seconds. In the figure below, this is illustrated by the impulse train in the output. Furthermore, the magnitude of the impulses representing the echoes are small relative to the original signal. As Such, they are difficult to detect. The solution to this problem is to take the autocorrelation of the cepstrum.

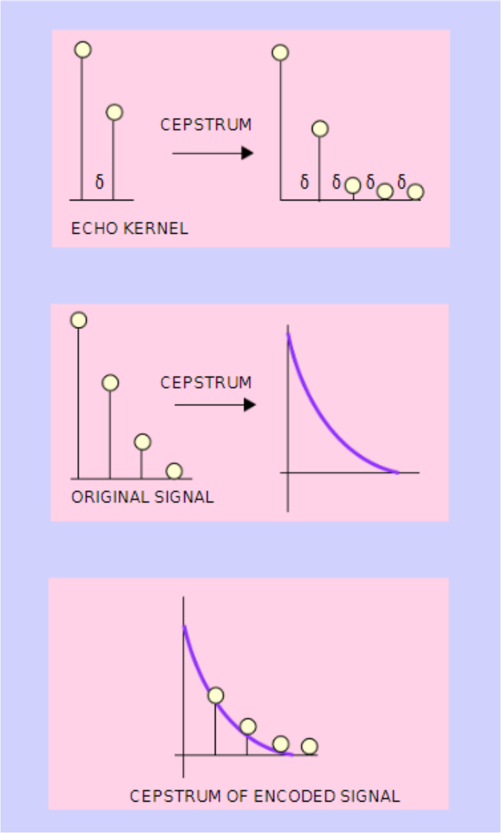


Figure 12.14: Cepstrum of the echo-encoded signal

We echo the signal once with delay (δ) using the kernel depicted in the figure 12.15. The result is illustrated in figure 12.16.

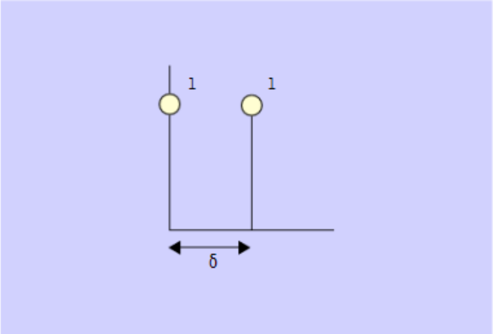


Figure 12.15: Echo kernel example

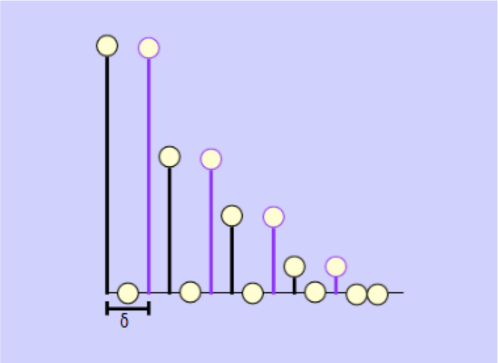


Figure 12.16: Echoed version of example signal

Only the first impulse is significantly amplified as it is reinforced by subsequent impulses. Therefore, we get a spike in the position of the first impulse. Like the first impulse, the spike is either ($\delta 1$) or ($\delta 0$) seconds after the original signal. The remainder of the impulses approach zero. Conveniently, random noise suffers the same fate as all the impulses after the first.

The rule for deciding on a one or a zero is based on the time delay between the original signal and the delay (δ) before the spike in the autocorrelation. Recall that a one was encoded by placing an echo ($\delta 1$) seconds after the original and a zero was placed ($\delta 0$) seconds after the original. When decoding, we assign a one if the magnitude of the autocorrelation function is greater at ($\delta 1$) seconds than it is at ($\delta 0$) seconds. A zero is assigned if the reverse is true. This is the same as deciding which kernel we used utilizing the fact that the "one" and "zero" kernel differ only in the delay before the echo.

Using the methods described, it is indeed possible to encode and decode information in the form of binary digits into a media stream with minimal alteration to the original signal, meaning that the output of the encoding process is changed in such a way so that the average human can not hear any significant difference between the altered and the original signal. There is little, if any, degradation of the original signal. Instead, the addition of resonance simply gives the signal a slightly richer sound. This method becomes less effective, however, on audio signals with a large enough silence gap, as the echo will sound more obvious in those cases.

12.8 Discrete Cosine Transform (DCT)

The discrete cosine transform is a technique for converting a signal into elementary frequency components. DCT can be employed on both one-dimensional and two dimensional signals, like audio and image, respectively. It is a spectral transformation, which has the properties of DFT, however DCT uses only cosine functions of various wave numbers as basic functions, and operates on real valued signals and spectral coefficients.

DCT of a 1-dimensional signal, and the reconstruction of the original signal from the DCT coefficients (using Inverse DCT) can be computed using the following equations (where$f_{\text{dct}}(x)$ is the original signal, and $c_{\text{dct}}(u)$ are the DCT coefficients):

$$C_{\text{dct}}(u) = \alpha(u)\sum_{x=1}^{N_{1t}} f_{\text{dct}}(x)\cos\left[\frac{\pi(2x+1)u}{2N_{1t}}\right], \text{for } u = 0,1,2,\ldots,N_{1t}-1$$

$$f_{\text{dct}}(x) = \sum_{x=1}^{N_{1t}} \alpha(u)\,C_{\text{dct}}(u)\cos\left[\frac{\pi(2x+1)u}{2N_{1t}}\right], \text{for } x = 0,1,2,\ldots,N_{1t}-1$$

$$\text{where}\alpha(u) = \begin{cases}\sqrt{\frac{1}{N_{1t}}} & \text{for } u = 0\\ \sqrt{\frac{2}{N_{1t}}} & \text{for } u \neq 0\end{cases}$$

Thus, a sender can apply DCT on regions of samples, and hide a message in the coefficients similarly to phase coding (i.e. adding or reducing $k/2$ if the hidden bit is one or zero respectively, with k being a small number chosen as part of the stego-key). The receiver can easily calculate back the original signal and extract the message.

DCT packs the energy of the signal into the low frequency regions, which provides an option of reducing the size of the signal without degrading the quality of the signal. It is, additionally, very fast computationally. The main disadvantage of this method, however, is that DCT may cause block artefacts when heavy compression is used. Thus, in the case of MP3, hearable noise may be caused as a result of applying DCT before compression (commonly, DCT causes “mosquito noise” in audio signals).

12.9 Discrete Wavelet Transform (DWT)

Another type of signal transformation, a Discrete Wavelet Transform is any wavelet transform for which the wavelets are discretely sampled. As with other wavelet transforms, a key advantage it has over Fourier transforms is temporal resolution: it captures both frequency and location information (location in time).

To use this transformation in steganography, we first apply DWT to each region of samples. The DWT used is a simple Haar wavelet – For an input represented by a list of 2^n numbers, the Haar wavelet transform may be considered to pair up input values, storing the difference and passing the sum. This process is repeated recursively, pairing up the sums to prove the next scale, which leads to 2^n-1 differences and a final sum. The message is then hidden, in a similar way to DCT seen above (importantly, the k value added to coefficients here should be one with 4 fractional digits, such as 0.0050. This is because the coefficients themselves have 4 fractional digits). Finally, inverse DWT will be applied. During extraction, DWT is applied again to calculate coefficients and retrieve the message.

This method creates less noise than DCT, and is a robust choice overall. Its main drawback is low data capacity, as larger regions are needed for DWT computation, which results in less bits hidden overall.

12.10 Part2_3_length Parity

This approach, created as a tool known as "MP3Stego", is a very robust steganographic method that was considered state-of-the-art for many years. The hiding process here takes place at the heart of MP3's encoding process, namely in the `inner_loop`. As discussed in 5.5, the inner loop quantizes the input data and increases the quantizer step size until the quantized data can be coded with the available number of bits. Another loop (`outer_loop`) checks that the distortions introduced by the quantization do not exceed the threshold defined by the psychoacoustic model. The `part2_3_length` variable contains the number of `main_data` bits used for scalefactors and Huffman code data in the MP3 bit stream. The idea is to encode the message bits as its parity by changing the end loop condition of the inner loop – continue running the `inner_loop` as long as the parity bit of `part2_3_length` is not equal to the corresponding message bit.

This works well as the noise created by the quantization will only slightly increase by crossing the threshold set by the psychoacoustic model, as we only continue to search for a quantizer step until one is found that creates a `part2_3_length` with the desired parity. To increase robustness even further, "MP3Stego" only hides the message in randomly chosen `part2_3_length` values; the selection is done using a pseudo-random bit generator based on SHA-1.

12.11 Huffman Index Approach

A modern steganographic method, this technique is based on transforming Huffman table indexes. It is probably one of the most robust, modern steganography techniques today that is also reversible and has good capacity potential.

Like 3.9, the hiding process will take place in the iteration loop part of the encoding process in MP3. Specifically, we will look at the table_select variable, that holds the Huffman table indexes for each channel, granule, and region in the frame. As we can see from table 12.1, some tables have the same maximum value, so they can encode the same range of values.

As an example, we consider the quantization values from a subregion: (1, 0, 0, 1, 0, 2). According to the MP3 standard encoding rules, a table with a maximum value no less than 2 can encode it. However, in order to minimize the encoded bit stream, the MP3 standard rules to encode with table 2 or 3, which are closest to the subregion's maximum value. At the same time, the standard rules to select a table which requires the smaller number of bits. In this example, we can see that the standard rules will select table 3 for encoding.

Table index	Max value	Linbits	Table index	Max value	Linbits
0	0	without	16	16	1
1	1	without	17	18	2
2	2	without	18	22	3
3	2	without	19	30	4
4	not used	without	20	78	6
5	3	without	21	270	8
6	3	without	22	1038	10
7	5	without	23	8206	13
8	5	without	24	30	4
9	5	without	25	46	5
10	7	without	26	79	6
11	7	without	27	142	7
12	7	without	28	270	8
13	15	without	29	526	9
14	not used	without	30	2062	11
15	15	without	31	8206	12

Table 12.1: Properties of Huffman Tables in the MP3 Standard

Quantized values	Codeword in Table 2	Codeword in Table 3
$(1,0)$	011	001
$(0,1)$	010	10
$(0,2)$	000001	00001
Total bits	12 bits	11 bits

The idea is to do data embedding by way of changing the selection of tables. The method works by the following embedding rules:

Table index	Table index (transformed)		Table index	Table index (transformed)	
	bit $= 1$	bit $= 0$		bit $= 1$	bit $= 0$
0	-	-	16	16	17
1	1	3	17	18	17
2	2	3	18	18	19
3	2	3	19	20	19
4	-	-	20	20	21
5	5	6	21	22	21
6	5	6	22	22	23
7	7	8	23	31	23
8	7	8	24	25	24
9	9	8	25	25	26
10	10	11	26	27	26
11	10	11	27	27	28
12	10	12	28	29	28
13	13	15	29	29	30
14	-	-	30	31	30
15	13	15	31	31	23

Table 12.2: Rules for Huffman Table Transform

Let's assume that, for the current region, the MP3 encoder chose table 10 as this region's Huffman table. If the current bit of the hidden message is 1, then we can continue without changing any information. However, if the bit is 0, we instead change the index of the chosen table to 11. Similarly, if the chosen table was originally 12, we will keep it as 12 if the bit is 0, but change it to 10 otherwise. During the extraction process, the receiver can directly judge if the hidden bit is 0 or 1 based on the chosen table.

This works so well as changing the Huffman encoding of the audio signal does not make any changes to the sound itself, only slightly increasing the amount of bits required to store the signal (and even not in a significant, noticeable way). The transformation is built this way for three major reasons: 1) table 0 has no Huffman codeword, and tables 4 and 14 are not in use, so these tables are not changed. 2) In order to minimize the change, the table that is selected in the swap is the same level as the original table. 3) In order to achieve blind extraction, each table only corresponds to one other table at the time of extraction. By such construction, if the table of the index is in $H_0 = \{3, 6, 8, 11, 12, 15, 17, 19, 21, 23, 24, 26, 28, 30\}$, it means the hidden bit is 0; if it is in $H_1 = \{1, 2, 5, 7, 9, 10, 13, 16, 18, 20, 22, 25, 27, 29, 31\}$, it means the hidden bit is 1.

This method is very robust, has good undetectability and yields good capacity in tests.

12.12 Other Methods

As research continues and adapts, new steganographic techniques (and variations of existing techniques) will inevitably surface. As an example, earlier variations of 3.10 are quoted in the research paper written about it, and though all have lesser traits than the final version, it is always possible that someone will come up with an upgraded idea.

Other methods might prove niche – as an example, one researcher proposed a method based on MP3 linbits of Huffman codewords by analyzing the structure of MP3 linbits. In fact, most codewords don't have linbits; At the same time, changing linbits has a great influence on the perceived quality of audio and it's irreversible. Thus, for most MP3 files, this method will be practically useless – however, in some edge cases it might work well enough.

The point that one needs to take from this is that MP3 has a lot of "weak spots" with potential to use (or abuse) with steganographic methods. A secure channel must be ready for as many possibilities as it can – after all, the idea behind steganography is to be a surprise at plain sight.

13 Conclusions

Table 13.1 compares the advantages and disadvantages of the different steganographic methods discussed above:

Method	**Advantages**	**Disadvantages**
Metadata Hiding (12.1)	Simple	Easy to counteract
Redundant Bits Hiding (12.2)	Simple, good capacity	Easy to counteract, may cause errors in decoding process
In-Frame Hiding (12.2.1)	Simple, hard to counteract, robust for very small data	Will create noise for larger data
LSB Coding (12.3)	Easy to implement, little effect on sound	Easy to counteract, susceptible to data loss
Genetic Algorithm Approach (12.3.1)	Better robustness than LSB	Hard to implement
Parity Coding (12.4)	Simple, robust, unlikely to be detected	As LSB Coding
Phase Coding (12.5)	Robust, unlikely to be detected	Low capacity - only useful for small data transfers
Spread Spectrum (12.6)	Message is spread across carrier signal - harder to detect	May create hearable white noise
Echo Hiding (12.7)	Little degradation of the original signal	May be noticeable in sounds with large silence gaps
DCT (12.8)	Computationally fast, little degradation of the original signal	May cause noise after compression
DWT (12.9)	Computationally fast, very robust	Low capacity
Part2_3_length Parity (MP3Stego) (12.10)	Very robust	Lower capacity than other methods
Huffman Index Approach (12.11)	Very robust, good capacity, hard to detect	-

Table 13.1: Comparison of MP3 steganography methods

As discussed, there are many methods and techniques for steganographic uses in MP3 files. This gives one a good amount of choices to decide between when in need of applying steganography to audio files, and the best method can be chosen in respect to factors such as simplicity, robustness or capacity. To protect from such methods, however, one should be ready for all of the possibilities – and continue to update their security, as new methods are researched or conceptualized all the time.

Part III
MP3 Steganography System Implementation

14 Overview

As part of this research project, we have implemented a steganographic system for MP3 files. The system, created in the Python language, is built of two components: a MP3 encoder and a MP3 decoder, both customized to support embedding and extraction of secret text messages. The steganographic method we have decided to showcase in this system is the Huffman Index Approach (see 12.11 on page 62). The source code for our system, as well as additional information about installation and usage, can be found in the attached Github of our project.

In this document, we will overview our system from a “bird’s eye” view, looking mainly at the different components and the flow of the code. To better understand minor nuances, such as mathematical equations or specific code conditions, we recommend diving into the actual code and reading the relevant code blocks.

Important to mention, our system was built with simplicity and readability in mind, and thus performance times are somewhat neglected. This is because our system is built as a basis, an explanatory demonstration for other ro create their own specific applications of MP3 steganography. We thus recommend using this system as a guide for your own implementation, instead of a practical tool for actual applications.

14.1 Dependencies

The project was created in `Python 3.9`. Besides a `Python 3.9` interpreter, the following Python modules are required:

- NumPy (for mathematical and array calculations)
- SciPy (for file handling)
- TQDM[1]
- bitarray
- numba[1] (this library compiles some code in real time for better performance)

[1] These modules help with output readability and run time performances, but are not required for the actual system logic.

14.2 API

Our system is contained in a Python library called `mp3stego`, which can be found on `pip`. To start, either download the origin code or import the library from `pip`, and create a `Steganography` object:

```
from mp3stego import Steganography

stego = Steganography(quiet=True)
```

The `quiet` variable is a boolean value that determines whether or not information about the encoding and decoding processes should be printed to the console. It is set to True by default.

Next, we will go over the different actions available with the system. To encode audio into MP3, you may use the `encode_wav_to_mp3` function. The input audio must be in a WAV file format.

```
stego = Steganography(quiet=True)
stego.encode_wav_to_mp3("input.wav", "output.mp3", 320)
```

The first argument is a path to the input WAV file, the second is a path to the output MP3 file. The third is the `bitrate` that should be used in the encoding process, with the default value being 320 (meaning 320 KB). You may use bitrates from 32 KB to 420 KB with jumps of 32 KB.

For decoding a MP3 file, you may use the `decode_mp3_to_wav` function. This will parse the chosen MP3 file and create a WAV file from the decoded PCM samples.

```
stego = Steganography(quiet=True)
stego.decode_mp3_to_wav("input.mp3", "output.wav")
```

The arguments will be the path to the MP3 file to decode, and the WAV output file, correspondingly.

To hide a secret string in a MP3 file, use the `hide_message` function:

```
stego = Steganography(quiet=True)
stego.hide_message("input.mp3", "output.wav", "String to hide in the file")
```

The first argument is the path to the input file to hide the message in. The second is the path to the outputted MP3 stego-file containing the message. The third is the message to hide. There are two optional arguments: the fourth argument is the path for the WAV file outputted during the decoding process (defaultly, it will be the same as the input MP3 file, but with the ".wav" ending), and the fifth argument is a boolean that, if set, will delete that temporary WAV file (defaultly set to True).

To reveal a secret string from a MP3 stego-file, use the `reveal_message` function:

```
stego = Steganography(quiet=True)
stego.reveal_message("input.mp3", "results.txt")
```

With the first argument being a path to the stego-file, and the second being a path to a text file to which the function will output the hidden message (this has to be a .txt file).

Finally, you can clean a MP3 file from any potential secret message that was hidden with our encoder using the `clear_file` function:

```
stego = Steganography(quiet=True)
stego.clear_file("input.mp3", "output.mp3")
```

With the first argument being the path to the input MP3 file, and the second being the path to the outputted clear MP3 file.

14.3 Website

For your own convenience, we have implemented a simple website design and corresponding server-side code, that utilize our API to allow users to perform steganography actions easily. For more information, see the corresponding GitHub page.

14.4 Implementing Your Own System

As discussed, after trying out our system, we recommend using it as the basis for your own implementation of steganographic systems in MP3. The following should be kept in mind when creating a system of your own design:

1. It is highly recommended to acquire a copy of the *International Standard (IS)* of the MP3 standard. It contains general guidelines, as well as specific values and rules to follow for the encoding and decoding processes.

2. Use this document as a companion to the code. When reading a block of code, refer to the corresponding section in part I of this document, as well as the documentation within the code files themselves, to better understand the theory and ideas behind the code lines.

3. Performance and security: as discussed, the system is built around the ideas of readability and simplicity. In actual operational systems, though, performance and security should also be valued. Ideally, a well-made steganography system should be built in a compiled language (such as `C` or `C++`) for best performance, and be built with security guidelines in mind.

4. Though we have implemented only one steganography method in our system as a showcase, more could be implemented in yours to allow greater variation in user choice or to showcase more types of threats, as you see fit. Refer to part II to learn more.

5. Finally, take the time to understand the ideas written in this document to the best of your ability. If you have read thus far, you probably understand by now that both the MP3 format and the methods for hiding information in MP3 files are very complex and require a good grasp of the algorithms used, the physics of sound, and

a general well understanding of computer science techniques. It is no easy feat to create a MP3 steganographic system from scratch. We hope this document and our code will help you make this process as simple as possible, but do not be afraid to use additional resources to research and more deeply understand any part in the process of creating and building your system.

15 Code Flow

15.1 System Flow

The `MP3Stego` system is built of a costum MP3 encoder and decoder. In order to apply steganography, the following steps take place during the embedding process:

1. A secret message is chosen by the user, as well as a MP3 file to serve as the carrier file.

2. The MP3 file is decoded, using our decoder, and a WAV file is outputted.

3. The WAV file is re-compressed into MP3 using our encoder. During the encoding process, the secret message is embedded into the file using the Huffman Index Approach. For convenience reasons, first the message's length is embedded, followed by a '`#`' character, and finally the message itself.

4. The newly created MP3 stego-file containing the hidden message is outputted.

To extract a secret message, a MP3 stego-file is inputted into our decoder, and during the decoding process, the message can be rebuilt (as per the steganography method described in part II). The message's length can be found until a '`#`' character is seen (in binary of course), at which point one can easily calculate the message itself.

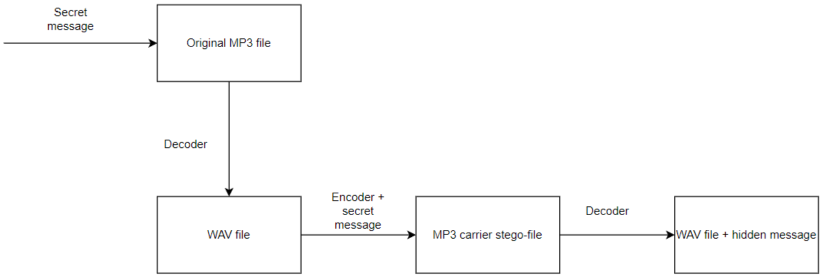


Figure 15.1: System Flow

15.2 Encoding Process

The following steps take place during the encoding process:

1. A MP3Encoder object is created. Bitrate, sample rate and MPEG version are found and saved in respective variables. All PCM samples from the inputted WAV file are divided into frames (1152 samples) and polyphase filterbank is applied using the init and sub_band_initialize functions. See part I subsection 5.1.

2. For each frame:

   (a) For each granule, and for each channel:

      i. MDCT is applied on the samples. See part I subsection 5.2.

      ii. FFT is applied on the samples, and feeds into a psychoacoustic model. See part I subsections 5.3 – 5.4.

      iii. The iteration_loop is applied to quantize the samples and generate relevant information. See part I subsection 5.5.

         A. The calc_scfsi function selects scalefactors for the samples.

         B. The max_resevoir_bits function is called at the beginning of each granule to get the max bit allowance for the current granule based on reservoir size and perceptual entropy.

         C. The outer_loop function is called to control quantization noise (see part I subsection 5.5, "Distortion control loop"). It also iteratively calls the bin_search_step_size and inner_loop functions that find the best quantizer step size for the samples in each region (see part I subsection 5.5, "Rate control loop"). The following functions are called inside these two functions:

            - quantize applies quantization on the samples.
            - calc_run_len calculates the size of the rzero, count1 and big_values region. See part I subsection 7.1.3.
            - subdivide subdivides the big_values region which will be used to separate Huffman tables. See part I subsection 7.1.3, "big_values".
            - part2_length calculates the number of bits needed to encode the scalefactors in the main data block. See part I subsection 7.1.3, "part_2_3_length".
            - big_v_tab_select selects a Huffman table index for the big_values region. new_choose_table chooses the most effective table possible based on the first choice.

      iv. Using all generated information, the format_bitstream function is called to format the frame into a MP3 frame. See part I subsection 5.8. The following functions are then applied:

         A. encode_side_info formats all relevant information for the side information of the current frame. See part I subsection 7.1.3.

         B. encode_main_data formats the main data part of the current frame. First, the scalefactors generated are formatted. Then, a Huffman table index is chosen for each region using the huffman_code_bits function. If a message is to be hidden in the MP3 file, the chosen indexes are transformed based on the current message bits and using the transformation rules of the steganographic method (see part II subsection 12.11). The regions are then encoded into Huffman codewords and formatted. See part I subsection 5.6.

   (b) The frame bits are added into a global array.

3. The frame array is written into a MP3 file.

4. The MP3 file is outputted.

15.3 Decoding Process

The following steps take place during the decoding process:

1. File metadata is parsed, as per the rules of ID3v2 format. See part I subsection 7.2.1.

2. The first frame is found, using the offset calculated from the ID3v2 (offset=0 if no metadata is present in the file).

3. For each frame:

   (a) The `init_current_header` function parses the frame's header into the relevant fields in a `FrameHeader` object. See part I subsection 7.1.1.

   (b) The `init_current_frame` function is called:

      i. Using the `set_side_info` function, the frame's side information is parsed and saved in a `FrameSideInformation` object. See part I subsection 7.1.3.

      ii. At this point, if a secret message was hidden in the file, the decoder can retrieve the relevant message bits from the current frame by checking the indexes found in the `table_select` variable. See part II subsection 12.11 for more information on the steganographic method.

      iii. The `set_main_data` function is called, to extract the main data part of the frame – both the scalefactors and the encoded Huffman codewords. The Huffman codewords are also decoded here. See part I subsection 7.1.4.

      iv. For each granule, and each channel:

         A. The `requantize` function applies requantization on the samples. See part I subsection 6.4.

         B. If the channel mode is Joint Stereo, and there is no mode extension, the `ms_stereo` function is applied. See part I subsection 6.6.

         C. For short windows, the `reorder` function is applied. See part I subsection 6.5.

         D. For other windows, the `alias_reduction` function is applied. See part I subsection 6.7.

         E. `IMDCT` is applied on all samples. See part I subsection 6.8.

- F. `frequency_inversion` is applied on all samples. See part I subsection 6.9.
- G. `synth_filterbank` is applied on all samples. See part I subsection 6.10.
- v. The `interleave` function adds the decoded PCM samples to a global PCM array.
- (c) All bits extracted from the secret message are saved into a global string.
- (d) Calculate the current offset, based on frame's size.
- (e) Go to next frame.

4. The PCM array is written into a WAV file.
5. The WAV file, as well as the secret string, are outputted.

15.4 Clearing Process

To clear a MP3 file of a potential secret message hidden using the Huffman Index Approach, we can simply re-decode and encode the file. This works since during the new encoding process, the most effective Huffman tables (the ones that will take the least amount of bits to encode the samples) will be chosen for the file samples, thus overriding any message that may have been hidden in the indexes.

15.5 Scenario Example

Here is included an example scenario where our system (or any MP3 steganography system) could be put to use:

Say our agent, Alice, has an open shell command line where she wants to run some commands or scripts. This shell is open on a remote server and is remote controlled, thus Alice must send commands remotely over a network. However, Alice does not want to send those commands openly over the network, as the channel she uses to communicate with the remote server is insecure. This can be caused by two reasons: A) Alice is an attacker in an "enemy" network. She has managed to control her own region in a server of that network (namely, the remotely controlled shell on the server), but does not want the ones controlling the network to see her commands sent to the shell, or even suspect that Alice's traffic are malicious commands. B) Alice actually controls the server in which the shell runs, but she suspects of *man-in-the-middle* attackers sniffing traffic on the channel. She does not want such a potential attacker to sniff out her commands (she could use cryptography to hide the content she sends over the channel, but she does not want the attacker to even suspect she sends those commands).

To protect her commands and hide their existence on the insecure channel, Alice turns to steganography. On her end, Alice downloads a MP3 file of an unsuspicious nature – say, a MP3 version of the hit song "Never Gonna Give You Up" by Rick Astley.

She then uses the custom MP3 steganography system to decode the file and re-encode it, embedding her secret message within – in her case, a shell script she wishes to run in the server. She then sends the MP3 stego-file through the insecure channel to the server. Any attacker sniffing the traffic over the channel will only see a MP3 file of "Never Gonna Give You Up" transfering to the server. This is probably less suspicious traffic than shell commands in plain text, or even weird encrypted messages – unless the attacker specifically knows Alice uses steganography, they will probably have no idea about the script hidden in the MP3 file.

When the file arrives on the server, Alice is in the clear. She can use the custom MP3 decoder to retrieve the script she hid inside the file, and with some code, have her shell run it. As only the channel itself is insecure, any sniffer will never even know Alice was the one who ran these scripts!

16 Conclusions

We have implemented a full steganographic system for MP3 files, including a custom encoder and decoder, and a message-hiding process based on the Huffman Index Approach described in part II of this document.

As discussed, this system should provide a great basis for you to understand the rationale behind the encoding and decoding processes, as well as steganographic methods for MP3 files. We truly hope that our code, as well as the rest of the information presented in this document, will help you, dear reader, build a steganography system of your own for your future applications

Good luck!

Bibliography

Part 1 - The Theory Behind MP3

- "MP3: The Definitive Guide", Scot Hacker, https://www.ime.usp.br/~rbc/lixo/ch02.html

- "MP3 Decoder in Theory And Practice", Praveen Sripada, http://www.diva-portal.org/smash/get/diva2:830195/FULLTEXT01.pdf

- "Principles of MP3", Harry Fairhead https://www.i-programmer.info/babbages-bag/1222-mp3.html

- "MP3 Tech", various articles and authors, http://www.mp3-tech.org/

- "The Audio File: Understanding MP3 Compression", Thomas Wilburn, https://arstechnica.com/features/2007/10/the-audiofile-understanding-mp3-compression/

- ID3 metadata, https://id3.org/id3v2.3.0

- FFT and MDCT processes, Wikipedia, https://en.wikipedia.org/wiki/Fast_Fourier_transform, https://en.wikipedia.org/wiki/Modified_discrete_cosine_transform

Part 2 - Steganography in MP3

- "A review of MP3 steganography methods", IJAERV, https://www.ripublication.com/ijaer18/ijaerv13n2_40.pdf

- "MP3Stego: Hiding text in MP3 files", Mark Noto, https://sansorg.egnyte.com/dl/xx2DvrODS0

- "Techniques for data hiding", W. Bender, D. Gruhl, N. Morimoto, A. Lu, https://www.researchgate.net/publication/220354258_Techniques_for_Data_Hiding

- "A comparative study of audio steganography techniques", Palwinder Singh, https://www.irjet.net/archives/V3/i4/IRJET-V3I4117.pdf

- "Audio steganography using DWT & DCT", Sumeet Gupta, Namrata Dhanda, https://www.iosrjournals.org/iosr-jce/papers/Vol17-issue2/Version-5/F017253244.pdf

- "High capacity reversible data hiding in MP3 based on Huffman table transformation", Dingwei Tan, Yuliang Lu, Xuehu Yan, Lintao Liu and Longlong Li, https://www.aimspress.com/fileOther/PDF/MBE/mbe-16-04-158.pdf