311289 Alexander Adelholm Brandbyge
251289 Frederik Hagelskjær
260387 Rudi Hansen
200757 Leon Bonde Larsen
040282 Kent Stark Olsen
160788 Kim Lindberg Schwaner

Inefficient network protocol

You know you like it

Faculty of Engineering
University of Southern Denmark
Niels Bohrs Allé 1
5230 Odense M
Denmark

www.sdu.dk/tek
+45 6550 7303
tek@tek.sdu.dk

Abstract

This project is based on the given assignment to create a protocol stack which use DTMF tones as information carriers, furthermore this has to be done through air by speaker and microphone. The team agreed on pursuing a secondary objective which was to develop an application programming interface for easy-to-use utilisation of the protocol stack.

It is natural therefore to use the layers described in the OSI-model to define the implementation of the protocol stack itself, and for the composition of the report. Interdisciplinarity is a big concern for this project as networking- and datacommunication theory is mixed with the basics of, digital signal processing, software development, and C++ programming skills. Thus, this report will reflect these skills used to analyse given problem statements and solving them.

Everything points in the direction of accomplishment of the objective as a fairly stable protocol stack was implemented, though it is not efficient compared to state of the art data communication systems which exist today. In time of writing, the application programming interface is still under development.

The result work as a proof of concept, that DTMF tones can be used as information carriers when the data is transported in air. The significance of creating this proof have given a great insight in topics regarding the basics of, networking and data communication, digital signal processing, software development, and C++ programming.

Contents

List of Figures

List of Tables

Listings

Abbreviations

API	Application programming interface
CRC	Cyclic redundancy check
DFT	Discrete Fourier Transform
DTMF	Dual-tone multi-frequency
FFT	Fast Fourier Transform
IIR	Infinite Impulse Response
RUDP	Reliable User Datagram Protocol
TCP	Transmission control protocol
UDP	User datagram protocol

Temp: Writing conventions

For your eyes only

Naming conventions

This is a list of how to write names of things

PortAudio is how they spell it at `http://www.portaudio.com/`

> "PortAudio is a free, cross-platform, open-source, audio I/O library."

Boost is spelled Boost:

> "Boost provides free peer-reviewed portable C++ source libraries"

LaTeX references

If one writes a reference to a *specific* Figure, Table, Chapter or similar, they are written with a capital letter. For example:

```
To see how a figure looks, see Figure \ref{fig:label}
```

eller

```
To read the first chapter, go to Chapter \ref{cha:chapter_one}
```

Chapter 1

Introduction

This project is devised as a part of the B.Sc. Robot Systems Engineering, 3rd term course at the Faculty of Engineering, University of Southern Denmark, Odense autumn 2011. The implicit goal is to obtain knowledge of computer applications for signal processing. The focus will mainly lay at understanding analogue and digital signals, and integrate this processing in computer applications where computer architecture, operating systems, and data communication will enter into as significant competences.

As the DSMI[1] model is used as a framework for this particular education, it is required for the students participating in this course to formulate a strategy for solving an assignment in teams.

1.1 Requirements

This project has in many ways an experimental approach, where it is largely up to the students to figure out how to solve the assignment. This is due to the requirement of building a protocol stack which uses DTMF tones as data carriers. The exact requirements of the assignment are listed below:

- Computers must communicate by exchanging DTMF tones in air.
- A protocol stack must be build, layered architecture.
- Information must be exchanged between computers.
- A distributed application must be developed in C++.
- It is required that the application performs some meaningful task.

1.2 Secondary Requirements

The team decided on expanding the list of requirements, this was done because guidelines regarding the tools for planning, documentation, and development of this project was needed. This was done because along with the theory it was desired to get a basic knowledge of these tools as they ease the administration of maintaining documents and software when several people are working at the same project at the same time.

- Easy-to-use application programming interface is developed as distributed software.
- The application programming interface has to be cross platform.

[1] The Engineering Education Model of the University of Southern Denmark

- The report itself is type setted in LaTeX and is written in English.
- Google[1] docs is used for internal documentation, minutes of meetings, etc.
- Google[2] calendar is used for time management.
- Trello[3] is used as the planning tool for deadlines, to do lists, etc.
- Git and GitHub[4] is used as version control system for the report and the software.

1.3 Report structure

The report will be split into several chapters which each explain the different areas that have been explored to fulfil the requirements of this assignment. In chapter 2 the general concepts will be discussed. Chapter 3 will delve at the physical layer, which define the rules of the data transmission itself and how it should handle this data regarding the upper layer. Chapter 4 will contain an overview of the data link layer, which explains concepts around the frame, flow control, and error control. Chapter 5 will explain the workings behind the process-to-process communication. The backbone which control the flow of data between layers is explained in chapter 7. Utilisation and concepts of the application programming interface which acts as the easy-to-use feature of the protocol stack is explained in chapter 6. Chapter 8 will explain about the test tool that have developed to test each layer separately for stability issues. Experiments conducted will be held in chapter 9. A discussion of the process, solutions, and the results takes place in chapter 10. The conclusion will reside in chapter 11.

Along with this report a CD will be attached, which contain all material produced during this process. The source and the distributed application will be included on the CD as well.

1.4 Project members

This section will serve as an introduction to each of the project group members. It briefly tells which parts of the project, each member has concerned himself with mostly and what role he has been filling. These roles originate from the Belbin team role model. As the project was begun, the group sat down and discussed which roles they would like to try and fill. They were selected out of a wish to learn what a particular role would involve, and not necessarily what each member is best at.

Alexander Has primarily been working on the general architecture of the project, and thedevelopement of the backbone class + associated buffers.

Frederik is a slem bandit

Kent has been working with the physical layer which is the lowermost layer of the OSI-model. It has demanded for his skills in terms of understanding the basics of, networking and data communication, digital signal processing, software development, and C++ programming. As a team player Kent is still challenged and he still needs to work on handling conflicts with greater dexterity.

Kim is a slem bandit

Leon is a slem bandit

Rudi is a slem bandit

[1] https://docs.google.com/
[2] https://www.google.com/calendar/
[3] https://trello.com/
[4] https://github.com/

Chapter 2

General concepts

Introduction.. overordnet hvad vil vi lave?

- An overview.
- Naming conventions? Such as, our library is the "Dtmf library"

2.1 Network layers and the OSI model

The task of enabling two or more processes to communicate across physical machines requires a lot of work, and a lot of errors have to be considered, depending on the medium used to transport data. Addressing, both at the process and machine layer has to be handled, and the interface to the physical medium needs to be considered. Grouping problems into logical sections help to solve this problem, as an otherwise very complicated procedure can be broken down into small self contained processes. One such way of grouping a network framework is the OSI model [2, pp. 27-42]. The OSI model is represented as a stack of processes (hereafter referred to as a "network stack"), which together enables a group of processes to communicate across physical and logical networks. In itself the OSI model is not an actual network stack, but it is a model of an architecture that is both robust and flexible. As seen in Figure 2.1 it defines seven layers in a networking application, where each layer corresponds to a logical section of the network protocol, and a message passes from the top layer in the first process, potentially all the way to the bottom of the stack, until it has arrived at the second process stack, after which it begins "climbing" the stack, and thereby making the message available to the second process.

Each layer interfaces to the layer above and/or below it, either adding or removing data from the package that is passed. This is either done directly, or by a third-party control mechanism depending on the choice of architecture. Layer one, two and three can be considered the network support layer, they are responsible for ensuring dataflow through the network of physical machines and they are expected to be active and usable, even though no client application is using the network on a machine.

2.1.1 Physical layer

This layers responsibility is to enable a stream of bits to be broadcast between physical machines, typically through a wire, by the means of electricity or in this case through air, by means of sound. Synchronization, encoding and the transmission speed, are all things that the physical layer needs to handle.

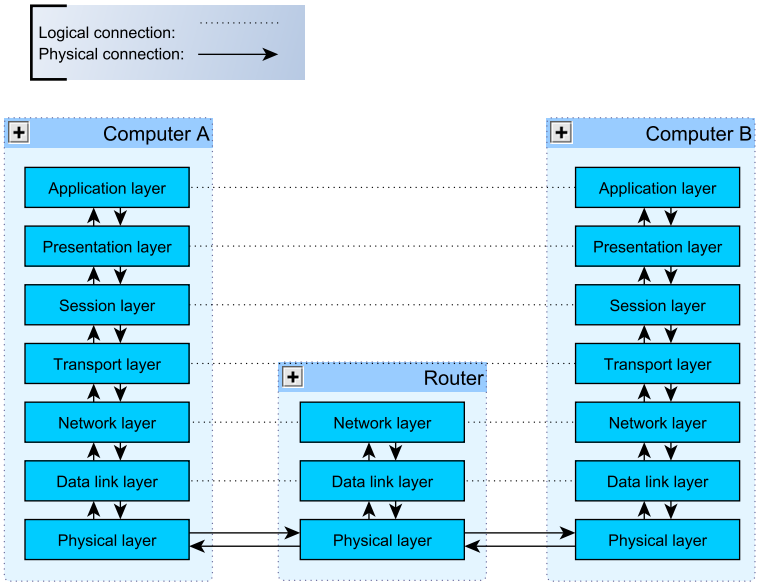

Figure 2.1: OSI model

2.1.2 Data link layer

The data link layer uses the physical layer to present an error free transmission line. Data coming from the higher layers, is divided into frames (small manageable units) which the physical layer can handle. Local addressing is also handled by the data link layer.

2.1.3 Network layer

The network layer is responsible for package routing across larger networks, especially across networks where the sender isn't directly connected to the receiver. It handles sending and receiving arbitrarily large blocks of data from and to the higher layers.

2.1.4 Transport layer

The transport layer ensures that a process can deliver a package to another process, thus making sure packages arrives in order and intact.

2.1.5 Session layer

This layer handles synchronization between two network processes, and dialog control (whether communication should be full duplex or half duplex).

2.1.6 Presentation layer

The presentation layer is concerned with ensuring that the data is interpreted correctly (int, float, string. etc), encryption and compression.

2.1.7 Application layer

This is the final application, using the network stack. For instance a chat client, email client etc.

2.2 Callback functions

Some of the functionality in the library depend on the concept of callback functions, and a short introduction to them is given here. A callback function, is a piece of executable code, that can be executed at a later time. This is widely used in order to allow users of libraries to define custom functionality, that the library can call without knowing what it is at compile time. For instance, the user of our library defines a callback function, that will be executed when a message is received.

2.3 Architectural goals

For many libraries developed, it is a priority that the final product is easy to use, efficient and powerfull. The same goals apply to the network library: It must be easy to use, the user should be able to start the library by just telling it what address it has, and it should be very easy to send a message, with nothing more than the data to send and an adress. Efficiency is more difficult, especially since the project requires using DTMF instead of an electric signal, but within that constraint, there is room for efficiency. Also, the user-application thread will need to spend as little as possible execution time in the network library code, and do as little as possible memory management related to networking operations, as this will minimize faults in the library by user actions.

2.3.1 Overall system architecture

The general architecture of this library, as seen in Figure 2.2, is to have one centralized backbone construct, which is the only part of the library to have any long term data storage and awareness of the network state. Within the backbone, resides a set of buffers used to store intermediate data and a set of layers, loosely corresponding to the layers of the osi model. The layers are: Api, transport, datalink and physical. The layers above the transport layer are deemed outside of the scope of this library, and instead an api layer is introduced to handle incoming and exiting data. Since we are not utilizing an electrical connection to transfer data, an extra layer is introduced below the physical layer, to handle playing and recording sound in a reliable way, this is the audio interface.

Layers are represented by an object that can encode or decode a sequence of bytes according to its rules. For instance the Data Link Layer can take a sequence of bytes representing a packet, and encode it to the form of frames. However each of the different layer representation has no way of controlling when it is being used, no contextual information(i.e. knowledge of any other object in the system), and are therefore considered pure function objects. The backbone monitors the buffers and the layers, and controls that transformation of data is done in a way that eliminates congestion as much as possible and ensures no data is lost. Since the backbone must be able to operate autonomously, it will be instantiated with its own execution thread. The data storage can then be regarded as an assembly line, where data is taken from one buffer, transformed and put into the next buffer, in order to apply all the necessary packaging that network communication requires, before the ”end data” is sent to the audio library or the Api layer (in the case of sending and recieving data respectively). Some of the layers might need attention based on other criteria than the condition of the buffers, it is the responsibility of the backbone to handle this as well.

Facade class

Any calls from the user goes through the API class and are passed on to the backbone. Data packages to and from the user are also buffered in the backbone to ensure low coupling. Since this is the only class the user can instantiate, it also has the responsibility of initializing the rest of the library upon creation. When a user application wishes to send a message, the

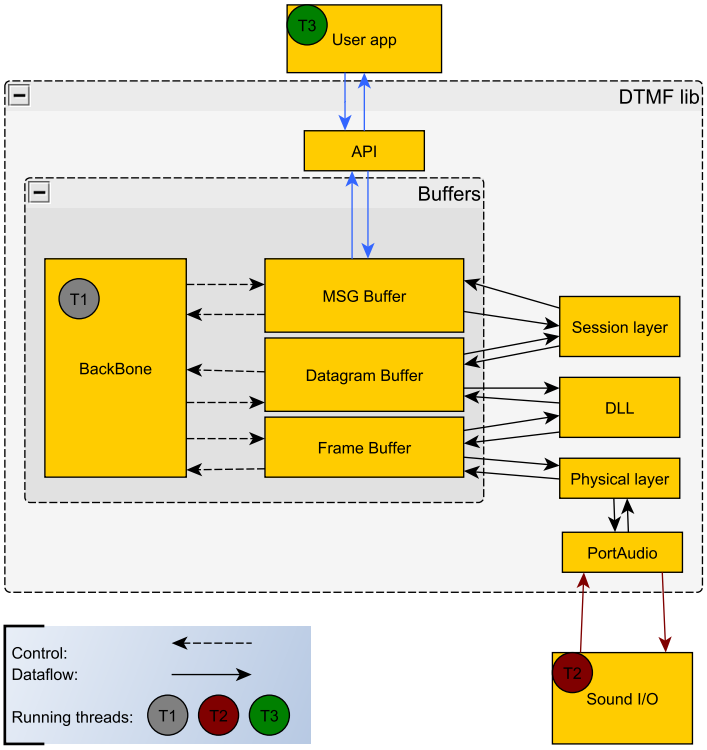

Figure 2.2: General DTMFLib architecture

API will provide a "message" data holder, which can be filled and passed back to the API, this removes the need to do memory handling from the user application. In order to receive messages from other systems, the user must register a callback function, that will be called by the network library when system has received a complete message. The lifetime of this message will be limited to the scope of the callback function.

The layer classes

The three layer classes reads data from a buffer, works on the data and writes it into the next buffer in the chain. Each layer may instantiate temporary internal buffers to store partially processed data between attention. Some of the layers will need to send a message to the layer before it, and when executing an encode or decode function, the layer is given access to both the appropriate send and receive buffers.

Chapter 3

Physical Layer

This chapter will be split into three parts, first part will explain the main considerations that have been made during the design phase of a physical layer which uses DTMF tones to act as information carriers. The design is based on the basics of networking and digital signal processing theory which means that the physical aspects will not be taking into considerations and thus not be in the scope of this document. The design will create the foundation for implementing the physical layer as software.

The second part will explain the implementation in greater detail regarding the physical layers interface, which functionality is exposed to the data link layer, and the internal workings of the BufferedSoundIO class.

The third part will contain a brief discussion of the thoughts made after the implementation was done.

3.1 Framework Composition

The physical layer defines the electrical and physical framework for receiving and transmitting signals through the media, and furthermore it defines encoding/decoding and alignment schemes for translating frames into signals and visa versa.

This project requires the use of DTMF tones as information carrier, and furthermore it is required that transmission are broadcast through the air. These requirements decides some of the properties of the physical layer, the first one is the media which is the air as sound waves are used for communication, and the communication can only take place as half-duplex as sending and receiving is impossible as the communication system works as a broadcast system.

3.1.1 DTMF as information Carrier

DTMF is an abbreviation for Dual Tone Multiple Frequency. It is a system of tones that are used by telephones when dialing a number. The system is an arrangement of four low tones and four high tones, they are arranged in a four-by-four matrix which gives the system sixteen combinations.

The idea is that these sixteen combinations formed by the DTMF matrix is used to transmit data between two or more computers. To let DTMF tones enter into data communication we have to apply the property of waves to carry bits. This can be done by letting each entry in the matrix consist of four bits. Four bits gives sixteen combinations which each can be assigned to an entry in the matrix. Below is shown the DTMF map which will be used in the physical layer for encoding and decoding.

Now by letting one computer play the tones through a speaker and another computer record it from a microphone then data is transmitted. The exchange of data is essentially an exchange of information and it is not satisfying exchanging information of the size of

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	0000	0001	0010	0011
770 Hz	0100	0101	0110	0111
852 Hz	1000	1001	1010	1011
941 Hz	1100	1101	1110	1111

Table 3.1: Table of the matrix with the bit combinations assigned to each entry.

four bits alone, cause this would make the system inefficient. The system need to be able to transmit a sequence of tones matching a given bit pattern. The number of tones played each second will determine the bit rate of the communication. As each tone hold four bits the bit rate can be written as below:

$$bitrate = \frac{numberOfTones \cdot numberOfBitsPerTone}{time} \tag{3.1}$$

3.1.2 Utilisation of PortAudio Interface

PortAudio exposes streams for recording or playing back sound through the sound card of a computer. PortAudio allow for developers to push raw audio data to the soundcards outgoing buffer and receiving raw audio data from the soundcards ingoing buffer. Thus, PortAudio is good foundation for the implementation of the physical layer as tone detecting, and tone generating algorithms can be laid on top of PortAudio, as basic mathematics which apply to digital signal processing. The PortAudio API provides the user with a callback function which implicitly exposes the sound card buffers for input and output. PortAudio furthermore gives the possibility for using more sound streams which then provide more callback functions which means that input and output streams can run asynchronously. The way PortAudio handles these streams will be explained in greater detail later.

PortAudio will be implemented as a part of the physical layer to create the interface between the developed protocol stack and the sound card. An example of this is shown in figure 3.1.

The DtmfPhysical and BufferedSoundIO will be implemented as two separated classes which will be bounded together and exposed as the physical layer to the rest of the protocol stack. The idea for implementing the physical layer as two seperate classes on top of PortAudio API, is to separate functionality between the physical layers internal layers. BufferedSoundIO class will be implementing the math needed to generate and detect tones while the physical layer class will implement the encoding schemes for transforming frames from the data link layer into a sequence of numbers which then can be pushed to BufferedSoundIO which then processes this sequence to generate a sequence of DTMF tones at the sending site. In the receiving site the receiving process will look a lot like the process for sending a frame, the BufferedSoundIO class will detect sequences of DTMF tones which will be transformed into a sequence of numbers. This sequence of numbers will then be sent to the DtmfPhysical and then be translated into a frame.

3.1.3 Goertzel Algorithm

To be able to detect if tones have been transmitted some algorithm for detection of tones have to be implemented. For this purpose the Goertzel algorithm is used, this algorithm has the ability to detect if a signal contain a specific frequency. Other algorithms exist which would provide the same information but those would be much more expensive regarding the computational complexity. Those other algorithms are called Discrete Fourier Transform (DFT)

and the other is called Fast Fourier Transform (FFT) which is a more efficient way to obtain the same information as the DFT algorithm. The computational complexity of DFT

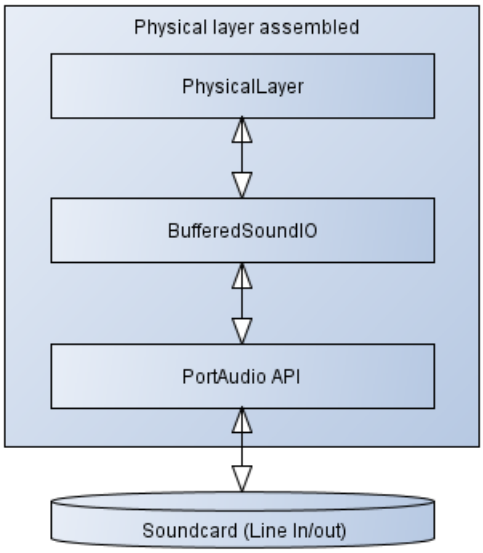

Figure 3.1: This is a model of the steps needed to implement the physical layer. The PortAudio API is delivered by PortAudio. BufferedSoundIO is handling the math for generating sound and detecting sound, and also the PortAudio API. The physical layer includes the encoding scheme and functionality exposed to the rest of the developed protocol stack.

according to [4, p. 124] is N^2, where N is the number of samples. For FFT the computational complexity is calculated to be $\frac{N}{2} \cdot \log_2(N)$, where N is the number of samples.

As the detection of tones has to occur as fast as possible it is desired to lower the cost of cpu power by using an algorithm which has the least computational complexity. The Goertzel algorithm therefore suit this need very well. The reason for this is that with a few pre-calculated constants and N iterations over N samples, a value is returned which indicate if a specific frequency is present in the incoming signal.

Essentially the Goertzel algorithm is a second order IIR filter which is dependent on current input and previous output, the filter is given as the difference equation shown below:

$$y(n) = x(n) + 2 \cdot \cos(2\pi \cdot f_0) \cdot y(n-1) - y(n-2), \tag{3.2}$$

where f_0 is the frequency of interest.

By Z-transform the following is obtained:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - 2 \cdot \cos(2\pi \cdot f_0) \cdot z^{-1} + z^{-2}} \tag{3.3}$$

As the above equation show, it is a second order IIR filter. This can be implemented as a direct form-II structure where the point W is of interest.

The point W will be used for calculating the frequency response at a specific frequency of interest. As this is taking place in discrete time, an expression of the frequency of interest is needed. In discrete time the frequency spectrum is divided into frequency bins, the size of these bins is determined by the number of samples and the sample rate. Obtaining the k'th bin can be done as shown below:

$$k = \frac{f_0}{f_s} \cdot N \tag{3.4}$$

where f_0 is the frequency of interest, f_s is the sampling frequency, and N is the number of samples.

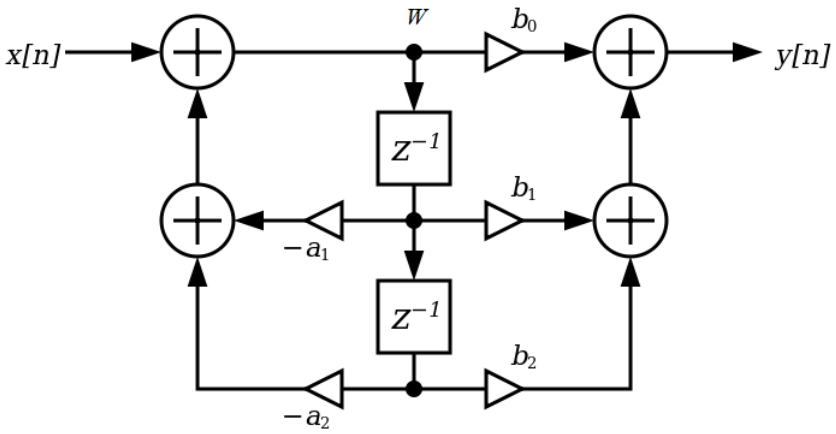

Figure 3.2: A direct form II implementation

Now it is possible to calculate the filter coefficients, which in the case with the Goertzel algorithm reduces to one single coefficient. This coefficient have to be calculated for each tone that want to be identified, but can be calculated in advance because its only dependent on the frequency of interest.

The coefficient is calculated with:

$$c = 2 \cdot cos(2\pi \cdot \frac{k}{f_s}) \tag{3.5}$$

Now that all constants can be calculated in advance the detection system only have to calculate the filter output, calculate the magnitude of the frequency and then it is able to decide based on the result if the frequency exist in a incoming signal. As detection of DTMF tones is needed for this application it needs to determine if two specific frequencies is present at the same time. But this wont be much of a problem as the constants for each frequency can be calculated in advance. The algorithm are implemented as a direct form-II structure, and need to iterate over the array of collected samples to be able to detect if a given frequency is present in a signal.

The computational complexity can therefore be written as one multiplication plus two additions per iteration per tone. This leads to:

$$numberOfOperations = 3 \cdot N \cdot M, \tag{3.6}$$

where N is the number of samples and M is the number of tones.

For one tone detection over 205 samples this is around 600 calculations, for detection of 8 tones simultaneously the equation above result in around 5000 calculation, and these are real calculations where as for DFT and FFT the computational complexity is much higher and the calculations are carried out with complex numbers.

This is the reason for choosing the Goertzel algorithm for detection of frequencies.

3.1.4 Synchronization

The physical layer will be handling the synchronization of the data stream. Keeping the data in sync enables the software to keep track of the given chunks of data from the upper layer, this is important to do because the physical layer at the receiving site will have to assemble the DTMF tones back into the exact same chunks of data for delivery to the upper layer. A way to solve this need is to wrap the content of data into a header and possibly a tail. The chuck of data received from the data link layer would be natural to wrap in a

header and a tail as an extra precaution to indicate if the frame is transmitted. This will ease the assembly of a frame because the software have an indication of the exact start of a frame and the exact ending of it as well.

Another problem with synchronization arise due to the mapping between 4 bit combinations and DTMF tones. This is because if there is a need for sending, 1111 and 1111 right after each other the system will not be able to identify the two tones corresponding to the given bit patterns from each other. It would therefore look like only one tone was transmitted instead of two. This actually apply to each combination which are followed up by its own combination of bits. Some kind of stuffing is needed to separate each tone from each other.

There are several ways in which this problem can be solved. One of the possibilities is to stuff the transmitted tones with a little bit of silence in between each other. This implementation would require some sort of timing scheme which would rely on precise timing so decisions, on how the recorded silence should be interpreted, can be made. Decisions that will define the transmission of a frame, will be as already mentioned, where does a frame start, where does it end, and the stuffing in between each tone. All these properties will be hard to manage duo to silence can be considered as white noise which is totally random. White noise is not in the scope of this document.

A second solution could be to do the stuffing at the data link layer in between equal bit patterns with a another bit pattern. The are several disadvantages duo to use of this method. First the synchronization would now be spread across two layers because the physical layer still would have to track the beginning of a frame and the end of a frame. Another disadvantage is that this method is unreliable as the defined bit pattern for stuffing at the data link layer could be generated by the data contained by the frame itself. This mean that when data containing the bit pattern for stuffing this data could be discarded on false reasons and then corrupt the frame.

A third and more elegant solution is to add a ninth frequency to the map seen in table 3.1. The new map will then look like the table below:

index		0	1	2	3
	DTMF	1209 Hz	1336 Hz	1477 Hz	1633 Hz
0	697 Hz	0	1	2	3
1	770 Hz	4	5	6	7
2	852 Hz	8	9	10	11
3	941 Hz	12	13	14	15
4	350 Hz	16	17	18	19

Table 3.2: This new mapping system create four new combinations of DTMF tones. The index notation can be used for implementation purpose.

In table 3.2 a new mapping of DTMF tones is shown. This new mapping system contain four new combinations of tones which can be used as control signals. The combination $1209Hz$ and $350Hz$ gives the number 16, this number could be used as a code for *here does the frame start*, 17 for *here does the frame end*, and 18 for *two equal bit patterns are presented after each other*. Code 19 can be leaved as a reserved control code.

This method for synchronization is also well connected to the rest of the data transmission at the physical layer as the implementation is based on a tone detection system, so by adding an extra tone a lot of new possibilities are offered, in exchange of complicated timing schemes or sharing the synchronization functionality between two layers. The detection system can now by identifying a shift in tone frequencies detect a new tone was sent and thus the system is able to register the tone.

Code	Representation
16	Start of a frame
17	End of a frame
18	Double tone
19	*Reserved*

Table 3.3: Show a table of control codes(tones).

3.1.5 Encoding scheme

The encoding scheme is handling the translation from frames into DTMF tones and visa versa. The encoding will be a two step process as frames are translated into a sequence of numbers which then are translated into a sequence of DTMF tones.

A frame can be considered as a stream of bits, this stream is the actual data the frame consist of. This stream can be divided into 4 bit sequences where each number represent the value from zero to fifteen both included. These sixteen combinations match up with table 3.1 and the four upper rows in table 3.2. After dividing the frame into nibbles[1] and each nibble is put into a list, this list now is a sequence of numbers. Each number is assigned to a combination of two tones according to table 3.2. These frequencies can now be calculated based upon the number representing each nibble.

Calculate low frequency:

$$f_l = \frac{number}{4}, \tag{3.7}$$

where f_l is a whole number, meaning if the outcome is not whole, digits after the comma are discarded. Number represent a number from the sequence numbers representing the data stream of a frame.

Calculate high frequency:

$$f_h = number\%4, \tag{3.8}$$

where f_h is a whole number from the modulus operator, this operator is used here to return the rest of the division.

If low and high frequencies are arranged in two arrays as they are listed, one for low tones and one for the high tones, then f_l and f_h represent an index of a tone in the frequency array concerned.

The generation of numbers from identified frequencies is also an easy calculation to carry out, as this would look like this:

$$number = f_l \cdot 4 + f_h, \tag{3.9}$$

This can be confirmed by table 3.2.

The latter calculations defines the encoding scheme between DTMF tones and numbers, which is one of the steps needed for the full encoding. The other step is to get a frame and translate it into a number sequence and likewise translating a sequence of numbers into a frame. At the sending site frames are split into nibbles and stuffed with control tones at relevant spots.

An Example:

12	1	1	15	10	11

Table 3.4: Represent a frame and each number is a nibble of that frame.

The example above would then have to be stuffed as below according to table 3.3.

[1] A 4 bit size

16	12	1	18	1	15	10	11	17

Table 3.5: Represent a frame as the number sequence after stuffing it.

As seen in table 3.5 the frame itself is wrapped in control codes telling the software where the frame start and end. In between nibbles of the same bit combination a control code is added to indicate that two equal tones are going to be played right after another as this frame is transmitted.

3.2 Physical layer software Implementation

Before implementation is started a few decisions have to be made. First thing is defining an interface which tell which functionalities that will be exposed to the upper layer. Second phase of the decision making is to develop flowcharts that define the flow of events that have to take place to make the physical layer as functional and stable as possible. A decision about using Boost C++ Libraries[1] was also made so utilisation of the supported ring buffer can take place. This decision was taken to avoid implementing this buffer type ourselves as this is a time consuming process, making it work and the optimization which follows. By using Boost we get this vital functionality handed over for free. This section will not go into every little aspect of implementing the software, but will give good overview of how the physical layer is utilized and also how it does work internally to process incoming and outgoing data. It is therefore recommended to take a look at the files located at src/physical/ to get the full view of the physical layer. Files that spoken of is listed below:

- DtmfPhysical.h - This file contain the interface for the implementation of DtmfPhysical class.
- DtmfPhysical.cpp - This file contain the implementation of DtmfPhysical class.
- BufferedSoundIO.h - This file contain the interface for the implementation of BufferedSoundIO class.
- BufferedSoundIO.cpp - This file contain the implementation of BufferedSoundIO class.

3.2.1 DtmfPhysical.h - Interface

As mentioned the first decisions will define the requirements of the interface. Decisions around the interface are listed below:

1. Constructor have to take parameters for controlling sample rate for incoming and outgoing sound streams, length of sample buffers, resolution, and number of channels. All the parameters have to be available for later tweaking purposes.
2. The interface need to have a send method which take a void pointer as parameter where the frame buffer can be passed as a reference so manipulation of the incoming buffer can take place at the physical layer. This is due to
3. The interface need to have a receive method which take a void pointer as parameter where the incoming frame buffer for the data link layer can be passed as a reference so manipulation of this buffer can take place at the physical layer.

[1] http://www.boost.org/

3.2.2 DtmfPhysical.cpp - Implementation

The interface implementation takes place in the DtmfPhysical class which will be the class for exposing received data to the protocol stack and receiving data from the protocol stack for data transmission.

The *constructor* of the physical layer instantiates the BufferedSoundIO which setup PortAudio for utilisation. The constructor passes the parameters through the constructor of the BufferedSoundIO.

The `send(void*)` method takes one parameter as a reference to the buffer exposed from the data link layer to the physical layer. This buffer will be of the type `boost::circular_buffer<Frame>`. The `send` method check if a transmission is going on as `send` is called. If not it resize an internal buffer in the DtmfPhysical so this buffer can hold all frames as one sequence of numbers, the type of the sequence buffer will be `boost::circular_buffer <unsigned int>`. This sequence is then pushed to the BufferedSoundIO class which internally holds a buffer of the same type. A simple flowchart of the `send` method is shown in figure 3.3.

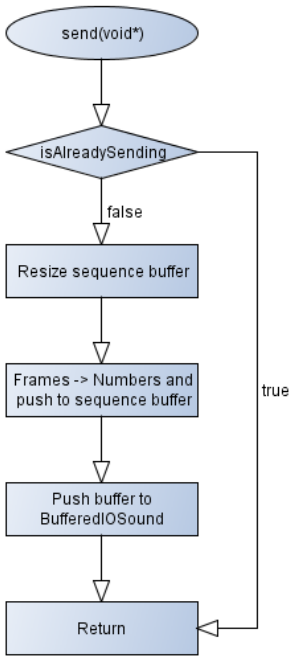

Figure 3.3: An overview of the internal workings in the `send` method

The `receive(void*)` method takes one parameter as a reference to the buffer which stores data for link layer is delivered from the physical layer. This buffer will be of the type `boost::circular_buffer<Frame>`. The `receive` method check if the internal frame buffer is empty. If this is not the case the frames are delivered up to the data link layer. A simple flowchart of the `receive` method is shown in figure 3.4.

3.2.3 Internal workings of the BufferedSoundIO

The interface of BufferedSoundIO will not be explained in greater detail as the instantiation is handled by the DtmfPhysical class.

The DtmfPhysical class is fairly simple compared to the BufferedSoundIO class. This is because BufferedSoundIO have to control two processes at the same time which are fired by PortAudio.

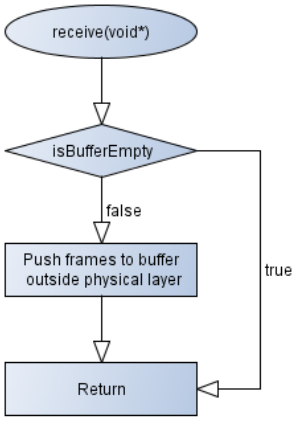

Figure 3.4: Flowchart shown here gives an overview of the internal workings in the `receive` method.

1. Output streaming
2. Input streaming

To understand BufferedSoundIO a little knowledge of how PortAudio works internally is needed. PortAudio provides the user of the API to initialize and start a sound stream through a computers sound card. This stream *of sound* or data is available through a callback function delivered by PortAudio. As PortAudio is programmed in C, this callback function is simply a function pointer[1] which allow for the user to create a function in his program that will be called when PortAudio is calling its own callback function. This callback is called according to the setup of a given stream, for instance, if the sample rate is $8kHz$ and the stream is initialized as only output with a buffer of 8000 samples, then will this stream fire the callback every second as the buffer is emptied every second. The same goes for input streams but a little different, every time the input buffer is full the stream callback is fired. To implement this type of function a well defined pattern have to be followed, the pattern is defined by PortAudio and come in the form of a function declaration with an arbitrary name.

```
1 static int paStreamCallback(const void *inputBuffer,
2                              void *outputBuffer,
3                              unsigned long framesPerBuffer,
4                              const PaStreamCallbackTimeInfo* timeInfo,
5                              PaStreamCallbackFlags statusFlags,
6                              void *userData);
```

Listing 3.1: PortAudios callback function declaration. This declaration is used for both input streams, output streams, and the two in combination

BufferedSoundIO is instantiated with DtmfPhysical and therefore resides in DtmfPhysical. BufferedSoundIO itself setup two streams where each represent either a input or output stream, this is done because the physical layer works on two streams of data seperately. As two streams are now present, a foundation for communication with sound is founded. These two streams are able to work asynchronously which might proof to be an advantage later on. If the input callback is fired twice as often as the output callback, could proof to make

[1] http://www.newty.de/fpt

the communication more reliable as this could address some of the overlapping issues which could lead to loss of data. Exact scientific tests on the topic will not be performed due to time constraints, but a rule of thumb is, be sure to fire the input callback at least twice as often as the output callback.

Output streaming

The output stream fires the internal `paOutputStreamCallback` function of BufferedSoundIO. As shown in listing 3.1 the last parameter taken by PortAudio callback `void *userData` is used so the user is able to pass arbitrary data as a reference into the callback. Regarding BufferedSoundIO this parameter is used to send a reference to the object itself in which the stream resides. Listing 3.2 show the exact implementation of `paOutputStreamCallback`.

```
1 static int paOutputStreamCallback(  const void *inputBuffer,
2                                     void *outputBuffer,
3                                     unsigned long framesPerBuffer,
4                                     const PaStreamCallbackTimeInfo*
                                          timeInfo,
5                                     PaStreamCallbackFlags statusFlags,
6                                     void *userData)
7 { return ((BufferedSoundIO*)userData) -> outputStreamCallback(
      inputBuffer, outputBuffer, framesPerBuffer, timeInfo, statusFlags);
       }
```

Listing 3.2: Implementation of `paOutputStreamCallback`

This way of implementing the callback makes the callback able to reach data which resides in the object of BufferedSoundIO, `paOutputStreamCallback` call the `outputStreamCallback` which is also a member function of BufferedSoundIO.

The `outputStreamCallback(...)` essentially generate sound according to entries in an internal buffer of BufferedSoundIO if there is any present, a sketch up of the flowchart of `outputStreamCallback(...)` is shown in Figure 3.5.

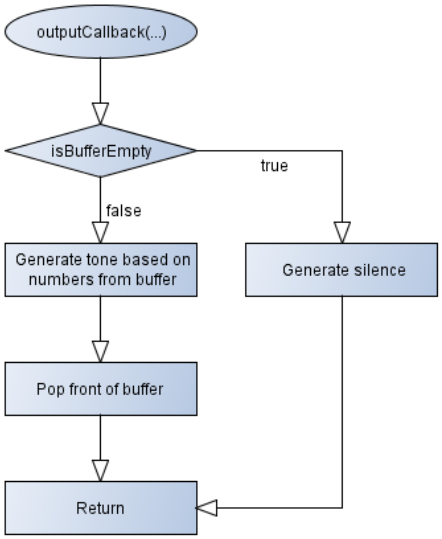

Figure 3.5: A sketch up of `outputStreamCallback(...)` function. For more details this can be found in BufferedSoundIO.cpp

Input streaming

Input streaming works in the exact same way as the output stream regarding the callback. BufferedSoundIO have a function named `paInputStreamCallback` which handles the callback from PortAudio, `paInputStreamCallback` call the `inputStreamCallback` function which handles the internal input buffer exposed by PortAudio. Figure 3.6 show a sketch up of `inputStreamCallback`.

This function is a little more complex compared to the output callback as this implements the Goertzel algorithm for detection of tones and also a decision making system which analyse the state of the signal to see if it is a valid signal. If the signal is valid a code is calculated, this code is now validated to see if it is contained in the mapping system of tones and codes. If the code is valid it is pushed to the input sequence buffer which is of the type `boost::circular_buffer<unsigned int>`. When ever the decision making system see the code 17, which corresponds to end of a frame a callback is made from BufferedSoundIO to DtmfPhysical asking the DtmfPhysical to pull the codes from the input sequence buffer which is held in BufferedSoundIO.

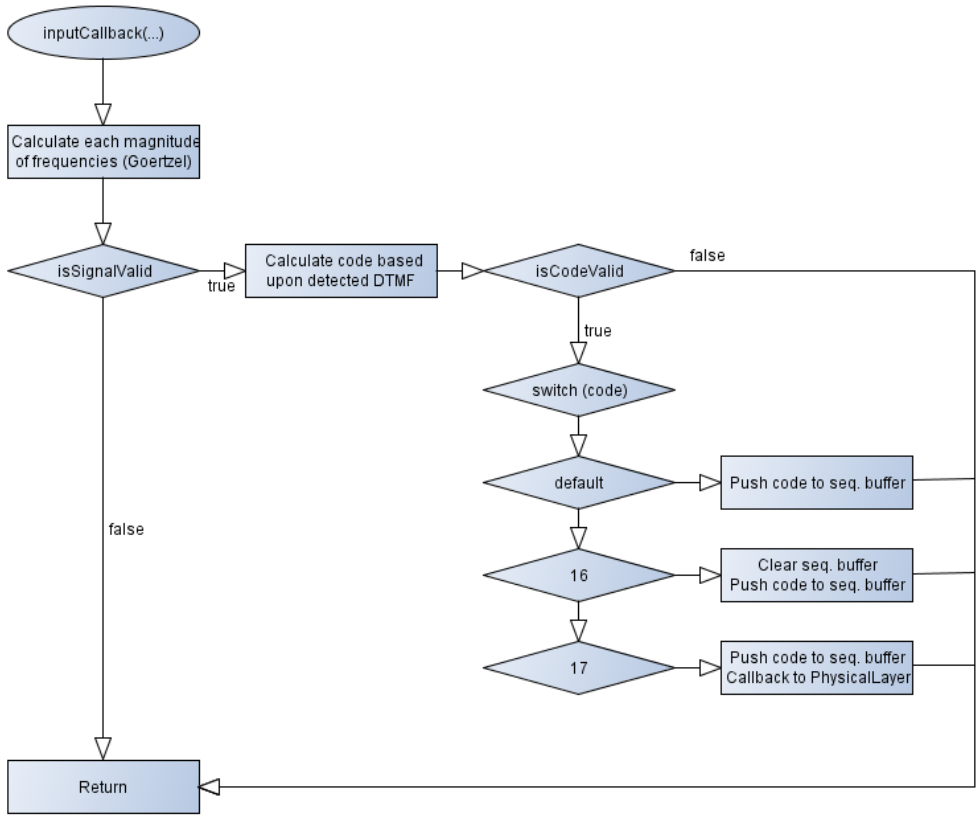

Figure 3.6: A sketch up of `inputStreamCallback(...)` function. For more details this can be found in BufferedSoundIO.cpp

3.2.4 Callback to DtmfPhysical

Everytime the input callback in BufferedSoundIO recognise the code 17, a callback is made back to the DtmfPhysical. This callback pull the codes from BufferedSoundIO, generate a frame from the information pulled, and store this frame in the frame buffer of DtmfPhysical. This ensures that frames and the functionality for generating sequences from frames and

generating frames from sequences are kept in the DtmfPhysical class. A sketch up of the function is shown in figure 3.7

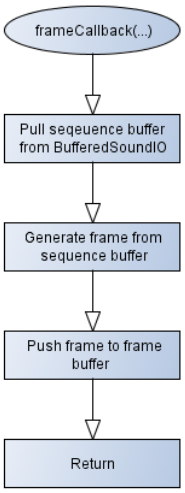

Figure 3.7: A sketch up of `frameCallback(...)` function.

3.3 Discussion

Here goes pip-hans...

Chapter 4

Data link layer

The main purpose of the data link layer is to present a flawless line of communication to the transport layer. This means that the data passed to the receiving transport layer must be exactly the same transmitted from the sending transport layer.

4.1 Overview and considerations

There are several subjects to examine before designing the data link layer. First of all the requirements must be identified.

- Short frames
- Error detection
- Error correction
- Pipelining
- Multipoint

4.1.1 Encoding

First thing to consider is the encoding of signals. Since there are sixteen different DTMF combinations, each tone can carry four bits.

The only type of error to consider is the situation where a tone is misinterpreted, which leads to a four bit burst error (reference). Therefore the system should be designed specifically to detect errors of this type. Since the media is considered to be very noisy, transmissions should also be kept as short as possible.

A two dimensional parity check will be able to detect burst errors of the proposed size, so this is the choice. Implementing the parity check as a four-by-four matrix will make it possible to transfer two bytes with a frame of three bytes.

Example: We want to transmit the two bytes 0011 0101 and 0101 1110. The data link layer puts these in a four by four matrix and calculates the parity bits by adding the rows and columns:

Instead of as normally done to increase the size of each row by one to contain the parity bit, the parity bits are transmitted together as a redundant byte. In the case of this example the transmission would be:

Each four bit nipple is now transmitted as a DTMF-tone. Should one of the tones be misinterpreted, the receiving data link layer would get a mismatch of the parity bits for example:

This will lead to the frame being discarded. Though in some cases it might be possible to correct the error and find the original nibble, this is not recommended, since more than

0011	0
0101	0
0101	0
1110	1
1101	

Table 4.1: Two-dimensional parity check

0011	0101	0101	1110	0001	1101

Table 4.2: Bytes to be transmitted

one tone might be corrupted. Furthermore it complicates the data link layer significantly, and the gain is low since the frame could be re-transmitted in six extra tones. Errors in the redundant byte will also lead to discarding the frame.

4.1.2 Flow control

The next thing to consider is flow control. Since the DTMF-system cannot be used as full-duplex, piggybacking (reference) is impossible. This means that eventually the receiver will have to reply. This reply will also be of six-tones to take advantage of the parity system, and therefore it might as well contain information about witch frames to resend. In other words a selective repeat system is preferred. To introduce a selective repeat system, additional redundancy is needed.

Pipelining must be considered to increase the speed of the system and is implemented as a three bit sequence number. This limits redundancy to a minimum while still benefiting from pipelining. The sender transmits eight frames (forty eight tones) and then waits for the receiver to reply with one frame (six tones). Should the receiver not respond or should the reply be lost, the sender will automatically re-transmit after a while.

4.1.3 Frame design

The physical layer will provide a starting point for each transmission. If this is was not the case, additional flags would be needed in between the frames, leading to the need of stuffing (reference) and to additional redundancy.

A frame size of three bytes is preferred for the proposed parity system. This means preferably transmitting one byte per frame. A smaller payload would mean increased complexity and lesser efficiency.

Normally bytes are transmitted in chunks of eight, but since the length of datagrams can vary, the sender must have a way of telling the receiver to process smaller chunks. This introduces the need for an End Of Transmission flag (EOT).

0011	0
0101	0
0000	0
1110	1
1000	

Table 4.3: Failed parity check

Next thing to consider is multipoint. There are three options: A token network(reference), a time division network(reference)) or a code division network(reference). Time division requires a level of timing the interface layer is unable to deliver. Code division leads to the need for larger frames or if implemented with the proposed frame size, a lot of unused frames in a small network. This leaves us with a token passing network, so this is the choice.

The selective repeat system and the token network introduces the need for different frame types. The proposed type field will consist of two bits, at the same time controlling the token and indicating frame types. Table 4.4 shows the meaning of the two bits.

00	Has no token	Reply from receiver
01	Has no token	Passes token to addressed station
10	Has token	Accepts token
11	Has token	Data frame for addressed station

Table 4.4: Protocol for type field

In reply from receiver frames each bit of the data byte corresponds to a sequence number and has value 1 for accepted and 0 for resend.

Since two bits is reserved for type, three for sequence number and one for flag, the remaining two bits will control the addressing. Both can be used for identifying the receiver, since info about sender is not needed at this level. Thereby the protocol allows networks of up to four stations.

Token passing is controlled entirely by the data link layer. When the token is offered, there is a window of response time, wherein the station must reply by accepting the token. If there are no reply during the window, the token is offered to the next station.

This leads to the following format of a frame:

type (2 bits)	address (2 bits)	sequence (3 bits)	EOT-flag (1 bit)	data (8 bits)	parity (8 bits)

Table 4.5: Final frame format

4.2 Implementation

The data link layer is implemented as a `DataLinkLayer` class and a Frame class. The `DataLinkLayer` object is instantiated by the backbone class and controls the network token and the processing of frames into datagrams and vice versa. When instantiating, the address and token is controlled by arguments. The Frame objects and datagram objects are instantiated in buffers by the backbone and presented to the data link layer as method arguments.

Two public methods are used to call the data link layer, one for upwards traffic (`decode`) and one for downwards traffic (`encode`). Both methods are called with pointers to the four accessible buffers as arguments. Furthermore the backbone can call a method (`needsAttention`) to know whether the data link layer need extraordinary attention because a timer has run out. Another method (`canTransmit`) tells the backbone whether the data link layer object holds the network token and thereby is able to send encoded data.

The methods of the classes are developed to realize the flowchart of the data link layer. Decisions on whether to implement a method in one class or another is done using the expert pattern.

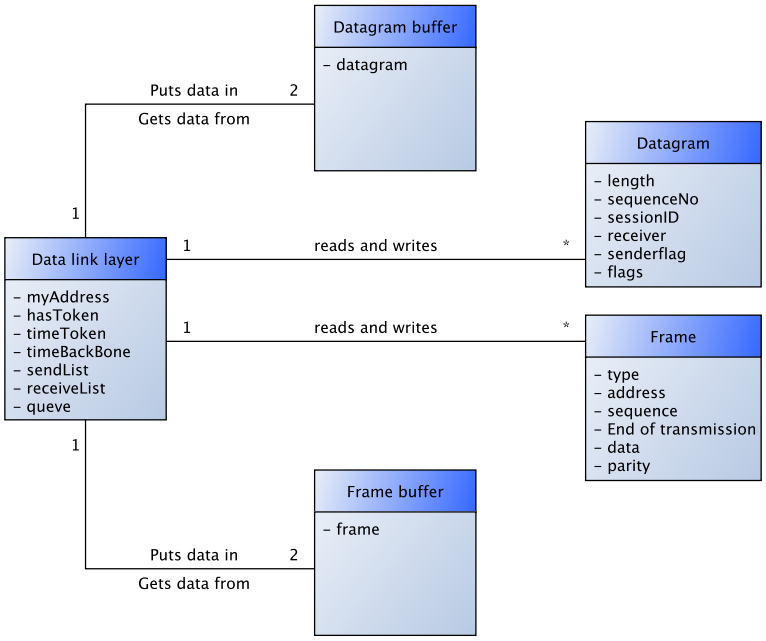

Figure 4.1: Domain model for data link layer

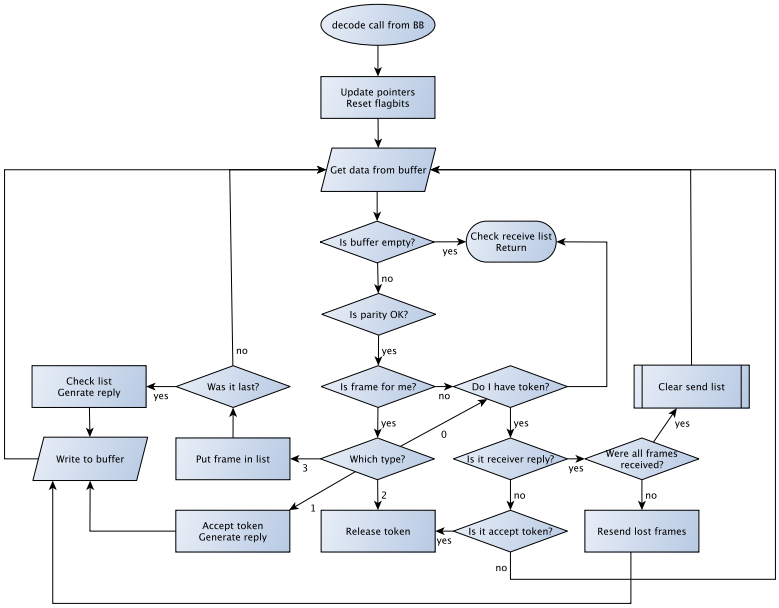

Figure 4.2: Flow chart for data link layer decode method

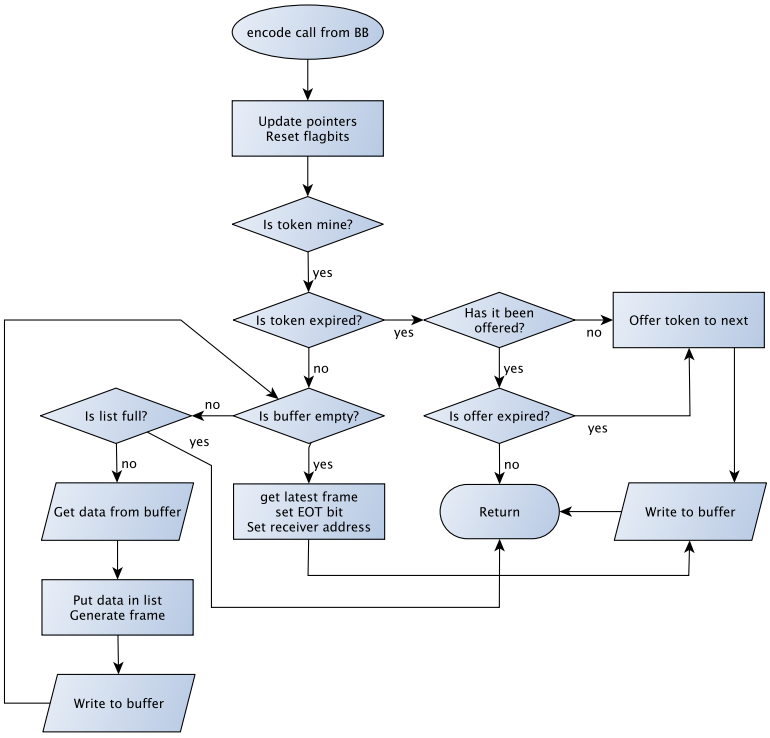

Figure 4.3: Flow chart for data link layer encode method

Chapter 5

Transport layer

In the OSI model, the session and transport layers handle process-to-process communication and sessions (chiefly used when a more permanent connection is required for synchronous transfer), respectively. When initially exploring ideas for this networking protocol stack, one wish was that it should be able to serve more than just one application at a time. This brings a need for such process-to-process delivery that the transport layer protocols provide. In this chapter, the design and implementation of a transport layer protocol (in this entire chapter referred to as *the protocol*) is described. To the extend possible, the above- and underlying layers are viewed as if they are unknown to the protocol, in order to provide a generic interface.

Protocol stack?

5.1 Design theory

In this section, the reasoning behind the design of the protocol is described. That is, any thoughts and ideas, about the inner workings of the protocol, that has ether been put into code or discarded for whatever reason. This whole section is, thus, a description of what went on before the actual software implementation took place.

5.1.1 Speed versus reliability

A dilemma, when formulating how the protocol should function is, that as high as possible speeds are desired. On the other hand, one must remember, that we want to make it as painless as possible for the user application to use the protocol (meaning the user should be able to expect successful delivery).

Ref. to the "project ideas" document

A UDP[1]-like protocol design would no doubt be one of the best ways to achieve a low overhead. The UDP protocol provides a connectionless service, where datagrams[2] are sent and then forgotten. That way the receiver does not have to spend bandwidth to acknowledge successful receptions. Additionally, the header size of a datagram is quite small, as there is no overhead used for error- or flow control. Datagrams are also sent without any way of knowing whether they are received in-order or not.

Using a TCP[3] inspired protocol instead would enable the transport layer to provide a much more reliable service. Now, in the TCP protocol, flow- and error control is added, as well as congestion control. Every packet is assigned a sequence number, ensuring they are passed on to the server application correctly, by the receiving transport layer protocol. Every packet is also acknowledged by the receiver, if it is successfully recorded. If not, the sender re-sends them. All these things combined makes TCP a lot more reliable, but it also increases it's header size to around three times the size of the UDP header. Especially if

[1] User datagram protocol.
[2] The transfer unit of the UDP protocol.
[3] Transmission control protocol.

one is not sending a lot of data in each packet, the header can take up a large percentage of the combined (header plus data) package size.

Although the services provided by the data link layer guarantees an error-free connection, Ref. we cannot be sure data is received in the correct order. The data link layer does not provide *that* guarantee. As there is an uncertainty here, we need to establish a way to ensure that data is passed on from the protocol to the above layer in the correct order. This can be done by using a system, where each outgoing data packet is assigned a sequence number.

Sequence numbering provides a way to make sure packets are delivered in the correct order, but with just the sequence number, it is not certain that the data bytes *inside* a given packet are in the correct order. For this a checksum calculation is called for.

All summed up, a protocol design, that will ensure in-order delivery of data, is needed. Features as flow- and congestion control is ignored. Mainly because the communication is going to take place via sound after all: We can safely say that the transfer speeds will not Ref. be high and congestion is unlikely to occur. After all, the communication line is inherently half-duplex, making the data flow one-way (at a single point in time) by nature.

Coincidentally the RUDP[1] described in [1] suits the needs well on many points. Therefore many of the properties, portrayed in the following parts, are based on those of the RUDP. Note that the RUDP document is an *internet draft*, meaning it is work-in-progress (although it has seemingly not been updated for a long time).

5.1.2 Addressing processes

The data link layer takes care of the node-to-node delivery, where each node has an address. There is a problem however: The receiving data link layer protocol strips the header, containing the node address, off before the information reaches the transport layer protocol. Figure 5.1 shows this: The solid lines are the actual data transmission path, while the dotted lines are the individual layers' "pseudo-transmission paths". **DH** is the data link layer protocol header and, as we can see, it is not passed on up through the layers.

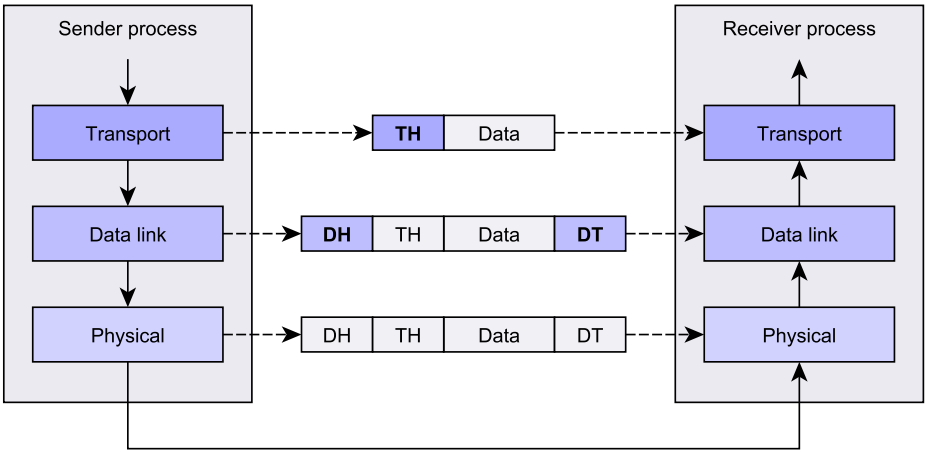

Figure 5.1: The transmission path from sending to receiving transport layer protocol

To obtain a way of distinguishing signaling processes from each other, another attribute must be introduced: A port number.[2] The port number, assigned to a certain server application, must be known to the client before it starts sending. That is the only way the

[1] Reliable UDP protocol.

[2] Port number, in this text, is separate from the port number normally associated with a process when it *binds* via some kind of Internet socket. Still, the term is used, as it's purpose is the same.

client process will know where to send data. In the protocol, we select the port number to be eight bits long. This reasoning behind this choice is, that it is unlikely that we have to server lots of applications at a time. Add to that, that we wish to keep a minimal overhead size (a higher port number would add more bits to the header), as we don't have a lot of bandwidth available to begin with.

5.1.3 Packet format

From the above layer the protocol expects to just receive a stream of data. The order in which the data is received is significant, but the data itself is not. When received, the data bytes will be packed into packets of a maximum length of 255 bytes (it can be smaller). A maximum length of 255 bytes seems like a sensible size.

Better reasoning/reference

All the attributes described above, leads to the packet header format as shown in Table 5.1. It is 8 bytes long, which leaves a maximum possible amount of 248 bytes of data per packet. The flags field will be treated throughout the next parts of this section.

0							7	8	15
Source port addr.								Destination port addr.	
SYN	ACK	NUL	RST	EAK	0	0	CHK	Total length	
Sequence number								Acknowledgment number	
Checksum									

Table 5.1: Transport layer protocol packet header

5.1.4 Sequence numbering

As mentioned, sequence numbering is introduced to make sure the receiving protocol passes on data to the above layer in the correct order. Every packet that is sent will have such a number attached. The sequence number is eight bits long. A packet will also have an eight bit acknowledgment number, which is closely related to the sequence number, as we shall see.

When a connection is initiated, a random sequence number between 0 and $2^8 - 1$ is selected. This number will be attached to a so-called SYN packet (the `SYN` flag is then set), which is used to establish a connection and synchronise the senders sequence numbers with the receivers. Each party must then increment this number by 1 before sending a response packet, except in a few special cases, which will be discussed. The sequence number is allowed to overflow, effectively resetting it to 0.

The acknowledgment number is also a number of 8 bits. It indicates, to the sender, the sequence number of a packet which the receiver has received correctly. If `ACK` flag is set it means the acknowledgment number is valid.

5.1.5 Error control

In order to avoid the risk of bytes shifting position inside one packet, a checksum is calculated. This checksum also serves as a strong error detection mechanism, although it should, in theory, not be needed as the data link layer protocol provides error control. The checksum type chosen is the cyclic redundancy check, or CRC, using the CRC-CCITT generator polynomial $x^{16} + x^{12} + x^5 + 1$. This provides a checksum size of 16 bits.

Elaborate on CRC + Ref.

The header of the packet is always getting it's checksum calculated. If the CHK flag bit is set, a checksum is calculated for the whole package. This is the default, and preferred, way of transmission.

5.1.6 Operation

Because of the need to synchronise sequence numbers, the protocol uses *handshaking* in the form of the SYN packet and the response to it. When a connection is first opened, the SYN packet, with a random sequence number attached, is sent. The receiver must then respond within a set time frame to verify the connection. This response must have an acknowledgment number equal to the sequence number just sent. Figure 5.2 shows the connection establishment taking place. Disconnected and connected refers to the protocols internal status; how it perceives itself.

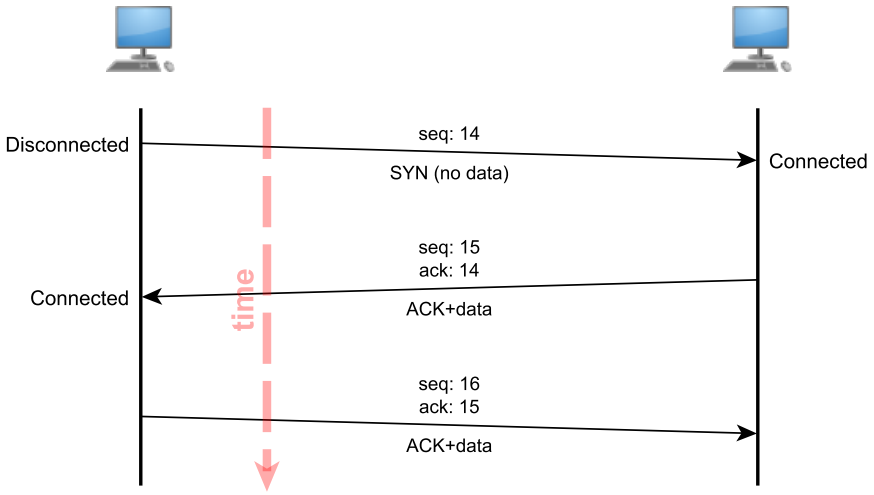

Figure 5.2: The transport layer protocol opening a connection

When a connection has been opened, communication keeps on happening as shown on Figure 5.2, except in the case where packets are received out of sync, or more than one packet has been received in a row before getting the chance to acknowledge them. If that happens we have a scenario as shown in Figure 5.3.

If the number of unacknowledged packets, either in- or out of sync, becomes too large (if the queue becomes too full), an extended acknowledgment packet (EAK flag set) is sent. This is a special packet containing no user data. Instead the data field consists of the unacknowledged sequence numbers. It is still limited by the maximum packet length, though. Table 5.2 illustrates the extended acknowledgment packet.

Header	Unack. seq. #1	Unack. seq. #2	···	Unack. seq. #N

Table 5.2: Transport layer protocol extended acknowledgment packet

In the case there is no data to send, but the user of the protocol wishes to gauge the connection status, a NUL packet (NUL flag set) can be sent. The NUL packet contains no user data, but still consumes sequence numbers normally. When a NUL packet is received, the receiver must respond to it within a specified time interval, or the connection is deemed broken.

The RST flag can be set to send a reset packet. When the reset packet is received, the receiver of the packet must stop sending data and flush it's data I/O queues. This can be

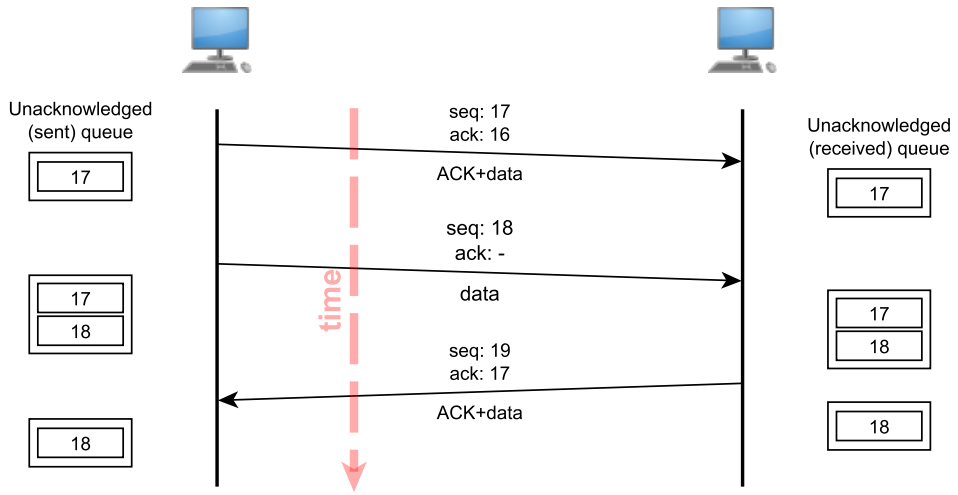

Figure 5.3: The transport layer protocol transmitting unequal amounts of packets

used to either reset the connection or close it entirely: If the `SYN` flag is also set, a reset will occur. A new, randomly generated, sequence number must be attached to the sequence number field in this case. If, on the other hand, the `SYN` flag is not set, the connection will terminate.

5.1.7 Retransmission handling

To accommodate the need for retransmission, the protocol will feature in- and output queues, which store whole unacknowledged packets. When an acknowledgment is received, the corresponding packet will be deleted from the queues. Each node will have a retransmission timer that, if it reaches zero, triggers a retransmission of any packets that has already been sent but are still unacknowledged. This retransmission timer is reset every time a packet, which requires acknowledgment, is sent. Only a set amount of retransmissions are allowed to occur before the connection is deemed broken.

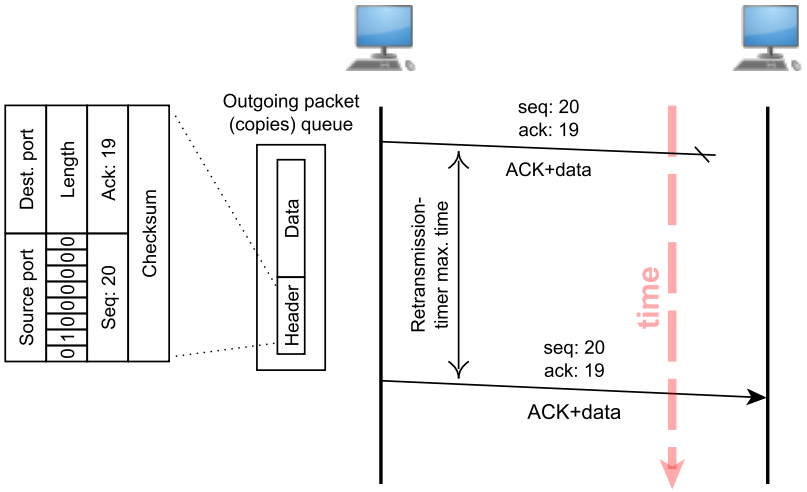

Figure 5.4: The transport layer protocol retransmission timeout

Figure 5.4 shows a packet with sequence number 20 being sent. It gets lost underway

though, and never reaches it's target. Luckily the sender had saved a copy of the sent packet, which he can then resend when the max. retransmission time has passed. When the receiver has later acknowledged the packet it is removed from the queue (this is not shown on the diagram).

5.2 Implementation

Armed with the specifications and design decisions mentioned in Section 5.1 it is time to step into the implementation phase. In practice this process is iterative, and the demands sometimes change with issues that emerge while working. This section aims to explain this process and describe some of the iterative steps that were taken.

5.2.1 General structure

To first give an overview of how the transport layer protocol is implemented, the general design structure is going to be clarified in the following: When first beginning the process of writing code, two main entities were identified. One is the protocol system itself, and other the packet data structure. These two are implemented as the C++ classes `DtmfTransport` and `Packet`.

The `DtmfTransport` class takes care of the connection and most of the features described in the design specifications. It is both the interface to the rest of the system and also the connection caretaker. With this, I mean that `DtmfTransport` is essentially initiating, maintaining and closing connections - although it all happens on command of the user (through the API). The `DtmfTransport` class makes use of the services provided by the `Packet` class to be able to set up, and manipulate, packet data structures.

`Packet` directly represents the packet format as it is specified in Section 5.1.3, meaning it basically consists of 8 bytes of header and up to 248 bytes of data. Add to that a number of methods to access or manipulate private members. The most important of these methods is probably `make()`, which is working as a so-called *lazy constructor*. Constructing lazily means that the `Packet` data members are not filled in before `make()` is called. Thus, `packet` objects can be passed around even without anything "in them".

5.2.2 Function classification

To better gain an overview of of which functions are doing what, they are split into two classifications: Essential- and compatibility functions. The essential functions are the ones providing functionality by directly implementing features, as described in the protocol design section. Compatibility functions are functions which are added to make the protocol compatible with other parts of the networking system.

Class	Essential	Compatibility
`DtmfTransport`	`encode()`	`toPacketQueueFromApi()`
	`decode()`	`packetFromCharBuffer()`
	`setPort()`	`packetToCharBuffer()`
	`connect()`	
	`close()`	
	`port()`	
	`connStatus()`	
`Packet`	`make()`	`makeFromArrays()`
	`calcChecksum()`	
	`flagSet()`	

Table 5.3: Transport layer protocol member function classification

Table 5.3 shows a list of member functions. Note that not all functions that are in the source files are listed. Those not listed are merely specialized versions of other functions, or functions to access private member data (and nothing else). As such they are not of great interest. The essential functions will be described more thoroughly in this section while the compatibility functions will only be touched briefly: They are mainly to fetch data out of buffers, to format data in a certain ways or to convert between object types. Yet they are important to mention.

5.2.3 Main protocol interface

As the protocol will need to function well with other parts of the networking system, the interface to it is of great importance. The other layers are developed simultaneously, therefore having well-defined interfaces - even before implementation is undertaken - is essential. Otherwise one would run into problems, where in- and output are of different formats and thus incompatible. To accomplish the goal of creating a "public" interface to the protocol, two in/out interface methods are used; `encode()` and `decode()`. These methods were the first part of the transport layer protocol to be defined. This happened in collaboration with the other team members. The rest of the protocol is built on top of these interface functions.

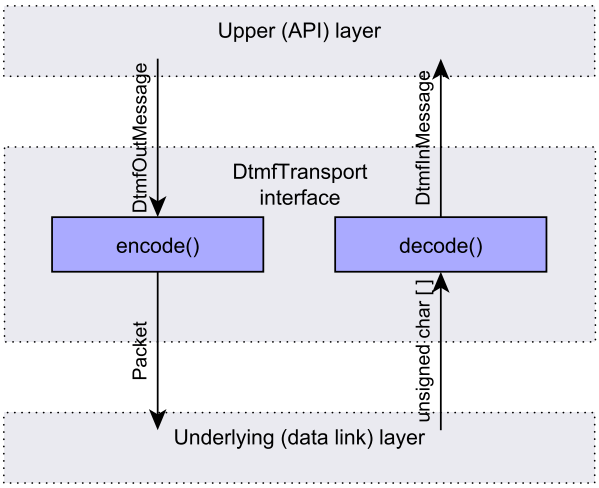

Figure 5.5: The transport layer protocol main interface methods

Figure 5.5 shows the `encode()` and `decode()` functions, and what they expect of in- and output from the above API and the underlying data link layer.

5.2.4 Encoding messages from the API

To first gain overview of the `encode()` method, we see that it requires a message in the form of a `DtmfOutMessage` object, as described in more detail in Section 6.2.3. This message object from the API contains the receivers port number as well as a pointer to an array of data. This is the data we want cut into pieces of the right size, to enclose in the `Packet` object.

What happens, when a suitable amount of data is "cut", is that `DtmfTransport` creates a new instance of the `Packet` object. All fields except for the receiver port number is filled out by `DtmfTransport` automatically without user interaction. This object then contains all of the information to pass on to the underlying layer.

5.2.5 Decoding messages from the data link layer

The method of choice, to pass data from the data link layer to the transport layer protocol, is chosen to simply be an array of bytes. These bytes are really stored in a buffer by the data link layer protocol. A pointer to this buffer is then passed on to the transport protocol. It is then the protocols responsibility to make sense of this array of bytes. This is done via the `packetFromCharBuffer()` method, which is a member of `DtmfTransport`. It examines the buffer to see if there is enough data to make out the header: If there is, a packet header will be assembled to then find out whether

Chapter 6

API

The application programming interface (API) layer is designed with the goal of presenting an easy-to-use interface for the Dtmf library. The following chapter goes trough some of the considerations and techniques used in the API layer design.

(The message buffers should be included in this section)

6.1 Usability considerations

The main responsibility for the API layer is to make it easy for the end user to set up and use the Dtmf library without the user having to worry about the layers below. The use of the API can be split into four parts: starting the library, sending a message, receiving a message and stopping the library.

6.1.1 Initializing the library

The main thing the library needs to know when starting is the address of the station on which it has been started, and whether it has the token or not (See data link frame design 4.1.3). These variables can advantageously be set by the constructor of the class. Because the library needs constant attention on the sound interface to be able to send and receive data, the send and receive logic will be running in a different thread called backbone thread, so the user code continues running in parallel with the Dtmf lib. Threading is described in 6.2.1.

6.1.2 Sending a message

Two situations are to be considered when designing the interface for sending messages. If the user wants to send a short amount of data and if the user wants to build up a message of several small data parts. To solve this, the API implements both methods on the API layer for sending messages. When a message is send, the API layer is responsible for transporting the user message from the user thread to the backbone thread, without causing thread race conditions (See 6.2.1).

6.1.3 Receiving a message

To avoid spending lots of user runtime on polling for new messages, callback methods are used. As the protocol supports several ports for receiving data, a callback method can be connected to each port. Note that if no callback is connected to a port, the data received on this port will be discarded.

6.1.4 Stopping the library

When closing the API, the allocated memory needs to be deallocated. This is done automatically by the deconstructor of the API. When the API deconstructor is called, it tells all threads to exit their tasks and shut down.

6.2 Implementation

6.2.1 Threading

Threading is used in the API layer to allow the concurrency that's needed between the user code, the send/receive logics and the callback method. By using threads, none of these will interrupt each others execution except when reading and writing from shared memory. Boost threads are used to make the implementation in the code easy and fulfil the multi platform requirements. When the API layer is started, it starts to child threads in the constructor before it returns.

Thread race conditions

One of the problems that can occur when using multiple threads is race conditions. A race condition[1] occurs when two or more threads are using shared memory at the same time. A good example is when two threads are trying to increase an integer in the memory by 1. Lets say that the value of the integer initially is set to 1. Both threads reads the value at the same time, so both threads reads a 1. They both increase the number by 1, and write it to the the memory. Now the integer has the value of 2, instead of the expected 3 because of the read collision.

Using mutexes to avoid race conditions

To solve the race conditions, mutexes (mutual exclusion objects) are introduced. Mutexes are locks that can be used to prevent two threads working on the same memory simultaneously. A mutex works by setting a mutex lock when entering an area of the code, and unlocking it when leaving this area. If another thread tries to lock the mutex while it's locked, the thread is paused until the mutex gets unlocked. By putting operations using shared memory inside mutexes, the threads takes turns using the data and race conditions can be avoided.

The volatile keyword

When compiling C++ code, the compiler tries to optimize the output application[2]. Practically this means that a variable can be buffered locally by a thread, and this will become a problem when to threads needs to access the same data. The volatile keyword can be used to tell the compiler to make the threads reload the value from memory every time it's used. This will ensure that the correct value is read from the value.

6.2.2 Thread description

Two threads are spawned when the API layer is initialized. The general idea behind these threads is to avoid interference between the Dtmf library and user application. The Dtmf needs to constantly monitor the surrounding sounds to be able to reply in time. Using separate threads avoids user algorithms stalling these important processes.

[1] http://support.microsoft.com/kb/317723
[2] http://msdn.microsoft.com/en-us/library/12a04hfd(v=vs.80).aspx

Callback thread

This thread is stopped by a mutex lock until new data arrives. When the backbone has put new data to the output buffer, it unlocks the mutex, allowing the callback to continue operation. The callback will start reading messages from the buffer. When reading a message, it looks up if there's a port defined for this message. If a callback is defined for the port, it will call the user-defined method with the message as argument. This allows the user to copy the data to a specified location. On exit the message is deleted. If there is no more messages in the queue, the callback thread locks the mutex.

Backbone thread

The backbone is described in the chapter 7

6.2.3 Method description

Initializing method

This is the constructor method of the API class. This method is called when the class is to be used. The first argument is the address of the machine the library is started on. The second argument is used for defining whether this node has the token when joining the network. Note that only one node can have the token at a time (See data link frame design 4.1.3). When this method is called, two threads are started. At this point some necessary pointers are exchanged, allowing the backbone thread to unsleep the callback thread when new data arrives. The threads are described in 6.2.2.

Add receiver port method

This method is used for adding listening ports, and assign a callback method to them. To define a class which can be used as callback object, it has to inherit from the `DtmfCallback class`. When a message is received the method named `callbackMethod` will be called on the defined object. Notice that this method will run in the callback thread. If the port is already defined, it will be overwritten. Only one callback method is allowed on each port.

Create new message method

This method is used when sending data. It returns an empty message, and enables the user to fill ind the message data manually. A reference is kept in the `DtmfApi`, so the message can be send itself by the build-in send method. This reference is used cleanup unused messages.

Send message immediately

When this method is called with arguments, the message is created internally and send to the que right away.

Deconstructings method

When the delete statement is called, the Dtmf library is shut down,. (See usage example D.5) When the lib is asked to shut down, it tells the running threads to exit their main loop and join the main thread. This may take a while depending on the state of the backbone. When threads are joined, all allocated messages will be deallocated.

6.3 Usage example

Usage examples are available in appendix (??)

6.4 Further improvements

Chapter 7

Backbone

The main controller of the application is the backbone, it is constructed by the API at launch, and after its construction, it starts its own thread and takes control of the network library.

7.1 The backbone class

The backbone class and its buffers, functions like a state machine, it controls the overall flow of the system by repeatedly checking the state of all buffers and deciding which buffer needs attention. Under all circumstances, as long as there is data in any buffer, the backbone will dispatch work to one of the layers, so the system only is idle when there is no data left to process. If more than two buffers have accumulated a sufficient amount of data, it is up to the backbone to choose the most urgent of the two buffers to work with, as this library has ample opportunity to choke itself with data, if it is not properly handled.

The end user application, does not decide how the backbone is allocated explicitly, instead, it will be created upon loading the library and instantiating by the API layer. This means that the startup of the library implicitly will be part of the facade pattern. The backbone class is also responsible for creation of the buffers and for handling settings and errors. The sizes of the buffers may be exposed through the API layer, for the end user to explicitly define buffer sizes, but the library should contain good default values, so the user does not need to configure this manually under most circumstances.

To prevent taking execution time from other processes in the user application, the backbone runs in a separate thread. This introduces thread racing conditions (See 6.2.1) at the API layer, when the user application wishes to send and retrieve a message, data may be lost, therefore the buffer must be constructed with a way to handle this situation, as this is outside the scope of the backbone itself.

7.1.1 Threshold values

Each of the buffers have an associated minimum and maximum value. Together with their size and data count, they define the response of the backbone when there are borderline congestion. The minimum value of a buffer is meant as a count of units that the system should aim to have in that buffer when possible. For instance the physical layer buffer needs to have above a certain amount of frames ready to play, because the library cannot guarantee when it can expect to execute next. Therefore the more data there is in the output buffers, the longer the network thread can wait before executing again. This is important, because the library will in the end no be running on a real time system, so execution time is not guaranteed. The maximum value indicates how much data there can be in a buffer before the system is close to congestion. Therefore whenever there is too much data in a buffer, the backbone should prioritize moving it out.

Too much outgoing data, or too little ingoing data in the system buffers, does not represent a stability threat, and the library will continue to operate at full, or close to full capacity under these circumstances, this means that the minimum threshold values doesn't apply to ingoing buffers, and maximum threshold values doesn't apply to outgoing buffers.

7.1.2 Execution flow

The method for determining which buffer to work with, is based on the following criteria:

In the case of sending messages, the physical layer buffer should never be empty unless all the other buffers are empty, otherwise the library will waste token time. This means that it is of high priority to move frames from the data link layer to the physical layer. At the same time, there must never be to many frames in the physical input buffer, otherwise data will be lost. Because buffer problems with the physical layer directly will result in erroneous operation of the library, these buffers must have the highest priority, when deciding where to "work". The inbound buffers has the highest priority, since faults here cause data loss, where as the other merely causes delays.

Next we have the frame buffers since, by the same logic, these can create congestion when there are to many frames to decode, which will later stall the physical layer, these will have high priority, but lower than the physical layer. And again the outgoing packet buffers will create a lack of frames if they don't deliver decoded packages.

The same argument goes for the transport layer, which of course has the lowest priority value. In the end the final priority queue is:

1. If amount of input frames >input frame maximum value, move frames from the physical layer.
2. If amount of output frames <output frames minimum value, move frames to the physical layer.
3. If amount of frames >frame maximum value, decode frames (produces inbound packages).
4. If amount of frames <frame minimum value, encode packets(produces outbound frames).
5. If amount of packets >package maximum value, decode packets(produces finished inbound messages).
6. If amount of packets <package minimum value, encode messages(produces outbound packages).

This prioritized flow (see figure 7.1) will ideally ensure that the system strives to stay in a "stable" state, during medium load time. Where, even if the library doesn't receive much processing time from the operating system, still manages to behave in a manner consistent with the protocol. When the system is a bit more relaxed, that is, all values are within working parameters, the system will assign work in a way that functions like round robin, that means, that the system checks all buffers once, and then if there is no work to be done, goes to sleep for a predetermined amount of time. The starting buffer to check, is advanced through the six buffers for each attempt at general work, in accordance with flow 7.2.

An important note: Whenever a buffer indicates that it needs to be emptied, it is not enough to just blindly empty the buffer, it is also necessary to check whether the output buffer can hold the result, if it can: great, if it cant then the buffer that is blocking the action needs to be emptied. The argument goes when a buffer indicates that it needs more items, if the buffer it fetches from is empty, then that buffer needs to be filled first.

Since there isn't a one-to-one correspondence between frames and packages or packages and messages, a buffer checker function needs to be created. One for determining whether there is room for a message to be decoded and one to determine whether the data link layer

can execute. The reason that the data link layers decode-and encode-functions aren't viewed independently is that the decode and encode function always need to be called right after one another (except when the layer indicates otherwise by the needs attention function), which effectively wraps up into a single data link action. Note that these checks depend on, among other parameters, the maximum size of packages and frames.

7.2 The buffer class

The buffer class itself is the container of all messages, packets and frames within the library. Within it resides three sets of sub buffers: Two message buffers, two package buffers and two frame buffers. The message are a bit different from the package buffers and frame buffers, and will be described independently.

They are used by the backbone to store messages, but the API (and by extension the user thread) can also pull or push messages from and to them, this introduces a concurrency problem, and the message buffer need to be able to handle the situation of simultaneous access flawlessly. Since the buffer is used as FIFO storage, it would make sense to be able to put a mutex (see 6.2.1) on only the last element of the queue, and implement it as a linked list. This would allow concurrent access that would not corrupt the list in any way since insertion in no way affected extraction, unless there is only one element in the queue, and here the mutex would guard it. When compared to the circular buffer offered by boost, or the simple array, there is a significant performance boost to the queue with a locked last element, as the flat array and circular buffer both would need a shared mutex at read and write, and both have a fixed size in order to be effective. An additional positive thing about the linked list, is that it can grow and shrink at will when needed. The downside is that the space is allocated on in the dynamic memory, and therefore possibly represent a point of memory leakage if not designed properly, and it takes up quite a bit of memory compared to the other two alternatives.

The frame buffer and package buffer have different needs compared to the message buffer. There is no need for handling concurrent access, and no explicit need to allocate more storage at runtime, as this will increase the running time of the individual layers. Therefore a type of circular buffer with fixed size is used instead, ideally with data allocated at startup, so the individual layers doesn't construct any objects at runtime, but instead uses those already allocated by the buffer. When a layer is used to decode or encode, the only parameters it receives are pointers to the buffers, as that is all the data they need.

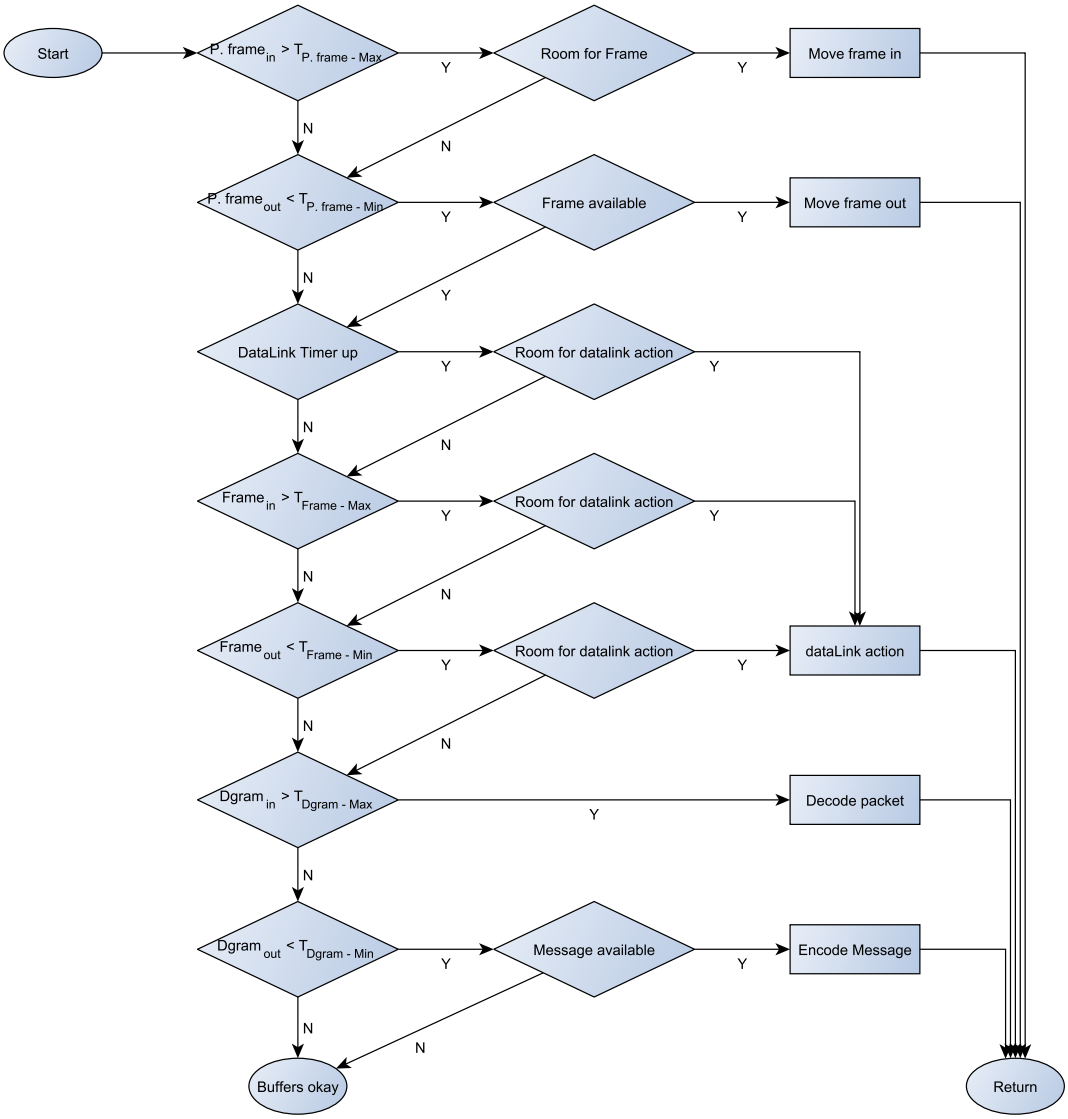

Figure 7.1: Backbone primary flow. This diagram illustrates the decision process when determining what buffers to service. ”P. frames” are frames still buffered in the physical layer

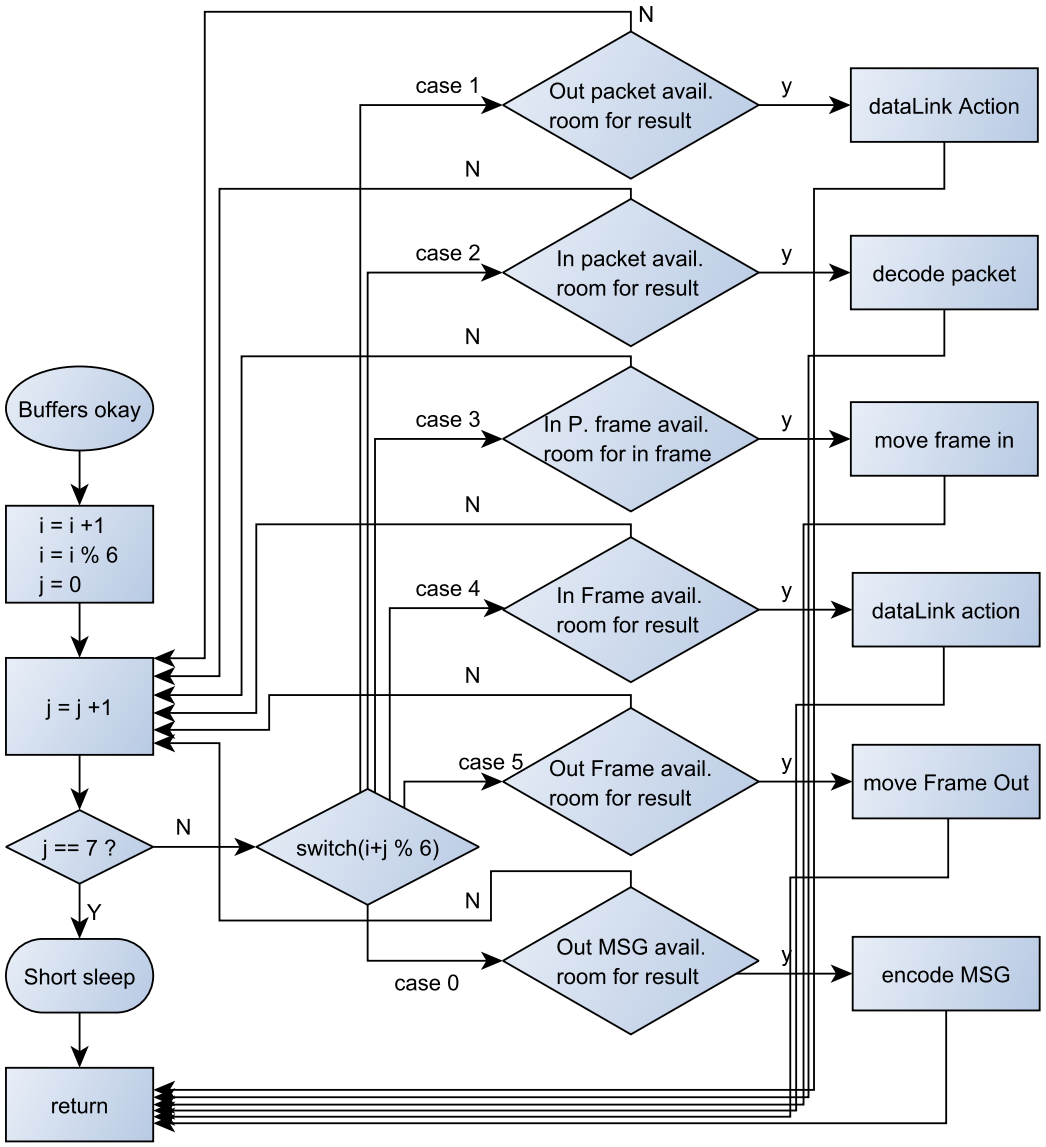

Figure 7.2: Backbone general flow. This diagram illustrates the decision process when the system is in a stable state, here the "switch" loop, ensures that all 6 actions are evaluated in order, but starting from a different one each time. "P. frames" are frames still buffered in the physical layer

Chapter 8

Test program

The strength in dividing development of a program into layers is that each layer can be developed, debugged and tested individually. If all layers are working individually, putting them together should prove much easier. Each of the layers are therefore debugged and tested individually before the final application is created. To be able to make easy and similar tests for all the layers, it was decided to develop a program for testing. Though there are two different ways to test a layer, the methods are the same.

8.1 Testing by substituting surrounding layers

To make it easy to change and manipulate the input and output, is it defined in .dat documents. Therefore the program doesn't have to be changed for different input, and the output can be easily extracted for further analysis. `Ifstream` and `Ofstream` are used to read and write from the documents. As the boost buffer is used as communication between the layers, is it also used as the test functions mean of communication with the layer tested. There are defined four buffers, two for input and two for output. Figure 8.1

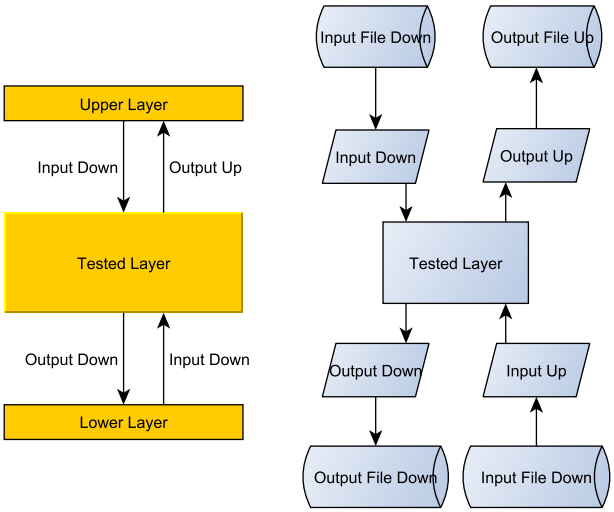

Figure 8.1: Substituting surrounding layers)

8.1.1 Testing a single instance of a layer

The test program is able to send data to a layer as if it was the neighboring layers. Thus the program makes it possible to give input to a layer and check the output. This method is used to check if a layer is capable of handling input and producing the expected output. This test is of course done throughout the development of the layers.

8.1.2 Testing communication between two instances of a layer

Another way to test a layer is to test actual communicating between two systems. In this test two instances of a layer is created, the upper layers are substituted as the single instance test. The lower layers are substituted with two buffers acting as respectively input and output for the two layers as seen in Figure 8.2 The two instances can then communicate, as if the lower layers are working perfectly. This way it is possible to test if all the functions of a layer is working, when communicating with another instance of itself.

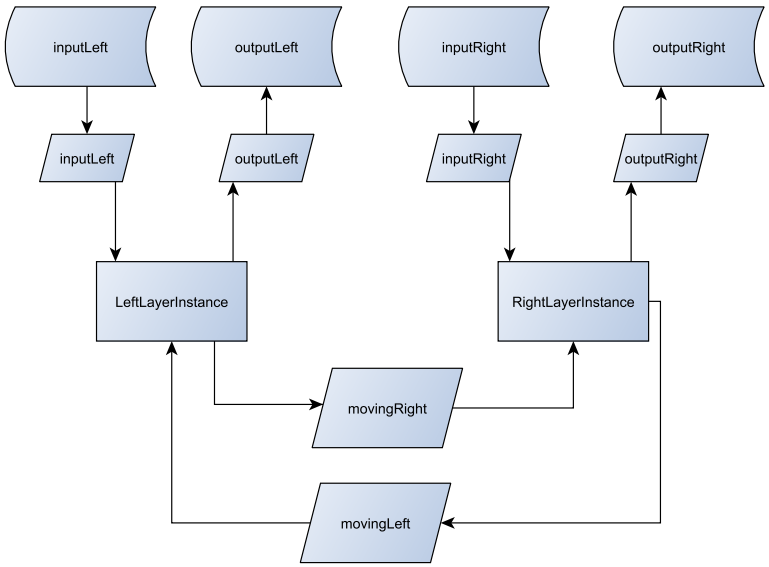

Figure 8.2: Two instances communicating directly trough buffer)

8.2 Creating a test program

In the beginning one simply includes the layer and in the function sets the name of the layer. It's is possible to change the defined names for the .dat files. The boost buffers are defined for the layer, so with this setup it's just running the program. Figure 8.3 shows the flowchart testing a single layer by feeding input to the inputbuffer, and after initiating the layer, collecting the output from the outputbuffer.

8.2.1 Physical Layer

The single instance test, whether the computer is able to send and receive sound have already been done while developing and debugging the layer. Therefore it is more logical to perform a test with two instances of the layer, examining how many of the sent messages are actually received by the other instance.

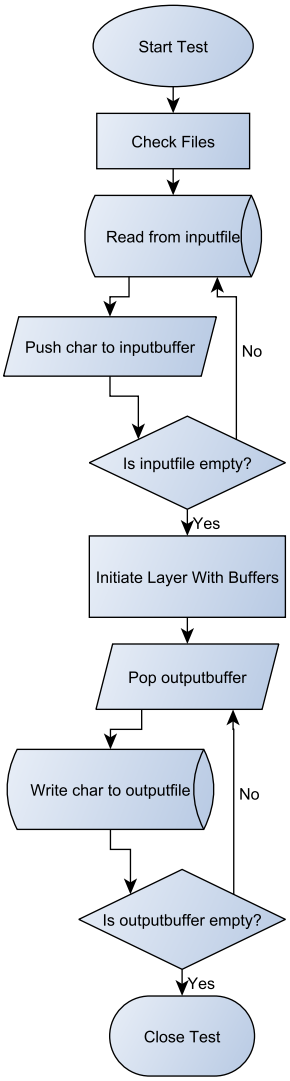

Figure 8.3: Flowchart for test function, testing a single direction

8.2.2 Data link Layer

The data link Layers main responsibilities is keeping track of the token giving permission to send and framing the messages before they are sent. To test that the framing are done properly, a test of a single instance of the data link layer is done. The two instance test is used for two purposes; testing the token and the ability to communicate with errors. The token test is mostly a debugging issue, but can also be used for optimization.

Testing the ability to communicate with errors is done by having to instances communicate, while inflicting a specified percent of errors. Thus it is possible to test the layers ability to maintain a connection with errors.

8.2.3 Transport Layer

More nEeded!

The transport layer is responsible verifying the checksum. An appropriate test is send-

ing a stream of messages to the transport layer, to see if only the correctly checksummed messages arrive. With the two instance test it would be possible to see if the transport layer is able produce and interpret the checksum.

8.2.4 Application Layer

The application layer is the users program interface. It is able to easily receive data as input and forward it to the transport layer so that it can be sent. A lot of single instance testing and debugging have to done to make sure that the application layer is able to handle all input correct. It is also necessary to create a program mimicking a users program, to test the application layers ability to transfer a file, using the application layers built-in functions.

The performance of the application layer is tested by building a file transfer program with the application layer recording the speed of transferring the files trough the application layer.

8.2.5 Layers Combined

In this test all the layers are connected and controlled by the backbone. This test is done as a potential user, with the options provided by the application layer. Using the file transfer program from the application layer, it is now tested by sending the file from the application layer test between two computers.

Chapter 9

Experiments

Throughout the development a lot of testing have been done to debug the program. This chapter is concerned about testing done towards the systems conditions. How much noise can it handle, what is the speed and ...

This is the result from these tests.

9.1 Psychical Layer

- physical tests? (throughput, background noise detection, etc)
- with test clas

9.2 Data link layer

- A test have been beformed were error was introduced to a signal
-

9.3 Transport Layer

- kim
-

9.4 Application Layer

- Moving a file trough the application layer
- with test class

9.5 Combined Layers

- How fast does it transfer a file
- Ping?

Chapter 10

Discussion

The main idea about implementing the DTMFBTINPL has been to find an approach where each member of the team was able to dig deeply into the essence of the professions while still making it easy to compile the different parts of the project into a coherent solution.

The chosen approach was to define clean interfaces between the software layers in the form of predefined buffers. This has proven to be a powerful tool, since each layer has been debugged and tested individually. Also a lot of potential problems about the overall flow- and software control was avoided, since all methods and loops were connected in the backbone program.

The architecture has provided a certainty of maximum reuse of code and have efficiently avoided unnecessary coupling in the software. Because of the bold ambition to create a compiled library for other developers to use, the problem reached a size, where efficient programming was imperative. Time was also an issue and again the procedure has proven most beneficial because each team member has been able to work and test independent of the others, thereby avoiding time wasted in waiting for other parts to finish.

The respective layers have been developed using an iterative approach, where the individual team member has worked on his part and then presented it to the rest of the group either by internal documentation on the project wiki, input to the report, demonstration programs or by giving short lectures in the subject. This way the team has been able to work efficiently while still pulling together.

The distribution of team roles has been a challenge, mostly because it implies a new way of thinking in behaviour and responsibilities instead of duties and fields of expertise. It was an informed choice to select the team roles from a point of learning and not a point of experience. This decision ensured that all members started on equal feet, but also imbued some difficulties of breaking old patterns. It has provided some serenity to the work, knowing that one does not have to be mindful of the entire project, trusting that other team members will do their part.

Professionally the project has given hands on experience of the OSI layers taught in data communication and a much greater understanding of the material. Developing a large software system leads to considering the UML language and advantages of object oriented programming, supporting the lectures in software development. Since all members of the team have contributed to the source code, a much greater experience in C++ programming has been obtained. Finally the analysis of the problem and to some extend implementing the tone detection system has lead to a greater understanding of digital signal processing.

Chapter 11

Conclusion

An API was developed - check The protocol stack was developed - check The crap worked
...

Bibliography

[1] T. Bova and T. Krivoruchka. *Reliable UDP protocol (Internet Draft)*. Tech. rep. Hosted on the the "Internet Engineering Task Force" (IETF) website. Cisco Systems, 1999. URL: http://tools.ietf.org/pdf/draft-ietf-sigtran-reliable-udp-00.pdf.

[2] Behrouz A. Forouzan. *Data Communications and Networking*. 4th. 1221 Avenue of the Americas, New York NY, 10020: McGraw-Hill, 2007. ISBN: 978-007-125442-7.

[3] J. Postel. *User Datagram Protocol (RFC 768)*. Tech. rep. Hosted on the the "Internet Engineering Task Force" (IETF) website. Information Sciences Institute (ISI), 1980. URL: http://tools.ietf.org/pdf/rfc768.pdf.

[4] Li Tan. *Digital Signal Processing: Fundamentals and Applications*. 1st. 84 Theobald's Road, London WC1Z8RR: Academic Press (Elsevier), 2008. ISBN: 978-0-12-374090-8.

Appendix A

PortAudio

As this project requires the usage of the sound card in a computer some interface is needed. A quick search on Google resulted in a countless number of possibilities for interfacing a C++ program with the sound card of a computer. The team decided on a set of requirements which the interface had to provide. Requirements are listed below:

- Easy-to-use application programming interface.
- Cross platform.
- Access to raw sample data.
- Variable sample speed.
- Support multiple streams.

The surface of several interfaces was scratched to see what they had to offer. Some of the possibilities are listed below:

RT Audio[1] was crap...

QT Multimedia[2] is developed by Nokia, and is a good suggestion as it comply with the requirements. The disadvantage is that core files from QT itself is needed which result in increased size of the developed software. Not that the size is specified as a requirement of the project itself but it also result in a superior complexity of the developed software itself, so QT Multimedia was discarded.

PortAudio[3] was finally chosen as the lowermost component of the software as it comply with the requirements mentioned above. The reason for this final decision was based on the simplicity of the usage of PortAudio. Beside this point it is extremely adaptable which make it possible to use in a wide range of applications.

[1] http://www.music.mcgill.ca/~gary/rtaudio/
[2] http://doc.qt.nokia.com/latest/qtmultimedia.html
[3] http://www.portaudio.com/

Appendix B

Boost C++ Library

Boost provides a repository for free peer-reviewed portable C++ source libraries. Boost provide a lot of quality components for developers to use and in the future some of them might become a part of the STL[1] library. The reason for choosing Boost is made because of the quality Boost can deliver, and this library is well documented. Furthermore Boost provide all the components needed for implementation of this project. Components used in development of this project are listed below:

Circular buffer is provided by Boost to give the developer the freedom to instantiate circular buffers of a given type. This type of buffer is also known as a ring buffer or cyclic buffer. It is essentially a data structure which forms a ring.

Thread is provided by Boost which enables a developer to manage multiple threads. A thread holds a process which is executed on a computer, allowing the use of multi threading enables a developer to execute multiple processes at the same time if the hardware and operating system is compatible with this behaviour, otherwise the multi threading is executed as multitasking, which is controlled by a time-division multiplexing process.

CRC is provided by Boost to enable a developer with an easy-to-use CRC component. CRC is an abbreviation for cyclic redundancy code. The use of CRC in this project is based on the desire to check if data have been correctly sent and received.

[1] Standard Template Library

Appendix C

Encoding at the physical Layer

Appendix D

Usage example

D.1 Creating instance

```
1 int main()
2 {
3 unsigned char myAddress = 1; // My address is 1
4 bool hasToken = true; // When starting I have the token. Only one node
      should have this flag set. When connecting to an already running
      DTMF network this should be set to false.
5
6 DtmfApi *api = new DtmfApi(myAddress, hasToken);
7 // DTMF API is now running, but no ports are enabled, so it can't
      receive any data.
8 (...)
```

Listing D.1: Creating instance example

D.2 Adding a port callback example

(This example assumes that an instance of API already has been created. See example D.1)

D.3 Sending data

(This example assumes that an instance of API already has been created. See example D.1)
The library support two ways of sending a message. Each with different advantages. First method - the fast way:

Second method - building a custom message:

D.4 Deleting instance

```
// Declare callback class
class MyPort1 : public DtmfCallback
{
    virtual void callbackMethod(DtmfInMessage * message)
    {
    // This function is called everytime data arrives. The incoming
        message only exist inside this method, so it's recommended to
        copy all needed information before exiting the function.
    // Fetch data from DTMF class to local memory (Assuming we have a
        dataContainer available in the MyPort1 class)
    m.getData(this->dataContainer*, 0, m.getMessageLength());

     // Data is now stored locally, return.
    return;
    }
}
// Make an instance of the class
MyPort1 * myPort1 = new MyPort1();
// Add the object as callback for port 1.
api->servicePort('1',myPort1);
```

Listing D.2: Adding a port callback example

```
unsigned char rcvAddress = '2';
unsigned char rcvPort = '1';
unsigned char * helloWorld = (unsigned char*)"HelloWorld";
api->sendMessage(rcvAddress, rcvPort, helloWorld, std::strlen(
    helloWorld));
```

Listing D.3: Sending data example 1

```
unsigned char rcvAddress = '2';
unsigned char rcvPort = '1';
unsigned char * hello = (unsigned char*)"Hello";
unsigned char * world = (unsigned char*)"World";
DtmfOutMessage message = api->newMessage(); // Create an empty message
message.setData(hello, 0, std::strlen(hello)); // Add hello to message
message.setData(world, std::strlen(hello), std::strlen(world)); //
    Append the world to message
message.send();
```

Listing D.4: Sending data example 2

```
delete(api); // DTMF API is now shutting down. This operation may take
    a while.
(...)
```

Listing D.5: Deleting instance example