Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTOL DEL TFC: Integració d'un modul d'audio en software de transmissió de video d'alta qualitat / Audio module integration into high quality video transmission software

TITULACIÓ: Enginyeria Tècnica de Telecomunicació, especialitat Sistemes de Telecomunicació

AUTOR: Gerard Castillo Lasheras

DIRECTOR: Jesús Alcober Segura

DATA: 6 de maig de 2011

Títol: Integració d'un modul d'audio en software de transmissió de video d'alta qualitat

Autor: Gerard Castillo Lasheras

Director: Jesús Alcober Segura

Data: 6 de maig de 2011

Resum

A continuació es presenta el projecte de fi de carrera "Audio module integration into high quality video transmission software".

Aquest projecte consta de la implementació d'un mòdul d'audio dins del sistema de transmissió de video d'alta qualitat, del software anomenat UltraGrid.

Per a aconseguir aquest objectiu es realitza l'estudi de quina arquitectura de so, dins de les disponibles conegudes específiques per al núcli Linux, és la més adequada. A més a més, de l'estudi de com es generen les mostres de so des de tots els dispositius capturadors que intervenen, a fi que aquest sistema sigui compatible amb tots els dispositius que s'usen als escenaris que es mostraràn. A més a més del protocol i encapçulament d'aquestes dades per tal d'assegurar-ne la bona adaptació dins els sistema i fer-lo el màxim compatible per a diferents dispositius, formats, mostrejos i altres treballs futurs.

Aquesta arquitectura de so escollida ha de permetre la integració del sistema de videoconferència UltraGrid amb les tarjetes de so integrades a l'ordinador. Així com permetre, el màxim possible, la captura i reproducció a temps real, així com sincronitzada amb el flux de video generat en paralel. Per tant, s'ha d'assegurar que es disposa d'una sèria de drivers i llibreries per a treballar-hi.

Es comprobarà que es podrà transmetre so mostrejat a qualitats màximes, les disponibles en funció de la pròpia tarjeta de so. Fins a generar fluxes específics com vocoders els quals permeten certa compressió, per tal de només transmetre veu, si es requereix.

Finalments es comprovarà el correcte funcionament del projecte i com es prova en casos reals.

Title: Audio module integration into high quality video transmission software

Author: Gerard Castillo Lasheras

Director: Jesús Alcober Segura

Date: 6 de maig de 2011

Overview

In this document, the bachelor thesis called “Audio module integration into high quality video transmission software” is presented.

This project consists on the implementation of an audio module into the high quality video transmission system, called UltraGrid.

To reach the objectives there is realized the study of which sound architechture, from all availabe known specific for Linux kernel, is the more appropiate. Also, the study of how are the audio samples generated from all involved capture devices, to do the whole system compatible with all the used devices into the shown scenarios. Furthermor, there is studyed the protocol and encpasulatin method of the audio data generated to ensure a good adaptation into the system and making it compatible for different devices, formats, sample rates and other future works.

This selected architechture has to permit the videoconference system UltraGrid with all computer sound card devices. And, as maximum as possible, the capture and playout at real-time, being synchronized with the video stream parallel generated. So, there has to be ensured that there are drivers and libraries to work with.

There will be demonstrated that it could be transmitted at highest qualities permitted by the maximum offered by the sound card. Also, there will be generated specific streams, like vocoders, that will permit some compression to transmitt only voice, if it is required.

Finally this is checked the proper project functioning and how it is prooved in real cases of use.

ACKNOWLEDGEMENTS

Als meu pare i la meva mare, els meus germans Sara i Eduard, a Laura Carreras i als amics que m'han donat suport fins al final.

A Francisco Iglesias i Pedro Lorente per els seus cops de mà i correccions durant el projecte.

A la resta de treballadors d'i2cat que m'han donat suport en algún moment del projecte, així com per consells o proves i testejos.

A David Cuenca i Davide Vega per a donar-me un cop de mà i consells, quan vaig començar amb el projecte.

I, finalment, a totes aquelles persones que han fet possible que aquest projecte es fes realitat, d'una forma o altre.

Gràcies a tots.

ÍNDEX

SUMMARY OF FIGURES

INTRODUCTION

As the main title explains, this project is involved in the integration of an audio module into high quality video transmission software.

The basic software is called UltraGrid [1], known as the first high quality transmission software for real time and uncompressed data.

The objective of this project is to improve the software involved, making it a complete multimedia software capable of working at the highest quality available depending on the context requirements, for example transmitting a real time opera from Liceu [2] to another theatre or using it for a videoconference, having both, video and audio, but always being able to choose one or both of them.

Here appear the project difficulties to solve, such as working at high audio sample rates or which audio format suits better for any virtual conference, among others and how to configure the whole audio module parameters. All these problems need to achieve minimum quality or to ensure the high quality possible of the equipment based, because of this is not the same to do the transmission of a violinist from a theatre than a familiar videoconference. So, a study of what quality, delay or bit rate suits better for every case will be required to solve the main problem of the project, always trying to reach the real time specifications and also being compatible with all hardware/devices used in the scenario.

In high quality video/audio transmission appears a different concept that is not doing only a face-to-face conference, like using Skype [3] . It is doing a room-to-room conference, which is the main goal of UltraGrid. Those scenarios seem so similar but have notable differences. For example, doing a face-to-face conference there is no need to reach high quality in video or audio stream, while the speakers could understand what they are watching and listening to, there is no need to improve for more quality. But, if a room-to-room is being done, as normal, the maximum quality available is required.

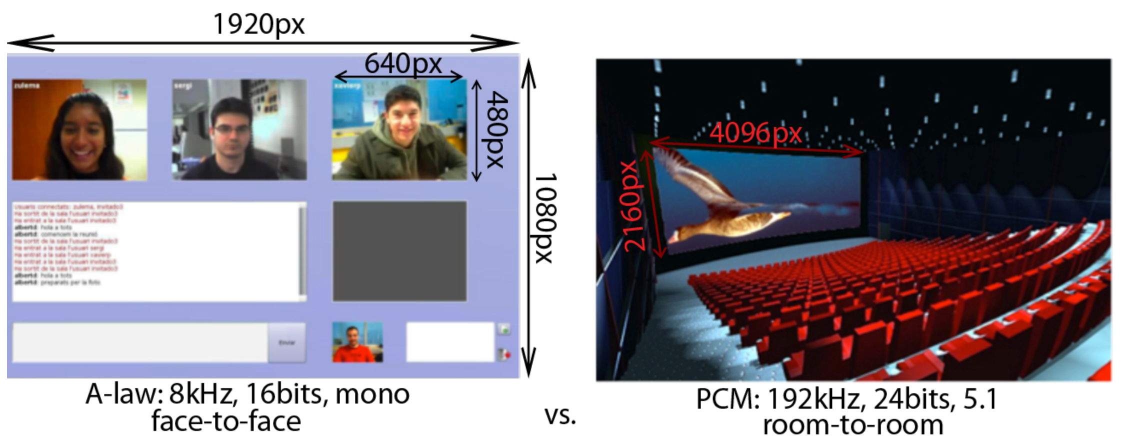


Fig. 7.1 Difference between face-to-face versus room-to-room

The example shown above explains graphically the differences between face-to-face and room-to-room. For a face-to-face there is no need to transmit video at high quality (FullHD [4]) and with a simple vocoder [5] the human voice is quite fine understood. While transmitting a room-to-room media it requires as much quality as possible, like transmitting 4K (4 times FullHD resolution) of video resolution and an audio quality like Blu-Ray.

This bachelor thesis is organized in four main parts:

1. Introduction.
2. Review of the state of the art and description of the technical approach deployed in this project.
3. Proposed solution and audio module generated.
4. Test, validation, conclusions and future works.

The introduction section is focussed on describe the problem description and this short explanation of the organization of the project.

During the second part, a short concepts list related to this project are described, concretely the European parent project, the basic concepts, the software involved and the test scenario that is going to be used. Then the technical approach deployed in this project will be described and will be helpful in how to implement the required audio module, understanding the problem to solve and its difficulties. This is how audio is digitized in capture side, treated, transmitted, then received, processed and reproduced.

In the third section there is the proposed solution using the most generic option to understand how audio module can be integrated into the other technologies available and trying to be as much as compatible as it can be, showing the problems that appears when trying to stream synchronous audio with video and at real-time, and trying to solve more problems that will appear, like all projects in development.

In the last part will be described three real scenarios where the project has been tested and validated. Also the environmental impact, the conclusions of this project, and then new research lines.

CHAPTER 1. REVIEW OF THE STATE OF THE ART

This chapter is focused in the main concepts involved. Trying to give the necessary information to understand the objective and purposes of the project. Explaining the parent project, the concept of videoconferencing, high definition, and digital sound and its treatment to process and transmit.

1.1 V3 Project

i2cat Foundation [6] has a research project called V3 [7] , which goal is to develop a experimental super high-quality system platform, regardless the display resolution, video and videoconference over advanced Internet networks. Nowadays there is not any platform of video of 4K resolutions for the user, and any platform of videoconference of high definition without compression implemented with free code or open source.

1.2 Videoconference

A videoconference is enabled by a system that is able to transmit and receive simultaneous and synchronous video and audio in real time between points that are far from each other, through the network.

The videoconference systems and, in general, interactive systems, and have strict requirements on time. One of the most important is the delay, which could break the interactivity between users. If we want an interactive conversation, we should have a delay of less than 150 ms (or 300 ms of RTT).

The goal of these systems is to enable an interactive communication between each other, like people do it in a face-to-face way.

1.3 HDTV

When we talk about High Definition Television [8] , we have the goal to have a better image than traditional analogue television. The quality and the clarity of an image depend on the resolution. High definition signal is digital and its aspect ratio is 16:9 rather than 4:3 in traditional television that offers a wide display of pictures.

1.3.1 HD resolution

High Definition standard describes four different models depending on the resolution of each image and the number of frames per second (fps). These are the 1280x720 HDTV (Standard 720p) or HD Ready, the 1920x1080 HDTV (Standards 1080p and 1080i) or Full HD and the 3840x2160 SHD. Their progression are quadratic, in a way that the format SHD (Super High Definition) and its equivalent in U.S 4k (2048x1080), represents four times the format Full HD or its equivalent 2k (2048x1080), that at the same time represents almost 4 times the size of the traditional television PAL or NTSC.

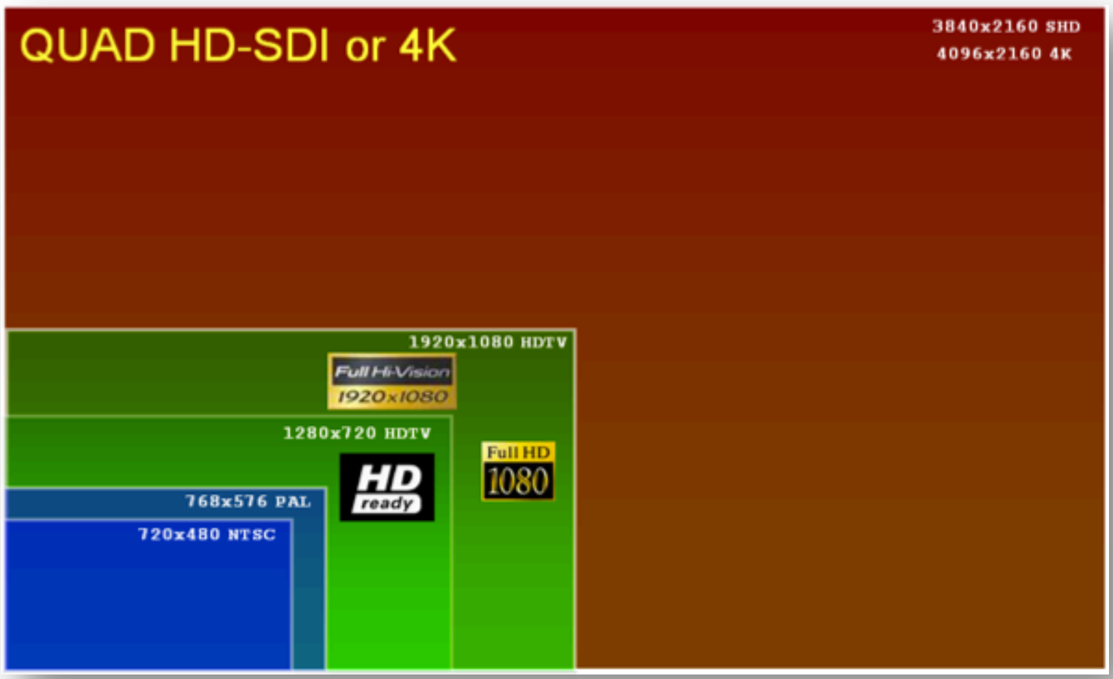


Fig. 1.1 Display resolutions

1.3.2 HD frame rate and scan

Another important thing in video is the frame rate; the responsible of the receiver brain perceives motion with static images. Frame rate is the number of images displayed per second. Experimental studies prove that our visual perception starts when viewing a movement displayed around 20 fps or more.
In video there are two ways of displaying images:

Progressive scanning (p): where each scan displays every line in the image raster sequentially from top to bottom. Possible frame rate are 23.98 / 24 / 25 / 29.97 / 30 / 60 depending of the zone.

Interlaced scanning (i): where each scan displays alternate lines (even or odd per field) in the image raster, and two complete scans are therefore required to display the entire image. Each scan is called field. Possible field rate are 50 / 59,94 / 60.

1.3.3 Colour encoding

A colour encoding is necessary for processing the image and then beeing able to reproduce in other equipment. There are some theories about colour encoding, which we only will explain a few aspects.

Usually RGB (Red, Green, and Blue) [9] encoding is used in electronic systems. This system assigns an equal weight for the primary colours (red, green and blue). With this encoding, it is possible to represent almost all visual colours in a black background.

Another colour space is YUV. YUV [11] is a way of encoding RGB [10] that offers some advantages. Firstly was thought for black and white compatibility, because YUV differs between luma component (Y), the brightness (1.1), and the chrominance (U and V), the colour. U is the difference between blue and luma (1.2). V is the difference between red and luma (1.3).

$$Y = 0{,}299\ R + 0{,}587\ G + 0{,}114\ B \qquad (1.1)$$
$$U = -0{,}147\ R - 0{,}289\ G + 0{,}436 \qquad (1.2)$$
$$B = 0{,}492\ (B - Y)\ V = 0{,}614\ R - 0{,}515\ G - 0{,}100\ B = 0{,}877\ (R - Y) \qquad (1.3)$$

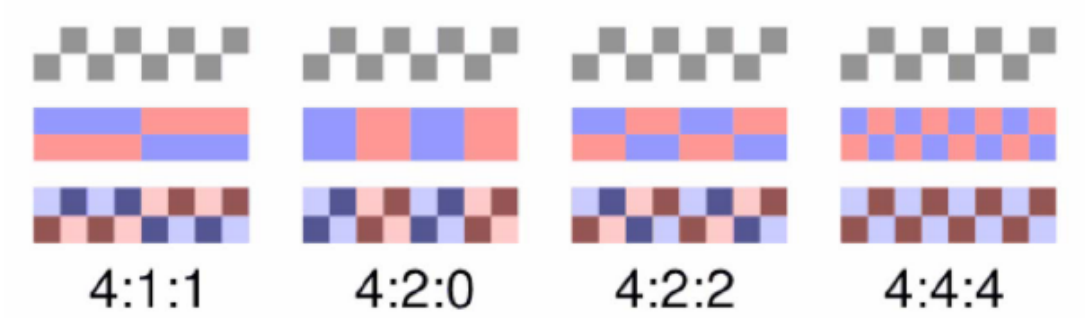


Fig. 1.2 Colour encoding possibilities

Since the human visual system is much more sensitive to variations in brightness than colour, a video system can be optimized by devoting more bandwidth to luma than to the colour difference components. In that sense, appear the 4:2:2 schemes that require two-thirds of the RGB bandwidth. This reduction results in almost no visual difference as perceived by the viewer. 4:2:2 are sometimes used in High Definition video.

1.3.4 HD bandwidth

The minimum bandwidth needed to assure that all the information can be transmitted at the same speed that is generated with propose of assuring that there will not be any kind of delay (no information stored). It can be calculated using the following formula:

$$BW = \text{height} \times \text{width} \times \text{bits/pixel} \times \text{fps} \times (Y+U+V)/4$$

1.4 Digital Audio

First of all, to get introduced into the digital audio concepts, a short outline of history is required to understand the context. Then, the basic concepts are exposed.

1.4.1 History

In the electrical communications history, the first reason for sampling a signal was to interlace samples from different telegraphy sources, and convey them over a single telegraph cable. Telegraph time-division multiplexing (TDM) was transmitted in an early age, in 1853 by Moses G. Farmer, an American inventor.

In 1903, the electrical engineer W. M. Minerused used an electro-mechanical commutator for time-division multiplex of multiple telegraph signals, and also applied this technology to telephony. He obtained intelligible speech from channels sampled at a higher rate of 3500–4300 Hz; below this was unsatisfactory. This was TDM, but used with pulse-amplitude modulation (PAM) instead of PCM.

In 1926, Paul M. Rainey of Western Electric patented a facsimile machine, which transmitted its signal with 5-bit PCM, encoded by an opto-mechanical analog-to-digital converter. The machine wasn’t produced.

British engineer Alec Reeves, unconscious of the previous tasks, conceived the use of PCM for voice communication in 1937 while he was working for International Telephone and Telegraph in France. He explained the theory and its advantages, but no practical use worked. Reeves filed for a French patent in 1938, and his U.S. patent was awarded in 1943.

The first transmission of speech by digital techniques was the SIGSALY vocoder encryption equipment used for high-level Allied communications during World War II from 1943. At this year, the Bell Labs researchers who designed the SIGSALY system became conscious of the use of PCM binary coding as already proposed Alec Reeves. In 1949, Ferranti Canada built a working PCM radio system for the Canadian Navy's DATAR system, which was able to transmit digitized radar data over long distances.

PCM, in the late 1940s and early 1950s, also used a cathode-ray coding tube with a plate electrode with encoding perforations. In the same way as using an oscilloscope, the beam was swept horizontally at the sample rate while the vertical deflection was controlled by the input analogue signal, causing the beam to pass through higher or lower parts of the perforated plate.

The plate collected or passed the beam, causing current variations in binary code, one bit at a time. Rather than natural binary, the grid of Goodall's later tube was perforated to produce a glitch-free Gray code, and produced all bits simultaneously by using a fan beam instead of a scanning beam.

The National Inventors Hall of Fame has honoured Bernard M. Oliver and Claude Shannon as the inventors of PCM, as described in 'Communication System Employing Pulse Code Modulation', U.S. Patent 2,801,281 (http://www.google.com/patents?vid=2801281) presented in 1946 and 1952 and awarded in 1956. Another patent by the same title was filed by John R. Pierce in 1945, and issued in 1948: U.S. Patent 2,437,707 (http://www.google.com/patents?vid=2437707). All of them published "The Philosophy of PCM" in 1948. [14]

Furthermore, pulse-code modulation (PCM) was used in Japan by Denon in 1972 for the mastering and production of analogue phonograph records, using a 2-inch quadruplex-format videotape recorder for its transport, but this was not used as a consumer product.

1.4.2 PCM/RAW, DPCM/DM/ADPCM/SND Formats & Speech companding

Pulse Code Modulation (PCM), developed in 1939, is a standard method for digitizing analogue audio signals. This format is an uncompressed, raw bit stream, linear (transmitted in a linear series meaning that the stream of the signal is sequential rather than random, the amplitude of both the of the input stream and output stream remain at a fixed ratio and a sinusoidal wave input source will result in a sinusoidal wave output signal at the same frequency), signed two's complement, fixed point encoded data file.

A PCM encoder (ADC) may sample analogue sound from 8,000 times per second (8 KHz) and use 8-bits to represent each sample. It is usually used at 44.1 KHz and 16-bit resolution to match CD Audio standards, and can encode at 96 KHz and 24-bit approximately, it will also depend on the device. The procedure uses only two alternating pulse values (1 and 0) duplicating binary code.

This codec makes a raw (RAW) data file (no header or footer information describing sample rate, sample resolution, monaural or stereo) and provides us with only 256 possible amplitude values (based on 8-bit binary numbering). When the codec is set to sample at the Audio CD level at 44.1 KHZ sample rate

(44,100 samples per second), with 16-bit resolution per sample (65,536 possible amplitude levels), this results in a file that requires 1,411 Kbps (kilobits per second, or 1.4MB) bit rate for the representation/playback of one second of stereo music. This result is obtained with the following formula:

Throughput (Bytes/s) = (sampling rate) x (bit depth) x (number of channels) / 8

DPCM, Differential Pulse Code Modulation, is a PCM codec that limits the amplitude difference between samples in order to improve data transmission efficiency. DPCM accomplishes this by predicting the next value after a sample and then encodes the difference between the predicted value and the actual value. SND can sometimes be a header less format. PCM digital sampling requires very active and accurate cut-off filters ("brick wall filter") in order to block a signal in excess of half the sampling frequency (Nyquist theorem [15]).

Delta Modulation is related to DPCM. However, DM uses a low quantization rate to indicate the slope of the amplitude of the signal (direction) so the prediction by DPCM will be more accurate.

1.4.2.1 Speech Companding

Logarithmic quantization of analog signal, as opposed to linear encoding, is also known as companding [16].

Companding is simply a system in which information is first compressed, transmitted through a bandwidth-limited channel, and expanded at the receiving end. It is frequently used to reduce the bandwidth requirements for transmitting telephone quality speech, by reducing the 13-bit codewords to 8-bit codewords. Two international standards for encoding signal data to 8-bit codes are A-law and µ-law. A- law is the accepted European standard, while µ-law is the accepted standard in the United States and Japan. This system is described by international standard G.711 [17].

Where circuit costs are high and loss of voice quality is acceptable, it sometimes makes sense to compress the voice signal even further. An ADPCM algorithm is used to map a series of 8-bit µ-law or A-law PCM samples into a series of 4-bit ADPCM samples. In this way, the capacity of the line is doubled. The technique is detailed in the G.726 standard [18].

The human auditory system is believed to be a logarithmic process in which high amplitude sounds do not require the same resolution as low amplitude sounds. The human ear is more sensitive to quantization noise in small signals than large signals. A-law and µ-law coding apply a logarithmic quantization function to adjust the data resolution in proportion to the level of the input signal. Smaller signals are represented with greater precision – more data bits – than larger signals. The result is fewer bits per sample to maintain an audible signal-to-noise ratio (SNR).

Rather than taking the logarithm of the linear input data directly, which can be computationally difficult, A- law/µ-law PCM matches the logarithmic curve with a piece-wise linear approximation. Eight straight-line segments along the curve produce a close approximation to the logarithm function. Each segment is known as a chord. Within each chord, the piece-wise linear approximation is divided into equally size quantization intervals called steps. The step size between adjacent codewords is doubled in each succeeding chord. Also encoded is the sign bit for the original integer. The result is an 8-bit logarithmic code composed of a 1-bit sign bit, a 3-bit chord, and a 4-bit step.

The sound file is also characterized by how many channels (mono or stereo) there are. Usually all signals eventually are combined into a single bit stream and are interleaved with each other (placed as follows in stereo mode: …LRLRLR..., where L is the left channel and R is de right channel).

For more information about speech companding refered to A- law/µ-law see ANNEX A.

PCM streams have two basic properties that determine their fidelity to the original analog signal: the sampling rate, which is the number of times per second that samples are taken; and the bit depth, which determines the number of possible digital values that each sample can take.

1.4.3 Sample rate and bits per sample

To enable computers and hard disk recorders to capture sounds, those sounds must be digitized. Digitized audio consists on a series of individual fixed frames that are interpreted by our brains as a continuous signal, many "snapshots" that we hear as continuous sound.

1.4.3.1 Sample rate

The frequency in which these audio snapshots are taken is referred to as the sample rate. The more snapshots are taken, the more detailed is the sound.
These pictures illustrate how sampling works:

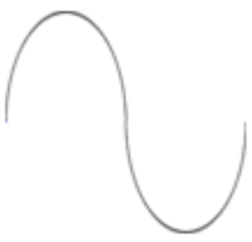


Fig. 1.5 Sine wave

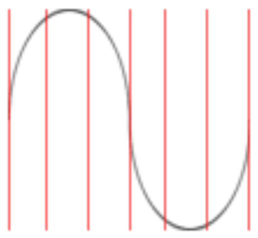


Fig. 1.3 Low sample rate

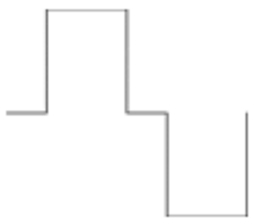


Fig. 1.4 Resulting waveform

In Fig. 1.5 is showed a sine wave with a frequency of one cycle per second. Fig. 1.3 represents the sample rate; in this case six samples per second, or 6Hz.

Each red line represents a snapshot of the sine wave at that single moment in time. Combining the snapshots appears Fig. 1.4.
There's not much resemblance to a sine wave. In fact, increasing the frequency of these waveforms to an audible pitch, the tonal difference would be amazing; the sine wave would be soft and dull, while Fig. 1.4 waveform would sound bright and piercing.
To get a better representation of a sine wave, appears the need to increase the sample rate. Changing the rate from 6Hz to 10Hz, is noticed a much better result:

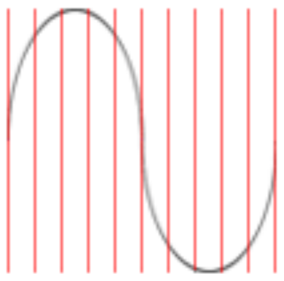


Fig. 1.6 Higher sample rate

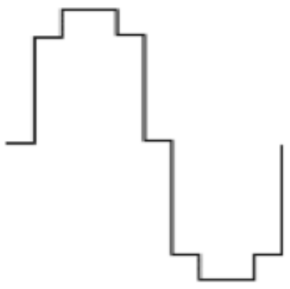


Fig. 1.7 Resulting waveform

Obviously, the more samples per second, the more the resulting waveform resembles the original.
In the world of digital audio recording, the most common sample rates are 44.1kHz and 48kHz; both significantly faster than the examples above.
On the golden edge of digital recording, sample rates of 96kHz-192kHz are beginning to appear. The higher the sample rate is, the greater the bandwidth or frequency responses.
In addition to the higher cost of fast converters, it is also important to consider the high cost of memory and disk space. A sound sampled at a rate of 192kHz will need four times more disk space as that same sound sampled at 48kHz.

1.4.3.2 Bit depth (bits per sample)

The other part of the digital audio equation is bit depth. Much like the sample rate defines the frequency response as it divides up the horizontal axis of the waveform, the bit depth defines the dynamic range of the sound as it describes the amplitude of the waveform at each sample point.
8-bit audio provides 256 separate levels for each sample. 16-bit audio provides up to 65,536 levels, and 24-bit samples gives 16.7 million different levels.

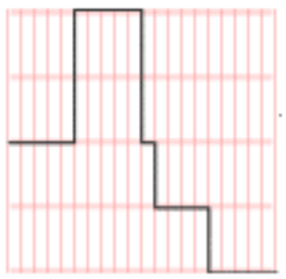


Fig. 1.9 Low bit depth

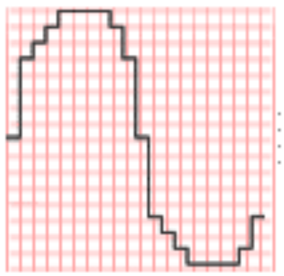


Fig. 1.8 Higher bit depth

Fig. 1.9 has twice the sample rate described in Fig. 1.6. This is because the low bit depth offers very little in terms of dynamic resolution. The higher bit depth shown in Fig. 1.9 results in significantly improved waveform resolution.
Higher sampling rates allow a digital recording to accurately record higher frequencies of sound. The sampling rate should be at least twice the highest frequency you want to represent (as the Nyquist theorem demonstrates).

Humans can't hear frequencies above about 20,000 Hz, so 44,100 Hz was chosen as the rates for audio CDs to just include all human frequencies. Sample rates of 96 and 192 KHz are starting to become more common, particularly in DVD-Audio, but many people honestly can't hear the difference.
Nowadays, in digital audio the most common sampling rates are 44.1 kHz, 48 kHz, and 96 kHz. A more complete list is shown in ANNEX B.

Since now we know how audio can be digitally generated and processed, but not how it should be transmitted into network, like internet.

1.5 Streaming: audio and video conference

To stream video and audio data is mandatory the use of an Internet protocol that suits the real time specifications. The best-known protocol is RTP [19].

The Real-time Transport Protocol (RTP) defines a standardized packet format for delivering audio and video over IP networks. RTP is used extensively in communication and entertainment systems that involve streaming media, such as telephony, video teleconference applications and web-based push-to-talk features.

RTP is used in conjunction with the RTP Control Protocol (RTCP). While RTP carries the media streams (e.g., audio and video), RTCP is used to monitor transmission statistics and quality of service (QoS) and aids synchronization of multiple streams. When both protocols are used in conjunction, RTP is originated and received on even port numbers and the associated RTCP communication uses the next higher odd port number.

It is one of the technical foundations of Voice over IP and in this context is often used in conjunction with a signalling protocol that assists in setting up connections across the network. RTP was developed by the Audio-Video Transport Working Group of the Internet Engineering Task Force (IETF) and first published in 1996 as RFC 1889, superseded by RFC 3550 in 2003.

If both audio and video media are used in a conference, they are transmitted as separate RTP sessions. RTP and RTCP packets are transmitted for each medium using two different UDP port pairs and/or multicast addresses. There is no direct coupling at the RTP level between the audio and video sessions, except that a user participating in both sessions should use the same distinguished (canonical) name in the RTCP packets for both so that the sessions can be associated.

One motivation for this separation is to allow some participants in the conference to receive only one medium if they choose. Despite the separation, synchronized playback of a source's audio and video can be achieved using timing information carried in the RTCP packets for both sessions.

The RTP header definition, as the RFC 3550 defines, per packet is:

Table 1.1 RTP header

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
V		P	X	CC				M	PT							Sequence Number															
Timestamp																															
SSRC																															
CSRC [0..15] :::																															

Where:

V: Version. Defines the RFC version of the RTP packet header (usually V=2).
P: Padding. If set, this packet contains one or more additional padding bytes at the end which are not part of the payload.
X: eXtension. If set, the fixed header is followed by exactly one header extension.
CC: CSRC Count. The number of CSRC identifiers that follow the fixed header.
M: Marker. The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream.
PT: Payload Type. Identifies the format of the RTP payload and determines its interpretation by the application. A profile specifies a default static mapping of payload type codes to payload formats. Additional payload type codes may be defined dynamically through non-RTP means. An RTP sender emits a single RTP payload type at any given time; this field is not intended for multiplexing separate media streams. See ANNEX C for a complete list of payload types.

Sequence Number: The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence.

Timestamp: The timestamp reflects the sampling instant of the first octet in the RTP data packet. As an example, for fixed-rate audio the timestamp clock would likely increment by one for each sampling period. If an audio application reads blocks covering 160 sampling periods from the input device, the timestamp would be increased by 160 for each such block, regardless of whether the block is transmitted in a packet or dropped as silent.

SSRC, Synchronization source: Identifies the synchronization source.

CSRC, Contributing source: An array of 0 to 15 CSRC elements identifying the contributing sources for the payload contained in this packet. For example, for audio packets the SSRC identifiers of all sources that were mixed together to create a packet are listed, allowing correct talker indication at the receiver.

Based on RFC 3550 recommendation, the packet header is not extended and additional payload header is defined.

For video transmission, UltraGrid implements RFC 4175 [20]; RTP Payload Format for Uncompressed Video definition, that is identified with payload type 96 into the RTP header when transmitting video data in UltraGrid. For example, if two lines of video are encapsulated, the payload format will be as shown in Fig. 1.10

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| V |P|X|   CC  |M|    PT       |       Sequence Number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Time Stamp                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             SSRC                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Extended Sequence Number    |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|F|          Line No            |C|           Offset            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |F|          Line No            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|C|           Offset            |                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               .
.                                                               .
.                 Two (partial) lines of video data             .
.                                                               .
+---------------------------------------------------------------+
```

Fig. 1.10 RTP payload format showing two (partial) lines of video

This RFC specifies a packetization scheme for encapsulating uncompressed video into a payload format for the Real-time Transport Protocol, RTP. It supports a range of standard and high-definition video formats, including common television formats such as ITU BT.601, and standards from the Society of Motion Picture and Television Engineers (SMPTE), such as SMPTE 274M and SMPTE 296M. The format is designed to be applicable and extensible to new video formats as they are developed.

When transmitting audio streams, the payload type is defined with number 97. Also, as a dynamic payload type (like 96), it permits a non-official and dynamic payload header definition.

But, in that case, for audio data is not used any type of standard, or RFC, to define the payload header. The payload format is as shown in next table:

Table 1.2 RTP dynamic payload header for audio data

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
V		P	X	CC				M	PT							Sequence Number															
Timestamp																															
SSRC																															
CSRC [0..15] :::																															
Position																Lenght (data's)															
Start Sample																Buffer Size															
Start Channel																Padding															

The reason of using that kind of payload format is to be as adapted as possible into standards that takes the definition of embedded audio data, and its structure, into HD-SDI video standards.

The HD serial interfaces provide for 16 channels of embedded audio. The interface uses the SMPTE 299M standard. In that case, an SDI signal may contain up to sixteen audio channels (8 pairs) embedded at 48 kHz, 24-bit audio channels along with the video. Typically, 48 kHz, 24-bit PCM audio is stored, in a manner directly compatible with the AES3 digital audio interface.

Next figure shows how the SMPTE HDTV standards combine:

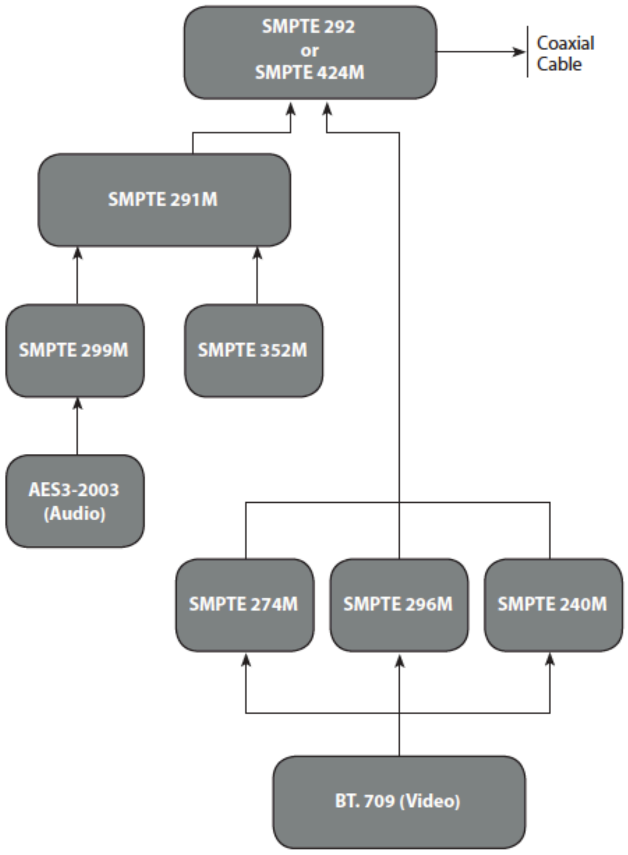


Fig. 1.11 SMPTE HD-TV Standards combination

1.6 Operating System (O.S.)

The entire environment is based on Linux [30] and some previous aspects must be known in advance, helping to understand how is treated audio in the O.S.

The global framework in Linux, that is not transparent to the user, is similar to the others UNIX Systems. Almost all the components of the system can be seen as file shape. This paradigm is applied to all the system, from the interaction with the hard system to the kernel tuning.

Linux Kernel [31] is the core of Linux system. Technically, Linux is the kernel, and the rest of the system is a set of applications that has not relation with the kernel. Because of it, it is also called GNU/Linux. The kernel offers an interface for the hardware and it is the responsible of the control of all processes that are running on the machine. In the kernel are loaded all the drivers of the system, the network stack, and parameters related to the processor, memory, buses, etc.

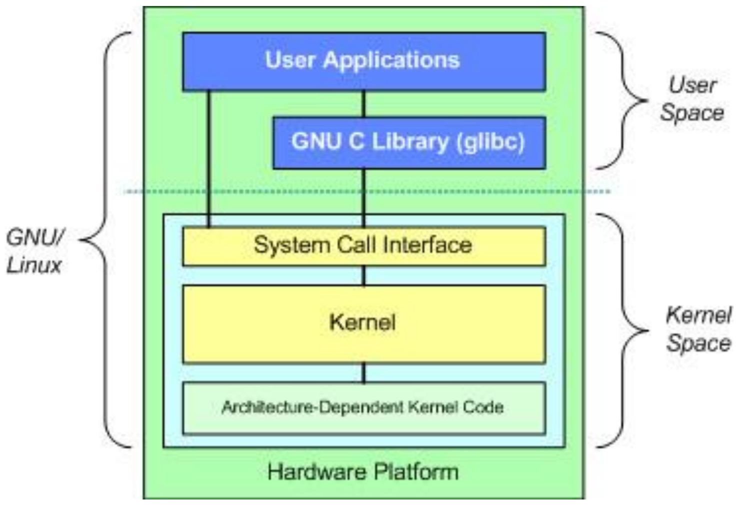


Fig. 1.12 The fundamental architecture of the GNU\Linux operating system

At the top there is the user, or application, layer. This is where the user applications are executed. Below the user space there is the kernel space. Here, the Linux kernel works.

There is also the GNU C Library (glibc). This provides the system call interface that connects to the kernel and provides the mechanism to transition between the user-space application and the kernel. This is important because the kernel and user application occupy different protected address spaces. And while each user-space process occupies its own virtual address space, the kernel occupies a single address space.

Kernel features and drivers can stay on the kernel as built-in or as a module. When features are built-in always are loaded in the kernel and consume memory and increase the size of the kernel. Otherwise if features and drivers are loaded as modules they should be load a runtime for doing its purpose. Both options are good because the first one gives speed and robustness and the second one give more flexibility to the system because drivers loaded only when are needed or used.

Since GNU/Linux exists there have been released so many distributions, UNIX-like operating systems built on top of the Linux kernel, of which can be distinguished between commercially-backed distributions, such as Fedora (Red Hat), openSUSE (Novell), Ubuntu (Canonical Ltd.), and Mandriva Linux (Mandriva), and entirely community-driven distributions, such as Debian and Gentoo.

Ubuntu is the current operating system used in this project, but some modifications from the original fresh installation have to be done if we want a complete environment working with audio, UltraGrid and all devices needed, as explained in ANNEX D that has been developed while working at i2cat.

CHAPTER 2. DESCRIPTION OF THE TECHNICAL APPROACH

2.1 Objectives

The objectives of this chapter are to focalize the context of the technologies available to work to, as will be shown in the test scenario part. Also, to understand a little better the software UltraGrid for implementing the audio module, and how we will go from its initial state to the final state with the audio module working properly. All of these steps will require the explanation of how works the audio architecture in GNU/Linux too, and especially how will be configured in Ubuntu O.S.

Also, some examples, with basic problems of the project, will be exposed during this chapter to understanding better the whole objectives to reach.

2.2 The Test Scenario

This part is trying to explain, also graphically, the test scenario and how it will be developed to reach goals. First of all, is described the basic specifications of the professional audio/video capture cards related to the context scenario, to work with. After this, is shown the basic and initial scenario, only with one audio option that didn't work fine at this point, having to be solved. This option became from the XenaHS [21] library implemented in UltraGrid at the beginning, and depending of its professional audio/video capture card. Next scenario is focused in the implementation of different audio/video options, since having the option to choose from different audio sources to which format suits better into the conference context.

2.2.1 Sound capturers

2.2.1.1 XenaHS Device

This is the first capture device introduced in the project scenario with AES/EBU audio inputs and outputs; 15-pin D-connector one female 15-pin D-connector is provided for six channels of AES/EBU inputs and outputs. The connector attaches to a six-XLR breakout cable supplied with the XenaHS card module. XenaHS AES/EBU audio is 24 Bit/48Khz. All AES inputs support asynchronous audio at 32-96Khz.
In the project it is used capturing the audio from a Shure microphone into an audio mixer (also it could be required for mixing more audio sources) and digitalizing it into an AD converter, then, with two (stereo mode) of the 15-pin D-connector one female 15-pin D-connector, it is processed to XenaHS capture card.

2.2.1.2 DeckLink Blackmagic

This capture card was introduced later. The principal reason is that it has more input options for video sources, like HD-SDI and HDMI.

About the sound in DeckLink [22] Studio, it features a massive 4 channels of balanced professional analog audio in and out, as well as 2 channels of AES/EBU digital audio in and out, for broadcast quality and sample accurate AV sync. DeckLink Studio also includes a sample rate converter on the AES/EBU input. When working with multi channel audio, DeckLink Studio can switch it's analog audio output 3 and 4 to AES/EBU digital audio outputs. That means you get an extra 2 digital audio output for a total of 6 channels of AES/EBU digital audio out! DeckLink Studio is perfect for 5.1 surround sound editing and multi language digital audio monitoring. But for now, it is only implemented into UltraGrid for capturing audio embedded from HDMI or HD-SDI sources.

2.2.1.3 Intel HD Soundcard

To have the choice of where the audio comes from into the same computer appears the option for using the integrated (or any special computer sound card) computer sound card. The scenario works with the Intel HD Audio cards [23].

Intel HD Audio delivers significant improvements over previous generation integrated audio and sound cards. The Intel HD Audio specification delivers the features needed for an improved audio experience.

Intel HD Audio is capable of delivering the support and sound quality for up to eight channels at 192kHz/32-bit quality, while the previous devices specification can only support six channels at 48 kHz/20-bit. In addition, Intel HD Audio is architected to prevent the occasional glitches or pops that other audio solutions can have by providing dedicated system bandwidth for critical audio functions.

That specifications make Intel HD Audio a good option to reach the professional level of the other capture cards listed above, and making it possible to be compatible with other capturer cards format.

One motivation to use integrated sound cards like that is to not depend from the fabricant libraries available for implementing the audio module into UltraGrid, but also to have the security that with a generic sound card there will be more options to format the audio stream, like sample rate or sample format.

2.2.2 Scenarios

The initial scenario is only prepared to work to one network direction (one computer can only be transmitter or receiver) as shown in the following figure:

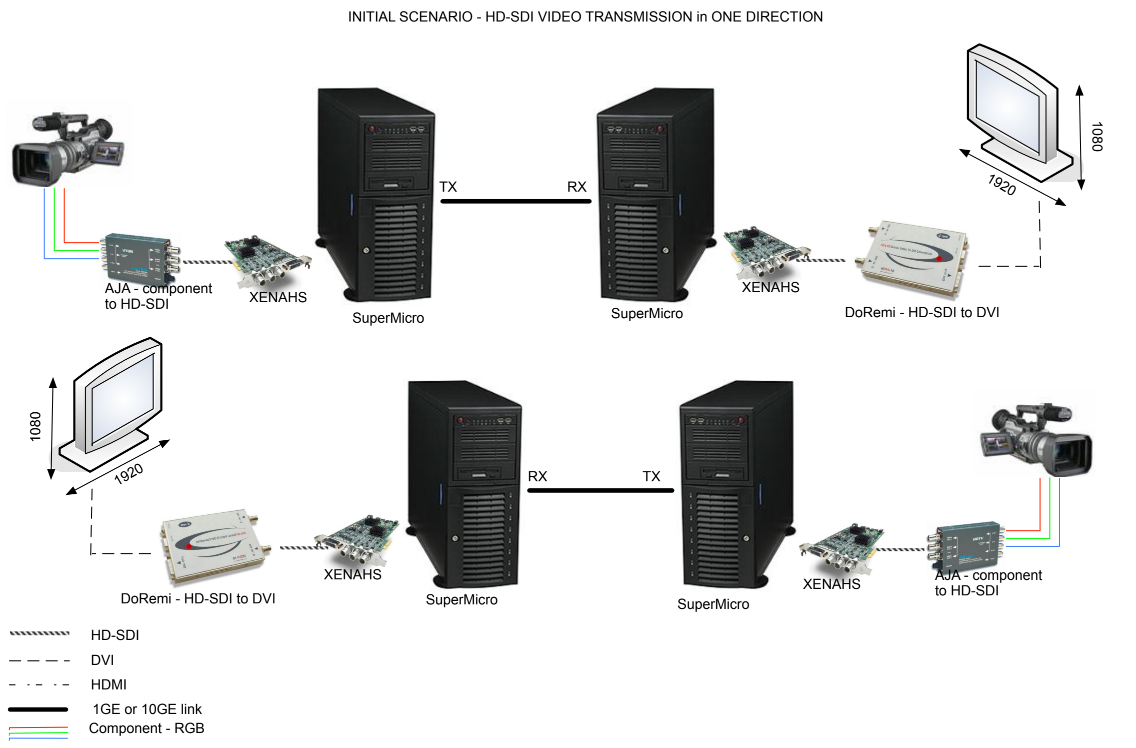


Fig. 2.1 Initial scenario

The initial implemented sound option was using the same professional capture card, XenaHS, and with its fabricant libraries. It worked at 48kHz, at 32 bits per sample (24bits per sample + 8 padding bits), and up to 6 channels.

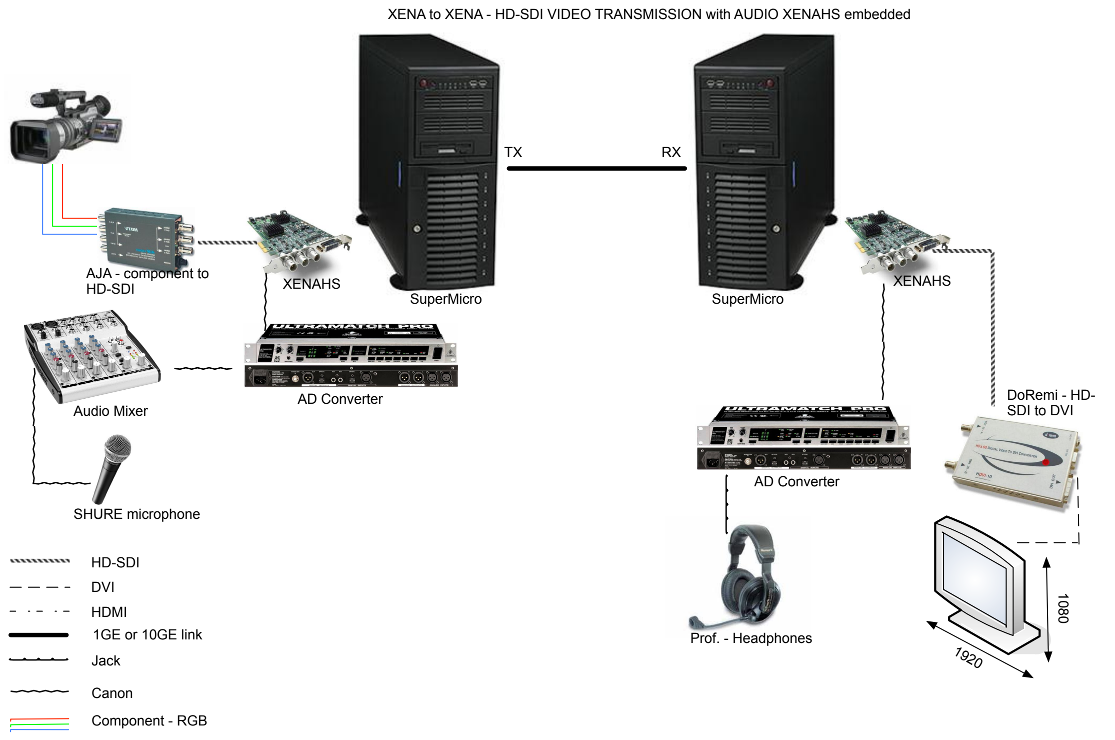


Fig. 2.2 Intermediate scenario with first audio option

The two scenarios shown need so many devices to work with XenaHS and only work in one network direction. Becoming a so expensive scenario. But having a quite list of options to transmit audio, from one channel input to six.
It is a good option to capture high definition data, for example from a theatre, and to transmit it to another theatre like cinema projection.

The scenario that is wanted to reach spans more options and can even be less expensive.

To reach the goal of having in a same computer the trasmission and reception appears the need to have faster CPUs and GPUs, and having to adapt the throughput for the network bandwith to reach a bidirectionall link (FullDuplex). This is done replacing the SuperMicros with an ASUS [24] motherboard that reaches 3GHz of sample clock and integrating it with an Nvidia [25] GPU. Also this motherboard integrates the Intel HD SoundCard.

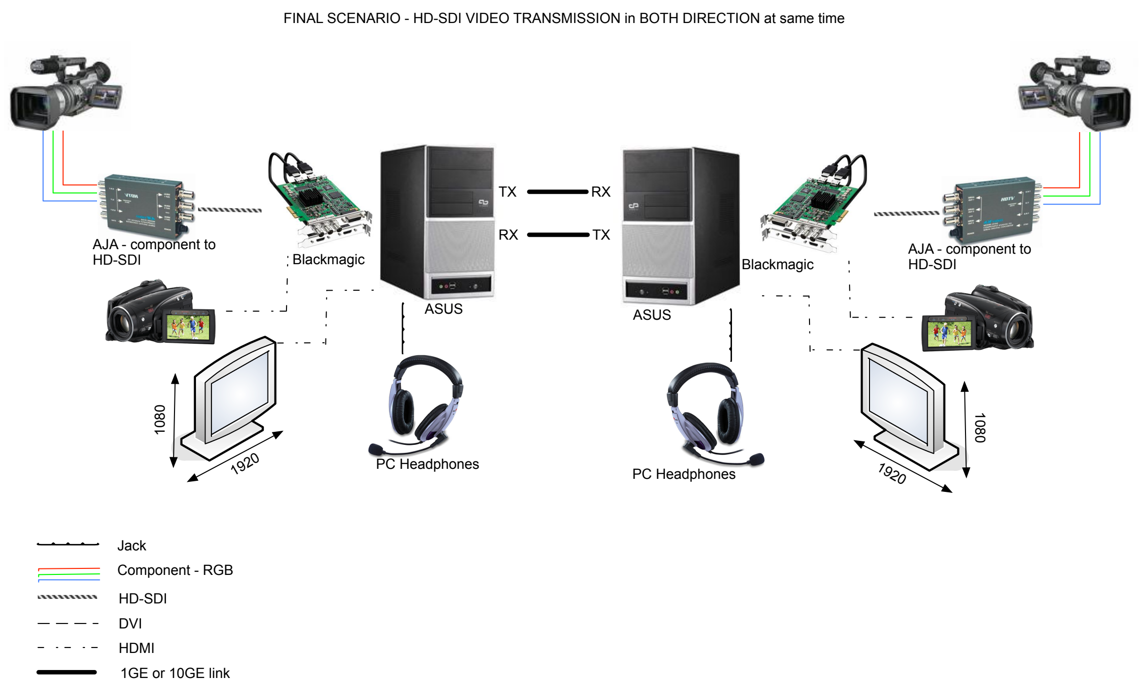


Fig. 2.3 Final scenario with Blackmagic card

This scenario can also have the option of chosing from audio embedded from the camera or from the headphones connected to the HD Intel Sound Card.

This final scenario doesn’t take care about the initial scenario, but, as it is raised in the project objectives, there will be the option to work with it as shown in next chapters.

Now, introducing how UltraGrid works will help to understand better how the audio module implementation will be possible.

2.3 UltraGrid

UltraGrid, the software involved in the whole project, is a high definition (HD) video conferencing and distribution system. It is also considered the first system capable of supporting uncompressed gigabit rate high definition video over IP. In fact, an UltraGrid node converts SMPTE 292M high-definition video signals into RTP/UDP/IP packets, which can be distributed across an IP network reaching transmission rates until 1.5 Gbps. This application is very useful for videoconference in high definition because only introduce transmission delay, doing null the encoding delay. In this way, we obtain a quality almost perfect and a resolution (like digital cinema) higher than present videoconference systems.

UltraGrid is a software Open Source initially developed by the ISI EAST [12] with the main goal to stress network launching a flow of only video of high quality (HD) without compressing, in order to provoke congestion. This software has double function capture / display and transmitter / receiver, correspondingly. Therefore, it takes video on real-time and packs it in IP datagram to be sent over network. In reception, it made opposite work, receiving, unpacking and retrieving HD-SDI signal for any device that can understand it. It can work like a tunnel SMPTE input and SMPTE output though the IP network.

This software is very modular and it is possible to add new features only adding the appropriate modules.

Until now and during the V3 project the next modules have already been developed:

- XenaHS module: this module works with the professional audio/video capture device.

- SAGE module: module that does an interface between UltraGrid and SAGE [28] for visualizing high definition streams on a title display.

- Module DV [26] (Digital Video): standard transmission and reception, with audio embedded. With this feature we can do a videoconference with a format more lightly for the network (around 30 Mbps [27]).

- Blackmagic DeckLink capture module, with audio embedded.

This project has the objective to add the module that gives the functionality to interact with the embedded audio captured, from the different devices described, and to generate a capture independent interface of audio data, directly from any audio card computer, such as a sound display that should be able to work with all the audio data received.

When having a video throughput of 1GB appears the motivation of optimize/reduce the data generated if audio stream is also generated:

2.3.1 Total generated bandwidth (case of use)

The bandwidth used in an UltraGrid transmission depends on the dimensions of the frame (height and width), the bits per pixel, the frames per second and the format of the pixel byte (UG uses YUV):

$$BW = \text{height} \times \text{width} \times \text{bits/pixel} \times \text{fps} \times (Y+U+V)/4$$

Transmitting a fullHD stream in NTSC mode with 8 bits per pixel and a 4:2:2 YUV format, the maximum rate achieved is:

$$BW_{fullHD} = 1920 \times 1080 \times 8 \times 29{,}97 \times (4 + 2 + 2) / 4 = 948{,}27 \text{ Mbps}$$

So, this video mode conference is reaching the 1GB Ethernet bandwidth.
This situation shows one of the problems of transmitting uncompressed video data, because if there is the need to transmit an audio stream with PCM in a notable quality, the rate will exceed the bandwidth becoming into a congestion situation.
To understand better this situation lets generate a stream in the same standard NTSC. And, for example, transmitting PCM at 48kHz with 16bits per sample and stereo mode:

$$BW_{audioCDlikeQuality} = 48000 \times 16 \times 2 = 1{,}46 \text{ Mbps}$$

Both streams are transmitted, so:

$$BW_{TOTAL} = BW_{fullHD} + BW_{audioCDlikeQuality} = 0{,}92 \text{ Gbps}$$

This generates a congestion situation that makes the whole system not working properly. Also there have to be added headers data from RTP packets, UDP packets and IP, and its stream payload headers, reaching up to 1Gbps everey time that an audio packet is transmitted (as a peak value).

Introduced UltraGrid, let's explain how it is its initial state and how will be implemented the audio module.

2.3.2 Initial state

The initial state of the audio module in the UltraGrid software was made for the capture in XenaHS device, that generates samples at 48000Hz, 32bits per sample and stereo mode (in fact transmits six channels, that only two of them are in use). So, at the beginning there is not a stable scenario when transmitting into a 1GB Ethernet.

In order to use XenaHS Device for the HD videoconference System, an external software for the audio transmission called "Robust Audio Tool" RAT [29] was used, as there was not an integrated stable version.

To understand faster how UltraGrid works, some UML (*Unified Modeling Language)* graphs are shown in ANNEX E.

With the UML shown is done an internal view of how whole internal loop was initial generated into UltraGrid.

Since now, as a result of other projects developed at i2cat, there have been done some improvements into the code that take into account buffer adaptation (with the video resolution and other purposes), video compression and the developed VBCC (Video Based Congestion Control).

These improvements shows the importance of how audio should be processed with video, that is audio stream should be processed as separated as it could be done from video stream because of the video treatment should not affect audio stream and its different treatment, and vice versa. Trying to maintain the sources synchronization.

2.3.3 Audio implementation

To implement an audio module, the best option is to create a device like done with XenaHS or Blackmagic, but only for audio stream. It should have three modes of interaction with video modules as shown in next high abstraction programme order execution figure:

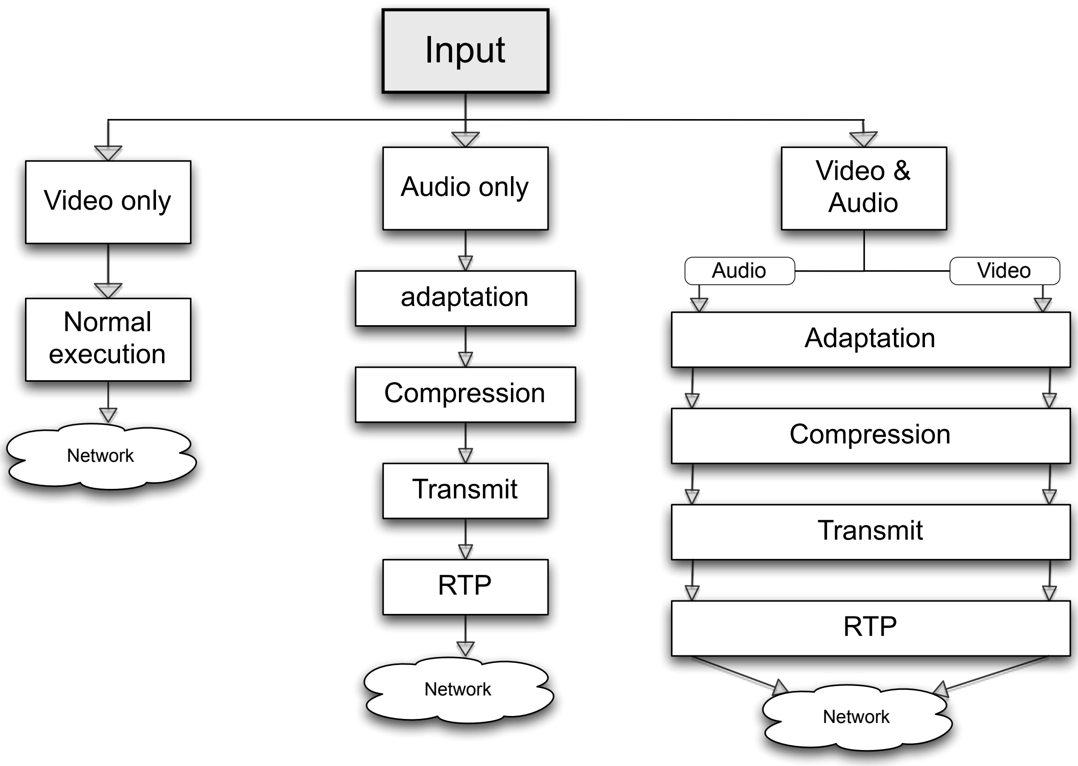


Fig. 2.4 Capture side high abstraction diagram

Those diagrams take into account the different cases of use of audio and video streams interaction. The first (left) is with no sound and only video transmission, like at the initial scenario with typical usage. The second case is with only audio streaming and the last case is transmitting both medias.

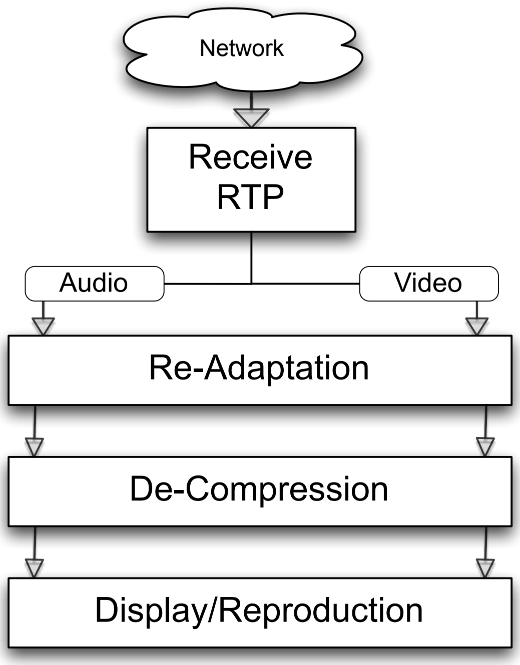


Fig. 2.5 Receive side high abstraction diagram

At reception side the streams are processed as normal. But if there is not one of both sources it does not affect the display/playout of the received media because if one media is not received it is not processed.

Here appears a synchronization contradiction because of if the synchronization is done with video source audio depends on it. But it can be solved, like will be explained in audio implementation chapter.

2.4 *Linux Sound*

Arriving at this point is required to understand how digital audio is treated in the Linux kernel and how it should be processed correctly for it to make the whole project work stable. Finally will be choosed the architecture that best suites.

This point is trying to explain how kernel should work with audio and, starting from the lower-level architecture explanation, it will show which of the architectures available will be chose for the main project.

2.4.1 Buffering

It is not useful to produce sound samples at the exactly rate required by the computer's audio hardware. If it is required to produce sound at CD quality, and the computer must be interrupted every 20us or a similar frequency of time to produce the next sample, the associated overhead would be unacceptably large and few of the required time to do the task would be spent in useful computation. For this reason, without exception, the computers' general purpose is to maintain a buffer for the input and output of sound samples.
Buffers that preserve the order in which data is added are known as FIFO (First In First Out) buffers or queues.

Fig. 2.6 illustrate the important concepts.

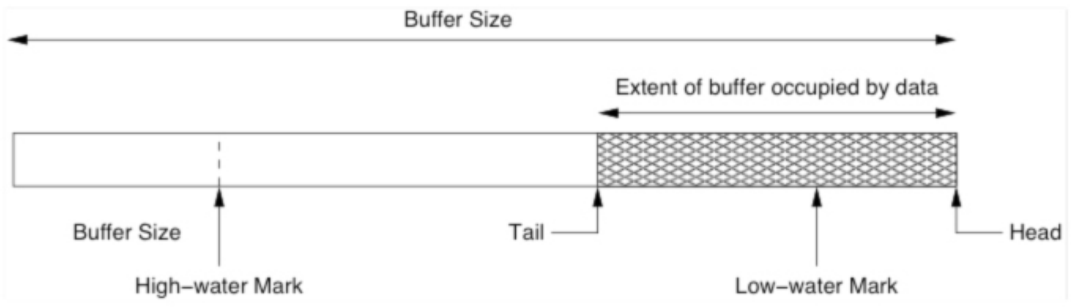


Fig. 2.6 A FIFO Buffer or Queue

As audio samples are generated by the application, they are added at the end of the queue, and the quantity of data gets increased.

In normal usage, a buffer suggests that unit of preallocated system resource is used to store the data with a consequent compile or run-time limit imposed on the maximum quantity of data which can be stored, as it has been configured. This is the case in most computer sound systems. A queue, on the other hand, suggests a dynamic structure that expands indefinitely until system resources are unavailable. One would tend to refer to a “sound buffer”, but to a “print queue”.

The buffered data is allowed to grow without any action being taken by the audio hardware until the data reaches the low-water mark. The low water mark is set by the system administrator and the application program to a level that, once reached, is really possible that the data in the buffer will be enough to ensure a continuous stream of samples delivered to the sound hardware, even if there are changes from time to time in the performance of the sound generating program.

Setting the lowest possible low-water mark means that sound generation will begin very quickly after the first audio samples become available, but there can be interruptions in the audio output if there are brief pauses in program execution. This is known as glitch effect, and for human ear is really notorious. This also can be a consequence, at network level, of packet lost and packet disorder.

Configuring a high low-water mark means that there may be a noticeable latency between samples generated by the application and sound produced from the computer. It also means that even if the system becomes heavily loaded and the program stalls for a considerable period, samples will be available to produce uninterrupted sound, becoming a glitch-free stream.

The application program is allowed, within limits, to choose the high and low-watermarks to adjust the performance of the sound subsystem. For example, a mail program which uses an audio queue to alert the user from an incoming mail will be unconcerned by latency and set a high low-water mark to ensure continuous playback under any circumstances, whereas a synthesizer program which generates sounds in response to key presses on a MIDI keyboard needs to make sure that the corresponding sound is emitted almost instantly, and will set a low low-water mark.

The concept of the high-water mark is less important in the context of a computer sound system, where the client program and the sound output device are almost always tightly coupled. If the buffer fills beyond the high-water mark, the sound subsystem may instruct the client application to cease the production of samples until the level of samples in the buffer falls below the high-water mark once again. The high-water mark is important in certain networking environments, where it might take some time for a “stop transmission” instruction to reach the source and there must be enough free space in the buffer to store information arriving after the stop instruction has been sent out if data loss has to be avoided.

However, this is not a relevant problem in a computer music system, because the application program is either running on the same computer as the sound hardware, or it is protected from the overrun problems by the operating system's data transport layer. Consequently, the high-water mark is almost always the same as the buffer size.

2.4.2 Kernel's job

If it were necessary to manipulate the sound hardware directly, every application would be required to understand the internal working of every piece of the sound hardware in any computer system. This would make the code unwieldy and difficult to maintain. It is the kernel's job to provide a degree of hardware abstraction, which says that all application programs may access to the same interface regardless of the installed hardware.

Now there is introduced the concepts of working with the lower-level architecture in Linux, and the known first sound hardware abstraction layer, called Open Sound System (OSS) [32] interface. This is done to understand how next audio architectures work.

2.4.3 Generic OSS functionality

OSS provides a number of devices that the application should address. What is of the most interest is the primary audio device /dev/audio. The system calls open(2), close(2), read(2) and write(2) permit the sound hardware to emit sound samples generated by an application, or to make available sound samples to be read by the application.

Unfortunately, reading and writing data between an application and a soundcard can be far more difficult than reading and writing a regular file. Consequently, several extra facilities are available to manipulate the behavior of the sound hardware via the ioctl(2) system call. This interface permits the setting and reading of parameters such as:

Sound hardware capabilities: request how many channels to work with is the sound hardware capable of recording and reproducing and what are the maximum and minimum sample rates.

Setting the hardware sample rate: requests a particular sample rate from the sound card.

Setting the number of channels: requests treating data as interleaved samples from or to a number of channels.

Requesting status information: request the current status of the sound card, as well as how many samples have been played or recorded since the device was opened.

Input/output buffer parameter manipulation: sets the total size and fragment size (effectively the low-water mark) of the audio buffers.

2.4.4 Generating high and low-water marks with OSS

OSS does not implement high and low-water marks directly, but it does permit the buffer to be divided into a number of fragments. This effectively permits the setting of a low-water mark because no output will occur before there is at least one full fragment in the buffer. For reasons about the implementation, the size of each fragment must be a power of 2.

Consider the buffer layout shown in Fig. 2.7.

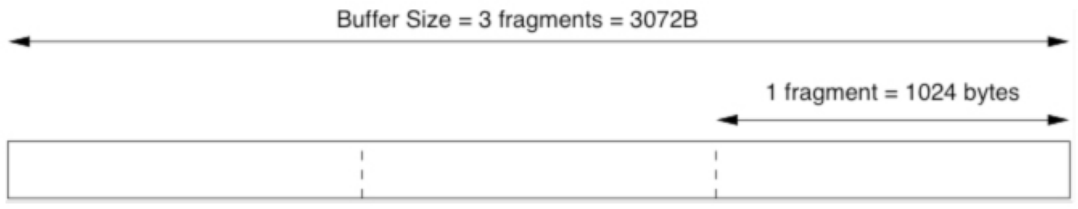


Fig. 2.7 Three fragments buffer sound

2.4.5 Client/Server Approach - Sound Daemons

The kernel provides a consistent interface for Unix applications to communicate with sound hardware, but problems arise when more than one application wishes to make sound at the same time. Generally, the act of opening the sound device locks out other applications from using it. As indicated in Fig 2.8, the first application to grab the sound device takes the control and may cause other applications to run without sound output, or even fail to start at all. The OSS interface defines a mixer device, but this is intended to mix a single sound source from the samples sent to the sound card with other sound sources available from the hardware such as CD/DVD audio, MIDI synthesizer outputs and so on. It will not mix different sample streams from multiple client applications.

As next figures indicate, there are two popular options to solve this problem. Remember when talking about operating system and basic architecture of GNU/Linux O.S.

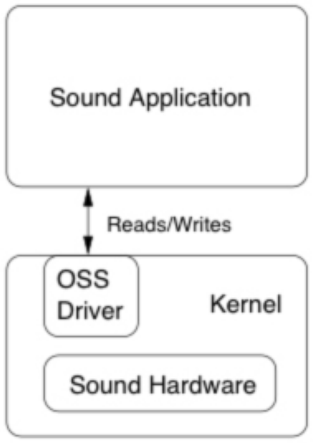


Fig. 2.8 Direct acces to devices. Little complexity on Kernel model.

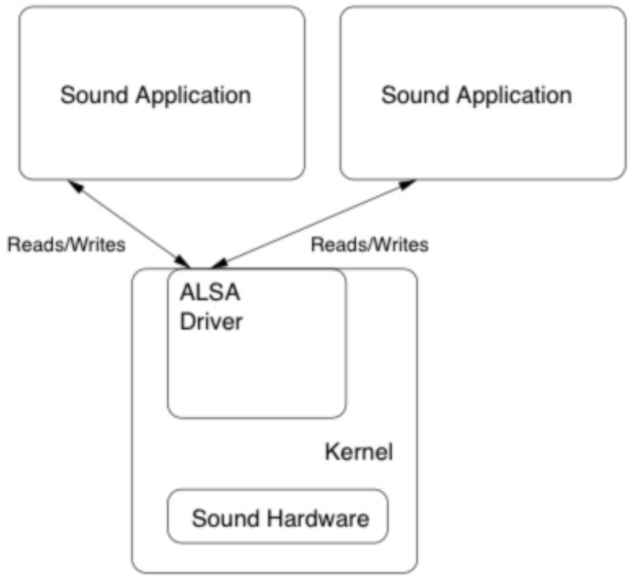


Fig. 2.9 Kernel complexity increase. Multiple connections via Kernel.

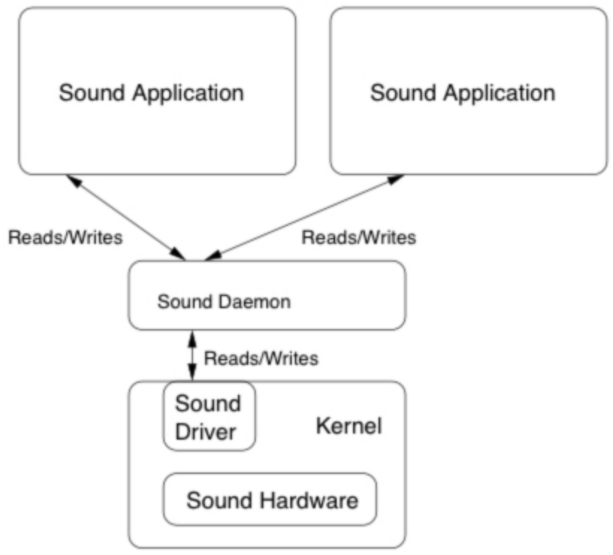


Fig. 2.10 Multi-stream access via Sound Daemon.

Some platforms provide kernel support for sound devices that can be opened by multiple applications (such as ALSA [33], the Advanced Linux Sound Architecture, Fig. 2.9).

Alternatively, a user-space program (like ESD [34], the Enlightened Sound Daemon, originally associated with the Enlightenment window manager or PulseAudio [35], used in the newer Ubuntu versions) can be run to act as a “sound server” (Fig. 2.10), accepting connections via a socket and performing mixing after sample rate and type conversion.

The latter approach is attractive from the architectural point of view, adhering to the principle of putting the minimum amount of code in the kernel where it is “locked and loaded”: kernel code, once loaded, occupies physical memory until it is unloaded again; it cannot be paged out like user-space programs.

Consequently, large and complex kernel drivers may impact upon the performance of the entire system even if the sound is currently not in use. However, it is questionable whether the overhead associated with copying the sound samples through a socket connection, even if it is on the local machine, outweighs the disadvantage of the large resident physical memory audio driver. Others argue that the closer binding of the application to the hardware possible with the ALSA approach is essential in high-performance audio applications such as hard-disk recorders and real-time music synthesis, and once admitted on that basis, it does not make sense to support the ESD method or PulseAudio as well.

Nevertheless, the ESD method is sufficiently popular to warrant it’s being covered here. Many open-source applications test for the availability of the Enlightend Sound Daemon and configure themselves accordingly.

As it should be, a sound architecture may provide a variety of audio-related functions, but it must include at least a low-level connection between the soundcard driver and kernel services and a higher-level audio software-programming interface to simplify applications development.

2.4.6 ALSA time

Until the 2.5.x development track, the Linux kernel utilized the OSS/Free audio API. OSS here stands for Open Sound System, an audio API originally written in 1992 by Hannu Savolainen. Independent developers contributed with drivers and other work to the OSS/Free system, but by the late 1990s the OSS API was showing its age. Around that time, a Czech system administrator named Jaroslav Kysela began work on what eventually became the ALSA project. Like Linux itself, ALSA began with rather modest goals: Jaroslav simply wanted more out of his soundcard than the existing API could deliver, and he was willing and able to meet the demands of the task. Like Linus Torvalds, the starter linux kernel developer, Jaroslav eventually found himself at the center of a group of talented developers, all dedicated to the development of a superior audio API for Linux.

ALSA's features and popularity grew at a point when users and developers began lobbying for utilizing ALSA to replace the OSS/Free system packaged with the kernel sources.

Testing began with the 2.5.x development kernels, and from the 2.6.x kernels onward ALSA has been the default Linux sound architecture.

As mentioned, ALSA's developers have concerned themselves with designing a truly superior sound system for Linux. On the ALSA Web site there is a description of the project's most significant features:

- Efficient support for all types of audio interfaces, from consumer soundcards to professional multichannel audio interfaces.
- Fully modularized sound drivers.
- SMP (Symmetric MultiProcessing) and thread-safe design.
- User space library to simplify application programming and provide higher level functionality.
- Support for the older OSS/Free API, providing binary compatibility for most OSS/Free programs.
- Licensed under the GPL and the LGPL.

Arriving at this point, is considered starting with a fresh installation of Ubuntu 10.04 or above. The most important reason for choosing Ubuntu, among a lot of other GNU/Linux distributions, is that it works with all device drivers that are going to be used; the more particular case is with the DeckLink Blackmagic card, because its producer driver has only worked with Ubuntu S.Os.

Ubuntu uses PulseAudio that works as ESD, but programming audio applications with PulseAudio library will cost some time latency in processing audio, as explained before.

So, the easiest and fastest solution is to work with PulseAudio as a Sound Daemon to interact with other sound applications and more instances of UltraGrid, like for example using one instance for transmitting and one other for receiving, in the same computer.

To implement the audio module, as talked above, the best option is using the ALSA library. Reaching the real-time specifications. The final motivation for using this API is, as said in the official ALSA project web site, that application programmers should use the library API rather than the kernel API. The library offers 100% of the functionality of the kernel API, but adds major improvements in usability, making the application code simpler and better looking. In addition, future fixes or compatibility code may be placed in the library code instead of the kernel driver.

In addition, ALSA has a capability called plug-in that allows extension to new devices, including virtual devices implemented entirely in software. ALSA also provides a number of command-line utilities, including a mixer, sound file player and tools for controlling special features of specific sound cards.

CHAPTER 3. PROPOSED SOLUTION

In this chapter is going to be described the generic solution for audio implementation, that is implemented with ALSA API, as it has been chosen above.

For a better understanding there will be described all processes since capturing audio, from computer audio card and its transmission, to reproducing the audio received. Configuring it to reach the lowest latency possible and generating the audio data synchronized with the video stream to assume a complete videoconference.

3.1 Buffering and sources synchronization

Sources synchronization mean that audio is defined as being clock synchronous with video if the sampling rate of audio is such that the number of audio samples occurring within an integer number of video frames is itself a constant integer number. The following table will show the relation with audio sample rates reached with HD Intel sound card and its audio buffer size, with both standard video sample rate used in UltraGrid.

Table 3.1 Audio sampling rate and synchronized audio buffer generation

Audio Sampling rate	Samples/frame, 29.97 fr/s video	Samples/frame, 25 fr/s video
192.0 kHz	6408/1	7680/1
96.0 kHz	3204/1	3840/1
48.0 kHz	1602/1	1920/1
44.1 kHz	1472/1	1764/1
32.0 kHz	1068/1	1280/1
16.0 kHz	534/1	640/1
8.0 kHz	268/1	320/1

The video and audio clocks must be derived from the same source since simple frequency synchronization could eventually result in a missing or extra sample within the audio frame sequence, known as underrun or overrun effect respectively, also as glitches. For 29.97 fps there have not been utilized integer numbers because of this could cause not real-time specifications (for example at 44.1kHz it should be 147147 samples per 100 video frames, and so on). This is rounding up to the first even number.

So, this synchronization is done at transmission side because of being capture side, but at reception side this is done comparing timestamps obtained at the RTP header when receiving both video and audio streams. This is automatically treated with both sides of the UltraGrid RTP module.

Generating this data flows indicates that we are generating the same presentation time (the same as latency time capture), this is because the rhythm is supposed to be taken by video source generation and audio is being generated at the same rhythm to be synchronized. Then, has to be said that it is not the whole latency but it is the latency that can be controlled by UltraGrid (its developers) and the other latency generated is from the capture devices, the displays, among others.

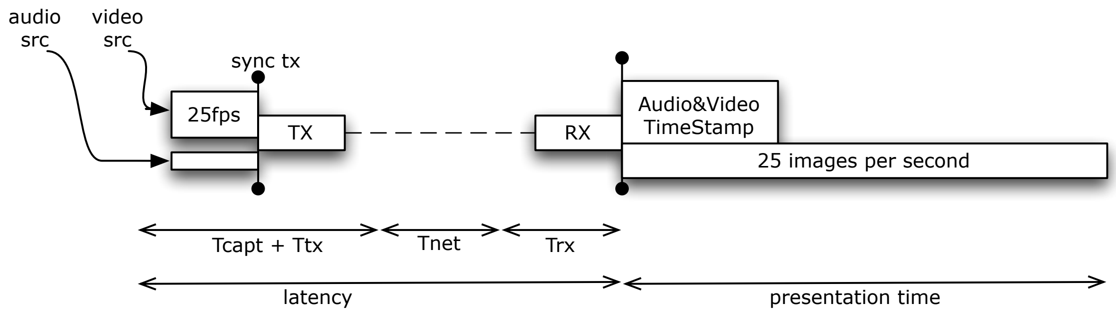


Fig. 3.1 Graphical timing of the whole involved process

It has to be said that presentation time is equal to Tcapt plus Ttx (capture time). Tnet is the time that the info is going over the network (it is treated as zero in this case).

Once knowing the audio buffer size required is time to do the appropriated ALSA initialization.

3.2 ALSA initialization

As it is done with OSS system to configure the start up ALSA defines its similar but particular steps of doing:

If it was not known with which parameters the sound card can work, then ALSA system in the O.S. have a command-line audio file player that helps to know this. Using the command:

```
#aplay –h
```

Then is listed the following recognized sample formats:

# Recognized sample formats are: S8 U8 S16_LE S16_BE U16_LE U16_BE S24_LE S24_BE U24_LE U24_BE S32_LE S32_BE U32_LE U32_BE FLOAT_LE FLOAT_BE FLOAT64_LE FLOAT64_BE IEC958_SUBFRAME_LE

IEC958_SUBFRAME_BE MU_LAW A_LAW IMA_ADPCM MPEG GSM SPECIAL S24_3LE S24_3BE U24_3LE U24_3BE S20_3LE S20_3BE U20_3LE U20_3BE S18_3LE S18_3BE U18_3LE
Some of these may not be available on selected hardware

These are all available sample formats into ALSA library from O.S but it is saying that some of theme should be specially treated like the ones that depends on plug-in generation, such as A_LAW.

The letter S means signed, U unsigned format sample. This is mentioned because it is not enough to know that a PCM sample is, for example, 8 bits wide. Whether the sample is signed or unsigned is needed to understand the range. If the sample is unsigned, the sample range is 0..255 with a centerpoint of 128. If the sample is signed, the sample range is -128..127 with a centerpoint of 0.

If a PCM type is signed, the sign encoding is almost always 2's complement. In very rare cases, signed PCM audio is represented as a series of sign/magnitude coded numbers. So, as common used in PCM standard generation, in this project Signed samples are chosen to work with. This election does the audio module to be compatible with the other capture devices from the scenario.

When more than one byte is used to represent a PCM sample, the byte order (big endian -BE- vs. little endian -LE-) must be known. Due to the widespread use of little-endian Intel CPUs, little-endian PCM tends to be the most common byte orientation and this is chosen.

So now audio module must be started up. This is doing the internal ALSA device opening and whole initialization, as done in ALSA capture side in UltraGrid. There have to be said that both sides do the same initialization structure, for capture or playback:

```
static void set_alsa_capture_params(struct vidcap_ALSA_state *st){

 struct vidcap_ALSA_state *s = (struct vidcap_ALSA_state *) st;
```

/* Open PCM device for capture (recording). */

```
    rc = snd_pcm_open(&s->handle,"default",SND_PCM_STREAM_CAPTURE, 0); //or _PLAYBACK
    if (rc < 0) {
      fprintf(stderr,"unable to open pcm device: %s\n",snd_strerror(rc));
      exit(1);
    }
```

/* Allocate a hardware parameters object. */

```
    snd_pcm_hw_params_alloca(&s->params);
```

/* Fill it in with default values. */

```
    err = snd_pcm_hw_params_any(s->handle, s->params);
    if (err < 0) {
        printf("Broken configuration for this PCM: no configurations available");
        exit(EXIT_FAILURE);
    }

/* Set the desired hardware parameters. */
  /* Interleaved mode */
    snd_pcm_hw_params_set_access(s->handle, s->params,SND_PCM_ACCESS_RW_INTERLEAVED);

/* Signed 16-bit little-endian format */  //endian because of INTEL Chipset
    snd_pcm_hw_params_set_format(s->handle, s->params,SND_PCM_FORMAT_S16_LE);

/* Two channels (stereo) */
    snd_pcm_hw_params_set_channels(s->handle, s->params, 2);

/* 48000 bits/second sampling rate (High quality) */
    rate = 48000;
    snd_pcm_hw_params_set_rate_near(s->handle, s->params, &rate, &dir);

/* Set period size to 1920 frames. */ //This is internally set to 1024, the nearest rounded
down into power of two frames buffer to 1920 frames, but reaching as samples as
required in audio buffer transmission when forcing capture of 1920 frames.
    s->frames = 1920;
    snd_pcm_hw_params_set_period_size_near(s->handle,s->params, &s->frames, &dir);

/* Write the parameters to the driver */
    rc = snd_pcm_hw_params(s->handle, s->params);
    if (rc < 0) {
      fprintf(stderr,"unable to set hw parameters: %s\n", snd_strerror(rc));
      exit(1);
    }
}
```

More treatment code about options, parameters listing, playback and capture with ALSA library is shown in ANNEX F.

Right now are known the basic ALSA API library functions to initialize ALSA device.

It has to be appreciated that frame size depends on the sample rate and the frames per second captured/reproduced. But the buffer size also depends on the utilized sample format and the number of channels. So, this is, for example:

Sample size at 48kHz and PAL stream: 48000/25 = 1920 (periods/fragments per audio frame)

Buffer size at 48kHz and PAL stream, 16bits per sample and stereo (standard PCM):

1920 x 16 x 2 / 8bits = 7680 bytes

But using an A-law format and one channel, the buffer size result:

1920 x 8 x 2 / 8bits = 3840 bytes

This shows how we can go into the line that separates from quality to throughput generated, and it is the never-ending fight for reaching the most accurate solution for every context.

3.2.1 Generic configuration (capture/playout)

The generic configuration for audio streaming, reaching high quality, is with a PCM of 48kHz of sample rate, 16 bits per sample and stereo mode.

This chose is because it is the most compatible mode to work with the scenarios proposed in this project.

This configuration gives high quality for transmitting at all human ear working spectrum. But it has to be remembered that if there is no need to transmit music with a vocoder there is spent less bandwidth.

3.3 Input/output buffer parameter manipulation

As it is internally required into ALSA API kernel module, the fragment sizes (known as periods by the ALSA community) have to be a power of two.

The restriction of having to be able of working with different sample formats and so on and having to reach real-time specifications gives the necessity of generating the fragment size that gives less latency for all options.

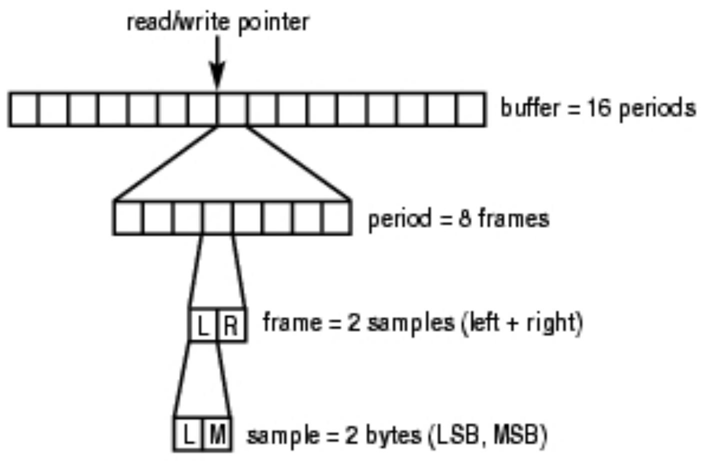


Fig. 3.2 Example ALSA buffering

The figure shown permits to understand how is generated the ALSA buffer size. This example is working at 16bits per sample and stereo, and configuring 8 frames per period and a buffer of 16 periods, so there is 512 bytes of audio buffer.

If the audio buffer size required to be synchronized with video source is of 7680 bytes (following the last example shown) the resulting high water mark is the same audio buffer size required.

The low-water mark should be as high as possible to reach the real-time specifications but with stability, as if it was all processed into the same soundcard, and maintaining it below the frames per second required and not reaching an overflow error.

Setting the period size with ALSA API (as shown in the initialization example) it sets the nearest below the needed sample size. For this example, sound would start to reproducing when receiving at least 1024 frames to reach the 1920 frames generated.

This become a good stable real-time specification (like using a computer sound player), but if a packet is lost there will be nothing to fill into the buffer required and there will appear a glitch into the continuous stream that can’t be eliminated. Also, here is shown the line that there is between real-time and stability/quality. So, this project is at this line.

3.4 Plugin generation

With plug-in generation there appear more options to implement other PCM digitalization and into different source treatment.

3.4.1 Dual

For example, there is the option to implement a dual audio system of transmission with the same bandwidth using two channels but both with mono audio source, being able to transmit two speakers from the same computer or room and expanding it up to the maximum channel treatment offered by the integrated audio card. Then this is done in reception side implementing a mono to stereo, and stereo to mono in transmission side, plug-in with ALSA system environment into the O.S. and opening it as the default device when starting up ALSA configuration.

3.4.2 A-law

From ALSA plug-in community is done an A-law dedicated plug-in that gives the option to implement a standard telephonic communication into audio stream treatment in UltraGrid, this is a vocoder that permits to reach the half throughput of the same stream that in stantard PCM. Starting to implement UltraGrid audio system into standards in audio videoconference’s system.

3.4.3 IMA-ADPCM

This is another implemented plug-in into ALSA API that permits generate with this vocoder the fourth part the bandwidth obtained from the same stream but with PCM formatting.

This plugin have the problem that it implements a vocoder that does a codification from an A-law formatted sample being affected from amplitude variation. So, a really good equalization must be done if there appears the necessity to work with the minimum possible bandwidth, and having to speak with a steady tone.

How to implement and configure a plugin with ALSA library is explained in ANNEX G.

3.5 ALSA programming

When packets are arriving these are processed to generate the expected audio frame. These are going from a prebuffer of two audio buffer size that permits to do a circular queue to fill one buffer when receiving the frame and reproducing the other one, and in capture side one buffer is filled when data-capturing and the other is processed to transmit, and so on.

3.5.1 Requesting status information

This part is really important to give, as much as ALSA API can, the maximum audio stability into UltraGrid.

Requesting status information permits to know which is the problem and how it can be solved as much as can be done when capturing or reproducing samples.

Having studied how to do that for both sides, transmission and reception, is shown an example of a generic x-run recuperation (x means over or under):

```
static void xrun(void){
    snd_pcm_status_t *status;
    int res;

snd_pcm_status_alloca(&status);

if ((res = snd_pcm_status(handle, status))<0) {
        printf(stderr, "status error in snd_pcm_status. sound halt\n");
        exit(EXIT_FAILURE);
}
if (snd_pcm_status_get_state(status) == SND_PCM_STATE_XRUN) {
        if ((res = snd_pcm_prepare(handle))<0) {
            printf(stderr, "prepare error in xrun. sound halt\n");
            exit(EXIT_FAILURE);
      }
      return;     /* ok, data should be accepted again */
}
        printf(stderr, "r/w error. sound halt\n");

        exit(EXIT_FAILURE);
}
```

This piece of code has been tested and it is not assuming real-time recovery at all.

Once detected that problem, it has been studied a better solution that gives UltraGrid more speed-recovery from packet-lost or any type of glitches.
The code generated does the following routine:

```
Enter to xrun() →     if(snd_pcm_recover(handle,r,0)<0) {

                             snd_pcm_drain(handle);

                             snd_pcm_reset(handle);

                             snd_pcm_prepare(handle);

                      }             → Continue capturing/reproducing
```

This section creates the concept of speed recovery. When faster the code generated is, from recovering the stream, it can be assumed that more quality treatment is done.

The snd_pcm_recover function recovers the stream state from an error or suspend. This functions handles -EINTR (interrupted system call), -EPIPE (overrun or underrun) and -ESTRPIPE (stream is suspended) error codes trying to prepare given stream for next I/O.

If the error is not solved then is used ‘snd_pcm_drain’, snd_pcm_reset and snd_pcm_preapre functions in that order. This is done because the first function stops the PCM preserving pending frames (for playback wait for all pending frames to be played and then stop the PCM. For capture stop PCM permitting to retrieve residual frames), the second reset PCM position and reduce PCM delay to 0 and the last function prepare PCM for use, so, in that order is reached a fast and quite good stream-recovery.

For more information about PCM interface see ANNEX H.

3.6 Capture side

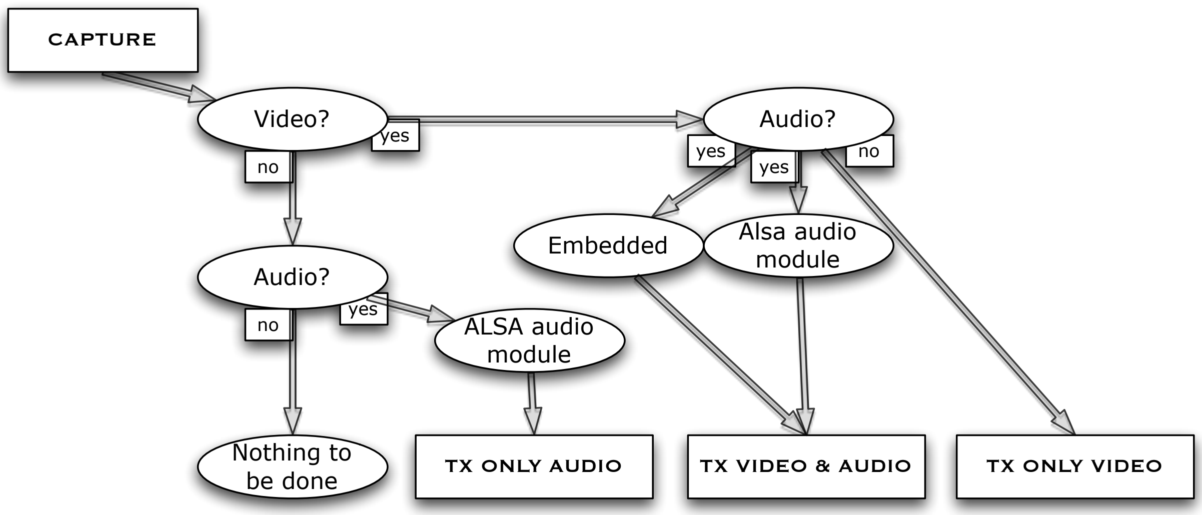


Fig. 3.3 Capture side diagram

The figure above explains how the transmission side algorithm will be done.

With the implemented module, called *audio capture*, there have done the basic functions to develop a capture device into UltraGrid.

A basic device has the following functions:

```
struct vidcap_type   *vidcap_DEVICE_probe(void);
void               *vidcap_DEVICE_init(struct map_d *md);
void                vidcap_DEVICE_done(void *state);
struct media_frame   *vidcap_DEVICE_grab(void *state);
```

The 'probe' function defines a method to assume the selected device and if it is prepared to start. Searching for its libraries into the O.S. , or drivers folder, and if the card is detected into the computer.

The 'init' function initializes all buffers and its capacity, as well as some specific structures and special call API card producer functions. Finally, here starts the audio 'thread' function, implemented internally as a function too, and its internal control semaphore.

'Done' function does the exit protocol to terminate all process started with the initialized device and releasing all buffers and preallocated memory.

'Grab' function is the responsible to copy the new data generated, from the main thread, to the RTP buffers and some header information. Always being controlled by the semaphore indications.

Main ‘thread’ function makes the loop go moving the circular buffer, being controlled by the general semaphore programmed. The circular buffer consists in two sub buffers; when one is being read the other is being written and reverse. Each sub buffer contains the audio data generated per video frame.

Also, and for future real-time control in audio parameters, the initialization of the ALSA parameters have been defined externally from the init function, and to have the complete control of which sample rate and format per sample is configured too.

3.7 Playout side

This side is similar as capturing but have the most delicate job that is reproducing the same as captured.

The playout side will follow the same criteria as in the section above. There is implemented a playout module, which does the basic functions to reproduce device into UltraGrid.

This side is programmed to be adapted with all kind of audio sources that can be streamed as shown in test scenario, but has to be implemented a default mode that is the same defined in capture side.

Into playout device there are the same functions, doing similar jobs. But the principal difference is that function grab is not required and there are implemented two functions instead. The function ‘getf’ that gets the frame from reception module (frame complete) and the ‘putf’ function that gives the expected frame to ALSA playout module implemented, doing the control of the circular reception buffer (a buffer with a size of two audio buffer size, as explained), like it is done with video stream.

The following image will show how it is done to do this whole process, differentiating from video and audio sources.

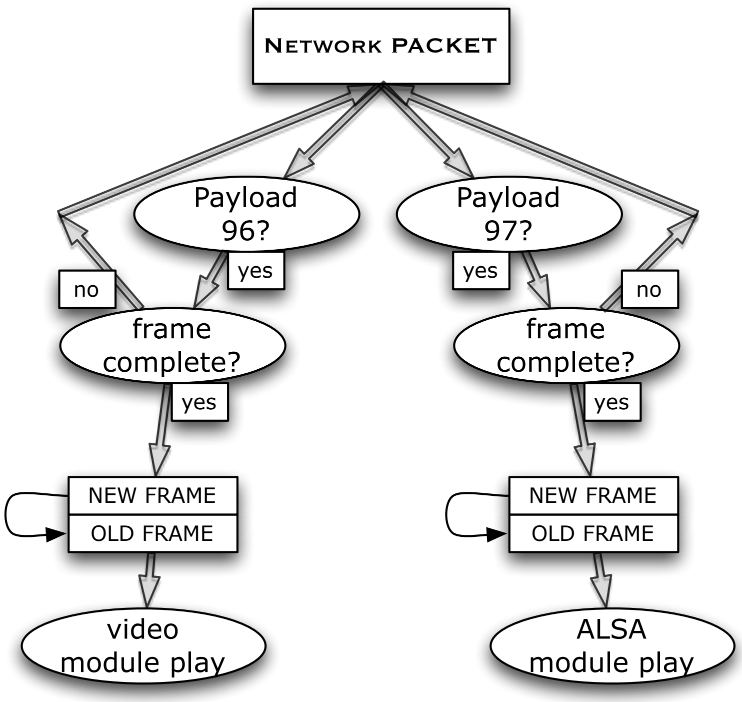


Fig. 3.4 Playout side diagram

In this picture is shown how audio is treated while packets are arriving from the net. This is that when packets are arriving, the RTP module gets the timestamp and permits display both sources synchronized. But, also, as an objective, if there is a video packet lost (or dropped, for example) audio is going on displaying its stream. Then, when there is another video packet it will only display frames with the same audio timestamp. This is how audio is not depending on video at playout time and will be done as a future work.

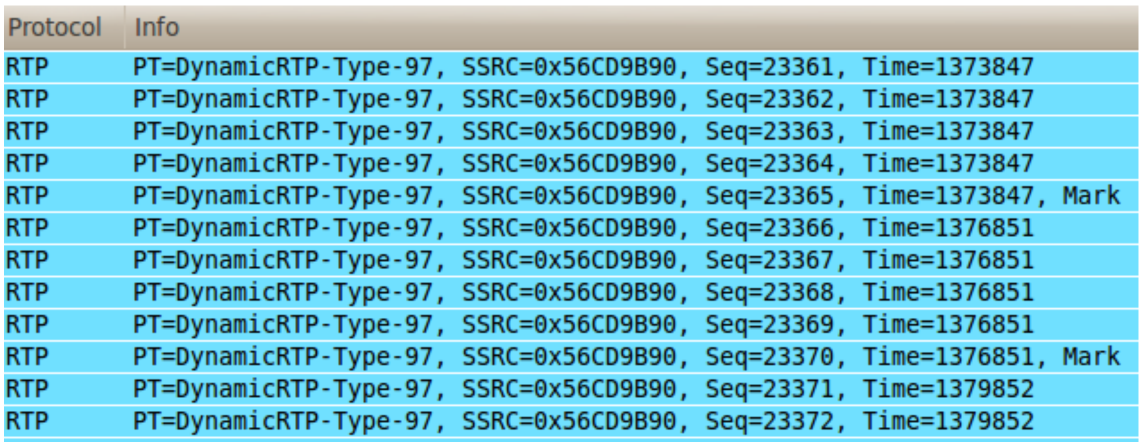
CHAPTER 4. AUDIO IMPROVEMENTS

This ending implementation chapter is going to explain how has been adapted every device specification from final scenario into the ALSA audio module implementation. The most important section is the first one that helps to understand what is happening and what is needed.

4.1 XenaHS Audio Module

Studying at the start up of this project the audio XenaHS implementation started before, has helped a lot doing the project in understanding how audio PCM is generated and can be processed, also with its treatment for RTP.

The following pictures cached from WireShark explain how the audio buffer is placed into RTP packet, that is observing the RTP packet generated from its payload to whole audio data provided to/from network.



Protocol	Info
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23361, Time=1373847
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23362, Time=1373847
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23363, Time=1373847
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23364, Time=1373847
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23365, Time=1373847, Mark
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23366, Time=1376851
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23367, Time=1376851
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23368, Time=1376851
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23369, Time=1376851
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23370, Time=1376851, Mark
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23371, Time=1379852
RTP	PT=DynamicRTP-Type-97, SSRC=0x56CD9B90, Seq=23372, Time=1379852

Fig. 4.1 RTP audio packets captured by WireShark

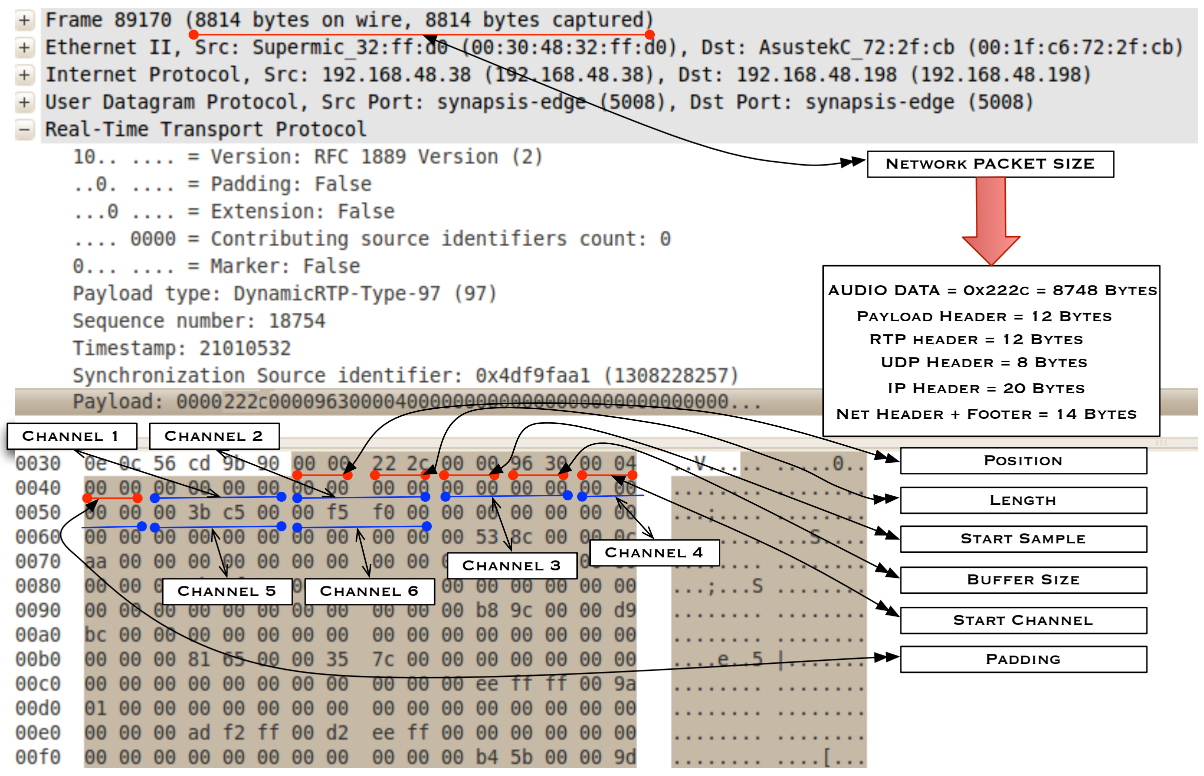
To give order to each packet there is the sequence number and to distinguish from frame to frame there is the timestamp.

There are five packets per frame generated. This is because of the whole data exceeds the maximum MTU (Maximum Transfer Unit) defined into the Ethernet interface to reach the 1Gb (9000Bytes of MTU). The following figure will describe it better:

Table 4.1 Frame reconstruction when receiving audio from XenaHS 48kHz, 6ch and setereo

	POSITION	LENGTH	START SAMPLE	BUFFER SIZE	START CHANNEL	PADDING
PAQ1	0x0000 = 0	0x222C	0	0x9630 ó 0x9618 =38448 ó 38424		0
PAQ2	0x222C =8748	0x222C	0	0x9630 ó 0x9618 =38448 ó 38424		0
PAQ3	0x4458 =17496	0x222C	0	0x9630 ó 0x9618 =38448 ó 38424		0
PAQ4	0x6684 =26244	0x222C	0	0x9630 ó 0x9618 =38448 ó 38424		0
PAQ5	0x88B0 =34992	0x0D80 ó 0x0D68	0	0x9630 ó 0x9618 =38448 ó 38424		0
TOTAL FRAME	9630 ó 9618 =38448 ó 38424	0x9630 ó 0x9618 =38448 ó 38424	0	0x9630 ó 0x9618 =38448 ó 38424	.->0->2->4->0->.	0

Analyzing the packet generation with WireShark there has been edited the following figure to explain how is data transmitted:



```
+ Frame 89170 (8814 bytes on wire, 8814 bytes captured)
+ Ethernet II, Src: Supermic_32:ff:d0 (00:30:48:32:ff:d0), Dst: AsustekC_72:2f:cb (00:1f:c6:72:2f:cb)
+ Internet Protocol, Src: 192.168.48.38 (192.168.48.38), Dst: 192.168.48.198 (192.168.48.198)
+ User Datagram Protocol, Src Port: synapsis-edge (5008), Dst Port: synapsis-edge (5008)
- Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: DynamicRTP-Type-97 (97)
    Sequence number: 18754
    Timestamp: 21010532
    Synchronization Source identifier: 0x4df9faa1 (1308228257)
    Payload: 0000222c000096300004000000000000000000000000000...
0030  0e 0c 56 cd 9b 90 00 00  22 2c 00 00 96 30 00 04   ..V.... .....0..
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0050  00 00 00 3b c5 00 00 f5  f0 00 00 00 00 00 00 00   ...;.... ........
0060  00 00 00 00 00 00 00 00  00 00 00 53 8c 00 00 0c   ........ ...S....
0070  aa 00 00 00 00 00 00 00  00 00 [illegible]   ........ ........
0080  00 00 [illegible] 00 00 00 00 00 00   ...;...S ........
0090  00 00 00 00 00 00 00 00  00 00 00 b8 9c 00 00 d9   ........ ........
00a0  bc 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00b0  00 00 00 81 65 00 00 35  7c 00 00 00 00 00 00 00   ....e..5 |.......
00c0  00 00 00 00 00 00 00 00  00 00 00 ee ff ff 00 9a   ........ ........
00d0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00e0  00 00 00 ad f2 ff 00 d2  ee ff 00 00 00 00 00 00   ........ ........
00f0  00 00 00 00 00 00 00 00  00 00 00 b4 5b 00 00 9d   ........ ....[...
```

Fig. 4.2 Captured WireShark image with exhaustive explanation

As it is in the initial audio scenario shown, it is only using stereo mode, so four channels are disused.

At this point of the project it was suggested doing a simple data compression. This compression consists in removing the first sixteen bytes of audio data generated (four channels at 24 bits per sample plus 8 bits of padding), because from 16th byte to 24th byte there was the audio data generated by XenaHS, because of working at 32 bits per sample and up to six channels it generates 24 bytes of sampled data. It was also plugging the 5th and 6th canon audio pin from XenaHS capture device:

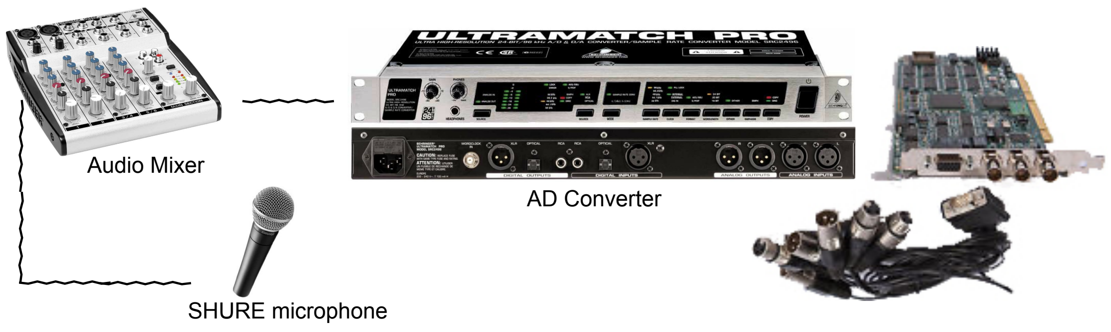
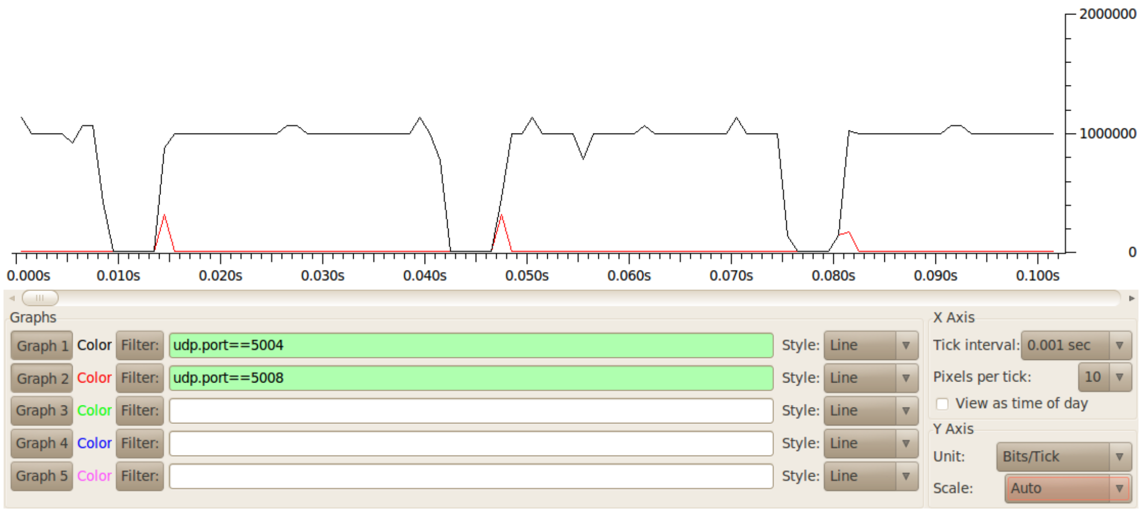


Fig. 4.3 Audio Configuration XenaHS capture card

The algorithm is as simple as removing the first sixteen bytes as said and reordering the data to be in the order as established but without the channels disused at the transmission point, and when receiving it ALSA receive it at 32 bits/48000kHz and 6 channels as can be done with the HD Intel Sound Card. Or without decompression, just reproducing it at stereo mode (2 channels), that doesn't affect at source synchronization time.

Remembering the bandwidth problem generated from transmitting FullHD resolution and audio uncompressed there have been calculated the throughput obtained with XENAHS audio samples:

$$BW_{fullHD} = 1920 \times 1080 \times 8 \times 29{,}97 \times (4 + 2 + 2) / 4 = 948{,}27 \text{ Mbps}$$
$$BW_{audioXENAHS} = 48000 \times 32 \times 6 = 8{,}78 \text{ Mbps}$$
$$BW_{audioXENAHS\_2CH} = 48000 \times 32 \times 2 = 2{,}92 \text{ Mbps}$$



Graphs

Graph 1 Color Filter: udp.port==5004 Style: Line
Graph 2 Color Filter: udp.port==5008 Style: Line
Graph 3 Color Filter: Style: Line
Graph 4 Color Filter: Style: Line
Graph 5 Color Filter: Style: Line

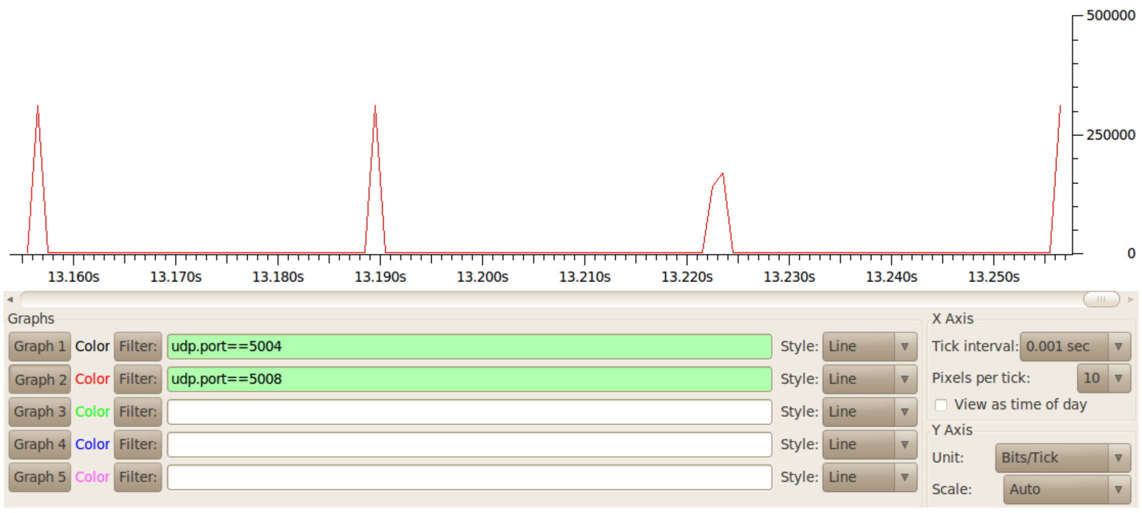
X Axis
Tick interval: 0.001 sec
Pixels per tick: 10
View as time of day

Y Axis
Unit: Bits/Tick
Scale: Auto

Fig. 4.4 Graphical timing WireShark captured packets. Audio and Video

This graphic is showing how is the throughput generated and in this particular case, as shown, there is reached the bandwidth limit, so there will be glitches at sound and dropped video frames.

The two following pictures will show how there is a really differential throughput generation when applying the compression or not. With no compression there are five packets and with compression there are only two packets.



Graphs

Graph 1	Color	Filter:	udp.port==5004	Style:	Line
Graph 2	Color	Filter:	udp.port==5008	Style:	Line
Graph 3	Color	Filter:		Style:	Line
Graph 4	Color	Filter:		Style:	Line
Graph 5	Color	Filter:		Style:	Line

X Axis
Tick interval: 0.001 sec
Pixels per tick: 10
View as time of day
Y Axis
Unit: Bits/Tick
Scale: Auto

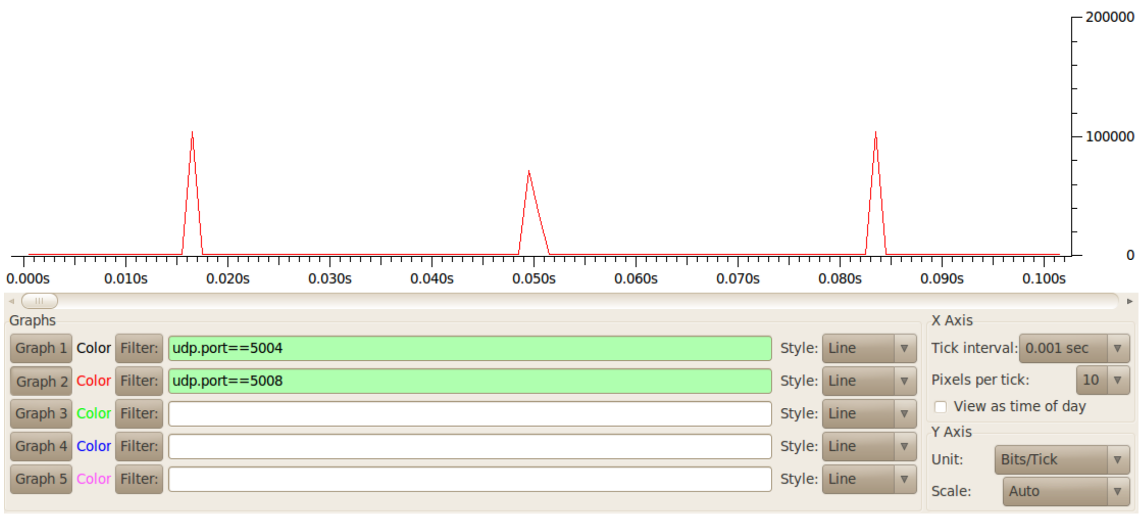
Fig. 4.5 Graphical timing Wireshark captured packets. 6 audio channels

Transmitting all six channels there is a peak-throughput of:

48000 x 32 x 6 x 29.97 = 269Mbits transmitting one audio packet (not doing the average as when calculating the bandwidth).

And transmitting only the two channels used there is:

48000 x 32 x 2 x 29.97 = 90Mbits when transmitting one audio packet.



Graph 1 Color Filter: udp.port==5004 Style: Line
Graph 2 Color Filter: udp.port==5008 Style: Line
Graph 3 Color Filter: Style: Line
Graph 4 Color Filter: Style: Line
Graph 5 Color Filter: Style: Line
Tick interval: 0.001 sec
Pixels per tick: 10
Unit: Bits/Tick
Scale: Auto

Fig. 4.6 Graphical timing Wireshark captured packets. 2 audio channels

So, now it is only generated the third part of the initial audio throughput and the proposed compression is reached.

$$\frac{BW_{audioXENAHS\_2CH}}{BW_{audioXENAHS}} = \frac{1}{3}$$

Fig. 4.7 Bandwidth proportion

Finally there are shown how are the new scenarios obtained as a result of these implementations.

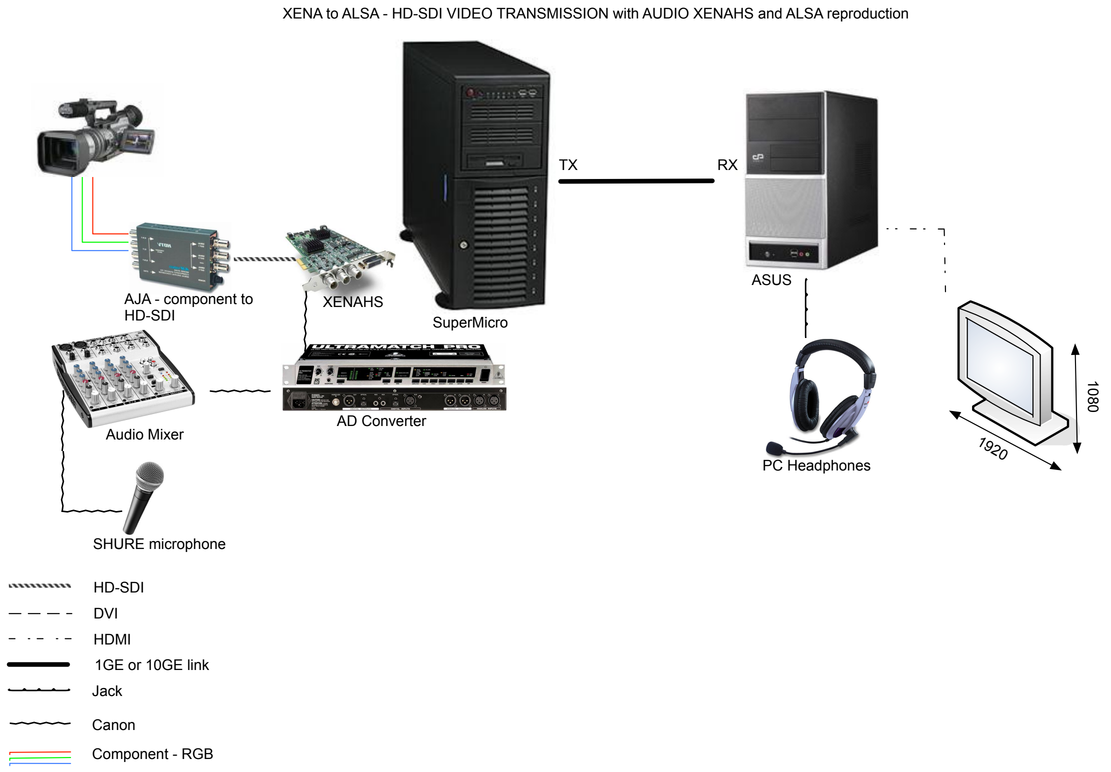


Fig. 4.8 XenaHS to ALSA scenario

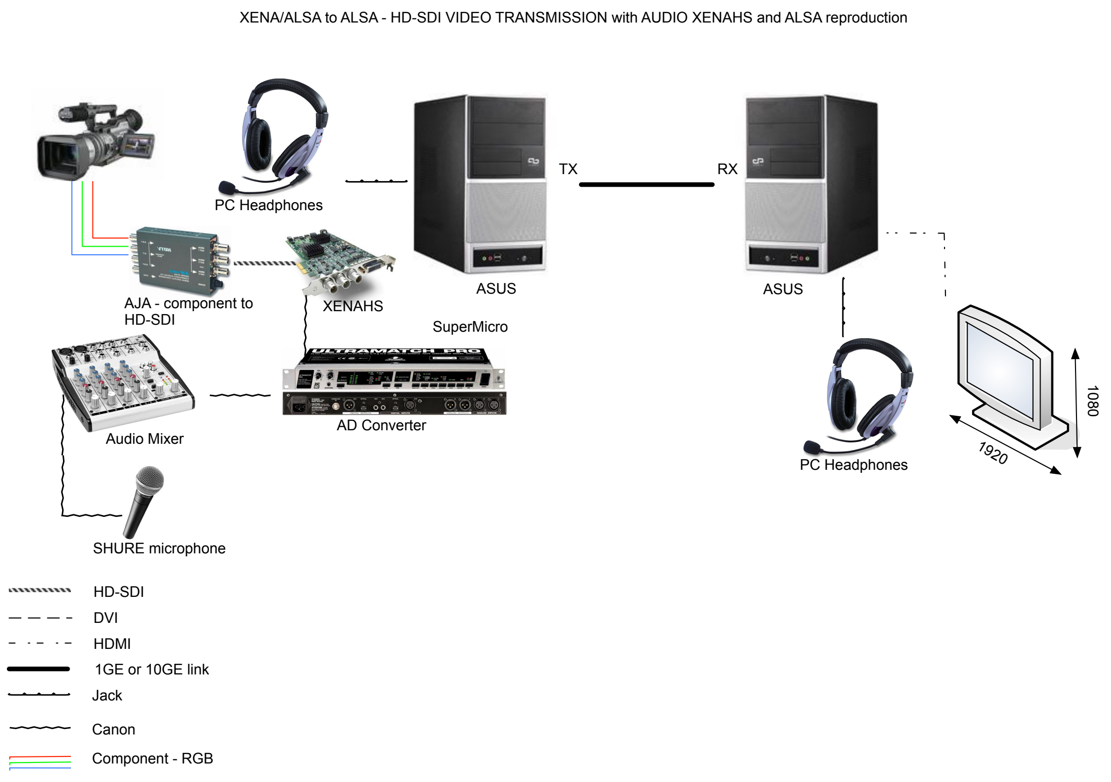


Fig. 4.9 Different options XenaHS to ALSA scenario

The last scenario shows that it could be done an election in capturing side from getting audio samples from XenaHS or from HD Intel Sound Card. This is done in transmission side generating a flag, and some other code generation, that permits activate the capture source election from XenaHS device or from integrated sound card with the ALSA implementation module.

In reception side the video stream is displayed by the SDL API [13] developed before into UltraGrid to not depend at reception side of having the XenaHS card and other dispositive required. This module takes the synchronization control with the audio data received and also the ALSA playout module have to be initialized with the correct parameters, the same as XenaHS captured samples.

It has to be said that at the initial programme analysis all seemed correct but it doesn't work correctly. It was consequence that one data structure was modified and XenaHS couldn't read the data structure expected, this is:

```
typedef struct{
      unsigned char         *video_data;
      unsigned char         *audio_data;
#ifdef HAVE_FIREWIRE
      int16_t                     *audio_dv;
#endif
      short                 mtypes;
} media_display_format;
```

The mistake was adding audio_dv pointer into the media_diplay_format structure. Then, XenaHS API implemented wasn't compatible with the generated structure.

The solution was simply to add the restriction to compile only with the firewire flag if XenaHS was not coming into use.

4.2 XenaHS Audio Module with ALSA/16bits

This module was required when working with Intel card of type AC'97 that didn't permit working with more than 20bits per sample. So when trying to capture with ALSA a stream from XenaHS this was not possible.

The proposed solution was processing the received bit-stream before reproducing it. Its structure was modified to become a PCM of 48kHz, 2 channels and 16bits per sample. It was transformed to assume the MSB bytes (2 Bytes) as a 16 bits sample and not taking into account the LSB bytes:

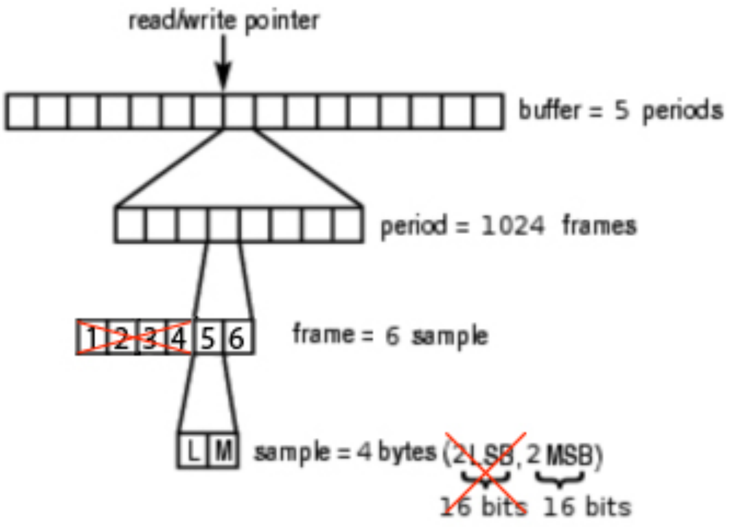


Fig. 4.10 6 channles 24 bits to 2 channels 16 bits

4.3 DV to ALSA

When implementing this audio module it was quite fast because of the knowledge acquired before. But there was no understanding reason of why it was reproducing so many glitches.

Analyzing the code into UltraGrid, there have been demonstrated that is not an audio modules problem, so it is out of scope right now. The problem is with the implementation of the linux firewire API into UltraGrid. This means that when it is captured into a file the data generated by the firewire from UltraGrid it sounds perfect, but when capturing it before transmitting then gliches appear. So, some arrangements are required to work correctly.

4.4 Blackmagic to ALSA

With this professional capture card there is integrated into a computer a complete high quality videoconference system.

The implementation with ALSA was so quick because it was only need to put into code a flag for choosing the stream origin (from Blackmagic or Intel HD), and if ALSA audio module was chosen only was required to activate it.

With this card we get the final scenario that can suit for all contexts required, from a face-to-face to a room-to-room videoconference, assuming HD-SDI and HDMI as principal source.

CHAPTER 5. TESTING & VALIDATION

This ending chapter is to demonstrate with some examples how this project has come to reality and how it can be tested and validated.

5.1 *MusicLab at CitiLab*

MusicLab is a multidisciplinary project that invites musicians and artists from many different fields to join into an open musician community. This project is realized at CitiLab. This is a center for social and digital innovation in Cornellà de Llobregat, Barcelona. It operates and distributes the digital impact on creative thinking, design and innovation from emerging digital culture.

This project took part into an experimetal session in MusicLab. Trying to test the simultaneosly music interpretation over internet. The objective was trying with musics, at different places, to play as synchronized as it can be done.

This emerged successful because it was possible to generate an operation protocol, despite the delay, but really little (real-time internet concept) thanks of the audio module implemented into UltraGrid.

Thanks to the succesfuly results it will studyed the option of doing a real-time performance in a convencional festival with this system.

For more information there is a link to know more about this experiment:
musiclab-taller-experimental-interaccio-simultania-online

5.2 *Audioconference centre unit*

With the following picture is going to described the concept:

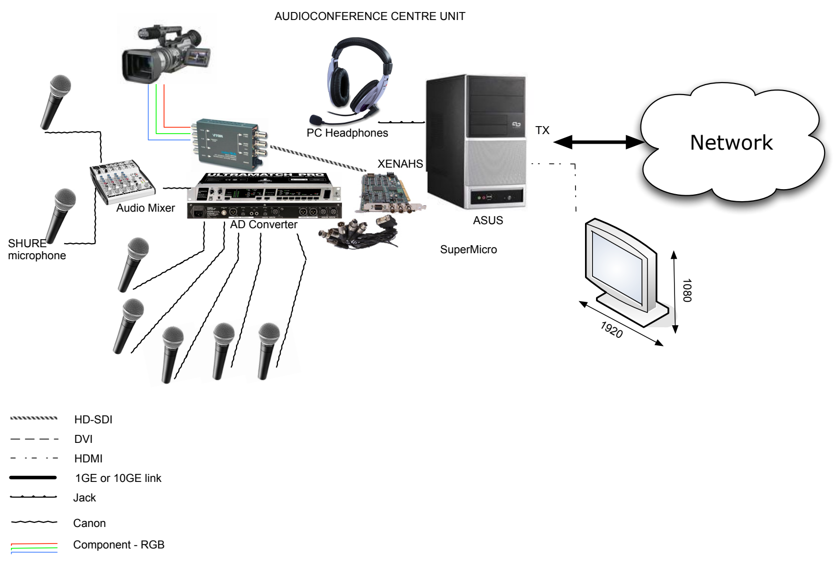


Fig. 5.1 Audioconference centre unit scenario

This is controlling all audio input sources to send only one (or more), switching it by software. What this option offers is to implement a really cheap option for generating an easy set up configuration videoconference centre unit, reaching the room-to-room level with the pieces shown. The final cost will depend on the number of microphones/devices required and so the maximum number of permitted audioconferencers.

This section is presented to be really easy implemented as a future work. The basic idea will be doing as in the following figure:

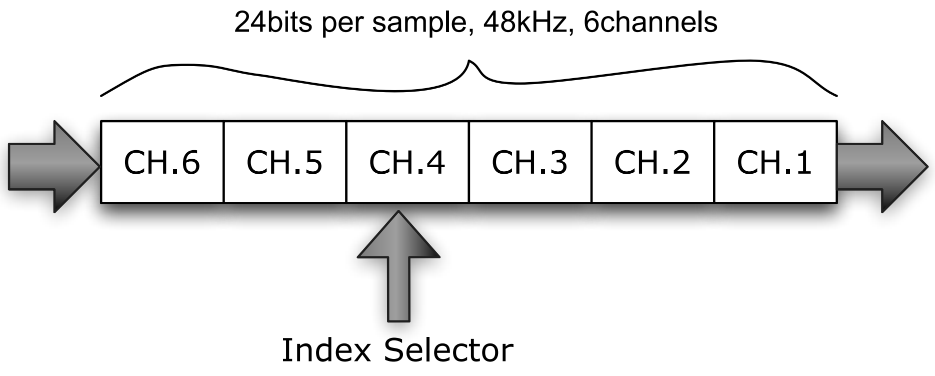


Fig. 5.2 Graphical explanation of software ndex selector

5.3 *Lab scenario*

This is the day by day scenario that can be tested at i2cat. Here can be mixed different options with all raised scenarios. This is a laboratory scenario. Here is where all practical issues have been studied and helped to understand how to going on developing the whole project.

CHAPTER 6. ENVIROMENTAL IMPACT

Nowadays, in a globalized world, working groups are distributed around the world. Several companies are geographical disperse and managers travel for meeting on headquarters. Also research groups work in a cooperative way with other international institutions and need to make face-to-face meeting. Also in telemedicine, doctors can save to travel huge distances for evaluate their patients, especially in medical specialists. These are few examples where high definition videoconference can be useful to save time, money and reduce pollution. Thinking in the audio side also could be a good platform for playing with different musicians and transmit high quality programs over the internet, having huge bandwidth but with no pollution impact, related on people transport.

Negative impact may be needs of high amount of bandwidth that are needed for making done the videoconference, needing the construction of new lines of optical fiber, satellite communication and aerial constructions.

In distributed display, working with an ordinary PCs, it’s a positive environment impact because these equipment can be used in other tasks when the distributed display is no required.

Finally, the different options of transmission into UltraGrid can be adapted into every case of bandwidth requirements and avoid the construction of new telecommunication lines. Also with audio adaptations there is no need of lots of expensive devices and with only one computer, and minimum professional or common devices, all video and audio processes can be done.

CHAPTER 7. CONCLUSIONS AND FUTURE WORKS

The main objective of this project is the audio integration into UltraGrid. As seen in testing and validation chapter, this has been achieved successfully because it has been reached stable transmissions sampled at 192kHz 16 bits stereo, synchronized with video at real-time. That reaches without drops or glitches with K or HDready video resolutions around 150 ms of delay. This is because 1Gb bandwidth is not filled.

K-resolution: 960 x 540 x 2 x 8 x 29,97 = 273Mbps

HD-ready resolution: 1280 x 720 x 2 x 8 x 29,97 = 421Mbps

Maximum quality audio achieved: 192000 x 2 x 16 = 5,86Mbps

It has been demonstrated that the best-known option to work out implementation related to free software/open source sound architectures is ALSA.

However, this kind of architectures can involve library modifications in a future. But this can be assumed that if there are modifications these will be useful to improve certain aspects of the library, and new integrations will be easy and fast, as said in ALSA’s community website.

What it also has been seen is that working with Ubuntu versions up to 10.04 implies an improvement in devices compatibility as well as the possibility of having Pulse Audio system by default and its compatibility with ALSA (even having to make some modifications as is explained in ANNEX D).

As it has been annotated in environmental impact, this type of system permits a decrease in pollution caused by transports. Even so, it is important to go further in studies about how to decrease in the dependence of high bandwidths.

So, as future works, there is a necessity of generating audio streams with loss compressions that ensure quality to transmit music/concerts or voice.

Furthermore, it is also important to generate the automation in the code to configure the parameters both in reception and in transmission depending on the required sample rate, sample format and the number of channels. In addition, to permit not having to depend on video if it is required at some time.

Other future works are the improvements related to firewire libraries implementation as it has been seen. Apart from the generation of an external application, done with socket communications, to permit real-time sound parameters modification and the selection of which audio source should be used.

Finally, it is also important to comment that it should always be taken into account the different exposed margins showed below depending on the context and required needs.

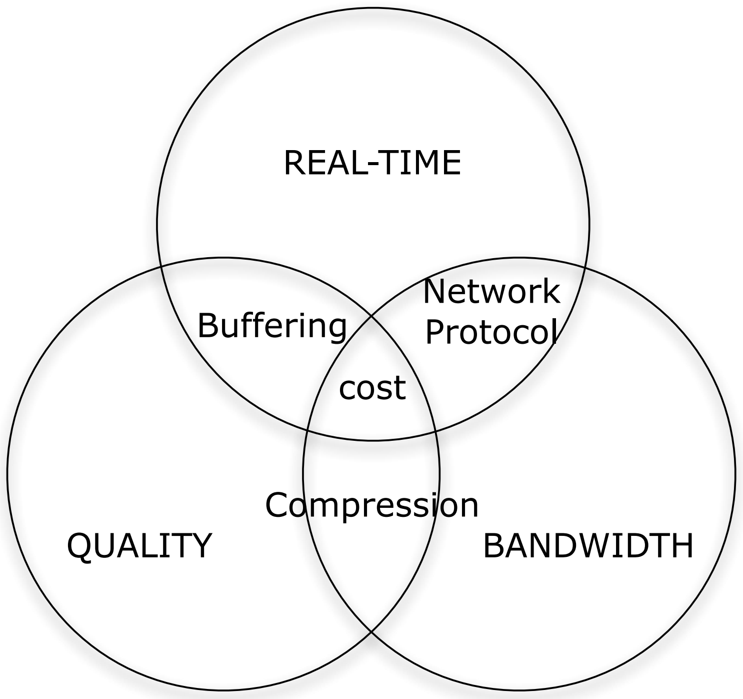


Fig. 7.1 Relationship and commitments

REFERENCES

[1] UltraGrid.
http://ultragrid.dcs.gla.ac.uk/

[2] Liceu.
http://www.liceubarcelona.cat/

[3] Skype.
http://www.skype.com/

[4] FullHD and HDready.
http://es.wikipedia.org/wiki/1080p

[5] Vocoder.
http://es.wikipedia.org/wiki/Vocoder

[6] i2cat foundation.
http://www.i2cat.net/ca

[7] V3 project.
http://wiki.i2cat.net/doku.php/i2cat:public:clusters:audiovisual:uhdgroup:uh dgrroup

[8] Wikipedia. High definition Television.
http://en.wikipedia.org/wiki/High-definition_television

[9] Wikipedia. Modelo de color RGB.
http://es.wikipedia.org/wiki/Modelo_de_color_RGB

[10] Wikipedia. RGBA color space.
http://en.wikipedia.org/wiki/RGBA_color_space

[11] Wikipedia. YUV.
http://es.wikipedia.org/wiki/YUV

[12] Information Sciences Institute East.
http://www.east.isi.edu/index.php

[13] Simple DirectMedia Layer.
http://www.libsdl.org/

[14] Pulse-code Modulation.
http://en.wikipedia.org/wiki/Pulse-code_modulation

[15] Nyquist theorem.
http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theor em

[16] Speech companding.
http://www.young-engineering.com/docs/YoungEngineering_ALaw_and_MuLaw_Companding.pdf

[17] G.711 – A-law and Mu-Law.
http://en.wikipedia.org/wiki/G.711

[18] G.726 – ADPCM.
http://en.wikipedia.org/wiki/G.726

[19] Real-time Transport Protocol.
http://en.wikipedia.org/wiki/Real-time_Transport_Protocol

[20] RTP Payload Format for Uncompressed Video.
http://www.faqs.org/rfcs/rfc4175.html

[21] AJA XenaHS.
http://www.aja.com/index.php

[22] Blackmagic – Decklink.
http://www.blackmagic-design.com/

[23] Intel High Definition Audio.
http://www.intel.com/design/chipsets/hdaudio.htm

[24] ASUS.
http://www.asus.com/

[25] Nvidia.
http://www.nvidia.es/page/home.html

[26] DV (Digital Video) over IP [DVTS].
http://www.sfc.wide.ad.jp/DVTS/

[27] Wikipedia. DV.
http://en.wikipedia.org/wiki/DV

[28] HDwiki. SAGE.
http://wiki.i2cat.net/doku.php/i2cat:public:clusters:audiovisual:uhdgroup:sage

[29] RAT – Robust Audio Tool.
http://www.sal.wisc.edu/~jwp/wiyn/rat.html

[30] Linux.
http://es.wikipedia.org/wiki/GNU/Linux

[31] Kernel.

http://www.kernel.org/

[32] OSS – Open Sound System.
http://en.wikipedia.org/wiki/Open_Sound_System

[33] ALSA – Advanced Linux Sound Architecture
http://en.wikipedia.org/wiki/Advanced_Linux_Sound_Architecture

[34] ESD – Enlightened Sound Daemon
http://es.wikipedia.org/wiki/Enlightened_Sound_Daemon

[35] PulseAudio.
http://es.wikipedia.org/wiki/PulseAudio

[36] D. Cuenca Gil, "Adaptive media content architecture regarding users capabilities"
http://upcommons.upc.edu/pfc/bitstream/2099.1/8044/1/memoria.pdf

[37] D. Turull Torrents, "*Performance and enhancement for HD videoconference environment*".
http://upcommons.upc.edu/pfc/bitstream/2099.1/5000/1/memoria.pdf

[38] F. J. Iglesias Garcia, "*Development of an integrated interface between SAGE and UltraGrid*".
http://upcommons.upc.edu/pfc/bitstream/2099.1/6068/1/memoria.pdf

ACRONYMS

BNC Bayonet Neil Concelman

DTV Digital Television

DV Digital Video

DVI Digital Visual Interface

RTT Round Trip Time

HDMI High-Definition Multimedia Interface

Fps frames per second

Full HD (or 2K) Full High Definition (1920x1080 pixels)

GE Gigabit Ethernet

Blu-Ray Blue Ray Disc

HD High Definition

HD-Ready High Definition - Ready (1280x720 pixels)

HD-SDI High Definition - Serial Digital Interface

HDTV High Definition Television IP Internet Protocol

ITU-R International Telecommunication Union – Radiocommunication

K resolution four times lower than 2K resolution (960x540 pixels)

NTSC National Television System Committee

PAL Phase Alternating Line (720x576 pixels)

RTP Real Time Protocol

QoS Quality of Service

RX Receiver computer

SAGE Scalable Adaptive Graphics Environment

SDL Simple Direct media Layer

SHD (or 4K or Quad HD) Super High Definition (3840x2160 pixels)

SMPTE Society of Motion Picture and Television Engineers

TV Television

TX transmitter computer

UDP User Datagram Protocol

UG UltraGrid

V3 Video, Videoconference and Visualization

YUV a colour space that is defined by luminance (Y) and two components of chrominance (UV)

YUV422 is YUV subsampling colour by a half

RAT Robust Audio Tool

BE Big Endian

LE Little Endian

API Application Programming Interface

VBCC Video Based Congestion Control

CPU Centra Processing Unit

GPU Graphics Processing Unit

Shure

O.S. Operating System

GNU GNU's Not Unix (an unix-like system)

UNIX Multitasking and multi-user computer operating system

AES/EBU Audio Engineering Society/European Broadcasting Union

RTCP Real-Time Transport Control Protocol

SNR Signal to Noise Ratio

ADPCM Adaptive Differential Pulse Code Modulation

CD Compact Disc

RAW not compressed data format

Hz Hertz

DM Delta Modulation

DPCM Differential Pulse Code Modulation

Glitches Technical problems. In audio slang can be stops and silence, among others, while audio capturing or playout.

ANNEX A. Speech companding (A- law/µ-law)

A-Law Compander

A-law is the CCITT recommended companding standard used across Europe. Limiting the linear sample values to 12 magnitude bits, the A-law compression is defined by Equation 1, where *A* is the compression parameter (*A*=87.7 in Europe), and *x* is the normalized integer to be compressed.

$$F(x) = \begin{cases} \dfrac{A*|x|}{1+\ln(A)} & 0 \le |x| < \dfrac{1}{A} \\[2ex] \dfrac{\operatorname{sgn}(x)*(1+\ln(A|x|))}{1+\ln(A)} & \dfrac{1}{A} \le |x| \le 1 \end{cases}$$

Table 1 illustrates an A-law encoding table. The sign bit of the linear input data is omitted from the table. The sign bit (*S*) for the 8-bit code is set to 1 if the input sample is negative, and is set to 0 if the input sample is positive.

Linear Input Data												A-law Encoded Output							
0	0	0	0	0	0	0	A	B	C	D	X	S	0	0	0	A	B	C	D
0	0	0	0	0	0	1	A	B	C	D	X	S	0	0	1	A	B	C	D
0	0	0	0	0	1	A	B	C	D	X	X	S	0	1	0	A	B	C	D
0	0	0	0	1	A	B	C	D	X	X	X	S	0	1	1	A	B	C	D
0	0	0	1	A	B	C	D	X	X	X	X	S	1	0	0	A	B	C	D
0	0	1	A	B	C	D	X	X	X	X	X	S	1	0	1	A	B	C	D
0	1	A	B	C	D	X	X	X	X	X	X	S	1	1	0	A	B	C	D
1	A	B	C	D	X	X	X	X	X	X	X	S	1	1	1	A	B	C	D

After the input data is encoded through the logic defined in the table, an inversion pattern is applied to the 8-bit code to increase the density of transitions on the transmission line, a benefit to hardware performance. XOR'ing the 8-bit code with 0x55 applies the inversion pattern.

Decoding the A-law encoded data is essentially a matter of reversing the steps in the encoding. Table 2 illustrates the A-law decoding table, applied after reversing the inversion pattern. The least significant bits discarded in the encoding process are approximated by the median value of the interval. This is shown in the output section by the trailing 1..0 pattern after the *D* bit.

A-law Encoded Input								Linear Output Data											
S	0	0	0	A	B	C	D	0	0	0	0	0	0	0	A	B	C	D	1
S	0	0	1	A	B	C	D	0	0	0	0	0	0	1	A	B	C	D	1
S	0	1	0	A	B	C	D	0	0	0	0	0	1	A	B	C	D	1	0
S	0	1	1	A	B	C	D	0	0	0	0	1	A	B	C	D	1	0	0
S	1	0	0	A	B	C	D	0	0	0	1	A	B	C	D	1	0	0	0
S	1	0	1	A	B	C	D	0	0	1	A	B	C	D	1	0	0	0	0
S	1	1	0	A	B	C	D	0	1	A	B	C	D	1	0	0	0	0	0
S	1	1	1	A	B	C	D	1	A	B	C	D	1	0	0	0	0	0	0

µ-Law Compander

The United States and Japan use µ-law companding. Limiting the linear sample values to 13 magnitude bits, the µ-law compression is defined by Equation 2, where µ is the compression parameter (µ =255 in the U.S. and Japan) and *x* is the normalized integer to be compressed.

$$F(x) = \frac{\operatorname{sgn}(x) * \ln(1+\mu|x|)}{\ln(1+\mu)} \qquad 0 \le |x| \le 1$$

The encoding and decoding process for µ-law is similar to that of A-law. There are, however, a few notable differences: 1) µ-law encoders typically operate on linear 13-bit magnitude data, as opposed to 12-bit magnitude data with A-law, 2) before chord determination a bias value of 33 is added to the absolute value of the linear input data to simplify the chord and step calculations, 3) the definition of the sign bit is reversed, and 4) the inversion pattern is applied to all bits in the 8- bit code.
Table 3 illustrates a µ-law encoding table. The sign bit of the linear input data is omitted from the table. The sign bit (*S*) for the 8-bit code is set to 1 if the input sample is positive, and is set to 0 if the input sample is negative.

Linear Input Data													µ-law Encoded Output							
0	0	0	0	0	0	0	1	A	B	C	D	X	S	0	0	0	A	B	C	D
0	0	0	0	0	0	1	A	B	C	D	X	X	S	0	0	1	A	B	C	D
0	0	0	0	0	1	A	B	C	D	X	X	X	S	0	1	0	A	B	C	D
0	0	0	0	1	A	B	C	D	X	X	X	X	S	0	1	1	A	B	C	D
0	0	0	1	A	B	C	D	X	X	X	X	X	S	1	0	0	A	B	C	D
0	0	1	A	B	C	D	X	X	X	X	X	X	S	1	0	1	A	B	C	D
0	1	A	B	C	D	X	X	X	X	X	X	X	S	1	1	0	A	B	C	D
1	A	B	C	D	X	X	X	X	X	X	X	X	S	1	1	1	A	B	C	D

After the input data is encoded through the logic defined in the table, an inversion pattern is applied to the 8-bit code to increase the density of transitions on the transmission line, a benefit to hardware performance.

XOR'ing the 8-bit code with 0xFF applies the inversion pattern.

Decoding the μ -law encoded data is essentially a matter of reversing the steps in the encoding. Table 4 illustrates the μ-law decoding table, applied after reversing the inversion pattern. The least significant bits discarded in the encoding process are approximated by the median value of the interval. This is shown in the output section by the trailing 1..0 pattern after the *D* bit.

μ-law Encoded Input								Linear Output Data												
S	0	0	0	A	B	C	D	0	0	0	0	0	0	0	1	A	B	C	D	1
S	0	0	1	A	B	C	D	0	0	0	0	0	0	1	A	B	C	D	1	0
S	0	1	0	A	B	C	D	0	0	0	0	0	1	A	B	C	D	1	0	0
S	0	1	1	A	B	C	D	0	0	0	0	1	A	B	C	D	1	0	0	0
S	1	0	0	A	B	C	D	0	0	0	1	A	B	C	D	1	0	0	0	0
S	1	0	1	A	B	C	D	0	0	1	A	B	C	D	1	0	0	0	0	0
S	1	1	0	A	B	C	D	0	1	A	B	C	D	1	0	0	0	0	0	0
S	1	1	1	A	B	C	D	1	A	B	C	D	1	0	0	0	0	0	0	0

ANNEX B. Digital audio sample rates

Table from: http://en.wikipedia.org/wiki/Sampling_rate

Sampling rate	Use
8,000 Hz	Telephone and encrypted walkie-talkie, wireless intercom and wireless microphone transmission; adequate for human speech but without sibilance; *ess* sounds like *eff*. Not for music.
11,025 Hz	One quarter the sampling rate of audio CDs; used for lower-quality PCM, MPEG audio and for audio analysis of subwoofer bandpasses.
16,000 Hz	Wideband frequency extension over standard telephone narrowband 8,000 Hz. Used in most modern VoIP and VVoIP communication products.
22,050 Hz	One half the sampling rate of audio CDs; used for lower-quality PCM and MPEG audio and for audio analysis of low frequency energy. Suitable for digitizing early 20th century audio formats such as 78s.
32,000 Hz	miniDV digital video camcorder, video tapes with extra channels of audio (e.g. DVCAM with 4 Channels of audio), DAT (LP mode), Germany's Digitales Satellitenradio (German), NICAM digital audio, used alongside analogue television sound in some countries. High-quality digital wireless microphones.
44,056 Hz	Used by digital audio locked to NTSC *color* video signals (245 lines by 3 samples by 59.94 fields per second = 29.97 frames per second).
44,100 Hz	Audio CD, also most commonly used with MPEG-1 audio (VCD, SVCD, MP3). Originally chosen by Sony because it could be recorded on modified video equipment running at either 25 frames per second (PAL) or 30fps (using an NTSC *monochrome* video recorder) and cover the 20 kHz bandwidth thought necessary to match professional analog recording equipment of the time. A PCM adaptor would fit digital audio samples into the analog video channel of, for example, PAL video tapes using 588 lines by 3 samples by 25 frames per second. Much pro audio gear uses (or is able to select) 44.1 kHz sampling, including mixers, EQs, compressors, reverb, crossovers, recording devices and CD-quality encrypted wireless microphones.
47,250 Hz	world's first commercial PCM sound recorder by Nippon Columbia (Denon)
48,000 Hz	The standard audio sampling rate used by professional digital video equipment such as tape recorders, video servers, vision mixers and so on. This rate was chosen because it could deliver a 22 kHz frequency response and work with 29.97 frames per second NTSC video - as well as 25 fps, 30fps and 24fps systems. With 29.97fps systems it is necessary to handle 1601.6 audio

	samples per frame delivering an integer number of audio samples only every fifth video frame. Also used for sound with consumer video formats like DV, digital TV, DVD, and films. The professional Serial Digital Interface (SDI) and High-definition Serial Digital Interface (HD-SDI) used to connect broadcast television equipment together uses this audio sampling frequency. Much professional audio gear uses (or is able to select) 48 kHz sampling, including mixers, EQs, compressors, reverb, crossovers and recording devices such as DAT.
50,000 Hz	First commercial digital audio recorders from the late 70s from 3M and Soundstream.
50,400 Hz	Sampling rate used by the Mitsubishi X-80 digital audio recorder.
88,200 Hz	Sampling rate used by some professional recording equipment when the destination is CD (multiples of 44,100 Hz). Some pro audio gear uses (or is able to select) 88.2 kHz sampling, including mixers, EQs, compressors, reverb, crossovers and recording devices.
96,000 Hz	DVD-Audio, some LPCM DVD tracks, BD-ROM (Blu-ray Disc) audio tracks, HD DVD (High-Definition DVD) audio tracks. Most pro audio gear uses (or is able to select) 96 kHz sampling, including mixers, EQs, compressors, reverb, crossovers and recording devices. This sampling frequency is twice the 48 kHz standard commonly used with audio on professional video equipment.
176,400 Hz	Sampling rate used by HDCD recorders and other professional applications for CD production.
192,000 Hz	DVD-Audio, some LPCM DVD tracks, BD-ROM (Blu-ray Disc) audio tracks, and HD DVD (High-Definition DVD) audio tracks, High-Definition audio recording devices and audio editing software. This sampling frequency is four times the 48 kHz standard commonly used with audio on professional video equipment.
352,800 Hz	Digital eXtreme Definition, used for recording and editing Super Audio CDs, as 1-bit DSD is not suited for editing. Eight times the frequency of 44.1 kHz.
2,822,400 Hz	SACD, 1-bit sigma-delta modulation process known as Direct Stream Digital, co-developed by Sony and Philips.
5,644,800 Hz	Double-Rate DSD, 1-bit Direct Stream Digital at 2x the rate of the SACD. Used in some professional DSD recorders.

ANNEX C. Payload types

PT	Name	Type	Clock rate (Hz)	Audio channels	References
0	PCMU	Audio	8000	1	RFC 3551
1	1016	Audio	8000	1	RFC 3551
2	G721	Audio	8000	1	RFC 3551
3	GSM	Audio	8000	1	RFC 3551
4	G723	Audio	8000	1	
5	DVI4	Audio	8000	1	RFC 3551
6	DVI4	Audio	16000	1	RFC 3551
7	LPC	Audio	8000	1	RFC 3551
8	PCMA	Audio	8000	1	RFC 3551
9	G722	Audio	8000	1	RFC 3551
10	L16	Audio	44100	2	RFC 3551
11	L16	Audio	44100	1	RFC 3551
12	QCELP	Audio	8000	1	
13	CN	Audio	8000	1	RFC 3389
14	MPA	Audio	90000		RFC 2250, RFC 3551
15	G728	Audio	8000	1	RFC 3551
16	DVI4	Audio	11025	1	
17	DVI4	Audio	22050	1	
18	G729	Audio	8000	1	
19	reserved	Audio			
20 - 24					
25	CellB	Video	90000		RFC 2029
26	JPEG	Video	90000		RFC 2435
27					
28	nv	Video	90000		RFC 3551
29 30					
31	H261	Video	90000		RFC 2032
32	MPV	Video	90000		RFC 2250
33	MP2T	Audio/Video	90000		RFC 2250
34	H263	Video	90000		
35 - 71					
72 - 76	reserved				RFC 3550
77 - 95					
96 - 127	dynamic				RFC 3551
dynamic	GSM-HR	Audio	8000	1	
dynamic	GSM-EFR	Audio	8000	1	
dynamic	L8	Audio	variable	variable	
dynamic	RED	Audio			
dynamic	VDVI	Audio	variable	1	
dynamic	BT656	Video	90000		
dynamic	H263-1998	Video	90000		
dynamic	MP1S	Video	90000		
dynamic	MP2P	Video	90000		

dynamic	BMPEG	Video	90000		

ANNEX D. Installing Ubuntu 10.04LTS with Blackmagic, Nvidia and UltraGrid (STABLE audio version)

Steps:

1.- Download a CD image .iso from http://releases.ubuntu.com/lucid/ and select desktop version i386, like:

ubuntu-10.04.2-desktop-i386.iso

2.- Burn it into a CD and then reboot the computer and boot it from the ubuntu CD.

3.- Select your language and then install ubuntu with ext4 journaling file system and 2GB of swap partition.

NOTE: try to have link to internet while installing the system.

4.- Before installation it is recommended to run X11 as root; so, first enter as normal user, open gnome-ternimal and type:

```
sudo passwd
```

Now type the new UNIX password (for root). And restart Xsession as root.

NOTE: don’t do any update if a pop window asks for it, only sudo passwd and restart.

5.- Install some essential packages:

```
apt-get install libsdl1.2-dev g++ gcc make libmad0-dev libavcodec-dev libpostproc-dev libmpeg2-4-dev libwxgtk2.6-dev libavformat-dev libdvbpsi4-dev libreadline5-dev libfreetype6-dev libjpeg62-dev libpng12-dev freeglut3-dev libxmu-dev libxi-dev xserver-xorg xfce4 ffmpeg libavcodec-dev libmpeg2-4-dev automake1.9 libtool gettext libpostproc-dev libavformat-dev libwxbase2.6-dev xfonts-100dpi libmagick9-dev libglew-dev libxv-dev python python-numarray python-wxgtk2.6 subversion vim bwm-ng ssh
```

(if some packages take problem remove them from that list or try to install the newest one if it is the case)

6.- Download DeckLink driver from http://www.blackmagic-design.com/support/software/ and select linux download link; uncompress and install the decklink .deb package for your system, and the MediaExpress package too.

7.- Install nvidia driver from the menu: System->Administration->Hardware Drivers.

Select the nvidia recommended one.

8.- Before rebooting your system you have to edit the vmalloc space for your kernel system and solve the incompatibility with your capture card and nvidia:

```
vim /boot/grub/grub.cfg
```

Then you need to find the "kernel" line for your current kernel and add *vmalloc=256M* at the end and save.

It should be like this:

```
menuentry 'Ubuntu, amb Linux 2.6.36.4-audio-test' --class ubuntu --class gnu-linux --
        class gnu --class os {
        recordfail
        insmod ext2
        set root='(hd0,10)'
        search --no-floppy --fs-uuid --set 03b57627-5155-4625-ade2-0f500fb6ff97
        linux   /boot/vmlinuz-2.6.36.4-audio-test root=UUID=03b57627-5155-4625-ade2-
        0f500fb6ff97 ro crashkernel=384M-2G:64M,2G-:128M quiet splash vmalloc=256M
        initrd  /boot/initrd.img-2.6.36.4-audio-test
}
```

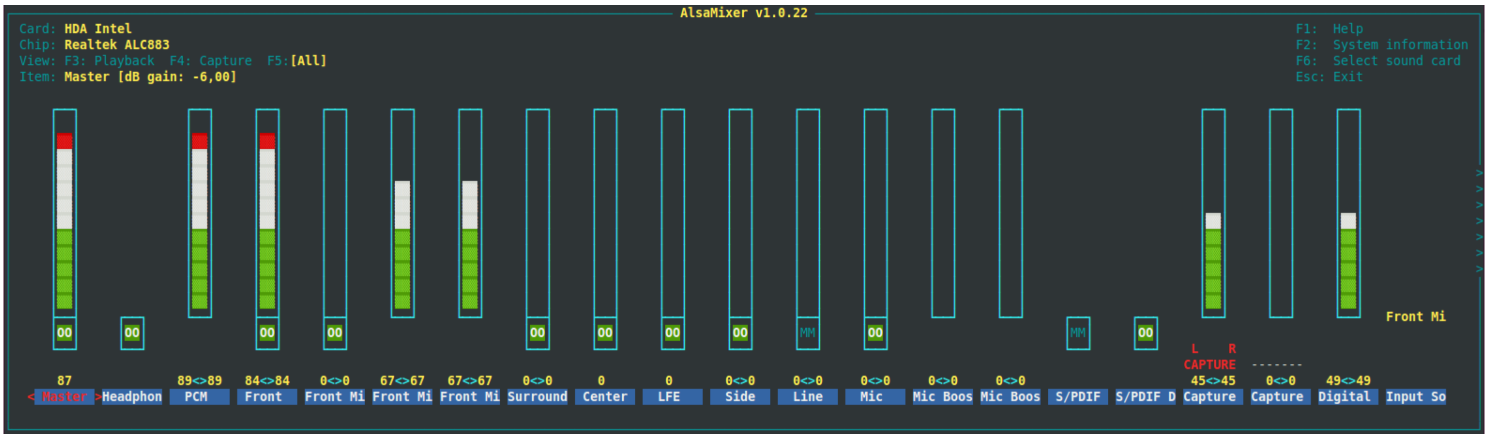
NOTE: don’t *update-grub*, or if you do that be sure you *vmalloc=256M* as done above.

9.- It is needed to add the user root to some system groups (to work with audio properly and only depend on alsamixer). Edit groups and add root, at least, to the audio groups.

```
vim /etc/group
groups: audio, video, plugdev, admin
erase from: pulse-access, pulse
```

Then, a good sound configuration should be like shown in the next figure, typing in the gnome-terminal:

```
alsamixer
```



```
AlsaMixer v1.0.22
Card: HDA Intel                                  F1:  Help
Chip: Realtek ALC883                             F2:  System information
View: F3: Playback  F4: Capture  F5:[All]        F6:  Select sound card
Item: Master [dB gain: -6,00]                    Esc: Exit
```

NOTE: since now you can configure audio levels and configure inputs and outputs from alsamixer only.

10.- Now reboot your system and enter, as always, as root.

If a prompt message is displayed by Blackmagic requiring to update the device firmware you should follow these steps:

```
# BlackmagicFirmwareUpdater
Usage: BlackmagicFirmwareUpdater [COMMAND] [ARGUMENTS]...

COMMANDS:
        status          Check if any cards needs its firmware upgraded
        update [card_id]       Update firmware versions

# BlackmagicFirmwareUpdater status
        blackmagic0 [DeckLink Studio 2]    0x06    PLEASE_UPDATE
# BlackmagicFirmwareUpdater update 0x06
Firmware update completed successfully for device: 'blackmagic0'. Please reboot your system now to activate new firmware
```

If you see the last message shown, now you are able to use blackmagic. So,now reboot please.

11.- Download and prepare ultragrid:

Download the latest repository of the ultragrid subversion from svn.i2cat.net:

```
mkdir  ~/where-you-want-to-work
cd     where-you-want-to-work
svn checkout http://svn.i2cat.net/repos/Ultragrid/UltraGrid
cd UltraGrid
```

sh setup-compression.sh

vim setup.sh (edit --enable-you-need line as is shown in some examples in that file, at the end of setup.sh)

To use with blackmagic and alsa type: *--enable-blackmagic --enable-alsa*

sh setup.sh

cd ultragrid/bin

./init.sh <mtu> <txqueuelen> ethX

cd ..

IN TX:

bin/uv -t blackmagic -b8 -B8 (or -B9 …. it depends on the capture mode) *<destination address> -M4 -A0 -a2*

IN RX:

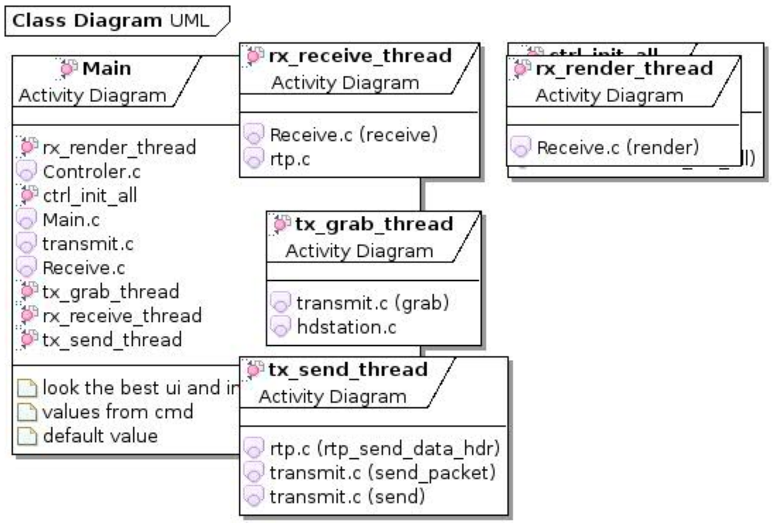
bin/uv -d sdl -b8 <sender address> -M4 -A0 -a1

NOW YOU ARE READY TO WORK WITH ULTRAGRID, BLACKMAGIC AND NVIDIA FOR SURE (be quite with sound connections and alsamixer).

NOTE: ANY ERROR ON ANYTHING ELSE DETECTED CONTACT WITH gerard.castillo@i2cat.net PLEASE!

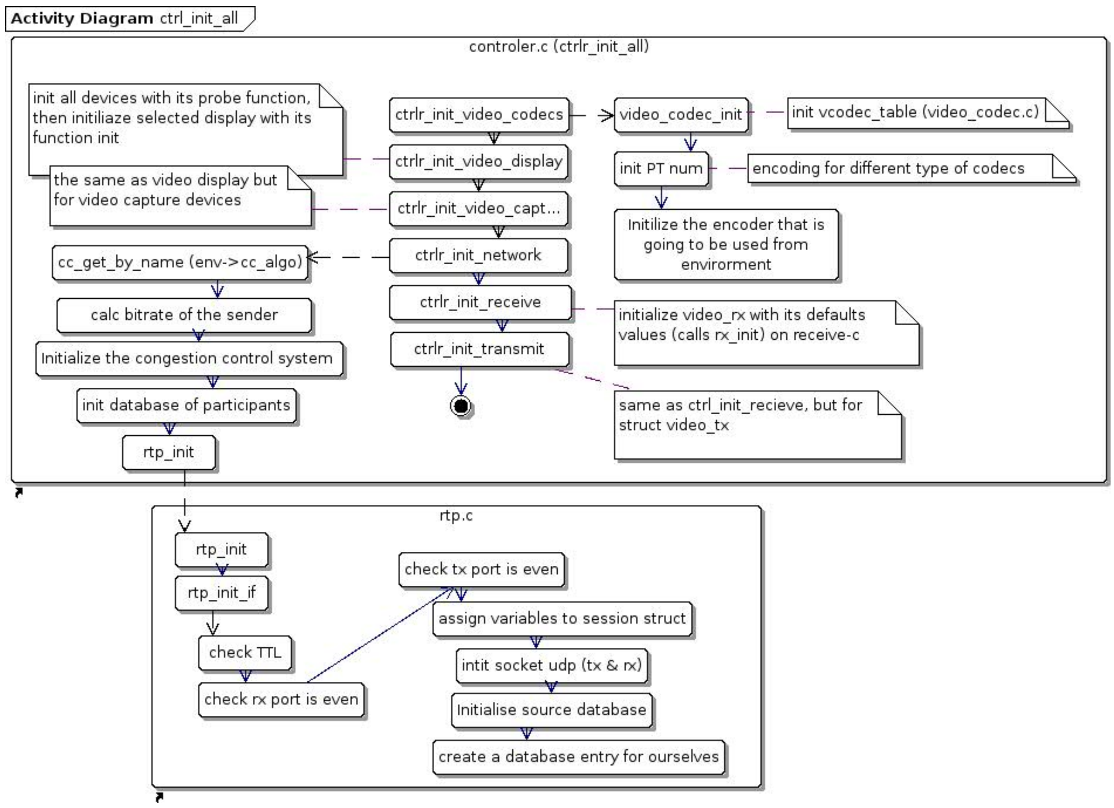
ANNEX E. Basic and initial UML

The first figure shown has the global UltraGrid structure:



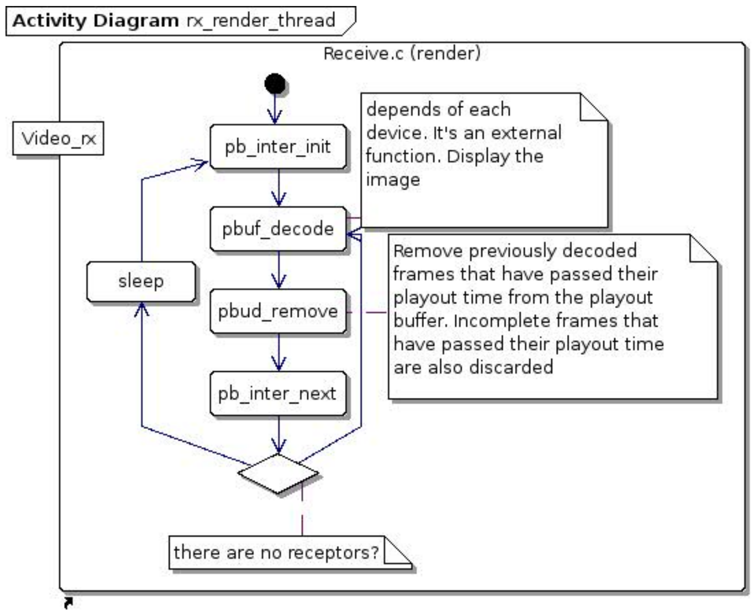
General UltraGrid UML

UltraGrid initializes in that order and then the threads have the global execution until the end. First of all it starts with the render thread to activate the decodification, followed by the start up of all the devices and parameters that will be used in execution time. Then, loop with render, grab, receive and send thread is done in that order.


Activity Diagram ctrl_init_all
controler.c (ctrlr_init_all)
init all devices with its probe function, then initiliaze selected display with its function init
the same as video display but for video capture devices
cc_get_by_name (env->cc_algo)
calc bitrate of the sender
Initialize the congestion control system
init database of participants
rtp_init
ctrlr_init_video_codecs
ctrlr_init_video_display
ctrlr_init_video_capt...
ctrlr_init_network
ctrlr_init_receive
ctrlr_init_transmit
video_codec_init
init PT num
Initilize the encoder that is going to be used from enviroment
init vcodec_table (video_codec.c)
encoding for different type of codecs
initialize video_rx with its defaults values (calls rx_init) on receive-c
same as ctrl_init_recieve, but for struct video_tx
rtp.c
rtp_init
rtp_init_if
check TTL
check rx port is even
check tx port is even
assign variables to session struct
intit socket udp (tx & rx)
Initialise source database
create a database entry for ourselves

Initialization UML

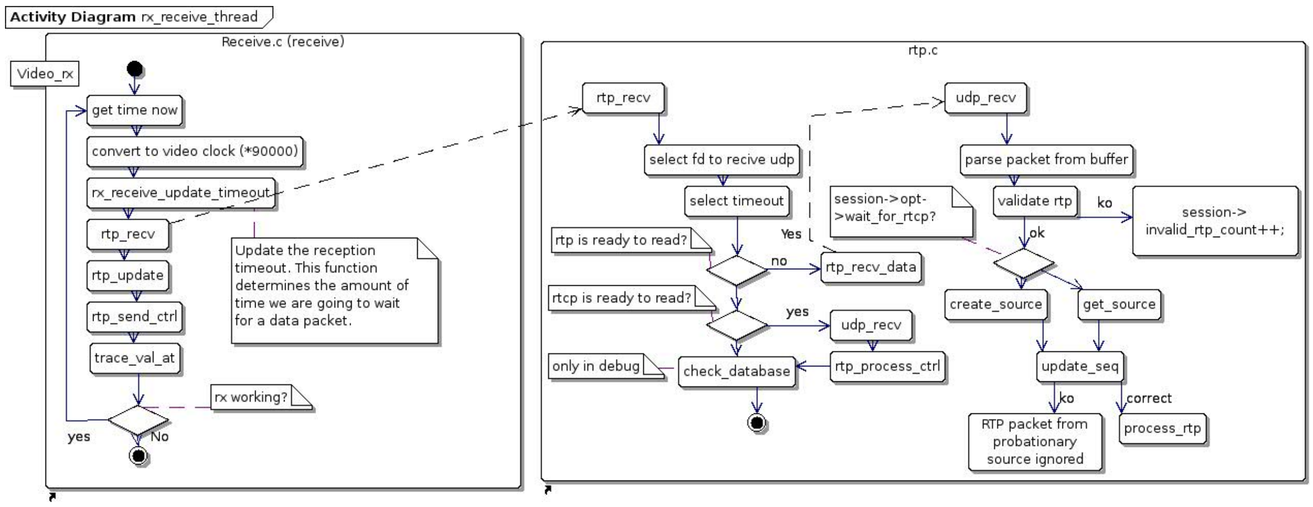
The general initialization starts up with the devices, displaying/decoding or capturing/encoding, followed by the network initialization that configures the RTP and RTCP parameters (for example, for having a loss-packet control) and its relation to work with data to receive or transmit (reception/transmitting ports, sockets UDP, session database, MTU…)



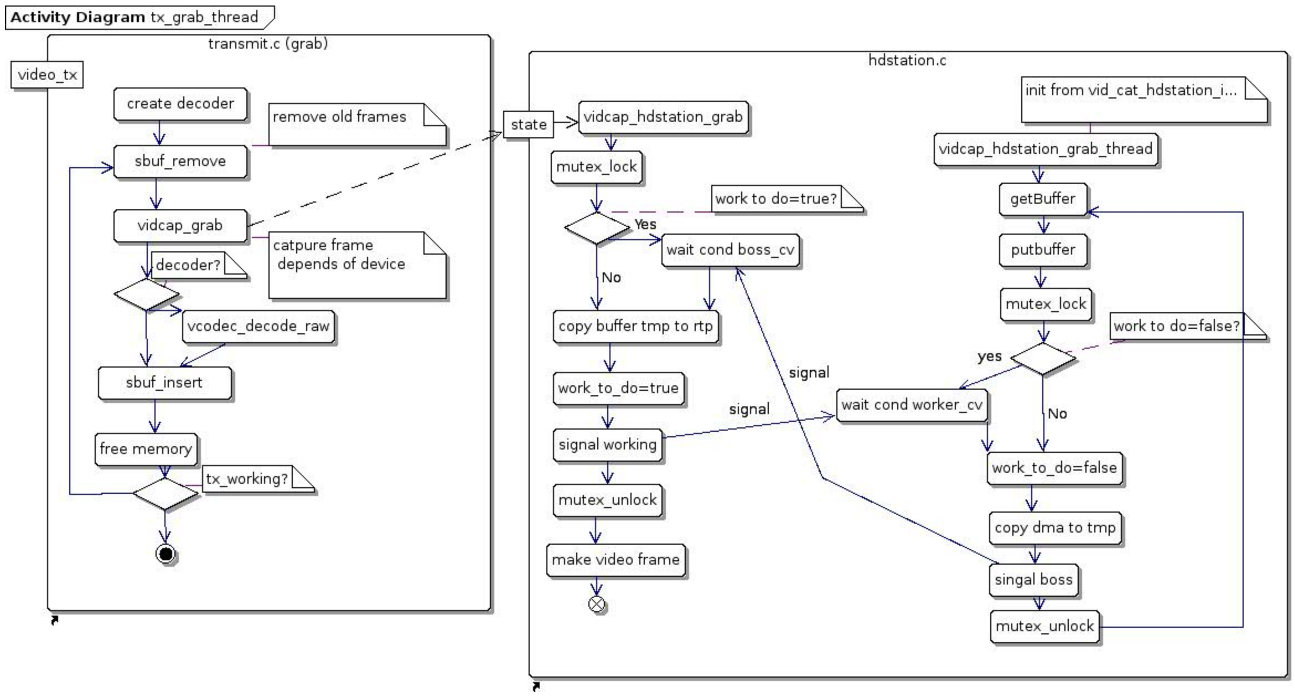
Activity Diagram rx_render_thread
Receive.c (render)
Video_rx
pb_inter_init
pbuf_decode
pbud_remove
pb_inter_next
sleep
depends of each device. It's an external function. Display the image
Remove previously decoded frames that have passed their playout time from the playout buffer. Incomplete frames that have passed their playout time are also discarded
there are no receptors?

Receive render thread UML

The render thread is the responsible for the data treatment from the network, decoding it, to the display. It capture packets from the net, process the headers and is going on creating the frames, both video and audio. When frame is complete is time to put it to the display. This thread is directly related to rx_receive_thread that takes control from receiving packets from the net to generating RTCP data.

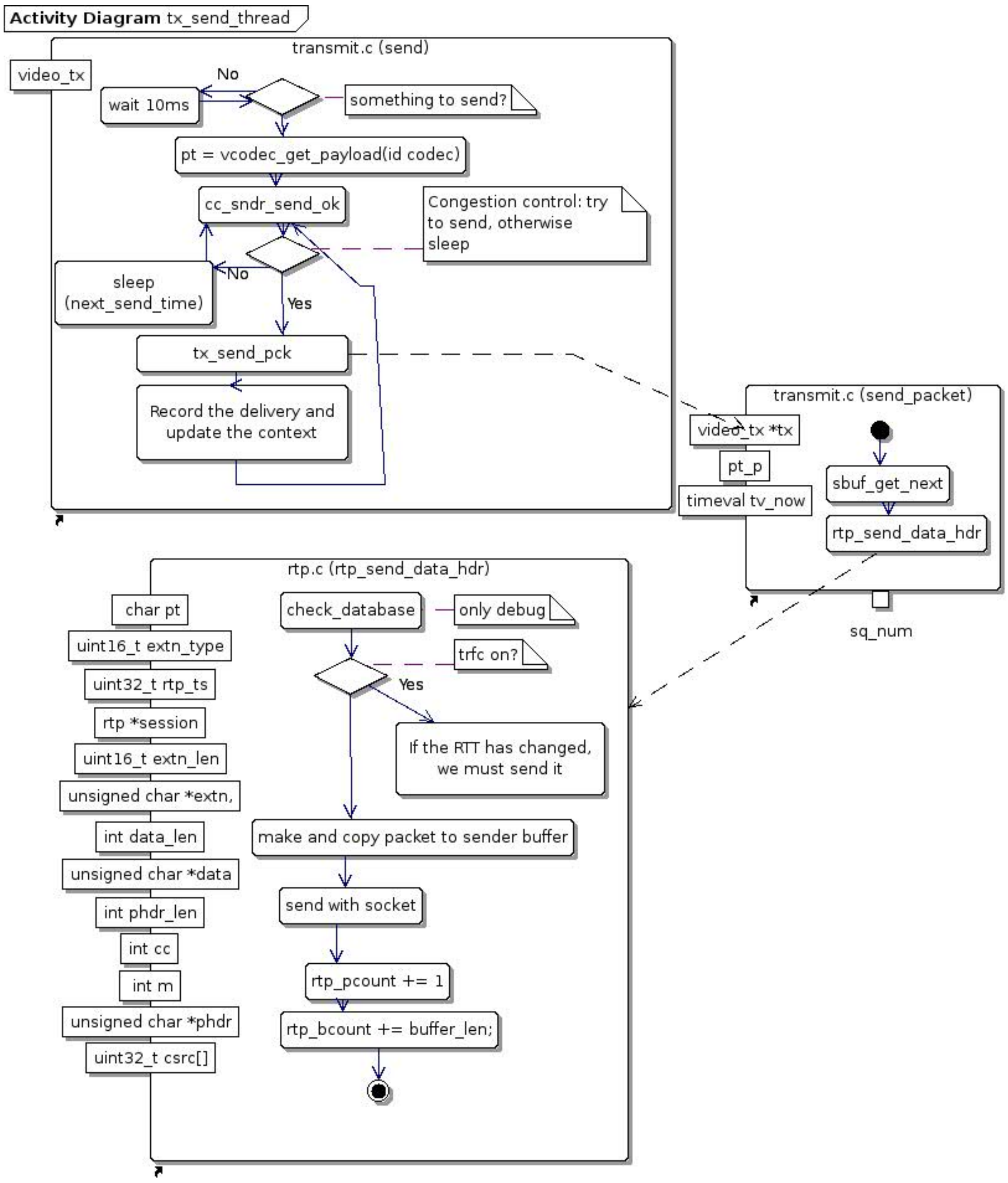


General receive thread UML



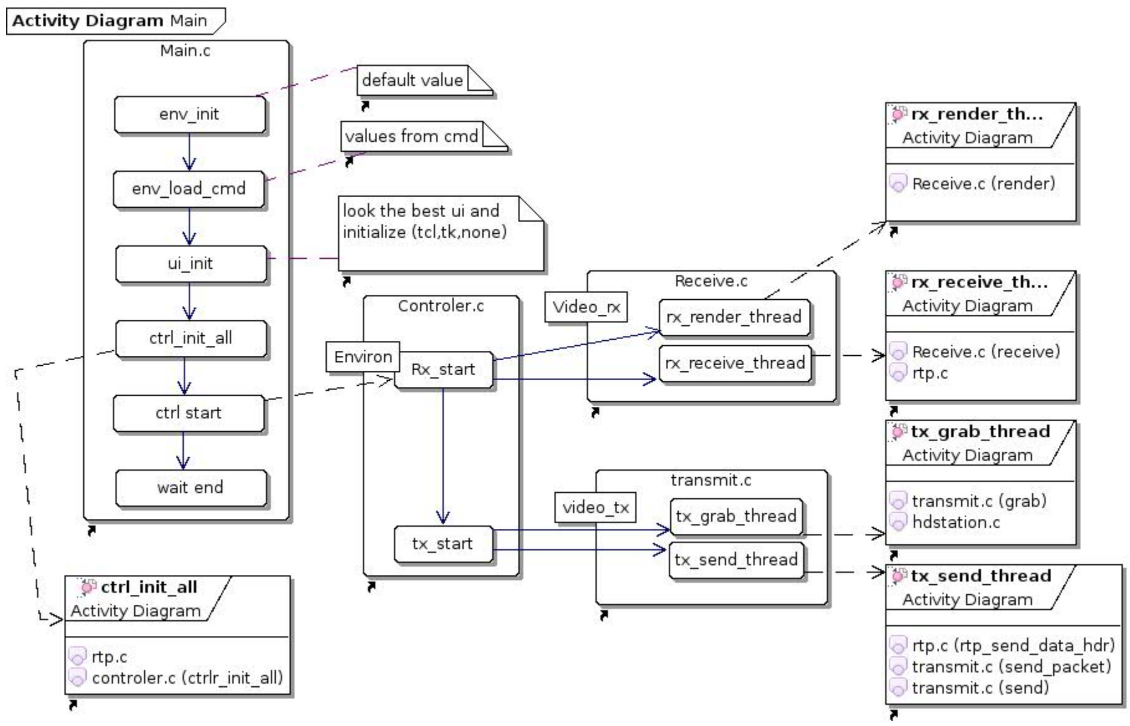
Transmit grab thread UML

This thread takes part into the whole loop from capturing from the camera to the network buffer. Here it is important to take into account of the audio data capture and its synchronization with video stream generation and transmission.



General trasmit thread UML

This thread is in charge of putting every frame generated and its payload into the RTP layer, fragmenting if it is necessary. Could be called as an RTP-transmit module.



Main execution UML

The ‘Main’ activity diagram shows more over the whole UltraGrid execution order, and how it is done.

ANNEX F. ALSA LISTING, OPTIONS, PLAYBACK AND CAPTURE EXAMPLES

Listing: permits to know the available parameters of the ALSA library into our kernel.

```
#include </usr/include/alsa/asoundlib.h>
int main() {
  int val;

  printf("ALSA library version: %s\n",
         SND_LIB_VERSION_STR);

  printf("\nPCM stream types:\n");
  for (val = 0; val <= SND_PCM_STREAM_LAST; val++)
    printf("  %s\n",
      snd_pcm_stream_name((snd_pcm_stream_t)val));

  printf("\nPCM access types:\n");
  for (val = 0; val <= SND_PCM_ACCESS_LAST; val++)
    printf("  %s\n",
      snd_pcm_access_name((snd_pcm_access_t)val));

  printf("\nPCM formats:\n");
  for (val = 0; val <= SND_PCM_FORMAT_LAST; val++)
    if (snd_pcm_format_name((snd_pcm_format_t)val)
      != NULL)
      printf("  %s (%s)\n",
        snd_pcm_format_name((snd_pcm_format_t)val),
        snd_pcm_format_description(
                   (snd_pcm_format_t)val));

  printf("\nPCM subformats:\n");
  for (val = 0; val <= SND_PCM_SUBFORMAT_LAST;
      val++)
    printf("  %s (%s)\n",
      snd_pcm_subformat_name((
        snd_pcm_subformat_t)val),
      snd_pcm_subformat_description((
        snd_pcm_subformat_t)val));

  printf("\nPCM states:\n");
  for (val = 0; val <= SND_PCM_STATE_LAST; val++)
    printf("  %s\n",
         snd_pcm_state_name((snd_pcm_state_t)val));

  return 0;
}
```

Opening: This example opens the default PCM device, sets some parameters, and then displays the value of most of the hardware parameters. It does not perform any sound playback or recording.

```
/* Use the newer ALSA API */
#define ALSA_PCM_NEW_HW_PARAMS_API

/* All of the ALSA library API is defined
 * in this header */
#include <alsa/asoundlib.h>

int main() {
  int rc;
  snd_pcm_t *handle;
  snd_pcm_hw_params_t *params;
  unsigned int val, val2;
  int dir;
  snd_pcm_uframes_t frames;

  /* Open PCM device for playback. */
  rc = snd_pcm_open(&handle, "default",
                    SND_PCM_STREAM_PLAYBACK, 0);
  if (rc < 0) {
    fprintf(stderr,
            "unable to open pcm device: %s\n",
            snd_strerror(rc));
    exit(1);
  }

  /* Allocate a hardware parameters object. */
  snd_pcm_hw_params_alloca(&params);

  /* Fill it in with default values. */
  snd_pcm_hw_params_any(handle, params);

  /* Set the desired hardware parameters. */

  /* Interleaved mode */
  snd_pcm_hw_params_set_access(handle, params,
                      SND_PCM_ACCESS_RW_INTERLEAVED);

  /* Signed 16-bit little-endian format */
  snd_pcm_hw_params_set_format(handle, params,
                              SND_PCM_FORMAT_S16_LE);

  /* Two channels (stereo) */
  snd_pcm_hw_params_set_channels(handle, params, 2);

  /* 44100 bits/second sampling rate (CD quality) */
  val = 8000;
  snd_pcm_hw_params_set_rate_near(handle,
                         params, &val, &dir);

  /* Write the parameters to the driver */
  rc = snd_pcm_hw_params(handle, params);
  if (rc < 0) {
    fprintf(stderr,
            "unable to set hw parameters: %s\n",
            snd_strerror(rc));
    exit(1);
  }

  /* Display information about the PCM interface */

  printf("PCM handle name = '%s'\n",
         snd_pcm_name(handle));

  printf("PCM state = %s\n",
         snd_pcm_state_name(snd_pcm_state(handle)));

  snd_pcm_hw_params_get_access(params,
                    (snd_pcm_access_t *) &val);
  printf("access type = %s\n",
         snd_pcm_access_name((snd_pcm_access_t)val));

  snd_pcm_hw_params_get_format(params, &val);
  printf("format = '%s' (%s)\n",
    snd_pcm_format_name((snd_pcm_format_t)val),
    snd_pcm_format_description(
                      (snd_pcm_format_t)val));

  snd_pcm_hw_params_get_subformat(params,
                  (snd_pcm_subformat_t *)&val);
  printf("subformat = '%s' (%s)\n",
    snd_pcm_subformat_name((snd_pcm_subformat_t)val),
    snd_pcm_subformat_description(
                    (snd_pcm_subformat_t)val));

  snd_pcm_hw_params_get_channels(params, &val);
  printf("channels = %d\n", val);

  snd_pcm_hw_params_get_rate(params, &val, &dir);
  printf("rate = %d bps\n", val);

  snd_pcm_hw_params_get_period_time(params,
                           &val, &dir);
  printf("period time = %d us\n", val);

  snd_pcm_hw_params_get_period_size(params,
                           &frames, &dir);
  printf("period size = %d frames\n", (int)frames);
```

```
snd_pcm_hw_params_get_buffer_time(params,
                                  &val, &dir);
printf("buffer time = %d us\n", val);

snd_pcm_hw_params_get_buffer_size(params,
                       (snd_pcm_uframes_t *) &val);
printf("buffer size = %d frames\n", val);

snd_pcm_hw_params_get_periods(params, &val, &dir);
printf("periods per buffer = %d frames\n", val);

snd_pcm_hw_params_get_rate_numden(params,
                                  &val, &val2);
printf("exact rate = %d/%d bps\n", val, val2);

val = snd_pcm_hw_params_get_sbits(params);
printf("significant bits = %d\n", val);

snd_pcm_hw_params_get_tick_time(params,
                                &val, &dir);
printf("tick time = %d us\n", val);

val = snd_pcm_hw_params_is_batch(params);
printf("is batch = %d\n", val);

val = snd_pcm_hw_params_is_block_transfer(params);
printf("is block transfer = %d\n", val);

val = snd_pcm_hw_params_is_double(params);
printf("is double = %d\n", val);

val = snd_pcm_hw_params_is_half_duplex(params);
printf("is half duplex = %d\n", val);

val = snd_pcm_hw_params_is_joint_duplex(params);
printf("is joint duplex = %d\n", val);

val = snd_pcm_hw_params_can_overrange(params);
printf("can overrange = %d\n", val);

val = snd_pcm_hw_params_can_mmap_sample_resolution(params);
printf("can mmap = %d\n", val);

val = snd_pcm_hw_params_can_pause(params);
printf("can pause = %d\n", val);

val = snd_pcm_hw_params_can_resume(params);
printf("can resume = %d\n", val);
```

```
  val = snd_pcm_hw_params_can_sync_start(params);
  printf("can sync start = %d\n", val);

  snd_pcm_close(handle);

  return 0;
}
```

Playback: This example reads standard from input and writes to the default PCM device for 5 seconds of data.

```
/* Use the newer ALSA API */
#define ALSA_PCM_NEW_HW_PARAMS_API

#include <alsa/asoundlib.h>

int main() {
  long loops;
  int rc;
  int size;
  snd_pcm_t *handle;
  snd_pcm_hw_params_t *params;
  unsigned int val;
  int dir;
  snd_pcm_uframes_t frames;
  char *buffer;

  /* Open PCM device for playback. */
  rc = snd_pcm_open(&handle, "default",
                    SND_PCM_STREAM_PLAYBACK, 0);
  if (rc < 0) {
    fprintf(stderr,
            "unable to open pcm device: %s\n",
            snd_strerror(rc));
    exit(1);
  }

  /* Allocate a hardware parameters object. */
  snd_pcm_hw_params_alloca(&params);

  /* Fill it in with default values. */
  snd_pcm_hw_params_any(handle, params);

  /* Set the desired hardware parameters. */

  /* Interleaved mode */
  snd_pcm_hw_params_set_access(handle, params,
                      SND_PCM_ACCESS_RW_INTERLEAVED);
```

```
/* Signed 16-bit little-endian format */
snd_pcm_hw_params_set_format(handle, params,
                         SND_PCM_FORMAT_S16_LE);

/* Two channels (stereo) */
snd_pcm_hw_params_set_channels(handle, params, 2);

/* 44100 bits/second sampling rate (CD quality) */
val = 44100;
snd_pcm_hw_params_set_rate_near(handle, params,
                           &val, &dir);

/* Set period size to 32 frames. */
frames = 32;
ssize_t Bpf= snd_pcm_frames_to_bytes(handle,frames);
printf("\n Bpf=%d \n",Bpf);
snd_pcm_hw_params_set_period_size_near(handle,
                         params, &frames, &dir);

/* Write the parameters to the driver */
rc = snd_pcm_hw_params(handle, params);
if (rc < 0) {
  fprintf(stderr,
          "unable to set hw parameters: %s\n",
          snd_strerror(rc));
  exit(1);
}

/* Use a buffer large enough to hold one period */
snd_pcm_hw_params_get_period_size(params, &frames,
                            &dir);
size = frames * 4; /* 2 bytes/sample, 2 channels */
buffer = (char *) malloc(size);

/* We want to loop for 5 seconds */
snd_pcm_hw_params_get_period_time(params,
                            &val, &dir);
/* 5 seconds in microseconds divided by
 * period time */
loops = 5000000 / val;

while (loops > 0) {
      //printf("playback satarted");
  loops--;
  rc = read(0, buffer, size);
  if (rc == 0) {
    fprintf(stderr, "end of file on input\n");
    break;
  } else if (rc != size) {
      fprintf(stderr,
              "short read: read %d bytes\n", rc);
    }
    rc = snd_pcm_writei(handle, buffer, frames);
    if (rc == -EPIPE) {
      /* EPIPE means underrun */
      fprintf(stderr, "underrun occurred\n");
      snd_pcm_prepare(handle);
    } else if (rc < 0) {
      fprintf(stderr,
              "error from writei: %s\n",
              snd_strerror(rc));
    }  else if (rc != (int)frames) {
      fprintf(stderr,
              "short write, write %d frames\n", rc);
    }
  }

  snd_pcm_drain(handle);
  snd_pcm_close(handle);
  free(buffer);

  return 0;
}
```

Capture: This example reads from the default PCM device and writes to standard output for 5 seconds of data.

```
/* Use the newer ALSA API */
#define ALSA_PCM_NEW_HW_PARAMS_API

#include <alsa/asoundlib.h>

int main() {
  long loops;
  int rc,err;
  int size;
  snd_pcm_t *handle;
  snd_pcm_hw_params_t *params;
  unsigned int val;
  int dir;
  snd_pcm_uframes_t frames;
  char *buffer;

  /* Open PCM device for recording (capture). */
  rc = snd_pcm_open(&handle,"default",SND_PCM_STREAM_CAPTURE, 0);
  if (rc < 0) {
    fprintf(stderr,"unable to open pcm device: %s\n",snd_strerror(rc));
    exit(1);
  }

  /* Allocate a hardware parameters object. */
  snd_pcm_hw_params_alloca(&params);

  /* Fill it in with default values. */
  err = snd_pcm_hw_params_any(handle, params);
  if (err < 0) {
     printf("Broken configuration for this PCM: no configurations available");
     exit(EXIT_FAILURE);
  }

  /* Set the desired hardware parameters. */
  /* Interleaved mode */
  snd_pcm_hw_params_set_access(handle,
params,SND_PCM_ACCESS_RW_INTERLEAVED);

  /* Signed 16-bit little-endian format */
  snd_pcm_hw_params_set_format(handle,
params,SND_PCM_FORMAT_S16_LE);

  /* Two channels (stereo) */
  snd_pcm_hw_params_set_channels(handle, params, 2);

  /* 44100 bits/second sampling rate (CD quality) */
  val = 48000;
  snd_pcm_hw_params_set_rate_near(handle, params, &val, &dir);

  /* Set period size to 1024 frames. */
  frames = 1920;
  snd_pcm_hw_params_set_period_size_near(handle,params, &frames, &dir);

 /* Write the parameters to the driver */
  rc = snd_pcm_hw_params(handle, params);
  if (rc < 0) {
    fprintf(stderr,"unable to set hw parameters: %s\n", snd_strerror(rc));
    exit(1);
  }

  /* Use a buffer large enough to hold one period */
  snd_pcm_hw_params_get_period_size(params,&frames, &dir);

  size = frames * 4; /* 2 bytes/sample, 2 channels(samples/frame) */

  buffer = (char *) malloc(size);

  /* We want to loop for 5 seconds */
  snd_pcm_hw_params_get_period_time(params,&val, &dir);
  loops = 5000000 / val;

  while (loops > 0) {
    loops--;
    rc = snd_pcm_readi(handle, buffer, frames);
    if (rc == -EPIPE) {
      /* EPIPE means overrun */
      fprintf(stderr, "overrun occurred\n");
      snd_pcm_prepare(handle);
    } else if (rc < 0) {
      fprintf(stderr,
              "error from read: %s\n",
              snd_strerror(rc));
    } else if (rc != (int)frames) {
      fprintf(stderr, "short read, read %d frames\n", rc);
    }
    rc = write(1, buffer, size);
    if (rc != size)
      fprintf(stderr,"short write: wrote %d bytes\n", rc);
  }

  snd_pcm_drain(handle);
  snd_pcm_close(handle);
  free(buffer);

  return 0;
}
```

ANNEX G. Configuring ALSA plug-in

First of all let’s create the .asoundrc file that is read by ALSA library when opens and searchs for devices.
This may be localized into the default user directory, so in a terminal type the following commands:

```
#vim ~/.asoundrc
```

When executing this command you are creating this hidden file, if it doesn’t exists. Then with vim let’s write the code to generate the plug-in needed:

```
pcm.alawUG {
        type alaw
        slave {
                pcm “default”
                format defaultFORMAT
        }
}

pcm.ADpcmUG {
        type adpcm
        slave {
                pcm “default”
                format defaultFORMAT
        }
}
```

→ Sample rate and format must be the same for both master and slave.

This file is creating a plug-in with A-law and IMA_ADPCM that takes as slave the default device configured into ALSAs library.

Now, to work with this generated plug-ins it must be done:

- Open the desired plug-in:

```
snd_pcm_open(handle, "alawUG", SND_PCM_STREAM_CAPTURE, 0);
```

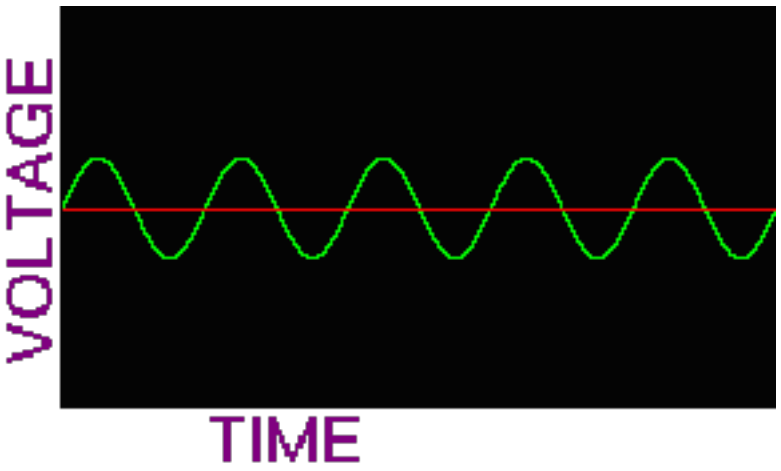
- Set the working sample format:

```
snd_pcm_hw_params_set_format( handle, params,
SND_PCM_FORMAT_A_LAW);
```

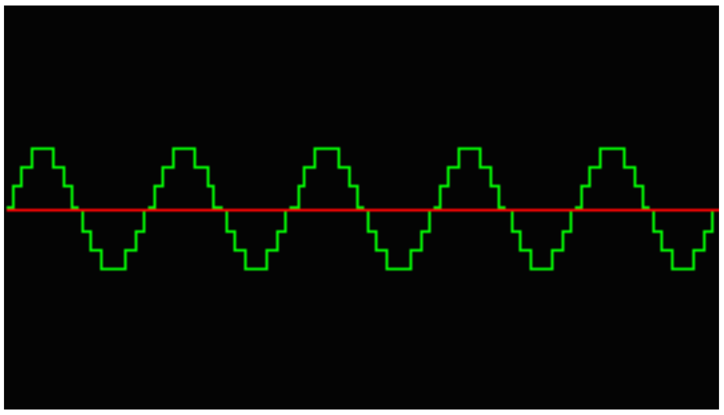
ANNEX H. PCM (digital audio) interface

From ALSA project web site – the C library reference

Although abbreviation PCM stands for Pulse Code Modulation, we are understanding it as general digital audio processing with volume samples generated in continuous time periods.
The analog signal is recorded via analog to digital converters (ADC). The digital value (de-facto a volume at a specific time) obtained from ADC can be further processed. The following picture shows a perfect sinus waveform:



Next image shows digitized representation:



As you may see, the quality of digital audio signal depends on the time (recording rate) and voltage resolution (usually in an linear integer representation with basic unit one bit).
The stored digital signal can be converted back to voltage (analog) representation via digital to analog converters (DAC).
One digital value is called sample. More samples are collected to frames (frame is terminology for ALSA) depending on count of converters used at one specific time. One frame might contain one sample (when only one converter is used - mono) or more samples (for example: stereo has signals from two converters recorded at same time). Digital audio stream contains collection of frames recorded at boundaries of continuous time periods.

General overview
ALSA uses the ring buffer to store outgoing (playback) and incoming (capture, record) samples. There are two pointers being maintained to allow a precise communication between application and device pointing to current processed sample by hardware and last processed sample by application. The modern audio chips allow to program the transfer time periods. It means that the stream of samples is divided to small chunks. Device acknowledges to application when the transfer of a chunk is complete.

Transfer methods in UNIX environments
In the UNIX environment, data chunk acknowledges are received via standard I/O calls or event waiting routines (poll or select function). To accomplish this list, the asynchronous notification of acknowledges should be listed here. The ALSA implementation for these methods is described in the ALSA transfers section.

Standard I/O transfers
The standard I/O transfers are using the read (see 'man 2 read') and write (see 'man 2 write') C functions. There are two basic behaviours of these functions - blocked and non-blocked (see the O_NONBLOCK flag for the standard C open function - see 'man 2 open'). In non-blocked behaviour, these I/O functions never stops, they return -EAGAIN error code, when no data can be transferred (the ring buffer is full in our case). In blocked behaviour, these I/O functions stop and wait until there is a room in the ring buffer (playback) or until there are a new samples (capture). The ALSA implementation can be found in the Read / Write transfer section.

Event waiting routines
The poll or select functions (see 'man 2 poll' or 'man 2 select' for further details) allows to receive requests/events from the device while an application is waiting on events from other sources (like keyboard, screen, network etc.), too. snd_pcm_poll_descriptors can be used to get file descriptors to poll or select on (note that wait direction might be diferent than expected - do not use only returned file descriptors, but handle events member as well - see snd_pcm_poll_descriptors function description for more details and snd_pcm_poll_descriptors_revents for events demangling). The implemented transfer routines can be found in the ALSA transfers section.

Asynchronous notification
ALSA driver and library knows to handle the asynchronous notifications over the SIGIO signal. This signal allows to interrupt application and transfer data in the signal handler. For further details see the sigaction function ('man 2 sigaction'). The section Asynchronous mode describes the ALSA API for this extension. The implemented transfer routines can be found in the ALSA transfers section.

Blocked and non-blocked open
The ALSA PCM API uses a different behaviour when the device is opened with blocked or non-blocked mode. The mode can be specified with *mode* argument in snd_pcm_open() function. The blocked mode is the default (without SND_PCM_NONBLOCK mode). In this mode, the behaviour is that if the

resources have already used with another application, then it blocks the caller, until resources are free. The non-blocked behaviour (with SND_PCM_NONBLOCK) doesn't block the caller in any way and returns -EBUSY error when the resources are not available. Note that the mode also determines the behaviour of standard I/O calls, returning -EAGAIN when non-blocked mode is used and the ring buffer is full (playback) or empty (capture). The operation mode for I/O calls can be changed later with the snd_pcm_nonblock() function.

Asynchronous mode
There is also possibility to receive asynchronous notification after specified time periods. You may see the SND_PCM_ASYNC mode for snd_pcm_open() function and snd_async_add_pcm_handler() function for further details.

Handshake between application and library
The ALSA PCM API design uses the states to determine the communication phase between application and library. The actual state can be determined using snd_pcm_state() call. There are these states:

SND_PCM_STATE_OPEN
: The PCM device is in the open state. After the snd_pcm_open() open call, the device is in this state. Also, when snd_pcm_hw_params() call fails, then this state is entered to force application calling snd_pcm_hw_params() function to set right communication parameters.

SND_PCM_STATE_SETUP
: The PCM device has accepted communication parameters and it is waiting for snd_pcm_prepare() call to prepare the hardware for selected operation (playback or capture).

SND_PCM_STATE_PREPARE
: The PCM device is prepared for operation. Application can use snd_pcm_start() call, write or read data to start the operation.

SND_PCM_STATE_RUNNING
: The PCM device has been started and is running. It processes the samples. The stream can be stopped using the snd_pcm_drop() or snd_pcm_drain() calls.

SND_PCM_STATE_XRUN
: The PCM device reached overrun (capture) or underrun (playback). You can use the -EPIPE return code from I/O functions (snd_pcm_writei(), snd_pcm_writen(), snd_pcm_readi(), snd_pcm_readn()) to determine this state without checking the actual state via snd_pcm_state() call. It is recommended to use the helper function snd_pcm_recover() to recover from this state, but you can also use snd_pcm_prepare(), snd_pcm_drop() or snd_pcm_drain() calls.

SND_PCM_STATE_DRAINING
: The device is in this state when application using the capture mode called snd_pcm_drain() function. Until all data are read from the internal ring buffer using I/O routines (snd_pcm_readi(), snd_pcm_readn()), then the device stays in this state.

SND_PCM_STATE_PAUSED
: The device is in this state when application called the snd_pcm_pause() function until the pause is released. Not all hardware supports this

feature. Application should check the capability with the snd_pcm_hw_params_can_pause().

SND_PCM_STATE_SUSPENDED

The device is in the suspend state provoked with the power management system. The stream can be resumed using snd_pcm_resume() call, but not all hardware supports this feature. Application should check the capability with the snd_pcm_hw_params_can_resume(). In other case, the calls snd_pcm_prepare(), snd_pcm_drop(), snd_pcm_drain() can be used to leave this state.

SND_PCM_STATE_DISCONNECTED

The device is physicaly disconnected. It does not accept any I/O calls in this state.

PCM formats

The full list of formats present the snd_pcm_format_t type. The 24-bit linear samples uses 32-bit physical space, but the sample is stored in low three bits. Some hardware does not support processing of full range, thus you may get the significant bits for linear samples via snd_pcm_hw_params_get_sbits() function. The example: ICE1712 chips support 32-bit sample processing, but low byte is ignored (playback) or zero (capture). The function snd_pcm_hw_params_get_sbits() returns 24 in the case.

ALSA transfers

There are two methods to transfer samples in application. The first method is the standard read / write one. The second method, uses the direct audio buffer to communicate with the device while ALSA library manages this space itself. You can find examples of all communication schemes for playback in Sine-wave generator example. To complete the list, we should note that snd_pcm_wait() function contains embedded poll waiting implementation.

Read / Write transfer

There are two versions of read / write routines. The first expects the interleaved samples at input (SND_PCM_ACCESS_RW_INTERLEAVED access method), and the second one expects non-interleaved (samples in separated buffers - SND_PCM_ACCESS_RW_NONINTERLEAVED access method) at input. There are these functions for interleaved transfers: snd_pcm_writei() snd_pcm_readi(). For non-interleaved transfers, there are these functions: snd_pcm_writen() and snd_pcm_readn().

Direct Read / Write transfer (via mmap'ed areas)

Three kinds of organization of ring buffer memory areas exist in ALSA API. Access SND_PCM_ACCESS_MMAP_INTERLEAVED has interleaved samples. Access SND_PCM_ACCESS_MMAP_NONINTERLEAVED expects continous sample areas for one channel. Access SND_PCM_ACCESS_MMAP_COMPLEX does not fit to interleaved and non-interleaved ring buffer organization.

There are two functions for this kind of transfer. Application can get an access to memory areas via snd_pcm_mmap_begin() function. This function returns the areas (single area is equal to a channel) containing the direct pointers to memory and sample position description in snd_pcm_channel_area_t structure.

After application transfers the data in the memory areas, then it must be acknowledged the end of transfer via snd_pcm_mmap_commit() function to allow the ALSA library update the pointers to ring buffer. This kind of communication is also called "zero-copy", because the device does not require to copy the samples from application to another place in system memory.
If you like to use the compatibility functions in mmap mode, there are read / write routines equaling to standard read / write transfers. Using these functions discards the benefits of direct access to memory region. See the snd_pcm_mmap_readi(), snd_pcm_writei(), snd_pcm_readn() and snd_pcm_writen() functions.

Error codes
-EPIPE
This error means xrun (underrun for playback or overrun for capture). The underrun can happen when an application does not feed new samples in time to alsa-lib (due CPU usage). The overrun can happen when an application does not take new captured samples in time from alsa-lib.
-ESTRPIPE
This error means that system has suspended drivers. The application should wait in loop when snd_pcm_resume() != -EAGAIN and then call snd_pcm_prepare() when snd_pcm_resume() return an error code. If snd_pcm_resume() does not fail (a zero value is returned), driver supports resume and the snd_pcm_prepare() call can be ommited.
-EBADFD
This error means that the device is in a bad state. It means that the handskahe between application and alsa-lib is corrupted.
-ENOTTY, -ENODEV
This error can happen when device is physically removed (for example some hotplug devices like USB or PCMCIA, CardBus or ExpressCard can be removed on the fly).

Managing parameters
The ALSA PCM device uses two groups of PCM related parameters. The hardware parameters contains the stream description like format, rate, count of channels, ring buffer size etc. The software parameters contains the software (driver) related parameters. The communication behaviour can be controlled via these parameters, like automatic start, automatic stop, interrupting (chunk acknowledge) etc. The software parameters can be modified at any time (when valid hardware parameters are set). It includes the running state as well.

Hardware related parameters
The ALSA PCM devices use the parameter refining system for hardware parameters - snd_pcm_hw_params_t. It means, that application choose the full-range of configurations at first and then application sets single parameters until all parameters are elementary (definite).

Access modes
ALSA knows about five access modes. The first three can be used for direct communication. The access mode SND_PCM_ACCESS_MMAP_INTERLEAVED determines the direct memory

area and interleaved sample organization. Interleaved organization means, that samples from channels are mixed together. The access mode SND_PCM_ACCESS_MMAP_NONINTERLEAVED determines the direct memory area and non-interleaved sample organization. Each channel has a separate buffer in the case. The complex direct memory organization represents the SND_PCM_ACCESS_MMAP_COMPLEX access mode. The sample organization does not fit the interleaved or non-interleaved access modes in the case. The last two access modes describes the read / write access methods. The SND_PCM_ACCESS_RW_INTERLEAVED access represents the read / write interleaved access and the SND_PCM_ACCESS_RW_NONINTERLEAVED represents the non-interleaved access.

Formats
The full list of formats is available in snd_pcm_format_t enumeration.

Software related parameters
These parameters - snd_pcm_sw_params_t can be modified at any time including the running state.

Minimum available count of samples
This parameter controls the wakeup point. If the count of available samples is equal or greater than this value, then application will be activated.

Timestamp mode
The timestamp mode specifies, if timestamps are activated. Currently, only SND_PCM_TSTAMP_NONE and SND_PCM_TSTAMP_MMAP modes are known. The mmap mode means that timestamp is taken on every period time boundary. Corresponding position in the ring buffer assigned to timestamp can be obtained using snd_pcm_htimestamp() function.

Transfer align
The read / write transfers can be aligned to this sample count. The modulo is ignored by device. Usually, this value is set to one (no align).

Start threshold
The start threshold parameter is used to determine the start point in stream. For playback, if samples in ring buffer is equal or greater than the start threshold parameters and the stream is not running, the stream will be started automatically from the device. For capture, if the application wants to read count of samples equal or greater then the stream will be started. If you want to use explicit start (snd_pcm_start), you can set this value greater than ring buffer size (in samples), but use the constant MAXINT is not a bad idea.

Stop threshold
Similarly, the stop threshold parameter is used to automatically stop the running stream, when the available samples crosses this boundary. It means, for playback, the empty samples in ring buffer and for capture, the filled (used) samples in ring buffer.

Silence threshold

The silence threshold specifies count of samples filled with silence ahead of the current application pointer for playback. It is usable for applications when an overrun is possible (like tasks depending on network I/O etc.). If application wants to manage the ahead samples itself, the snd_pcm_rewind() function allows to forget the last samples in the stream.

Obtaining stream status

The stream status is stored in snd_pcm_status_t structure. These parameters can be obtained: the current stream state - snd_pcm_status_get_state(), timestamp of trigger - snd_pcm_status_get_trigger_tstamp(), timestamp of last pointer update snd_pcm_status_get_tstamp(), delay in samples - snd_pcm_status_get_delay(), available count in samples - snd_pcm_status_get_avail(), maximum available samples - snd_pcm_status_get_avail_max(), ADC over-range count in samples - snd_pcm_status_get_overrange(). The last two parameters - avail_max and overrange are reset to zero after the status call.

Obtaining stream state fast and update r/w pointer

The function snd_pcm_avail_update() updates the current available count of samples for writing (playback) or filled samples for reading (capture). This call is mandatory for updating actual r/w pointer. Using standalone, it is a light method to obtain current stream position, because it does not require the user <-> kernel context switch, but the value is less accurate, because ring buffer pointers are updated in kernel drivers only when an interrupt occurs. If you want to get accurate stream state, use functions snd_pcm_avail(), snd_pcm_delay() or snd_pcm_avail_delay().
The function snd_pcm_avail() reads the current hardware pointer in the ring buffer from hardware and calls snd_pcm_avail_update() then.
The function snd_pcm_delay() returns the delay in samples. For playback, it means count of samples in the ring buffer before the next sample will be sent to DAC. For capture, it means count of samples in the ring buffer before the next sample will be captured from ADC. It works only when the stream is in the running or draining (playback only) state. Note that this function does not update the current r/w pointer for applications, so the function snd_pcm_avail_update() must be called afterwards before any read/write begin+commit operations.
The function snd_pcm_avail_delay() combines snd_pcm_avail() and snd_pcm_delay() and returns both values in sync.

Managing the stream state

The following functions directly and indirectly affect the stream state:

snd_pcm_hw_params

The snd_pcm_hw_params() function brings the stream state to SND_PCM_STATE_SETUP if successfully finishes, otherwise the state SND_PCM_STATE_OPEN is entered. When it is brought to SETUP state, this function automatically calls snd_pcm_prepare() function to bring to the PREPARE state as below.

snd_pcm_prepare

The snd_pcm_prepare() function enters from SND_PCM_STATE_SETUP to the SND_PCM_STATE_PREPARED after a successful finish.

snd_pcm_start

The snd_pcm_start() function enters the SND_PCM_STATE_RUNNING after a successful finish.

snd_pcm_drop

The snd_pcm_drop() function enters the SND_PCM_STATE_SETUP state.

snd_pcm_drain

The snd_pcm_drain() function enters the SND_PCM_STATE_DRAINING, if the capture device has some samples in the ring buffer otherwise SND_PCM_STATE_SETUP state is entered.

snd_pcm_pause

The snd_pcm_pause() function enters the SND_PCM_STATE_PAUSED or SND_PCM_STATE_RUNNING.

snd_pcm_writei, snd_pcm_writen

The snd_pcm_writei() and snd_pcm_writen() functions can conditionally start the stream - SND_PCM_STATE_RUNNING. They depend on the start threshold software parameter.

snd_pcm_readi, snd_pcm_readn

The snd_pcm_readi() and snd_pcm_readn() functions can conditionally start the stream - SND_PCM_STATE_RUNNING. They depend on the start threshold software parameter.

Streams synchronization

There are two functions allowing link multiple streams together. In the case, the linking means that all operations are synchronized. Because the drivers cannot guarantee the synchronization (sample resolution) on hardware lacking this feature, the snd_pcm_info_get_sync() function returns synchronization ID - snd_pcm_sync_id_t, which is equal for hardware synchronized streams. When the snd_pcm_link() function is called, all operations managing the stream state for these two streams are joined. The opposite function is snd_pcm_unlink().

PCM naming conventions

The ALSA library uses a generic string representation for names of devices. The devices might be virtual, physical or a mix of both. The generic string is passed to snd_pcm_open() or snd_pcm_open_lconf(). It contains two parts: device name and arguments. Devices and arguments are described in configuration files. The usual place for default definitions is at /usr/share/alsa/alsa.conf. For detailed descriptions about integrated PCM plugins look to PCM (digital audio) plugins.

Default device

The default device is equal to plug plugin with hw plugin as slave. The defaults are used:
defaults.pcm.card 0 defaults.pcm.device 0 defaults.pcm.subdevice -1
These defaults can be freely overwritten in local configuration files.
Example:
default

HW device
The hw device description uses the hw plugin. The three arguments (in order: CARD,DEV,SUBDEV) specify card number or identifier, device number and subdevice number (-1 means any).
Example:
hw hw:0 hw:0,0 hw:supersonic,1 hw:soundwave,1,2
hw:DEV=1,CARD=soundwave,SUBDEV=2

Plug->HW device
The plughw device description uses the plug plugin and hw plugin as slave. The arguments are same as for hw device.
Example:
plughw plughw:0 plughw:0,0 plughw:supersonic,1 plughw:soundwave,1,2
plughw:DEV=1,CARD=soundwave,SUBDEV=2

Plug device
The plug device uses the plug plugin. The one SLAVE argument specifies the slave plugin.
Example:
plug:mypcmdef plug:hw plug:'hw:0,0' plug:SLAVE=hw

Shared memory device
The shm device uses the shm plugin. The two arguments (in order: SOCKET,PCM) specify UNIX socket name (for example /tmp/alsa.socket) for server communication and server's PCM name.
Example:
shm:'/tmp/alsa.sock',default shm:SOCKET='/tmp/alsa.sock',PCM=default

Tee device
The tee device stores contents of a stream to given file plus transfers it to given slave plugin. The three arguments (in order: SLAVE,FILE,FORMAT) specify slave plugin, filename and file format.
Example:
tee:hw,'/tmp/out.raw',raw

File device
The file device is file plugin with null plugin as slave. The arguments (in order: FILE,FORMAT) specify filename and file format.
Example:
file:'/tmp/out.raw',raw

Null device
The null device is null plugin. This device has not any arguments.

Examples
The full featured examples with cross-links can be found in Examples section (see top of page):
Sine-wave generator

alsa-lib/test/pcm.c example shows various transfer methods for the playback direction.

Minimalistic PCM playback code

alsa-lib/test/pcm_min.c example shows the minimal code to produce a sound.

Latency measuring tool

alsa-lib/test/latency.c example shows the measuring of minimal latency between capture and playback devices.