MULTIMEDIA PLAYER IMPLEMENTATION ON EMBEDDED SYSTEMS

A Thesis Submitted to
the Department of Electrical and Electronics Engineering of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Electrical and Electronics Engineering

by
Yusuf Engin TETIK

December 2008
İZMİR

We approve the thesis of **Yusuf Engin TETİK**

Asst. Prof. Dr. Şevket GÜMÜŞTEKİN
Supervisor

Asst. Prof. Dr. Mustafa A. ALTINKAYA
Committee Member

Asst. Prof. Dr. Tolga AYAV
Committee Member

1 December 2008

Prof. Dr. M. Barış ÖZERDEM
Head of the Electrical & Electronics Engineering Department

Prof. Dr. Hasan BÖKE
Dean of the Graduate School of Engineering and Sciences

ACKNOWLEDGMENT

First of all, I would like to express my gratitude to Dr. Şevket Gümüştekin and Dr. Orhan Coşkun for their valuable suggestions, support, guidance and assistance in preparation of this thesis. Besides, I would like to state my appreciation to my colleagues, Ahmet Şahin and Yusuf Selçuk Ateşkan, for their valuable suggestions and support.

I would like to dedicate this thesis to my family.

ABSTRACT

MULTIMEDIA PLAYER IMPLEMENTATION ON EMBEDDED SYSTEMS

There has been a surge in the number of digital audio and video content in recent years. Advances in the compression and storage technologies and improvements in the speed of internet connection have enabled widespread use of multimedia content. A wide variety of devices have been introduced to decode and play these media contents. Initially designed as a mere voice communication device, the mobile phones nowadays come equipped with a variety of multimedia capabilities including media players despite their limited system resources.

Nowadays, huge servers host dramatically increased audio and video contents. Users prefer to watch these contents while streaming rather than downloading them first. So, streaming media players are responsible to present multimedia contents without annoying interrupts.

This thesis firstly introduces challenges in design and implementation of a streaming media player and then proposes solutions. Main challenges are keeping audio-video synchronization and server-client synchronization and detecting stream type, handling of multithreaded operations and buffer management. Audio-video synchronization problem is solved by using audio as master stream. Server-client synchronization problem is solved by designing a playback mechanism that keeps synchronization with the server by tuning the playback rate of a streaming media without losing lip-sync between audio and video. The proposed streaming player also has a feature of identifying the type of a media stream very rapidly without using a discrete stream inspector module. The presented design is heavily multithreaded which is implemented on Linux platform, moreover it is also convenient for and implementable on any multithreaded platform.

ÖZET

GÖMÜLÜ SİSTEMLERDE ÇOKLU ORTAM OYNATICI GERÇEKLEŞTİRİMİ

Son yıllarda sayısal ses ve görüntü içeriğinin sayısı belirgin bir şekilde yükseldi. Sıkıştırma ve saklama teknolojilerindeki ilerlemeler ve Internet bağlantı hızındaki iyileşmeler çoklu ortam içeriğinin yaygın bir şekilde kullanılmasına imkan tanıdı. Bu ortam içeriklerini çözebilen ve oynatabilen geniş bir yelpazede cihazlar tanıtıldı. İlk başta sadece ses iletişim cihazı olarak tasarlanan taşınabilir telefonlar, kısıtlı sistem kaynaklarına rağmen, bugünlerde çeşitli çokluortam içerkilerini oynatabilme yeteneğine sahip şekilde geliyorlar.

Bugünlerde, büyük sunucular son derece hızlı bir şekilde artan ses ve görüntü içeriğine sahiplik ediyorlar. Kullanıcılar, bu içerikleri cihazlarına tümüyle indirdikten sonra izlemek yerine daha içerik cihazlarına akarken seyretmek istiyorlar. Dolayısıyla, akan ortam oynatıcıları, çoklu ortam içeriklerini rahatsız edici kesilmeler olmadan sunmak zorundalar.

Bu tezde, ilk olarak akan ortam oynatıcı tasarlanmasında ve gerçekleştirilmesinde karşılaşılan zorluklar ve daha sonra da bu zorlukları aşacak çözümler öneriliyor. Temel zorluklar şu şekilde sıralanabilir; ses-görüntü eşlemesi ve sunucu-istemci eşlemesini sağlamak, akan içeriğin türünün belirlenmesi, eş zamanlı işlerin yönetimi ve bellek yönetimi. Ses-görüntü eşlemesi sorunu ses baz alınarak çözüldü. Sunucu-istemci eşlemesi sorunu ıse akan ortam içeriğinin oynatılma hızını ses-görüntü eşlemesi bozulmayacak şekilde ayarlayabilen bir çalma mekanizması tasarlanarak çözüldü. Önerilen tasarım, akan ortam oynatıcısının ortam türünü ayrı bir ortam türü tanıyıcı modüle ihtiyaç duymadan çok çabuk bir şekilde tanımasına da olanak veriyor. Ağırlıklı olarak eş zamanlı işlerden oluşan önerilen tasarım Linux üzerinde gerçekleştirilmiş, bununla birlikte eş zamanlı iş koşturabilen henhangi bir platform için uygun ve böyle bir platformda gerçekleştirilebilir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Media players have the most important role for any kind of digital entertainment systems such as DVD players, DVB set-top boxes, IPTV set-top boxes, PCs and even for handheld devices like cellular phones. The number of digital audio and video content on Internet increased dramatically in recent years. The advances in IP technology and infrastructure such as fibre optic technology make Internet appropriate for audio and video streaming which require high bandwidths (Conklin, et al. 2001). For instance, IPTV enables watching TV channels through Internet with high quality. User generated content, such as YouTube's videos, is also another important reason of increasing multi media data. The increasing multimedia contents are enjoyable only with the presence of a well done media player. So, a media player which will not degrade QoS (Quality of Service) or QoE (Quality of user Experience) is more important than it has been even before.

The player described in this thesis is a multimedia player which is capable of playing various media formats. It is a streaming media player which plays contents streamed over a channel by a server. Capability of playing various formats is a challanging task, because an abstraction layer or in other words an interface is needed by the player to access and use various formats (Sethuraman, et al. 2005). For instance, various video codecs such as MPEG-2, MPEG-4 (Liu 2001) and H.264 must be accessable over a common interface, and designing this interface requires at least a basic understanding of video coding. Furthermore a deep investigation of video codecs is required if hardware decoding blocks will be designed for the target embedded system. Because, video decoding requires high computation power, hence using a software video decoder for embedded systems is not an efficient way. Instead, programmable DSP blocks or hardware accelerators are used for video decoding on embedded systems (Sethuraman, et al. 2005).

Digital multimedia streaming (Dapeng, et al. 2001), especially video and voice streaming over IP has been one of the most popular issues for recent years. There are two types of digital media streaming (Suarez, et al. 2005) according to the origin of data

to be streamed: live streaming and streaming of previously processed and stored multimedia data. First one requires realtime encoding and processing schemes at server side and generally streamed as broadcasts. The latter one does not have encoding time limits, so better quality at lower bitrates can be achieved by giving video encoders more time. Video on Demand (VoD) streaming is a good example for the latter one.

The proposed media player design is implemented on Linux which is widely used on many embedded devices. According to (COTS Journal Online 2004) Linux is not designed for embedded systems and using Linux in embedded system may have risks such as interrupt latency, thread response time and device drivers. However, the number of embedded devices that host Linux as operating systems continues to increase year by year as seen in the Figure 1.1.

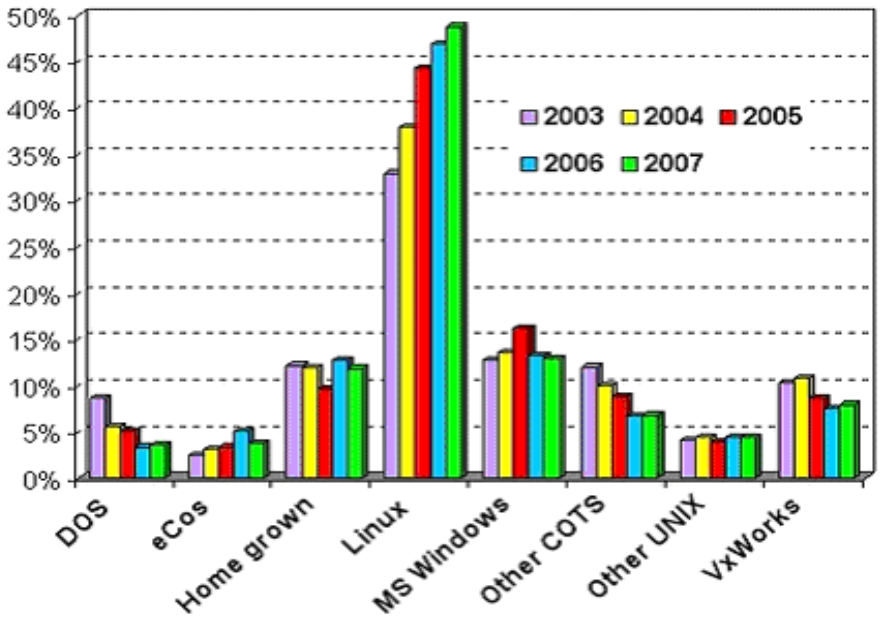


Figure 1.1 Embedded OS sourcing trends

(Source: Linux Devices 2007)

Some of the top reasons behind the popularity of Linux on embedded devices according to the survey by (Linux Devices 2001) are as the following:

- Source code is available and free
- No runtime royalties
- Robustness and reliability

• Linux has excellent networking support

• There are more drivers and tools available

• Lots of programmers are familiar with Linux

• It’s not from Microsoft

Although Linux is used for the implementation of the streaming multimedia player proposed in this thesis, it is also possible to use another multithreaded operating system since the proposed player architecture has no dependency on Linux.

The support for common container formats such as avi, program stream, transport stream, mov, asf; common audio codecs such as mp3, musicam, ac3 and common video codecs such as MPEG-2, MPEG-4, DivX, H.26x is critical for a multimedia player.

The proposed streaming multimedia player have the capability of playing from a variety of sources via various protocols such as http, mms, rtp, rtsp, rtcp, (Liu 1999) udp, etc.

The seamless playback of media streams which requires a tight synchronization with streaming server is a very challenging task to accomplish. However this is a must feature for a streaming media player, and proposed media player in this thesis uses a patent pending solution for this problem.

The lip synchronization between audio and video is also very important in aspect of QoE. The lip synchronization implies a perfect synchronization between audio and video data. This is also called as intra synchronization.

The economic utilization of mostly limited system resources is very critical for embedded systems. Hence, streaming media player must be designed in a way that makes efficiently use of system resources possible.

Trick modes including seeking to a backward or forward position must be supported by streaming media player. Implementing trick modes is also a very challenging task. Because audio and video synchronization must be re-established after trick mode operation is realized and so audio and video must be processed in a synchronized way during trick mode to enable a smooth resume possible.

The proposed streaming media player design is heavily multithreaded, hence requires a multithreading operating system. It involves a modular structure with each module designed to realize a single task such as decoding the video, de-multiplexing the media stream or playing audio, etc. Modular structure makes efficient implementation of the features discussed above possible.

Audio and video synchronization is critical to multimedia systems (Georganas

1996). The proposed design will employ a precise audio and video synchronization scheme. This scheme utilizes audio presentation time stamps to update player's master clock, resulting in a smoother and inter-synchronized playback.

Chapter-2 introduces the basic principles of the digital video coding (Conklin, et al. 2001) and the structure of media streams. There is a strong relationship between the structure of the media streams and the design of the streaming media player.

Chapter-3 explains the components and the logic behind the design of the streaming media player. Chapter-3 also explains how the player works. How the multithreaded operations and circular buffers are handled is explained in detailed. Steps required to play an ordinary media stream is explained in detail. This chapter also explains how the stream inspection is realized efficiently by the proposed design.

Chapter-3 also tries to explain one of the most challenging issues for streaming media players, that is, audio-video synchronization problem. The proposed solution for this problem offers using audio stream as master stream which will be explained in detail in this chapter. Lastly, other most important issue in designing streaming media players, that is, server-client synchronization is explained. A patent pending solution (filed to European and US patent offices) for this problem is proposed.

Chapter-4 explains the details of the implementation. Main implementation modules of the player are explained in detail. Pseudo codes written in C are used to show the implementation details of the modules.

Chapter-5 explains the proposed iddias in brief and concludes the thesis.

CHAPTER 2

VIDEO DE/CODING

In recent decades video coding has been a popular research area in Electrical Engineering and Computer Engineering, strongly because of developments in digital signal processing and advances in computer technology.

A digital video is made up of the individual still images or "frames" that, when played in sequence, are able to give the impression of movement. The impression of motion and continuum is due to the limitations in the human visual system. Video processing is tightly related to image processing, hence in fact, video is nothing else than sequentially recorded images.

Video coding generally refers to digital video coding, because it is much more efficient to process video signals after they are digitized from analog signals. Lossless digitization of analog signals can be realized according to **the Nyquist sampling theorem** which states that the sampling frequency should be at least twice the highest frequency contained in the signal.

$$f_{sampling} \geq 2\, f_{highest} \qquad (2.1)$$

If analog signal is discretely sampled at a rate that is insufficient to capture changes in the signal **aliasing** will arise. “The wagon wheel effect can be a good example of aliasing for video films. This is because continuously varying images are being discretely sampled at a rate of 24 frames/sec. The Nyquist sampling theorem tells us that aliasing will occur if at any point in the image plane there are frequency components, or light-dark transitions, that occur faster than f_s / 2, which in this case is 12 frames/sec. But in many situations the light-dark transitions may be occurring faster than this, such as a wagon wheel or propeller rotating at high speed (Olshausen 2000). ”

Consider a wagon wheel with eight spokes turning at a rate of 2.5 revolutions per second in the clockwise direction. In this case, the wagon wheel will move by 83% of the spoke spacing each frame, since;(Olshausen 2000)

$$((2.5 \text{ revs / sec }) \text{ x } (8 \text{ spokes /rev })) / (24 \text{ frames / sec}) = 0.83 \text{ spoke / frame} \qquad (2.2)$$

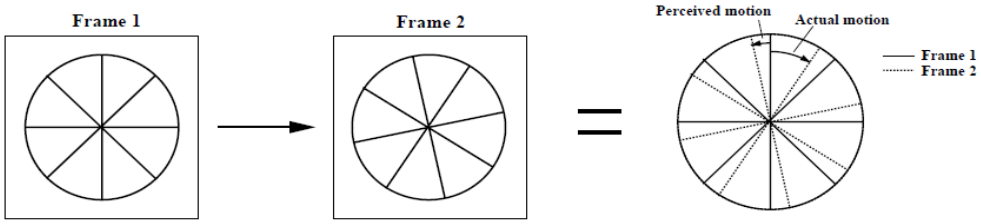


Figure 2.1. Aliasing in video films.

(Source: Olshausen 2000)

"The problem the brain faces in viewing these frames in rapid succession is that there are two interpretations. One interpretation is that the wheel has moved by 83% of the spoke interval in the clockwise direction. The alternative interpretation is that it has moved by 17% of the spoke interval in the counter-clockwise direction. It turns out that the brain prefers the latter interpretation, and so as a result you perceive the wheel moving backwards (counter-clockwise) at a slower speed than it is actually moving. (Olshausen 2000)"

2.1. Structure of Digital Images

A video stream is composed of sequentially recorded digital images (Netravali and Haskell 1988) and a digital image is represented by samples arranged in a two dimensional array. **Pixel** is the name of each sample in this array, which refers to picture element. So each pixel is located uniformly on a surface that is called as spatial domain. It holds an integer value that represents the intensity of light or a color.

It is common to use 256 gray levels to represent an image in gray tones, in this case each pixel will hold a single value between 0-255. The brightness step size 1/256 is close to what a human eye can perceive, in other words the quality of an image according to a human will not change significantly by using more than 256 gray levels. The other reason of choosing the number of gray levels as 256 is related to computer science. By this way, each pixel can be represented by a byte, that is 8 bits and can represent $2^8 = 256$ distinct values.

In Figure-2.2 a grayscale image of the planet Venus at 200x200 pixels resolution is shown. "When this image was acquired, the value of each pixel corresponded to the

level of reflected microwave energy. A grayscale image is formed by assigning each of the 0 to 255 values to varying shades of gray (Smith 1997) .”

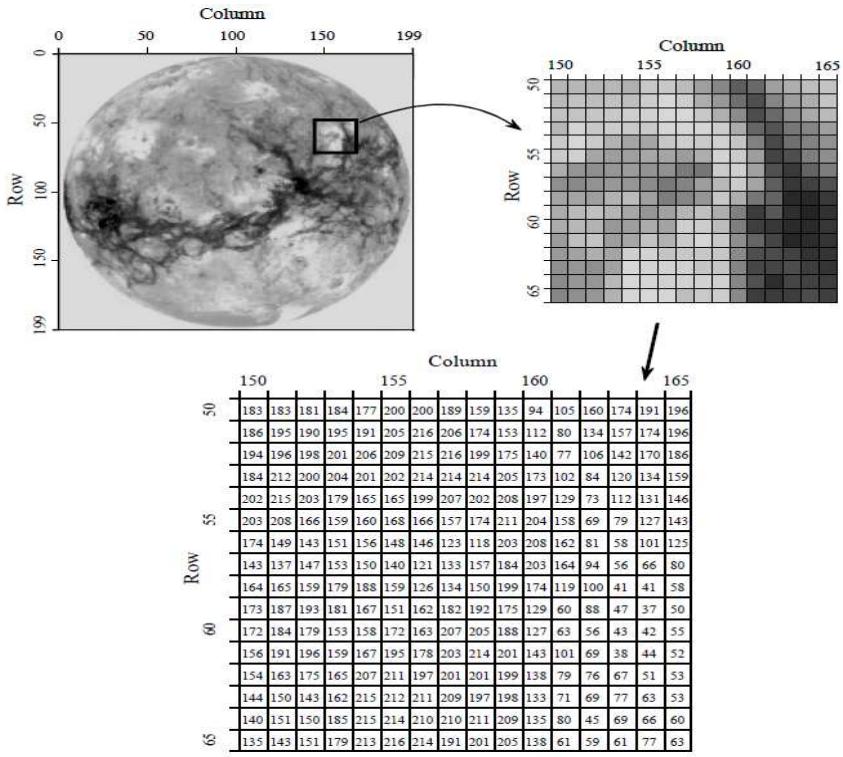


Figure 2.2. Structure of a grayscale image

(Source: Smith 1997)

Color can be added to the digital image by various ways. One way is adding chroma components to the grayscale image which is called as luma component. In this way, a colorfull digital image is composed of three components; the Y component determines the brightness of the color (referred to as luminance), while the Cb (blue difference) and Cr (red difference) components determine the color itself (called as chromas). In other words, each pixel will have 3 components, each component will hold a single value between 0-255, that means each component has 256 level of quantization and takes one byte from memory. As result, each pixel is represented by 3 bytes in memory. By using three components 256x256x256 = 16.8 million different colors can be defined, hence a colorspace is formed, called as YCbrCr colorspace which is a subspace of all colors of real life. The terms YCbCr and YUV are used interchangeably, however the term YUV usuallu used as the analog correspondence of YCbCr with scale factors.

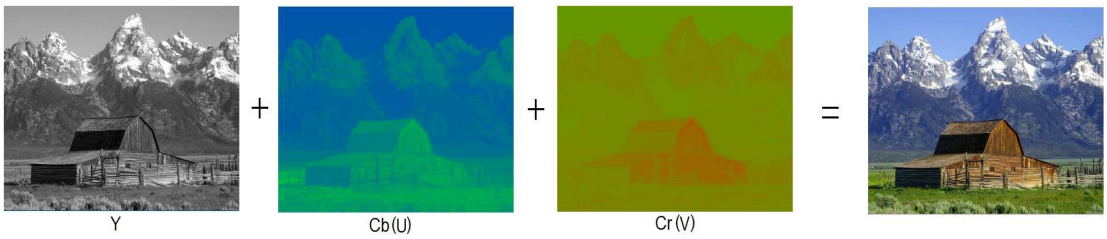


Figure 2.3. Forming a colorfull digital image

The other way of forming color images is using RGB colorspace. In this format each pixel is composed of R (red), G (green) and B (blue) components. In other words, each component represents the intensity of one of the three primary colors: red, green, blue. The set of all possible colors that can be composed by mixing these three primary colors is called as **gamut**. Conversion between RGB and YUV colorspaces is possible by using following formulas;

From RGB to YUV :

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (2.3)$$

It can also be represented as:

$$Y = 0.299R + 0.587G + 0.114B \quad (2.4)$$
$$U = -0.147R - 0.289G + 0.436B \quad (2.5)$$
$$V = 0.615R - 0.515G - 0.100B \quad (2.6)$$

From YUV to RGB :

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0.140 \\ 1 & -0.395 & -0.581 \\ 1 & 2.032 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \quad (2.7)$$

It can also be represented as:

$$R = Y + 1.140V \quad (2.8)$$
$$G = Y - 0.395U - 0.581V \quad (2.9)$$
$$B = Y + 2.032U \quad (2.10)$$

YUV colorspace is the standart color encoding system for analog television system worldwide (NTSC, PAL). Main reason of this decision is historical, because the early television sets were designed for black and white signals hence video cameras

were only capable of generating B&W signals until 1950s. After color signals were developed, a method compatible with B&W TV infrastructure is needed. Y signals were already being transmitted by the current system at that time, so engineers find out UV signals as solution. U and V signals were color difference signals and can be calculated from original RGB colorspace and the lumaY signals by using the formulas seen at above.

Most of the image and video compression formats prefer YUV as the colorpace. Because, the human eye is more responsive to brightness rather than color. Furthermore, when the resolution of chroma frames are set as half of the resolution of luma, human eye cannot perceive the difference. So, half of the bandwidth required to transmit or the storage to save the image/video is saved. The most common color encoding format for image/video encoders is YUV420 by which chromas' resolution set half of the luma's.

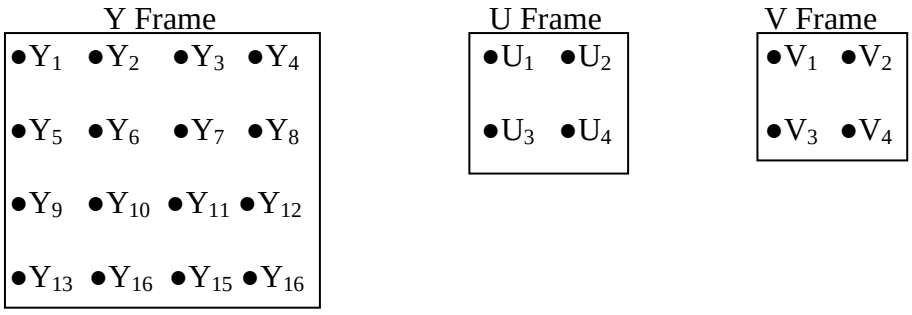


Figure 2.4. The structure of YUV420 color encoding format

The result colorfull frame, that is composed of the component frames at the above will be as the following:

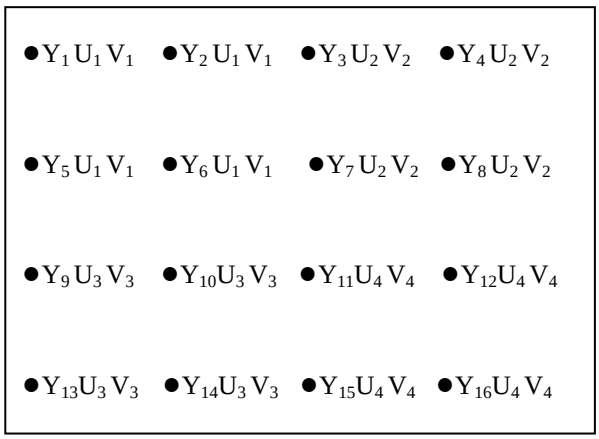


Figure 2.5. Composing colorfull image from YUV420 rames

The structure of a raw (not compressed or encoded) YUV420 video bitstream includes ordered sequence of frames, Y frame which is followed by U and V frames. The YUV420 video streams are often stored in files with ".yuv" extension.

At the above in the Figure 2.5. each capital letter (Y,U,V) represents a byte, hence holds a value between 0-255. The position of these bytes in the video bitstream will be as the following;

$Y_1\ Y_2\ Y_3\ Y_4\ Y_5\ Y_6\ Y_7\ Y_8\ Y_9\ Y_{10}\ Y_{11}\ Y_{12}\ Y_{13}\ Y_{14}\ Y_{15}\ Y_{16}\ U_1\ U_2\ U_3\ U_4\ V_1\ V_2\ V_3\ V_4$ eof

2.2. Video Compression Techniques

Digital video compression has made it possible to store, stream and transport large amounts of video content which was once impractical due to the excessive size of the data files required to convey the necessary information. Digital video compression, especially the MPEG (Gall 1991) formats and particularly the MPEG-2 format (Puri 1993) is widely used in devices ranging from DVD players to satellite and terrestrial set top boxes to network video servers and receivers.

The amount of data require to transmit or store raw (not compressed) digital video is far too much. For instance, for the majority of countries using PAL (including Turkey), the number of frames (digital images/pictures) showed per second is 25. The resolution of each frame is 720x576. Assume that the color encoding format of raw frames are YUV420, then required bandwith to transmit this video signal can be calculated as the following;

$$25*(\underbrace{(720*576)*1}_{\text{Y frame}} + \underbrace{(360*288)*2}_{\text{U and V frames}}) = 25*1.5*720*576 = 15{,}552{,}000 \text{ byte/sec} = 14.8 \text{ Mbyte/sec} \quad (2.11)$$

Transmitting or storing video data without compressing is not an efficient way, since video frames have spatial and temporal redundancy at remarkable amounts. Video compression techniques aim to remove these redundancies to lower required bandwidth to transmit video signals. The **spatial redundancy** is observed because of the pixels that are replicated within a single frame of video. The **temporal redundancy**

arises when there similarites between consecutive frames which is the common case, since there are only 40 milliseconds (1sec/25 fps) between successive frames.

If a video frame is compressed by just removing spatial redundancies, then it is called as **intra-frame** (intra coded frame), however if temporal redundancies are also removed, then it is called as **inter-frame** (inter coded frame).

Commonly used video codecs (MPEG-2, h.264) use a **block based coding** scheme. That is, video images are divided into **blocks** (such as 8x8 blocks) before being processed. For instance, MPEG-2 firstly divides an image into 16x16 squares which are called as **macroblock**. Then, each macroblock is divided into 8x8 squares, called as **block**s.

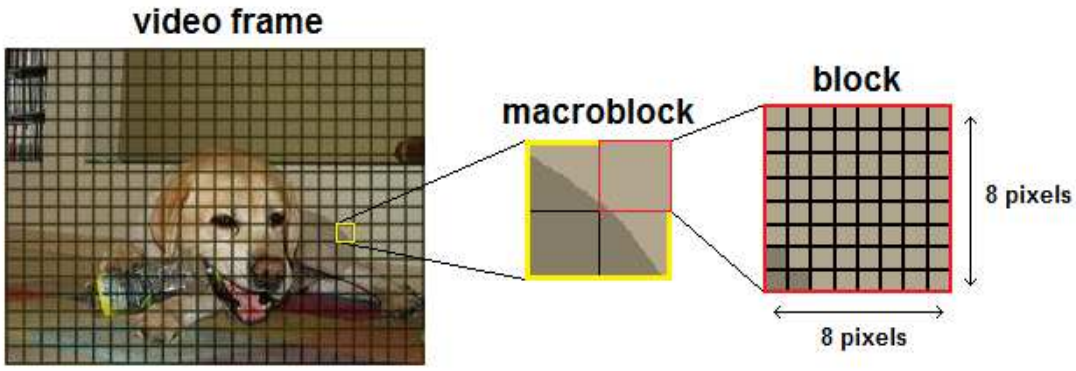


Figure 2.6. A video frame divided by macroblocks

In MPEG-2, 16x8 blocks are also used when images are divided into **field**s which are created by dividing the video image into two parts; odd rows of the image compose the first part(or field), and even rows compose the second one.

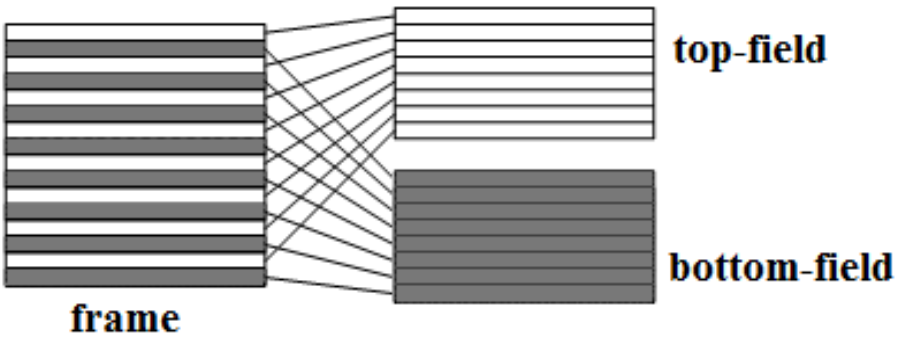


Figure 2.7. Fields in MPEG-2

The conventional coding schemes (such as MPEG 2) use a coding methodology which can be called as “divide and compress”. So, firstly overall picture sequence is divided into group of pictures. Then each picture is, divided into group of macroblocks called as **slice**, and then each slice is divided into macroblocks and each macroblock is divided into blocks. In other words, video stream is divided into parts according to a hierarchy as showed in Figure 2.8.

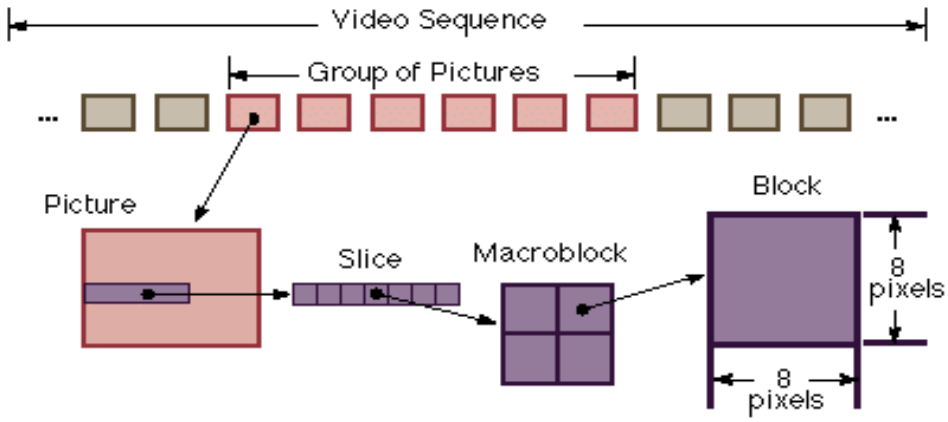


Figure 2.8. MPEG-2 Video stream data hierarchy
(Source: Basith 1996)

2.2.1 Pre-processing

Video images may contain noise which effects the compression of video in a negative way, since commonly used compression techniques such as DCT (Discrete Cosine Transform) are based on correlation of the pixels in an video image which is degraded by the noise. So, filtering out noise have a great importance before preprocessing.Filtering video images in a way to make them more suitable for compression may also be very effective. It may be realized by filtering out non-essential visual information from the video signal. Furthermore, some lossy operations or filters may be used to make video images more compressable such as using blurring filters.

The quantization is commonly used in video codecs such as MPEG-2, h.264. Actually quantization is not a pre-processing technique, and it is used after DCT operation. In other words, after DCT operation is applied to a block, a new block at the

same size that holds DCT coefficients of input block is generated. The generated DCT coefficients are integer values, and these coefficients are divided by a positive integer (such as 2) to represent the coefficient with fewer bits. For instance, if an integer number is divided by 2, the number of required bits to represent it will be one bit less. However, this is a lossy operation because when we try to reconstruct the number by multiplying the division by 2, if the divided number is odd then the result of multiplying will be the original number minus one. This method is known as quantization (reconstruction by multiplying is called as dequantization) and it is a very frequently used method in image/video compression although some precision is sacrificed in order to increase compression ratio.

Quantization is usually done after DCT. This is especially effective, if the used numbers in these operations are floating or double numbers. Otherwise, if integer numbers are used, quantizing before DCT may increase compression ratio, because after quantizing increases correlation of pixels in a block. So, when DCT operation is applied there will be fewer number of coefficients. So, any filter that increase the correlation of pixels in the block will increase the compression ratio. Of course filtered signals must be recoverable without loss or with acceptable loss. For example, the following proposed method increases the correlation and it is possible to reconstruct the original block after the filter is applied.

2.2.1.1 Pre-Process Filter Example

This method aims to increase the correlation of the pixels in a block by converging value of each pixel to a specific number. It is reasonable to select this target number as the average of the pixel values in the block.

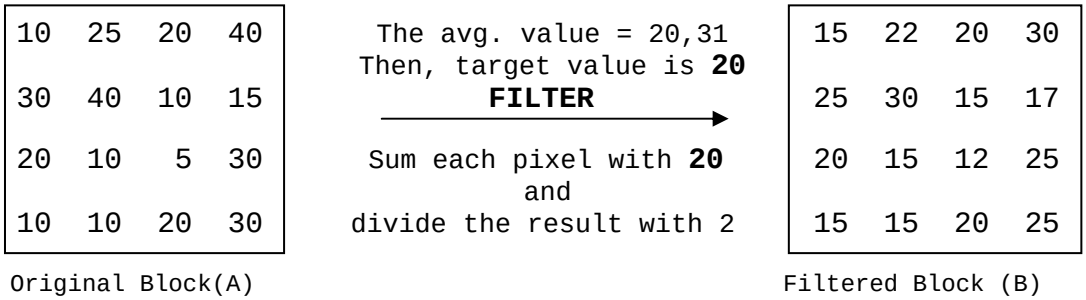


Figure 2.9. Filtering of a 4x4 block with the proposed method

The proposed filter increases the correlation of the block, so after the DCT is applied there remains fewer number of coefficients to transmit. In order to reconstruct the original block, the target number used to generate new block is needed. However, it is not necessary to transmit this number to the decoder, because the target can be calculated by using filtered block. The average of the pixels in the block will give the number used as the target value.

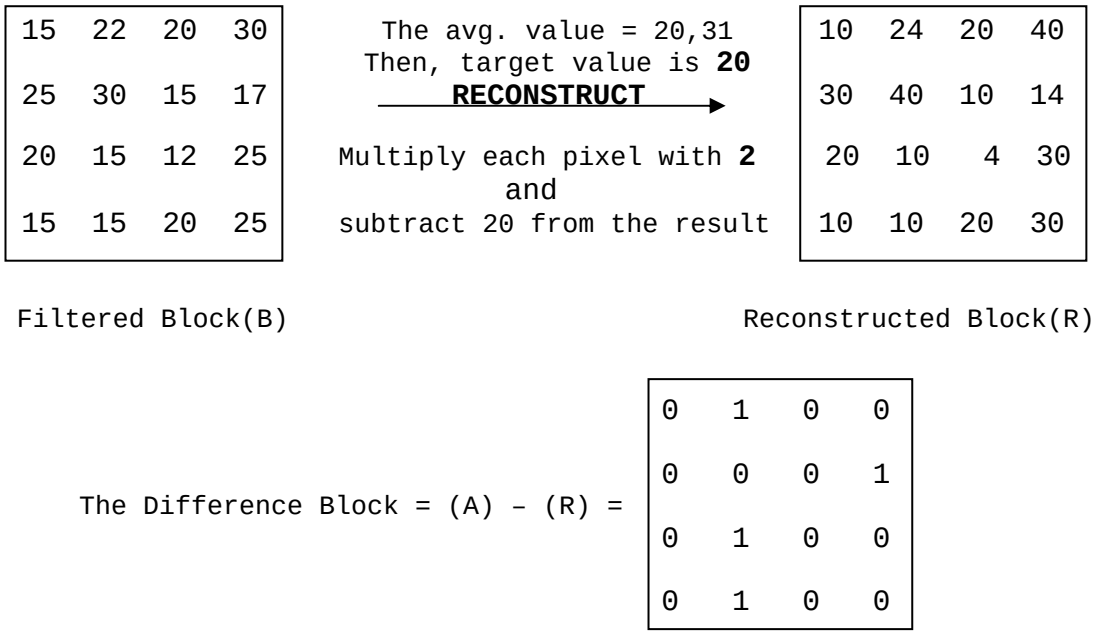


Figure 2.10. Reconstruction of 4x4 block after the proposed filter is applied

2.2.2 Intra Frame Coding (Eliminating Spatial Redundancy)

Spatial redundancy represents redundant data within a single frame of video. So intra frames are coded independently from other frames. Spatial redundancy occurs because adjacent pixels in a single frame are often correlated. In MPEG terminology, such frames are referred as **I frames**. An I frame may be thought of as a key frame or reference video frame which acts as a point of comparison to other frames during encoding, decoding and playback. Pictures between two I frames including first I frame is called **Group of Pictures** (GOP) in MPEG terminology. For instance, in MPEG-2 a GOP is generally composed of 12 or 24 pictures.

2.2.2.1 Step by Step Intra Frame Coding (Encoding)

Although each digital video compression format has its own particular characteristics, a number of common features are also present. In the following, basic methods used in intra coding will be given.

2.2.2.1.1 Divide Frame Into Macroblocks

A raw picture of video is composed of three frames: Y,U,V. If the resolution of raw picture is **CIF** (352x288), then the resolution of Y frame will also be in CIF resolution. The color encoding format is generally selected as YUV420, if so then U and V frames will be in the half resolution **QCIF**(176x144) of the Y frame.

Then Y frame is divided into **macroblock**s (MBs), each macroblock is a 16x16 pixels square. The chroma frames (U and V) which are in the half resolution of the Y frame are divided into **block**s, with each block in 8x8 pixels resolution. So a macroblock of Y frame corresponds to a block of U and a block V frame. In other words, a macroblock of original picture is reconstructed by using the corresponding macroblock of Y frame, and corresponding blocks of U and V frame as shown in the Figure 2.11.

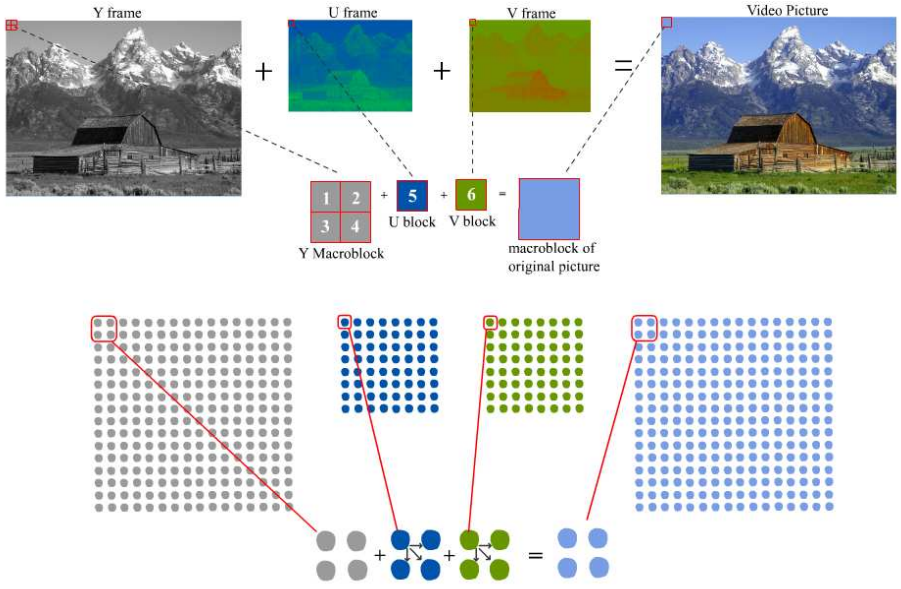


Figure 2.11. Reconstruction of the picture by using blocks of YUV420 frames

In order to reconstruct the original picture, chroma (U and V) frames must be upscaled to the size of Y frame. This is generally accomplished by replicating each pixel of a chroma frame as shown at the below of the Figure 2.11. Although, it is possible to use any other interpolation method to upscale chroma frames. The effect of using better algorithms to upscale chroma frames will not be perceivable by human eyes. This is the reason of the idea behind using YUV420 frames instead of YUV422 frames (each component frame is in the same size of the composed picture).

2.2.2.1.2 Discrete Cosine Transformation (DCT)

DCT is the most common transformation method used in video compression. Video codecs such as MPEG1-2-3-4 and H261-262-263-264 all use compression provided by the nature of DCT. “DCT is an orthogonal mathematical transform that is used to remove spatial redundancy by concentrating the signal energy into only a few coefficients (Mikaeli and Ying 2004)” The mathematical expression of DCT is given in (2.12).

$$F(u,v) = 0.25\, C(u)\, C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos(((2x + 1)u\pi)/16) \cos(((2y + 1)u\pi)/16) \quad (2.12)$$

u, v, x, y = 0, 1, 2, … 7
(x, y) are spatial coordinates in the sample domain
(u, v) are coordinates in the transform domain
$C(w) = 1 / \sqrt{2}$ for w = 0,
C(w) = 1 for w > 0

The DCT is applied to each block of YUV frames, in other words 8x8 DCT operation takes each block as input and generates a new 8x8 block which is composed of DCT coefficients as seen in the Figure 2.12. “Each DCT coefficient indicates the amount of a particular horizontal or vertical frequeny within the block. DCT coefficient at the position (0,0) is the DC coefficient that represents average sample value (Jack 2005).” Since DC value conveys much more information than any other DCT coefficient, it has great importance and is treated specially while packeting to transport. “Since natural images tend to vary only slightly from sample to sample, low frequency coefficients are typically larger values and high frequency coefficients are typically

smaller values (Jack 2005).” So, non-zero DCT coefficients are seem to be occur near DC value. In other words, the probability of occurring of a non-zero DCT coefficient reduces as its distance to DC value increases.

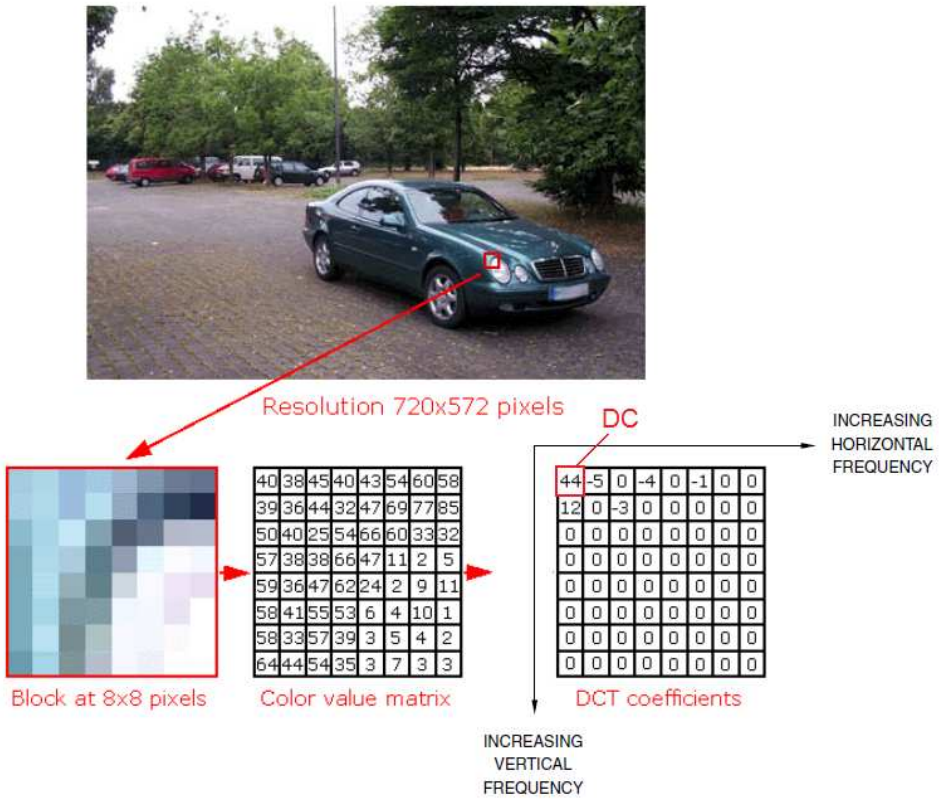


Figure 2.12. 8x8 DCT of a block
(Source: Mikaeli and Ying 2004)

2.2.2.1.3 Quantization

The quantization is a commonly used video compression technique (Viscito and Gonzales 1991). Actually, quantization is a lossy compression technique which is generally used after DCT operation. In other words, after DCT operation is applied to a block, a new block at the same size that holds DCT coefficients of input block is generated. The generated DCT coefficients are integer values, and these coefficients are divided by a positive integer (such as 2) to represent the coefficient with fewer bits. For instance, if an integer number is divided by 2, the number of required bits to represent it will be one bit less. However, this is a lossy operation because when we try to

reconstruct the number by multiplying the division by 2, if the divided number is odd then the result of multiplying will be the original number minus one. This method is known as quantization (reconstruction by multiplying is called as dequantization) and it is a very frequently used method in image/video compression although some precision is sacrificed in order to increase compression ratio.

Each coefficient of 8x8 DCTized block is not quantized with the same value. Because the most significant low frequency coefficients are grouped around the DC coefficient, and farther from the DC value both the amplitude and the significance of the DCT coefficients decreases. “So, higher frequencies (which are farther from DC value) are quantized more coarsely than lower frequencies, due to visual perception of quantization error. This results in many DCT coefficients being zero, especially at the higher frequencies (Jack 2005).” In order to quantize each coefficient with a different value, a **quantization matrix** is used. This is a 8x8 matrice which is already known by decoders or sent to decoders within encoded stream. The quantization matrix used for intra frames by MPEG2 is showed in the Figure 2.13.

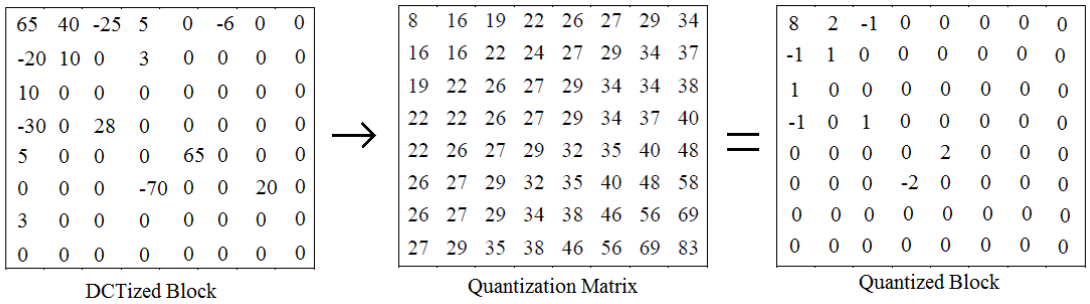


Figure 2.13. Quantization of a 8x8 block

2.2.2.1.4 Zig-Zag Scanning

After a block is DCTized and then quantized, non-zero DCT coefficients are grouped around the DC coefficient as seen in the Figure 2.13. In the next step, the quantized block will be coded by using run-length coding. However, before run length coding, the DCT coefficients are moved from 8x8 matrice to a one dimensional array by scanning matrice in a zig-zag order. The reason of scanning in zig-zag order is to

produce long runs of zero coefficients and to make the array more proper for run-length coding. Since DC value stores the most of information, it is treated specially and transmitted independently.

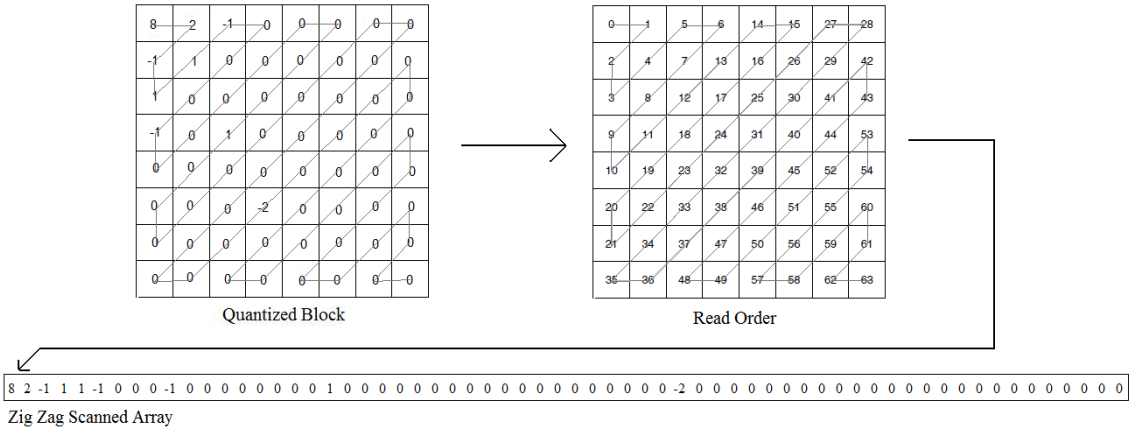
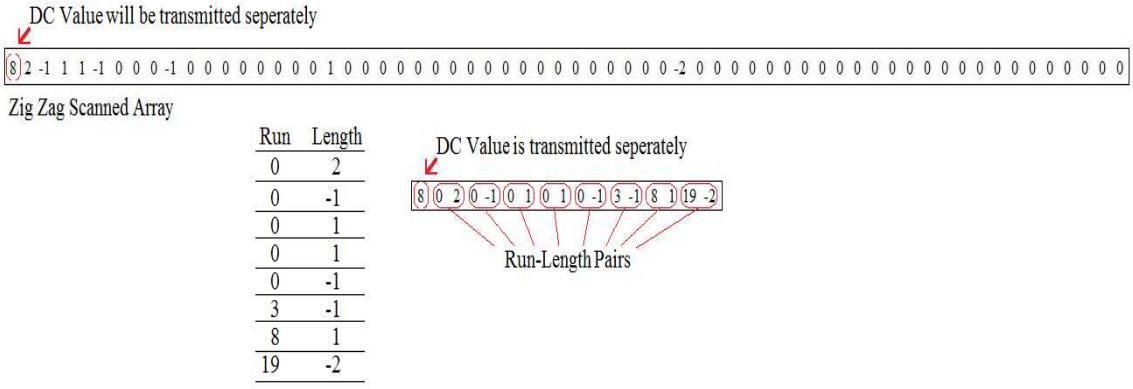


Figure 2.14. Zig-Zag scanning of quantized DCT block

2.2.2.1.5 Run-Length Coding

One-dimensional array, which is generated by zig-zag scanning of quantized DCT block as shown in Figure 2.14, is coded by a technique called run-length coding. Run-length coding aims to represent the one-dimensional array with fewer number of elements. This is accomplished by using run-length pairs; **run** indicates the number of zero coefficients that a leading a non-zero coefficient which is called as **length**.



Run	Length
0	2
0	-1
0	1
0	1
0	-1
3	-1
8	1
19	-2

Figure 2.15. Run-Length coding

So, the zero coefficients at the end of the array are not represented by a run-length pair, since there is no non-zero coefficient that ends the run. The run-length coding of one-dimensional array in the Figure 2.14 is showed in the following Figure 2.15.

2.2.2.1.6 Variable Length Coding (VLC)

Run-Length pairs which are generated by using zig-zag scanned array are coded with a lossless compression technique before transmission. This technique assigns a binary code to each run-length pair according to the pair's occurance probability. For instance, if a run-length pair is frequently occurred, so it has high probability of occurance, then the length of assigned binary code to this pair will be shorter. So, the length of assigned binary code is determined according to pair's occurance probability. Hence, run-length pairs will be represented by binary codes that have variable lengths. That is why this technique is called as **variable length coding**.

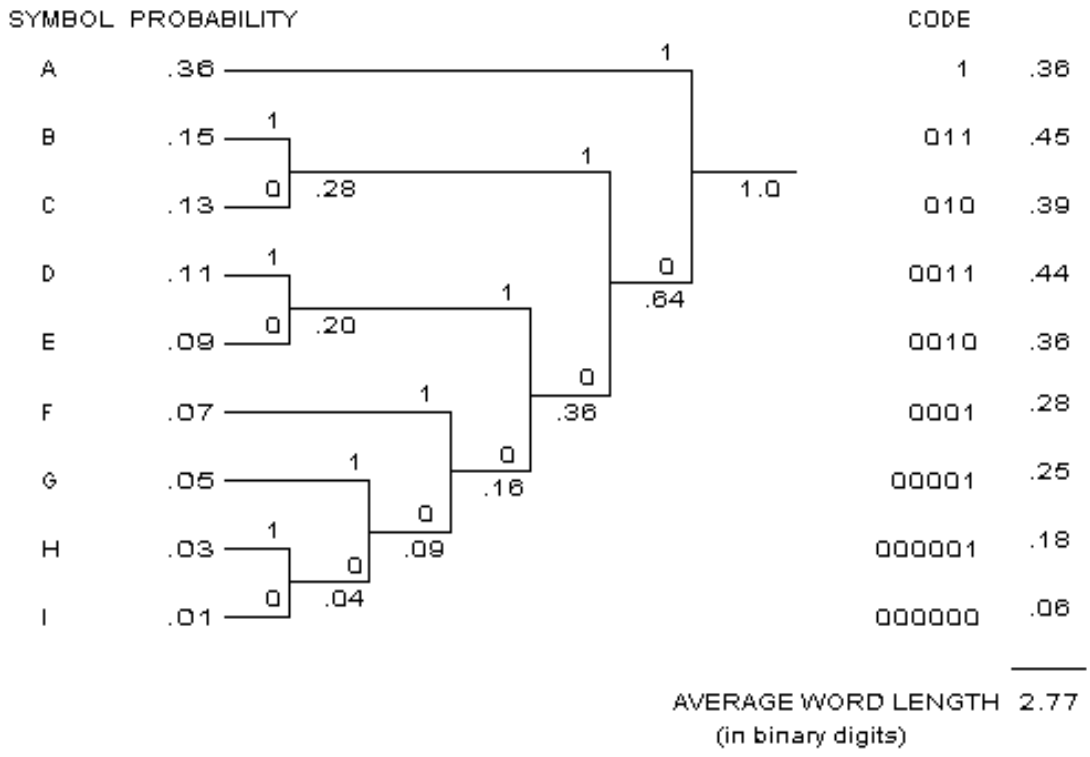


Figure 2.16. Variable length coding (Huffman Coding)
(Source: Mikaeli and Ying 2004)

This technique can also be called as **statistical coding**, or **entropy coding**. Entropy coding benefits from the fact that, “sometimes certain values are used more often than others, so a coding system can be designed to transmit the high-occurance values with a short(few bit) code, meanwhile, less frequently occurring values can use longer bit codes. In this way, we can achieve a significant improvement in the transmission efficiency. The Morse code of Telegrapchy system uses this principle. Huffman coding is the most common method for statistical coding and it’s widely used in video compression algorithms.”

2.2.2.1.7 A New VLC Method

In intra frame encoding, encoders divides the frames into n x n (8 x 8 for MPEG-2) square blocks. These blocks are then DCT transformed, quantized and variable length encoded. There are fixed variable length code tables for most of the encoding schemes such as MPEG-1 and MPEG-2. These tables are designed particularly for 8x8 blocks considering the probability of the runs and levels that may come from such a block.

The design of these tables affects the degree of compression. These tables must be able to cover all possible run and the level pairs. Even the least probable run and level pair must be able to be represented. This restricts the efficiency of such tables.

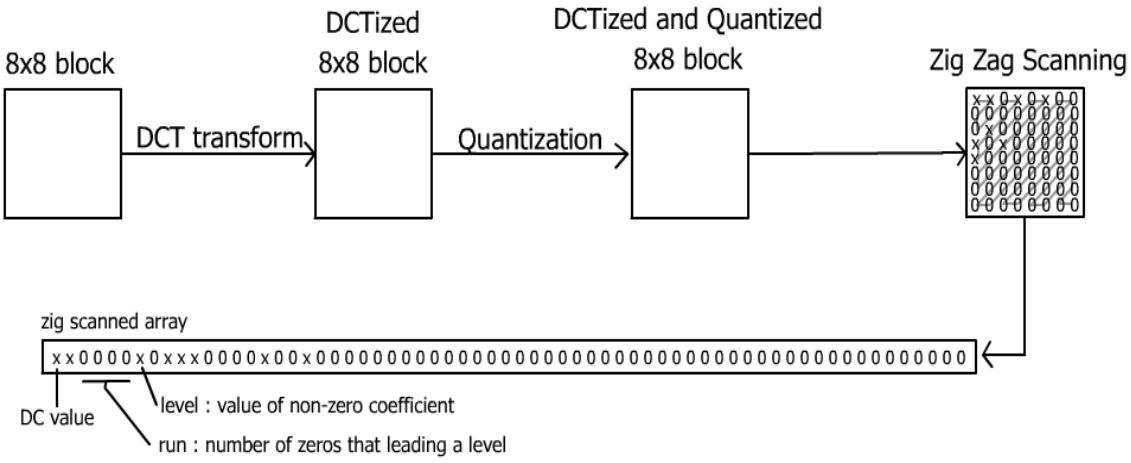


Figure 2.17. Intra frame encoding

When an nxn (8x8) block from a natural image is DCT transformed, the most significant low frequency coefficients are grouped around the DC coefficient. Farther from the DC value both the amplitude and the significance of the DCT coefficients decreases. In the proposed method, instead of using a vlc table for the whole nxn (8x8) block a special vlc table is designed for a smaller m x m (m < n) block around the DC value. This way the efficiency of the new vlc table is increased. That is, smaller variable code lengths are used to encode the most significant DCT coefficients. Designing such a special vlc table for a smaller block size will result in more compression than using the standart tables for the larger nxn block for the same quality level. The less significant DCT coefficients lying outside the smaller (mxm) block are eliminated resulting in further compression. The size of the inner block can be variable depending on the particular application and target rate and distortion.

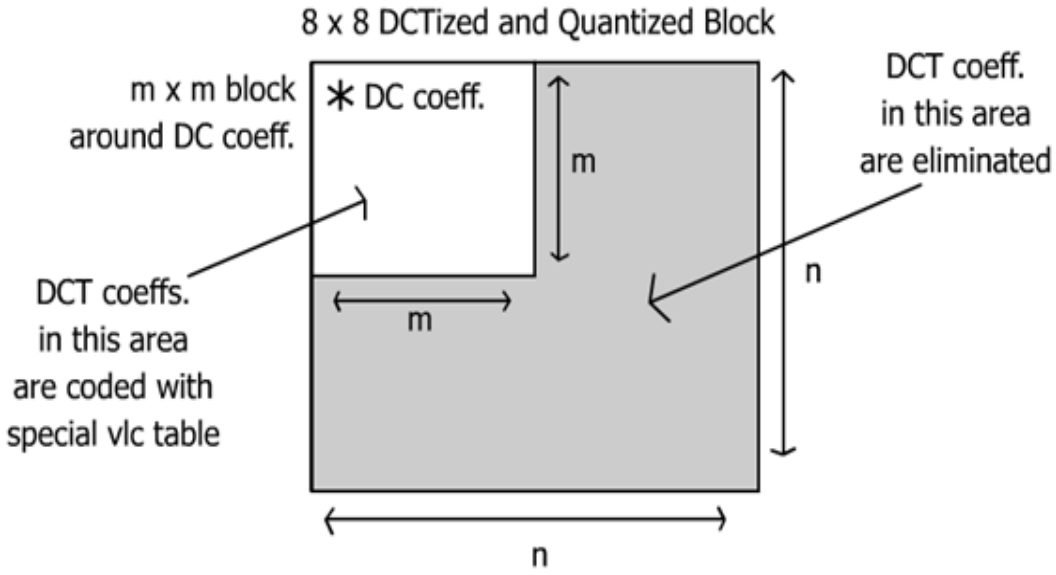


Figure 2.18. Coding of a part of DCTized block

2.2.3 Inter Frame Encoding (Eliminating Temporal Redundancy)

The **temporal redundancy** arises because of the similarites between the consecutive frames. The frames that are compressed by eliminating the redundant data, as shown in Figure 2.19., between itself and a reference frame are called **inter frames**.

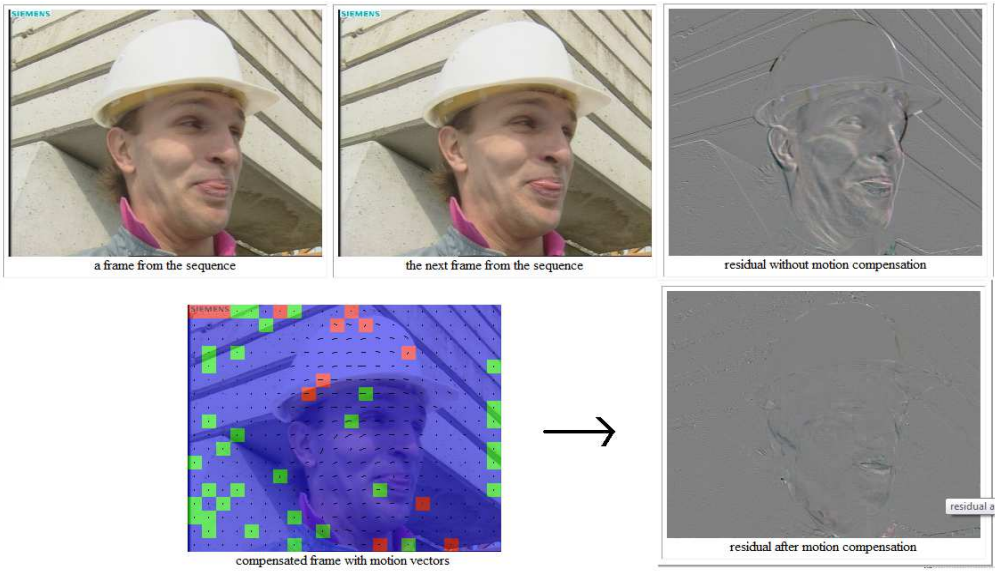


Figure 2.19. Difference of consecutive video frames
(Source: Compression 2002)

In video encoding particularly in MPEG encoding (Haskell, et al. 1997) , there are three types of frames; I, P and B frames. **I frames** are intra frames which are used as reference frames for the subsequent B and P frames. I frames do not refer to other frames to be decoded. **P frame**s refer to the I frames through motion vectors to be decoded. **B frame**s refer to the I or P frames or both to be decoded through motion vectors. Therefore B and P frames are called the **predicted frame**s and I frames are called the **reference frame**s.

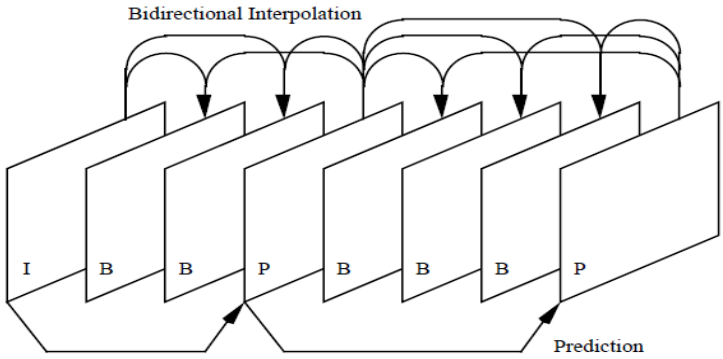


Figure 2.20. Intra (I) and Inter (B and P) frames
(Source: ISO/IEC 13818-2 1995)

2.2.3.1 Motion Compensation (Motion Prediction)

Motion compensation method aims to find similar blocks between consecutive frames by searching each block of inter frame (predicted frame) in the reference frame (Stiller and Konrad 1999).

Encoders search for predicted frame blocks in reference frame that results in the closest match, then subtracts the found reference block from the one in the predicted frame for which the search is done. This is called **block-based motion compensation**. The block size is generally constant (16x16 in MPEG-2), however it is also possible to use variable block sizes to define an object more accurately (Chan, et al. 1990).

When the closest match found, the motion vector is calculated and the difference block is transformed and coded. The aim is to code least amount of information for the predicted frame in order to achieve least possible bit-rate for the encoding. Motion vectors are two dimensional vectors that point to the block in the reference frame which is closest to the current block in the predicted frame.

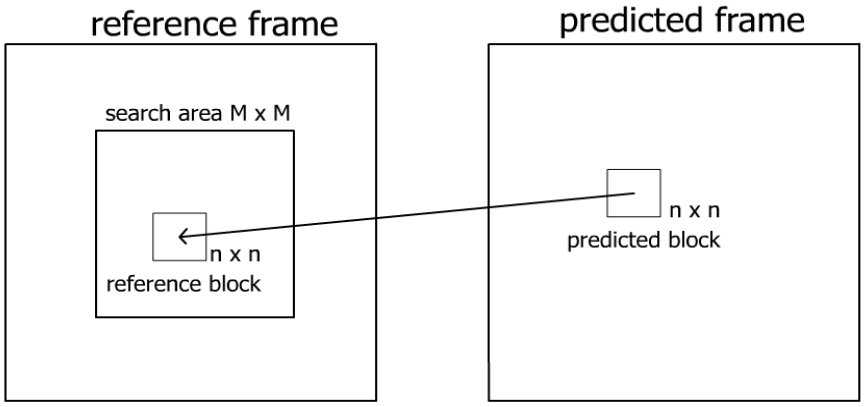


Figure 2.21. Motion search for predicted block

2.2.3.2 Motion Search Algorithms

Motion search algorithms (Jamkar, et al. 2002) use one of different possible matching criterions such as SAD (Sum Absolute Differences), MAD (Mean Absolute Difference) or MSE (Mean Square Error) to find the best match. Among various

criterias, SAD is the most popular one. For a block size of 16x16, the criterias mentioned can be defined as;

$$SAD = \sum_{i}^{16}\sum_{j}^{16} \left| B_{pred}(i , j) - B_{ref}(i , j) \right| \tag{2.13}$$

$$MAD = (1/256)\sum_{i}^{16}\sum_{j}^{16} \left| B_{pred}(i , j) - B_{ref}(i , j) \right| \tag{2.14}$$

$$MSE = (1/256)\sum_{i}^{16}\sum_{j}^{16} (B_{pred}(i , j) - B_{ref}(i , j))^2 \tag{2.15}$$

Firstly, an arbitrary block is selected from search area at the reference block according to a searching algorithm. Comparing all possible candidate blocks in the search area is known as **full motion search.** If the search area is the reference frame, that is all blocks in the reference frame will be tested to find the best matched one, then it is called as **global motion search.**

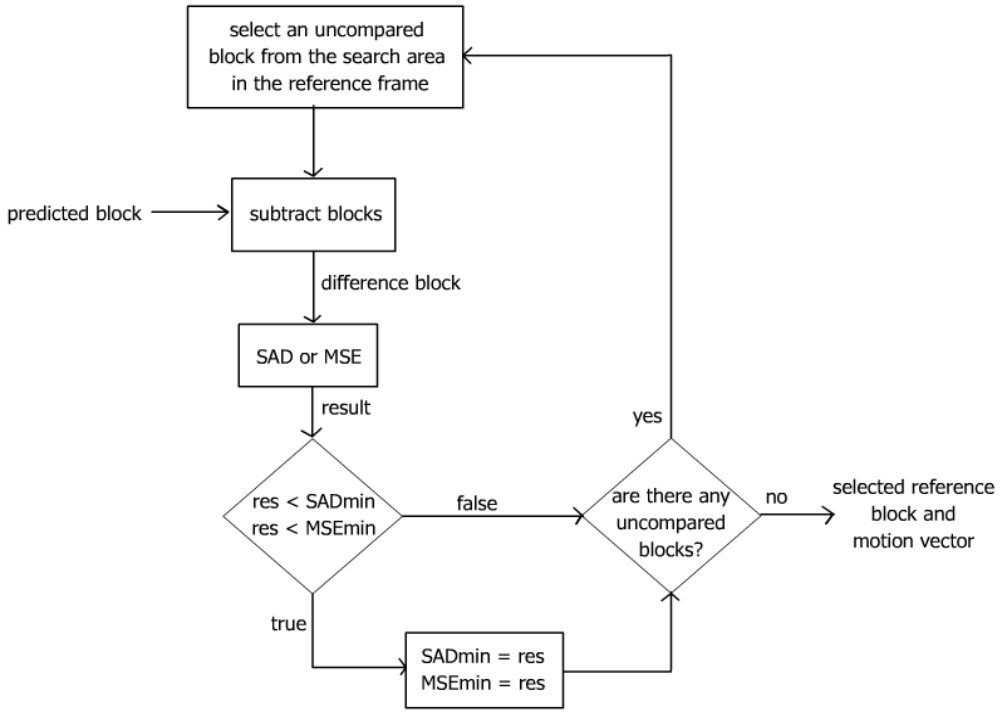


Figure 2.22. Full motion search

Generally all of candidate blocks in the search area are not tested, since it is not a very efficient. Most motion search algorithms compare only a part of all possible blocks in the search area. Generally the size of the search area is selected as 32 x 32 pixels. In MPEG-1-2, motion search is realized by using macroblocks(16x16) of luma frame(Y). A separate motion search for chroma frames(U and V) is not computed. Instead, the same vector that is computed for luma macroblock is used for corresponding U and V macroblocks. If color encoding format is YUV420, the coordinates of motion vector found for Y macroblock is divided by 2 to find the motion vectors of corresponding U and V blocks. In the following motion search algorithms (Turaga and Alkanhal 1998), for demonstration purposes it is assumed that searched block size 8x8 and search area is 16x16.

2.2.3.2.1 Three Step Search

Three step search is a very widely used search algorithm since its simplicity and performance. Firstly, a step size is set such as 8 pixels. This step size shows the distance of firstly selected candidate blocks to the center of search area.

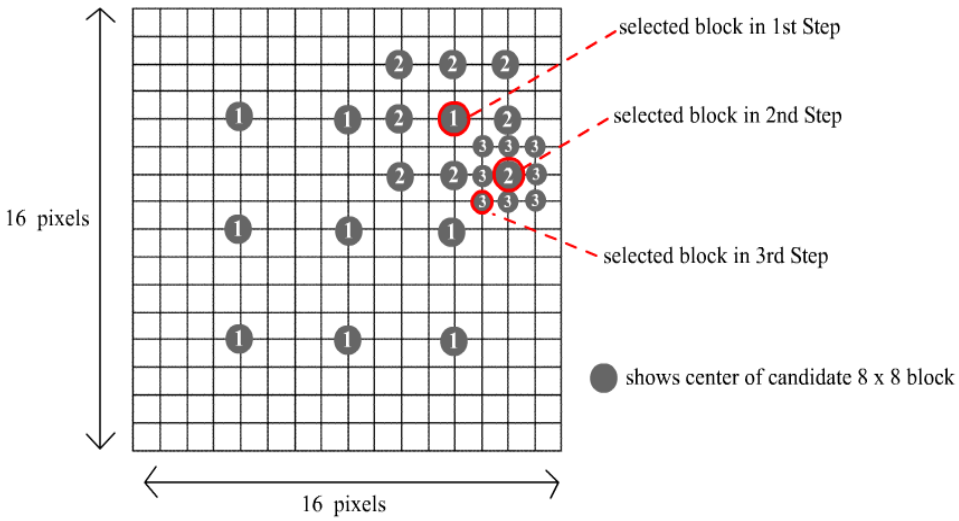


Figure 2.23. Three-step search (initial step size : 4)

At each step, SADs (Sum of Absolute Differences) for 9 candidate blocks (one is at center and others are at the main and inter-main directions) are computed, and the

block with the minimum SAD is selected as the center position for the next step and the step size is halved. This step is repeated until the step size becomes smaller than 1. The selected block with the minimum SAD at the last step is the matched block.

2.2.3.2.2 One At a Time Search

One at a time algorithm, as the name implies, picks one candidate block at each comparison step. This is a very simple and effective block based motion search technique.

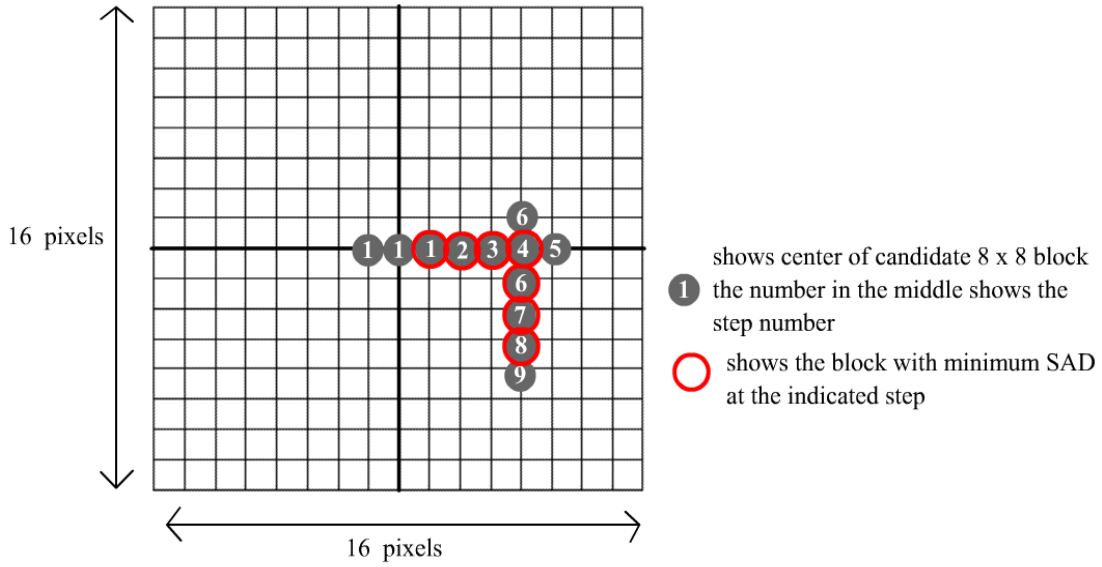


Figure 2.24. One at a time search

This algorithm tries to find the block with minimum SAD on the horizontal direction at first. So, three blocks on the horizontal direction about the center of the search window is selected as first candidates. If the block with the smallest SAD is at the center then the vertical search begins.

Otherwise, the search continues in the direction of the block with smallest SAD while the next candidate block in this direction has a smaller SAD than the current block. So, going on the same direction if the next candidate block has a larger SAD, then horizontal search is ended and vertical search begins from this point.

Vertical search works in the same manner with horizontal search, so the firstly it is decided to go upward or downward according to the SADs. And after one of either directions is selected, the search in this direction continues until a block with larger SAD is found.

2.2.3.2.3 Logarithmic Search

This algorithm is very similar to TSS (Three Step Search) and was introduced at the same time with TSS. Although this algorithm requires more steps than the TSS, it can be more accurate especially when the search window is large (Turaga and Alkanhal 1998).

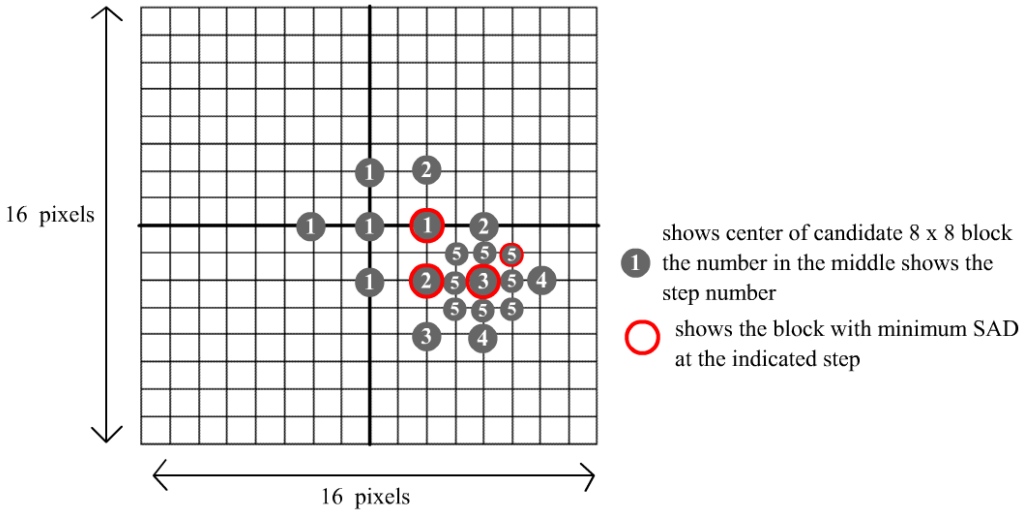


Figure 2.25. Logarithmic search

Firsty a step size is initiated as in TSS. At each step, SADs (Sum of Absolute Differences) for 5 candidate blocks (one is at center and others are at the main directions at the distance of selected step size) are computed, and the block with the minimum SAD is selected as the center position for the next step. If the positon of best match is at the center, then the step size is halved. This step is repeated until the step size becomes equal to 1. When the step size becomes 1, all the nine block around the

center are chosen for comparison and the best among them is picked as the required block (Turaga and Alkanhal 1998).

2.2.3.3 A New DCT Based Motion Search Criteria

Classical motion search algorithms, as discussed above, uses SAD (Sum Absolute Differences) or MSE (Mean Square Error) criteria to find the best match. However, both of these methods do not automatically guarantee the least coefficient count for the currrent block. There is no direct relation between the SAD or MSE metrics and the number of DCT coefficients to be coded.

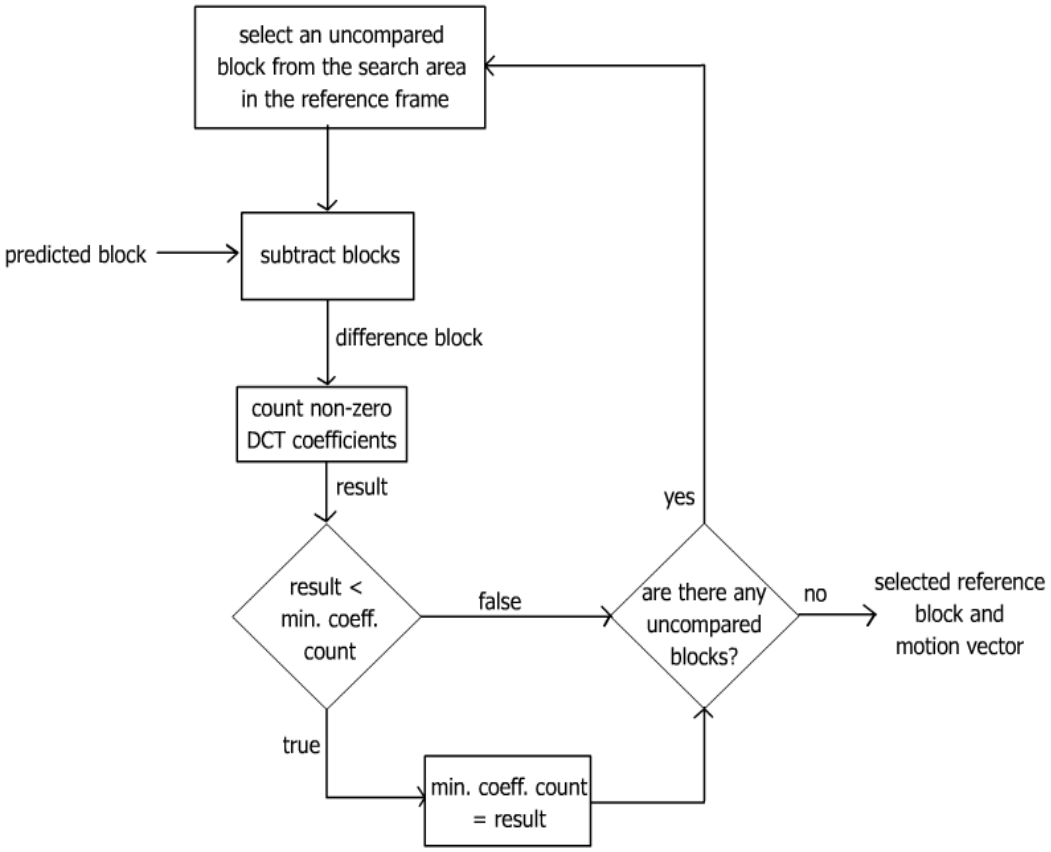


Figure 2.26. Full motion search with DCT based matching criteria

Instead, the proposed method directly uses the number of quantized dct coefficients as a measure of block matching. This method guarantees the least count of

DCT coefficients to be VLC encoded. Each time the current block in the predicted frame is compared with a block in reference frame, the number of quantized DCT cofficients are found by subtracting both blocks taking DCT of the result and applyıng the quantization. The block in the reference frame that gives the least number of quantized DCT coefficients is selected and its motion vector is calculated

2.3. The Structure of Media Streams

What a media player does is roughly saying converting media bit stream into real life audio and video signals. So, understanding how a media bitstream is generated has a critical importance before starting to design a media player. And for the same reason the aim of this section is to introduce the structure of media streams briefly.

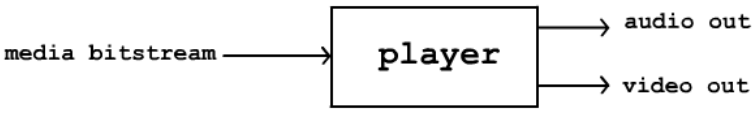


Figure 2.27. Media player as a black box

A media bitstream contains one or more elementary streams of video and audio, as well as other data. A typical media stream is generated by multiplexing audio and video bitstreams into one single bitstream. The player reads media bitstream as input and generates audio and video from that bitstream. Before going on to the design of media player, analyzing the structure of the media bitstreams will provide better understanding.

Media bitstream is generated by an encoding system which is composed of one or more encoders and a multiplexer as seen in the Figure 2.28. Raw (as they are captured) audio and video streams go through the audio and video encoders respectively. The encoder encodes raw stream and feeds the encoded stream to the multiplexer. Each feeded bitstream to the multiplexer can be called as “elementary stream” as in MPEG standard. Then, elementary streams are packetized and a packet header is generated for

each packet. The packet header shows the characteristic of each packet, such as the type (video, audio, etc), presentation or decoding time of elementary stream unit that the said packet conveys (Shibata, et al. 1995). In other words, each elementary stream is packetized with specific headers to the elementary stream and to the packet itself. Multiplexer combines these packets into one single bitstream and also adds information such as how many elementary streams are multiplexed, multiplexing rate, maximum required buffer size after de-multiplexing, type of the elementary streams or the clock reference.

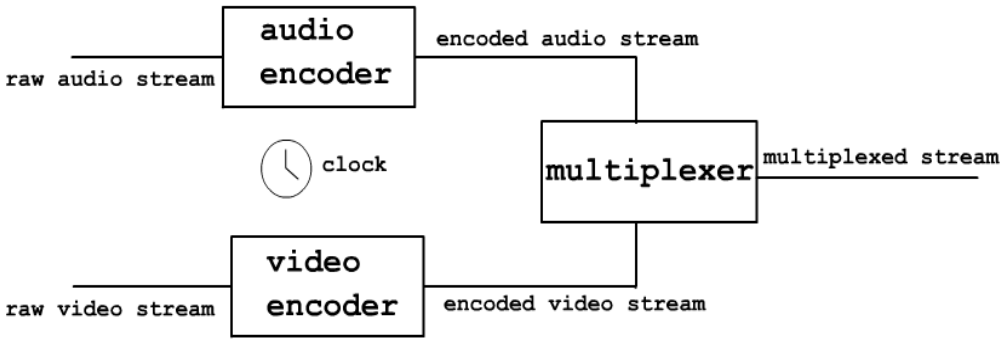


Figure 2.28. The encoding system that generates media bitstream

Data packs that convey the information related with the multiplexing can be thought as a layer which can be called as system layer as in MPEG streams (Hemy, et al. 1999). Elementary stream packets compose a different layer, called compression layer, which lies beneath the system layer as seen in the Figure 2.29.

System layer and multiplexing method describes container format of the media stream. Container formats aim to multiplex one or more elementary streams into one single bitstream. For instance, assume that there are two elementary streams, one is audio and the other is video that is the most common case. Firstly, audio and video data are captured by distinct devices, and then encoded by separate encoders as seen in the Figure 2.28. If it would possible to capture and record audio and video as a single signal, obviously, multiplexing will be unnecessary.

An other important point is the difference between recording way of the audio and the video, that is, video is not very coarsely sampled compared to audio. Video data is

composed of sequential pictures which are captured 25-30 times per second, hence gives an impression of continuity. So while playing video stream, each picture stays on screen about 30-40 milliseconds. However these sequential pictures seems like continuous to us because of sampling rate of our eyes and incredible interpolating capability of our brains. Pictures, can also be called as frames, are the key units for the video recording process. Audio signals are recorded much shorter intervals producing audio frames. Either audio or video encoders work in a frame based manner.

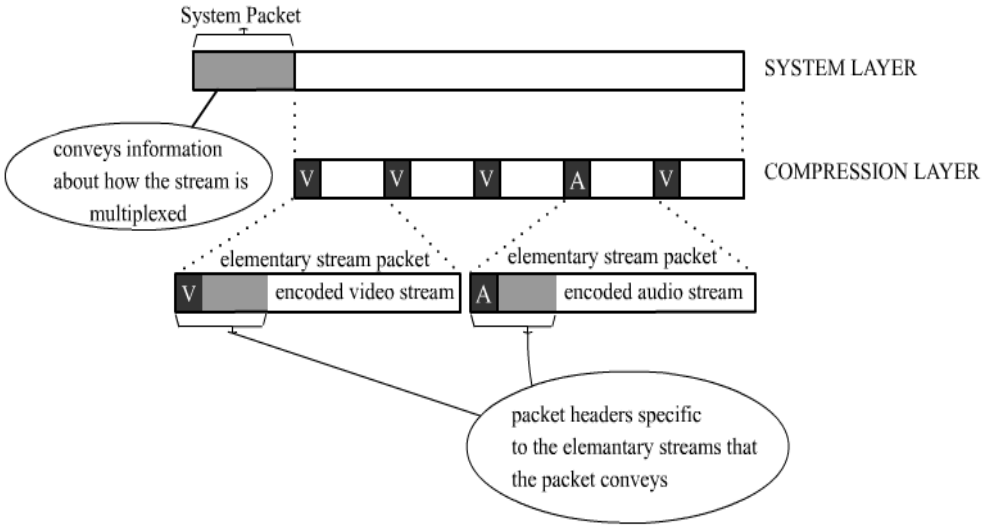


Figure 2.29. The layers media bitstream

The most challenging issue in multiplexing is how to maintain synchronization between audio and video streams (Blakowski and Steinmets 1996). Audio and video synchronization is achieved by injecting timing information such as presentation time, reference clock, and time interval between sequential frames into multiplexed stream (Tryfonas and Varma 1999). The details of audio and video synchronization will be given in the further sections.

2.3.1. Container Formats

Container formats are designed to contain and convey various types of multimedia elementary streams, generally compressed by standardized encoders.

There are various container formats some of which are more suitable for streaming of media and some of which are more suitable for trick modes and local storage. Container formats add an overhead to file size. So same content in different containers may have different file sizes.

Generally container formats suitable for streaming adds more overhead because they are designed to be more error resilient. Container formats have the capability to hold many type of audio and video streams, as well as other media streams. However, some container formats may be dedicated only for one type of stream such as audio stream containers like WAV (short for Waveform), AIFF (Audio Interchange File Format) or XMF (Extensible Music Format).

2.3.1.1. MPEG-2 Containers

ISO/IEC 13818 which is also known as MPEG-2 coding standard has 11 parts (Table 2.1) currently. First part, ISO/IEC 13818-1, also known as system part, describes MPEG-2 container which explains synchronization and multiplexing of one or more elementary streams such as audio and video, or other data into one single stream (Gall 1991). Although, containers described in MPEG-2 part 1 (ISO/IEC 13818-1 1994) typically convey MPEG coded video or MPEG coded audio streams, it is also possible to multiplex elementary streams coded with other common encoders such as h264, ac3 by using MPEG-2 containers.

Table 2.1. Parts of the MPEG-2
(Source: ISO/IEC 13818-1 1994)

ISO/IEC 13818-1	Systems
ISO/IEC 13818-2	MPEG-2 Video
ISO/IEC 13818-3	MPEG-2 Audio
ISO/IEC 13818-4	Conformance testing
ISO/IEC 13818-5	Software simulation
ISO/IEC 13818-6	Extensions for DSM-CC
ISO/IEC 13818-7	Advanced Audio Coding (AAC)
ISO/IEC 13818-8	Video, extension to 10-bit input symbols
ISO/IEC 13818-9	Extension for real time interface for system decoders
ISO/IEC 13818-10	Conformance extensions for Digital Storage Media Command and Control (DSM-CC)
ISO/IEC 13818-11	Intellectual Property Management and Protection (IPMP)

The burden of program stream headers to overall bitrate is minimal, however it is not enough error resilient for communication purposes. Hence, it is used as a container format for storage purposes such as in DVDs. The program stream is also more suitable for realization of trick modes than transport streams.

2.3.1.1.1. MPEG-2 Program Stream

MPEG-2 program stream is a very commonly used container format. The name of an ordinary MPEG program stream file ends with the .mpg, .mpeg or .vob extension. ISO/IEC 13818-1 describes program stream as "a stream definition which is tailored for communicating or storing one program of coded data and other data in environments where errors are unlikely, and where processing of system coding, e.g. by software, is a major consideration (ISO/IEC 13818-1 1994)." Program stream is as stated before more suitable for storage purposes, since not error resilient for communication.

Table 2.2. Syntax of pack header
(Source: ISO/IEC 13818-1 1994)

Syntax	Number of bits
pack_start_code 0x000001BA	32
'01'	2
system_clock_reference_base [32..30]	3
marker_bit	1
system_clock_reference_base [29..15]	15
market_bit	1
system_clock_reference_base [14..0]	15
marker_bit	1
system_clock_reference_extension	9
marker_bit	1
program_mux_rate	22
marker_bit	1
marker_bit	1
Reserved	5
pack_stuffing_length	3
for(i=0;i<pack_stuffing_length; i++){ stuffing_byte }	8

MPEG-2 program stream is composed of packs and packets. Packs convey packets in their payloads. In order to demultiplex program streams as shown in Table 2.2, firstly a pack header must be found. Each pack header starts with a 32 bit pack start code 0x000001BA which is used to identify a pack header Table 2.1. The pack size is variable and the size of the pack is written in the pack header. In VOB files each pack is 2048 bytes. “Program streams may be either fixed or variable bitrate. (ISO/IEC 13818-1 1994)” Bitrate of program stream can be calculated by using multiplexing rate and system clock reference (SCR) values in pack header.

Packets, which are conveyed in the payloads of packs, are generated by packetizing encoded elementary streams. Hence packets that are described in MPEG-2 specification are known as Packetized Elementary Stream (PES) packets. “Transport streams and program streams are each logically constructed from PES packets”, so PES packets can be used as basic units while converting Transport Stream to Program Stream or vice versa. Each PES packet consists of only one type of elementary stream. PES packet header gives information about the elementary stream such as whether it is audio or video and its encoding type, presentation and decoding timestamps, etc.

Table 2.3. Demultiplexing Program Stream

(Source: ISO/IEC 13818-1 1994)

Syntax	Number of bits
MPEG2_program_stream(){ do{ pack() } MPEG_program_end_code}	32

pack(){
pack_header()
while (nextbits() = = pack_start_code_prefix)
{
PES_packet();
}
}
	Variable Length

After packet header is readed according to the Table 2.2, then payload or in other words, bytes until the next pack header are stored as PES packet content. Then, PES packet header is read and payload of PES packet is feed to the appropriate decoder.

2.3.1.1.2. MPEG-2 Transport Stream

MPEG-2 transport stream container is also described in ISO/IEC 13818-1 system part and is designed to be convenient for transport purposes, hence used in broadcast applications such as DVB and ATSC.

PES packets are key structures for both of MPEG containers. So, converting transport stream to program stream or vice versa can be done by using PES packets. Transport streams are composed of packets which are 188 bytes in length and has a 13-bit packet id (PID) number that identifies the packet as shown in Table 2.3. Transport Stream can convey one or more program contents as multiplexed in a single bitstream.

Table 2.4. Syntax of a Transport Stream packet
(Source: ISO/IEC 13818-1 1994)

Syntax	Number of bits
sync byte 0x47	8
transport_error_indicator	1
payload_unit_start_indicator	1
transport_priority	1
PID	13
transport_scrambling_control	2
adaptation_field_control	2
continuity counter	4

if (adaptation_field_control = = ‘10’ || adaptation_field_control = = ‘11’){
adaptation_field()
}
	Variable Length

if (adaptation_field_control = = ‘01’ || adaptation_field_control = = ‘11’){
for (i=0;i<N;i++){
data_byte
}
}
	8

One program content is composed of one audio and one video elementary stream in general. However there can be other elementary streams such as audio for

multilanguage support or subtitles, etc. in a single program. Each elementary stream has a unique PID, so an elementary stream is extracted by obtaining the packets with this unique PID.

The Program Map Table (PMT) is also one 188 byte transport stream packet that shows the PIDs of the elementary streams for a specific program. PMT tables have also unique PID numbers and these pid numbers can be extracted from Program Association Table (PAT) packet. PID of a PAT packet is must be zero and is constant and same for all transport streams, however PID of PMT tables and by the way PID of elementary streams are not predefined and must be read from PAT and PMT tables.

2.3.1.2. Audio Video Interleaved (AVI)

The Microsoft Audio Video Interleaved (AVI) is another very common container format (Zimmerman 2003), and actually it is a specialization of Resource Interchange File Format (RIFF) which is composed of packets called lists or chunks. Chunks are the simplest units and have a 4-byte identifier in human readable form such as “avih” or “idx1” and 4 byte chunk size. Lists can be comprised of lists or chunks and start with “LIST” characters and the 4 byte identifier and 4 the byte list size as seen in the Figure 2.30.

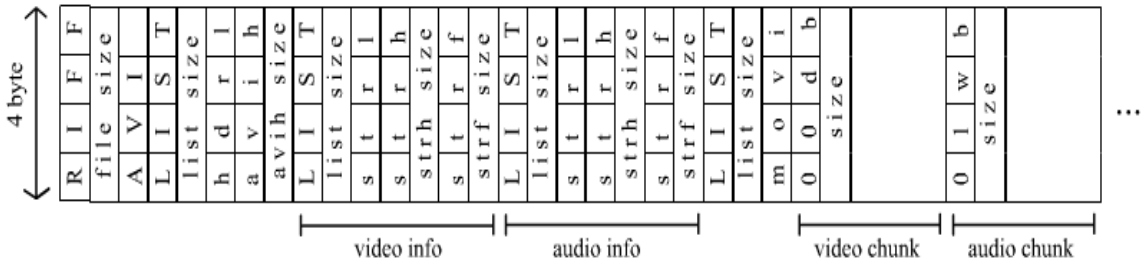


Figure 2.30. Graphical Representation of AVI container format

CHAPTER 3

THE PROPOSED MEDIA PLAYER

3.1. The Design of the Media Player

The presented player in this thesis has a modular and heavily multithreaded design that makes it more maintainable and scalable. As seen in the Figure 3.1, a streaming media player is composed of buffers which are shared via concurrent processes (threads) and modules which are responsible to realize a specific task such as demuxing, decoding, etc.

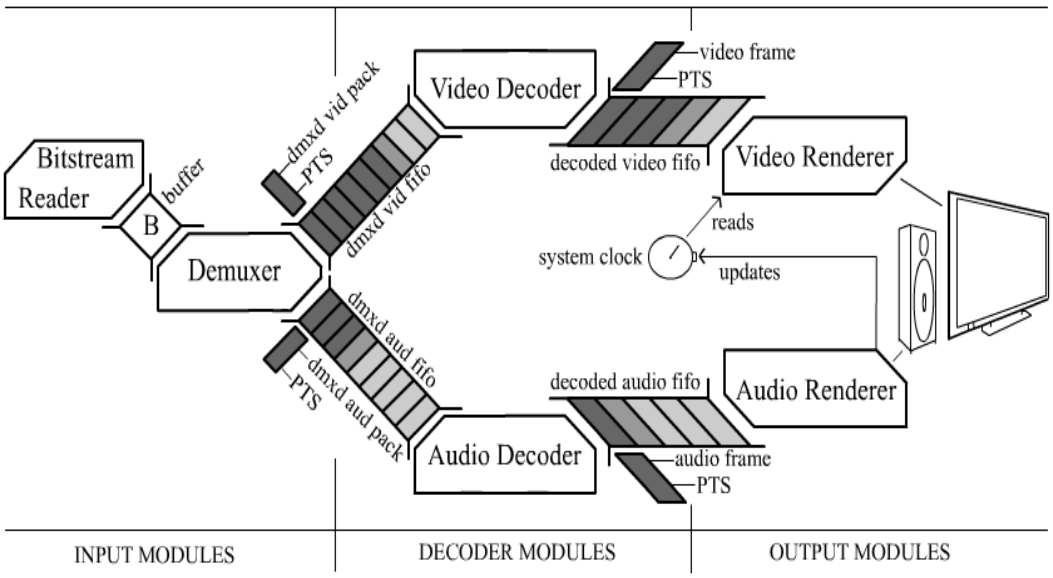


Figure 3.1. The design of the media player

3.1.1. Multithreaded Design

Playing a media stream requires implementation of tasks such as reading media stream from network or a storage device, de-multiplexing read media stream into its elementary streams, decoding de-multiplexed elementary streams and rendering

decoded elementary streams. As seen in the Figure 3.2 each task has its own thread, therefore tasks are running continuously and independently. Each task reads input data from a circular buffer and writes to another circular buffer after processing it. So, implementation of a task can be done in any way if it is compatible with the read and write interfaces of these circular buffers. This makes the abstract implementation of tasks possible and the design gains modularity. Furthermore delay in a task can be compensated by circular buffers between the said task and its dependent tasks.

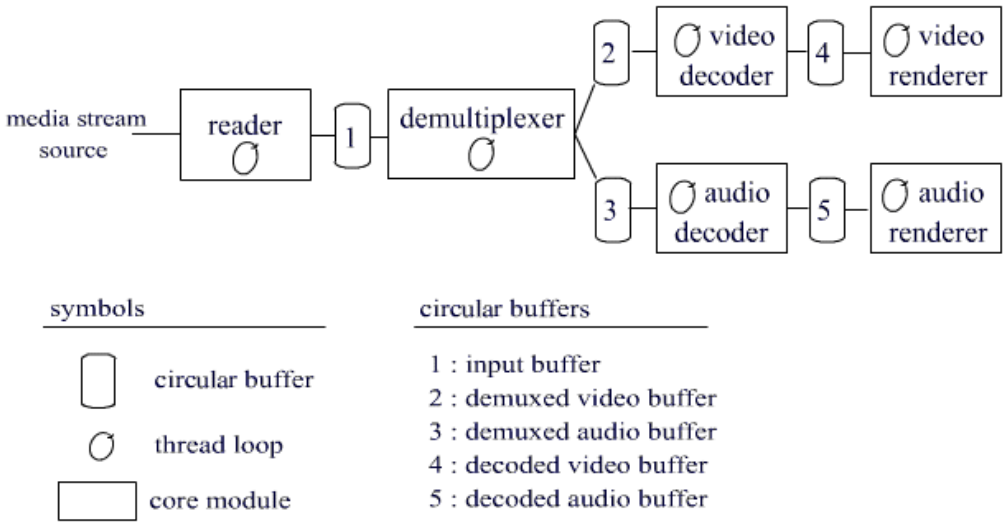


Figure 3.2. Threads and circular buffers of the proposed streaming media player

Since playing audio is a continuous process, it must be done within a dedicated thread. Audio driver of the system may implement this thread internally, so a dedicated thread created by the application programmer may be unnecessary. Although, realizing said tasks serially without using dedicated threads is possible, it requires a tight timing, and therefore this design is more convenient for real time operating systems.

3.1.2. Modules

The presented player uses modules to realize the required tasks in order to play a media stream. A module is implementation of a task such as decoding video, de-

multiplexing media stream, etc. Modules have predefined interfaces for input and output parameters. Therefore modules can implement tasks in any ways according to these interfaces. This hides details of implementation of the task by a module from the other modules that uses the said module.

The modules can be divided into 2 parts as; core modules and plug-in modules. Core modules use plug-in modules to realize a specific task such as mpeg decoding. For instance video decoder module is a core module which uses mpeg decoder module to decode mpeg coded streams. Therefore mpeg decoder module is a plug-in module which will be linked to video decoder core module if the video is mpeg coded. However, if the video is coded with another encoder such as h264, then h264 decoder module will be linked to video decoder core module as a plug-in module.

3.1.3. Circular Buffers

There are five circular buffers used in the proposed architecture shown in Figure 3.2 which have the same functionality and structure:

• Input Buffer is between the reader and the de-multiplexer module. It conveys read data from reader to the de-multiplexer.

• De-multiplexed video data buffer is between the de-multiplexer and the video decoder thread. It conveys the de-multiplexed and encoded video data pack and a PTS value assigned to this pack.

• De-multiplexed audio data buffer is between the de-multiplexer and the audio decoder thread. It is used to store de-multiplexed and encoded audio packs with PTS values attached.

• Decoded video data buffer is between the video decoder and the video renderer. It holds the decoded video frame packs with PTS values.

• Decoded audio data buffer works between the audio decoder and the audio renderer. It conveys the decoded audio frames and their PTS data.

3.1.4. Thread Synchronization on Circular Buffers

For each of the circular buffers, the same problem emerges; a thread produces and adds data pack to the circular buffer and another thread consumes and gets data pack

from the circular buffer. Two different threads may access the same buffer at the same time or the buffer may be completely full or empty. This problem is called “the producer-consumer problem” or “bounded buffer problem”. One should avoid the access of different threads at the same time, stop the producer thread when the buffer is full or stop the consumer thread when the buffer is empty. Proposed design for streaming media player requires sharing of data over circular buffers between simultaneous tasks as graphically reprensented at Figure 3.3.

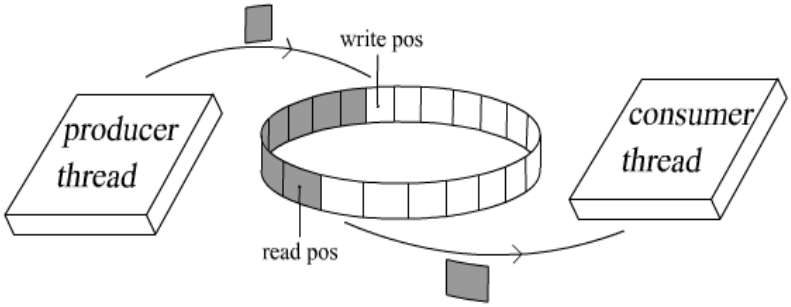


Figure 3.3. Producer - Consumer problem for circular buffers.

In presented player, this problem is solved by using two semaphores and a mutex for each of circular buffers. Writing to and reading from same position at the same time is prevented by using a mutex.

Table 3.1. The pseudo code for producer consumer problem

```
// initializations
#define PACK_CNT 10
full_smphr = init_smphr(PACK_CNT, PACK_CNT);
empty_smphr = init_smphr(0, PACK_CNT);
data_mutex = init_mutex();
//producer
Wait_for_available_slot(full_smpr);
Get_mutex(data_mutex);
Add_pack();
Release_mutex(data_mutex);
Increment_available_slot(empty_smphr);
//consumer
Wait_for_available_slot(empty_smpr);
Get_mutex(data_mutex);
Get_pack();
Release_mutex(data_mutex);
Increment_available_slot(full_smphr);
```

Assume that the function prototype to initialize a semaphore is init_smphr(available_slot_num, max_slot_num), then the pseudo code in the Table 3.1 can be written as solution to producer-consumer problem.

Selecting reasonable sizes for the circular buffers is also an important issue. In the computation of the size of the de-multiplexed video buffer one should make sure that the total duration of de-multiplexing, decoding and presentation of a frame must not exceed 1000 / fps milliseconds. For example, for a video stream with 25 frames per second, sum of de-multiplexing, decoding and presentation must not exceed 1000 / 25 = 40 milliseconds.

$$\text{de-multiplexing} + \text{decoding} + \text{presentation} <= 1000/\text{fps}; \quad (3.1)$$

Above inequality also enforces following inequality;

$$\text{decoding} < 1000/\text{fps}; \quad (3.2)$$

Assuming the worst case, in which decoding takes 1000/fps milliseconds, de-multiplexing will continue to fill circular buffer. So, if bitrate of video stream is B kbps, then FIFO buffer will be filled with B fps Kbits. This is the worst case, so we can compute the size of de-multiplexed video FIFO buffer using the following formula:

$$\text{dmxd\_vid\_cbuf\_size} = \text{video\_bitrate} / \text{fps}; \quad (3.3)$$

Size of the de-multiplexed audio circular buffer can be computed by applying the same methodology. If playing an audio frame takes N milliseconds, then

$$\text{dmxd\_aud\_cbuf\_size} = \text{audio\_bitrate} * N /1000; \quad (3.4)$$

Most of the system memory is used by decoded video circular buffer, because a pack in the decoded video circular buffer is composed of a decoded video frame and a PTS value. The memory required to save a decoded video frame is very large when compared to the decoded audio frame, encoded video or encoded audio frame.

Size of the memory to save a decoded frame (in YUV420 colorspace) is 1.5 * frame_width * frame_height bytes. In order to use memory efficiently, the buffer required to save a decoded frame pack is allocated before saving each decoded frame and is de-allocated after the decoded frame is presented. So if there are no decoded frames in the decoded video circular buffer, the decoded video circular buffer will not take any memory.

Pack count for the decoded video and audio circular buffers can be set to unity; since it is more reasonable to store audio or video frames in de-multiplexed circular buffers before they are decoded.

3.2. How the Player Works

The media player processes media bitstream in order to generate audio and video output. This process includes following steps;

- Firstly, reader module must be selected according to the stream source
- Secondly stream type must be identified.
- Thirdly stream must be demultiplexed with a proper demuxer.
- Demultiplexed elementary streams must be decoded by proper decoders.
- Lastly, decoded streams must be rendered with proper renderers without losing lip-sync between audio and video outputs.

3.2.1. Deciding Appropriate Reader Module

The presented player aims to access a variety of sources such as lfs (local file system), http, mms, udp, rtp, rtsp, rtcp, dvb, dvd, etc. Therefore, a distinct reader module must be implemented for each protocol. Furthermore, each reader module must be used via a predefined common interface; hence the other modules in the player will be unaware of which reader module is selected actually.

For this reason, a core reader module is implemented which is the common gateway to the reader modules. Selected reader module will be used via this core reader module as seen in the Figure 3.4.

It is not necessary for a reader module to support all features written in common interface. For instance, seek feature while reading will not be supported for dvb, udp or rtp. Because these protocols are not suitable for seek operation.

Selection of the appropriate module can be done by parsing the filename or by a parameter which is given by the user. For instance, if the filename is started with http://, then http reader module will be selected. Additionally, user can enforce the player to select a specific reader module via command line or graphical user interface.

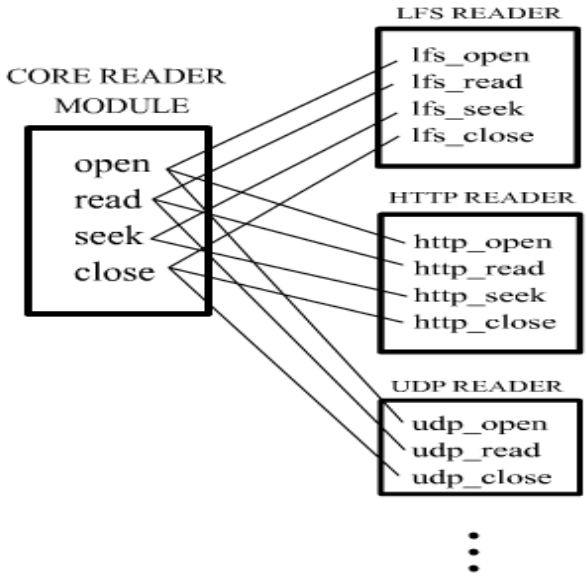


Figure 3.4. Core reader module and plug-in reader modules

3.2.2. Inspecting Stream Type

In order to start playback of a media stream, the player must detect features of this stream. At first container type must be detected. Although filename extension of a media stream usually shows its container type such as avi, ts, mov, mpg, etc., deciding container type by just according to the extension is not a proper way. So, it is also required to scan stream to find unique symbols specific to the container type to be sure that the stream's container type is what the extension shows if there is a filename extension. Otherwise, inspecting of the stream will continue until its container type is

detected. If the player cannot find container type of the stream, it will return that playback is failed because of unknown container type.

For instance, transport streams are usually named with extension ts. or trp. after they are recorded. So, if the extension of the name of a media file is ts or trp, it will be rational to check stream if it is really a transport stream at first. For this reason, distinct features of a transport stream from other container types must be known. That is a transport stream is composed of packets with 188 bytes and each packet has a sync byte as first byte which is equal to 71 in decimal, so a part of the stream can be scanned to check if this policy is valid.

The amount of the stream required to detect container type is variable for different container types. However, scanning more stream will give more accurate results in general.

The container type is the first required data to inspect the media stream, after the container type of the stream is found, it is possible to gather information like how many elementary streams that media stream includes, or codec types of the elementary streams. For this reason, according to the container type, headers that give information about the elementary streams of that media stream must be found. The place and structure of these headers is distinct for each container type. For instance, for avi format this information can be found just at the start of the stream, however for mpeg streams it may require to scan stream for a while to reach the required headers.

Encoding types of the elementary streams must be found at first to be able to select appropriate decoder modules for decoding. In general, this information can be found at the system layer, that is container type of the media stream, however it also requires checking elementary stream is really encoded with the encoder that is written in system layer.

For this reason, the media stream must be de-multiplexed for a while until a part of the elementary stream is extracted. It is attempted to decode the extracted elementary stream by the decoder offered in the headers of container. If it is decodable then the offered decoder is selected as decoder module, otherwise the other decoder modules that-the player supports tried until the proper one is found. If an appropriate decoder cannot found, the player returns that the playback is failed because of unknown codec type.

If the media stream includes video elementary stream, it is also required to inspect frame rate and frame size in width and height as well as its codec type. However, said

features of the video stream may not be found in the container headers. In that case, the features of the video stream must be extracted from the video elementary stream which requires de-multiplexing of the media stream to extract the video elementary stream.

The frame size is required to allocate decoded frame buffers and to initialize video renderer module. Video rendering and video decoder module also requires frame rate of the video stream which will be used when assigning or validating presentation timestamps of the decoded video pictures to provide a smooth playback.

If the media stream includes audio elementary stream, it is also required to inspect channel count, sampling frequency as well as its codec type. However, said features of the audio stream may not be found in the headers of the container. In that case, the features of the audio stream must be extracted from the audio elementary stream which requires de-multiplexing of the media stream to extract the audio elementary stream.

In general, the media players include a stream inspector module which inspects and gathers all of the required data before the start of playback. However, the presented player does not include a discrete stream inspector module. Instead required information is gathered as the media stream flows through the modules which makes inspection of media stream rather faster and does not require a complex stream inspector module.

As discussed above, to gather information such as frame size for video or sampling rate for audio may require de-multiplexing and decoding media stream for a while in a discrete stream inspector module. Therefore, the stream inspector module needs de-multiplex and decode capability for a variety of containers and codecs which increases the complexity of the inspector module.

As seen in Figure 3.5, inspection and gathering all of required data is not realized by a single module. When data reach to the de-multiplexer core module, it tries to find out the container type of the media stream, since proper de-multiplexer plug-in module will be selected according to the container.

After container type is found, demultiplexer module begins to realizing its task without need of any other information about the media stream. Video decoder core module waits for buffer number 2, the de-multiplexer will extract elementary video stream to this buffer. If stream reached to video decoder core module it first tries to inspect stream to find out the codec type of the video stream.

After codec type found, the video decoder core module calls the proper decoder and decoding loop begins. After the decoding of the first frame we know the frame size,

and frame rate precisely. Therefore, we can initialize the video renderer after the decoding of the first frame.

Features of the audio elementary stream are extracted at the audio decoder core module with the same way. If media stream contains only one elementary stream, for instance if there is no audio in the media stream, the buffer 3 will never be filled, so the audio core decoder module will never be started for the said media stream.

So as seen in the Figure 3.5, the required data is inspected at where it can easily be gathered which makes stream inspection simpler and faster.

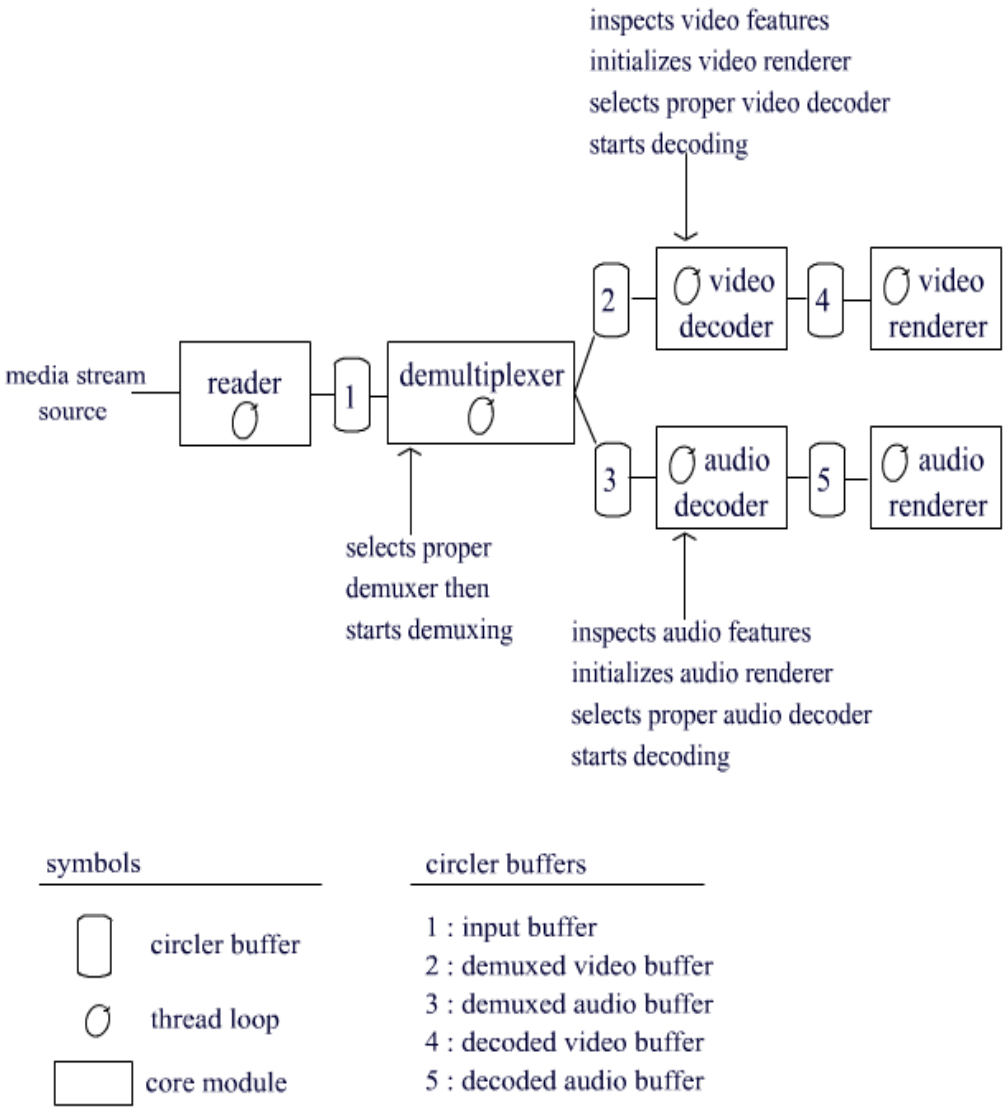


Figure 3.5. Stream inspection as the media stream flows through modules.

3.2.3. Demultiplexing

As discussed in the previous section, the structure of the media streams, media streams are composed of one or more multiplexed elementary streams.

The media streams are multiplexed in various ways which are defined by their container type. The presented player aims to play various media formats such as avi, ts, mpg, mp4, mov, etc. and each of these container formats require a distinct de-multiplexer implementation. Furthermore, there must be a common interface for these various container formats, so the demultiplexer core module can be linked to the proper demultiplexer plug-in module which performs the demultiplexing for a specific container format via this common interface. So, the other modules of the player will be unaware of the actual demultiplexer plug-in module, and will interact only with the demultiplexer core module which is already linked to the actual demultiplexer plug-in module.

At first, demultiplexer core module tries to find appropriate demultiplexer plug-in module by analyzing data that it takes as input. After the proper plug-in module is selected, the core module starts a demultiplexing thread which has a demultiplexing loop that calls get_new_packet() continuously. get_new_packet() returns new demultiplexed packet and its features. (e.g. is it audio or video, does it have presentation timestamp? etc.)

Demultiplexer core module has its own thread running and it continues to demultiplex while one of demultiplexed output buffers (buffer 2 and 3 in the Figure 3.5) is not full. Other-wise demultiplexer will wait for the relevant decoder to decode demultiplexed packs. However this is not a desired condition in the case of live streams, because it will cause overflow and hence data losses.

3.2.4. Decoding

There are various video and audio compression formats such as MPEG1/2 video, H.264, DivX for video and MPEG audio, ac3 for audio. Each compression format, i.e. encoding method, requires a distinct decoder. The presented player aims to support various audio and video codecs. Therefore the presented player has decoder plug-in modules, each of which implements a different codec. There are two decoder core

modules in the presented player, one is for audio and the other is for video. The decoder core modules provide a common interface for decoder plug-in modules.

Decoder modules also have their own threads, and decoding continues while relevant demultiplexed circular buffers (with the number 2 and 3 in the Figure 3.5) is not empty or relevant decoded data circular buffers (with the number 4 and 5 in the Figure 3.5.) is not full.

For instance, the video decoder module reads encoded video data and a PTS value from demultiplexed video data buffer 2 in Figure 3.5. When the decoding of each frame is completed, last read demultiplexed pack's PTS value is assigned to this frame and the frame is stored in the decoded video data buffer 4 in Figure 3.5. However, not all frames have a PTS value read from the bitstream. For such frames, PTS value is computed from the old pts values and frame rate using the following formula:

$$New\_PTS = Old\_PTS + 1000 / frame\_per\_seconds; \quad (3.5)$$

$$Old\_PTS = New\_PTS; \quad (3.6)$$

The video decoder core module, firstly tries to detect codec type of the demultiplexed video stream that it reads from the circular buffer 2 in Figure 3.5. After the codec type is detected, the proper video decoder plug-in module is selected accordingly. Then, the video decoder core module starts a loop that calls video_decoder_decode continuously. This function is the main decoding function that must be implemented by each video decoder plug-in module. It hides most of the complexity of decoding process and provides a simple and efficient interface to interact with the decoder. It is required that the video decoder plug-in module operates on a frame-by-frame basis to be compatible with the core decoder module.

For instance, the C like pseudo code in Table 3.2 shows the video decoder core module that calls video decoder plugin module in a loop. VideoDecoderContext is the structure that holds information about the decoded video stream such as hor_size, ver_size or frame_rate. Except codec_id and private data, all of the members of this structure must be set by the decoder plug-in module. Private data is a generic void pointer argument. It is set to non-NULL only if the decoder in use needs a private argument not specified by the other fields.Since encoders and hence decoders, work

with YUV frames VideoFrame is the structure that is defined to hold the luma and chroma components of the decoded frame.

Table 3.2. The interaction between the core and the plug-in video decoder modules.

```
bool decode loop()
{
        unsigned char * demuxed_video_start;
        unsigned int demuxed_video_size;
        int got_video_frame;
        VideoFrame video_frame;
        VideoDecoderContext video_context;

        DmxVideoCircularBuffer_read(&demuxed_pack);
        demuxed_video_start = demuxed_pack.data;
        demuxed_video_size = demuxed_pack.size;
        while (demuxed_video_size > 0)
        {
                decode_byte_cnt = video_decoder_decode( &video_context,
                                                        &video_frame,
                                                                &got_video_frame,
                                                                  demuxed_video_start,
                                                                  demuxed_video_size);
                if (decoded_byte_cnt < 0)
                        printf("Error at video decoding.");

                if (got_video_frame)
                {
                        DecVideoCircularBuffer_write(video_frame);
                }
                demuxed_video_start += decoded_byte_cnt;
                demuxed_video_size  -= decoded_byte_cnt;
        }
        return true;
}
```

Video decoder core module should transfer a new chunk of input data that is buf_ptr in the following prototype, to the decoder by the video_decoder_decode function as seen in Table 3.2. VideoDecoderContext may be updated by the de-coder plug-in module, therefore transferred as a pointer. The size of the given coded data chunk must be given to the decoder plug-in module side by using buf_size. Video decoder plug-in module will continue to decode given input coded data until a frame is decoded or end of buffer is reached. If decoder plug-in module achieves to decode a frame with given buffer, decoded frame will be returned in structure named VideoFrame, and got_video_frame will be set to true. Return parameter of this function will show how many bytes consumed from the buf_ptr. If got_video_frame is true, i.e. a frame is successfully decoded; the return value can be less than buf_size. Other-wise it must be equal to buf_size which means all of given buffer chunk is consumed and no frame is decoded.

3.2.5. Rendering

Implementation of the video output is platform dependent. For instance, directX library can be used on Windows, where directfb is used on Linux. Moreover, SDL can be used for both of these OS. Furthermore, there may be many options to implement video rendering on a single platform. Therefore, the presented player has video rendering plug-in modules each of which uses a distinct rendering library that can be accessed via a video renderer core module.

Implementation of the audio output will also be differing from platform to platform. And there also be many ways to implement audio rendering on a single platform such as ALSA and OSS on Linux. Therefore, audio renderer plug-in modules can be implemented for distinct libraries. Each audio renderer plug-in module must be implemented according to the common interface defined by the audio renderer core module.

The Renderer core modules have their own threads run-ning. Audio renderer core module has a great importance in providing inter synchronization, since it sets and updates system clock according to the PTS values of audio frames and when writing data to the audio driver’s buffer blocking IO method must be used.

Video renderer core module displays the video frames according to the frame rate by using PTS values and the system clock which is updated by the audio renderer periodically to maintain intra and inter synchronization.

In fact, the presentation of video frames by the video renderer may not always be with a constant framerate, instead a fine adjustment in the frame rate can be done to ensure synchronization between server and client applications with-out introducing a perceptible distortion.

For example, for a video stream with 25 fps framerate, the presentation duration of each frame must be 40 milliseconds. However in case of live streams, this value may need to be greater or smaller than 40 milliseconds to provide synchronization between server and client applications.

Furthermore, to keep video in sync with audio, some video frames may be presented longer or shorter than normal. Generally, these adjustments need not be applied very often, and the adjustment is done within a certain limit so that the smoothness of video playback is not lost.

3.3 Synchronization

Synchronization problems can be divided into two category: audio-video synchronization problems which is also called as intra synchronization and server-client synchronization problems which is also known as inter synchronization.

3.3.1. Audio-Video Synchronization

Audio-video synchronization is one of the most challenging issues in media player architecture design. This is due to the fact that audio and video data are captured by distinct devices, and then encoded by separate encoders as seen in the Figure 3.6. If it would possible to capture and record audio and video as a single signal, obviously, there will be no synchronization problem.

If audio and video would be transferred through different channels without multiplexed and delays because of en-coding or channel property are ignored, it would be easier to play audio and video in synchronization. However, obviously total time for encoding, decoding, transporting of audio and video streams will not be equal. As seen

in the Figure 3.6., if dA is the time difference between an audio frame is captured and played, and dV is the time between a video frame is captured and showed, then synchronization can be provided if and only if dA is equal to dV.

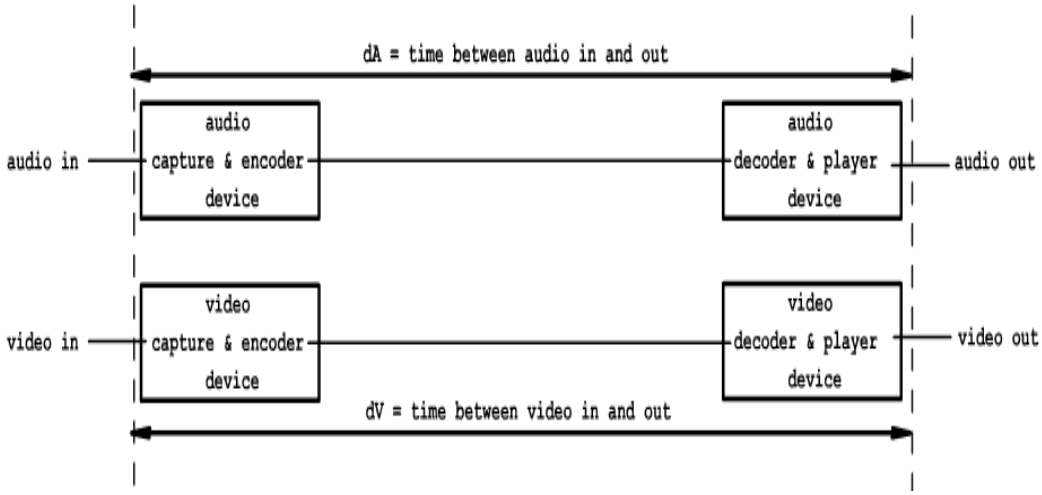


Figure 3.6. Synchronization of audio and video in case of separate channels.

However, if the audio and the video stream are multiplexed into one single bitstream at encoder or server side as seen in the Figure 3.7, then multiplexed stream will be send over one single channel to the client or decoder side where it will be demultiplexed to the audio and the video streams. The channel between the encoder and the decoder system can be thought as a storage device as well as any network link. The encoding and the capturing duration of the audio and the video streams are not equal, therefore the multiplexer must follow a strategy to make it possible for the player system to play audio and video in synchronization.

There are two widely known strategies used in multiplexing for the sake of audio video synchronization. First strategy is to note entering times of the video and the audio frames to the recording system. Each of the audio and the video frame can be attached with a timestamp that shows their capture time. Actually, the capture timestamps can be thought as the presentation times of frames, hence called as presentation timestamps. So, the presentation timestamps are assigned according to the recording system clock, which is generally referred as the system clock. The position and order of the audio or

video frames on the multiplexed stream does not bear any timing information. Instead, timing and synchronization is achieved through the use of time stamps inserted into the multiplexed stream by the encoding system. These stamps (audio and video PTS) must be correctly used to attain a perfectly synchronized presentation. The use of these PTS values in a proper way is up to the media player.

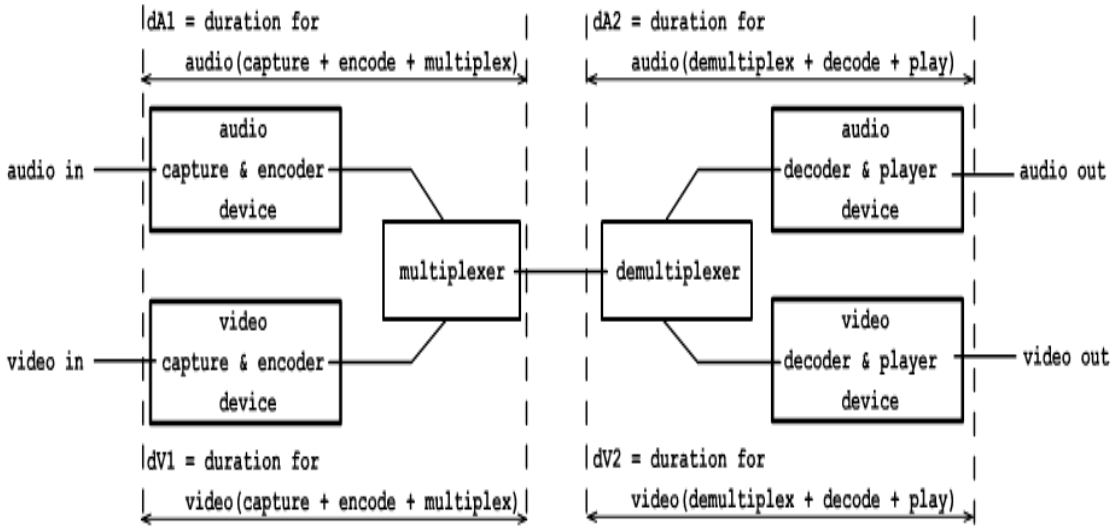


Figure 3.7. Synchronization of audio and video in case of single channels.

The second strategy is simpler but not convenient for transporting over networks. Since the frame rate is constant during the video sequence, the time interval is equal and constant for two consecutive video frames. Audio sampling rate which shows how many audio samples must be played in a second is also a constant value. So, instead of attaching a presentation timestamp to each frame, the sampling rate of the audio stream and the frame rate of the video stream can be inserted to the headers of the multiplexed stream as in avi containers. The presentation time (in milliseconds) of each frame can be calculated by using the frame rate and frame number as in the following:

$$\text{time interval} = 1000 / \text{frame rate} \tag{3.7}$$

$$\text{PTS of ith frame} = \text{start\_offset} + (\text{i} \times \text{time interval}) \tag{3.8}$$

First pts of audio or video can be a non-zero value. In other words, pts values of the first audio and first video frame can be not equal. So, if the pts value of the first frame is not zero, then this value can be used as start_offset in calculation of pts values of later frames as in the above formula.

The presented player requires a pts value for each decoded (video and audio) frame which can be obtained from multiplexed stream or can be calculated by using the frame rate as in the above formula. The player also requires a master clock in milliseconds resolution which will show the current time in player domain. The system that the player works on must already have a clock that works at least at milliseconds resolution. Therefore the rate of the system clock can be used as reference for the master clock. So, there will be an offset between the master clock and the system clock. In other words the master clock can be calculated any time by adding this offset to the system clock.

Audio must be played continuously. Playing of audio is realized by feeding the buffer of the audio driver. In the conventional methods, the audio frame is feeded to the audio driver's buffer if the audio frame's pts value is equal to the player's master clock. Then the audio driver begins to play this buffer. There are critical problems which arise mainly from playing audio due to the fact that audio driver, as well, has its own clock. Thus, the time required to play an audio frame depends on the audio driver's clock rate. Assume that an audio frame begins to play on the right time according to its PTS value. If audio driver's clock rate is faster, the playing of the audio will be finished early, so next audio frame will not be played just after previous audio frame. There will be perceptible gaps (silence) between audio frames. Similarly, if the audio driver's clock is slower, then the audio frame will be played longer and again there will be perceptible distortions. Because, the audio driver has its own clock and the clock rate of the player's audio driver's clock may not be equal to the encoding system's audio driver's clock. Conventional player's solve this problem by dropping or duplicating audio frame which causes perceptible defects.

However, there is another known solution of this problem. In this solution method, audio driver's buffer is feeded with the next audio frame when just after the playing of current audio frame is completely finished. So, audio pts values are ignored. Obviously this will prevent gaps or overwritings, however synchronization between audio and video will be lost by the time. Because if the audio driver's clock rate is faster, audio will begin to playing before corresponding video frames. Otherwise audio

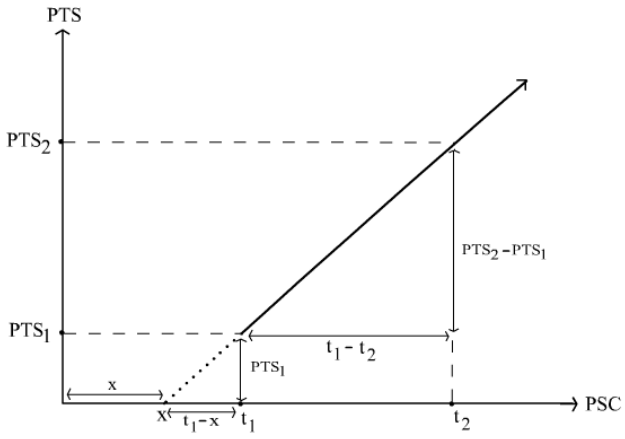
will be late for corresponding video. Therefore playing rate of video is must be updated according to the playing rate of audio.

Audio driver's buffer is set to blocking io access. So, audio driver will prevent overwriting, and any write request will be waited until the driver's buffer become empty. If the audio driver's buffer is empty next waiting audio frame is feed to the said buffer, and pts value of this frame is assigned as master clock. As discussed above, master clock is some offset plus system clock; therefore actually this offset will be updated.

Audio driver's buffer is set to blocking io access. So, audio driver will prevent overwriting, and any write request will be waited until the driver's buffer become empty. If the audio driver's buffer is empty next waiting audio frame is feed to the said buffer, and pts value of this frame is as-signed as master clock. As discussed above, master clock is some offset plus system clock; therefore this offset will be updated.

Indeed we have three independent clocks to reconcile before ensuring a seamless playback, first the players system clock, second the encoders time stamps which are assigned according to the encoder's clock and third the audio drivers own internal clock.

Another problem with the audio is the delay which occurs just before playing an audio frame because of the initialization of the audio buffer. Due to this delay, an audio frame which is planned to start playing at time T, will be played after a delay, at time T + dt. So, this dt value that shows delay that must be taken into account to reach an accurate synchronization.



t1, t2, …tn : nth audio frame start playing

PSC: Player's System Clock.

Figure 3.8. Relation between players' clock and audio timestamps

Figure 3.8. describes the relationship between the three clocks. The horizontal axis is the player's clock PSC, the vertical axis is the PTS values received from the stream. The points on the horizontal axis t1, t2, tn represent the times that the audio driver asks for a new audio sample. The difference between the successive points (tn-tn-1) is nearly constant. Instead of dealing with the audio drivers clock directly, it is more useful to know the t values for our purpose. There is a first order relation between the PTS values and the PSC.

$$PSC = ?*PTS + x \tag{3.9}$$

$$? = (t1-t2) / (PTS2-PTS1) \tag{3.10}$$

$$x = offset \tag{3.11}$$

From Figure 3.8, it is understood that the rate of increase in PTS values may not necessarily follow the rate of increase in system clock, due to the obvious fact that the player system and encoder system have two independent system clocks, that is, the slope of the line in Figure 3.8 may not exactly be 1. Therefore, in calculating the offset value x, this must be taken into account. Instead of accepting the initial offset value x = t1 – PTS1, the exact offset value can dynamically be computed each time a new PTS value is read from the stream by the equation (3.12);

$$offset = x = t1 – (PTS1 * (t1 – t2)) / (PTS2 – PTS1) \tag{3.12}$$

By using the equations above, we indeed couple the en-coders clock to our actual player clock through a first order relation. Using this relation, we first convert the PTS value of a frame to a new time value and if this value matches with the players clock we present the frame. The coefficients in the above equations will be dynamically updated each time a new PTS is received resulting in a more accurate synchronization as playback continues.

There is another thing that the Figure 3.8 and the above relations hint. This is the fact that audio drivers clock is the master since each time the audio driver asks for a

new sample, we feed it and all other things including video presentation follows this as slaves. For instance, if the pts value of the just decoded video frame is equal to the master clock it will be showed. Since master clock is updated by audio driver, and video player uses master clock to compare video frame's clock, audio and video will be synchronized. This is natural because in a media playback system audio is most delicate part to handle as it must be perfectly continuous.

3.3.2. Server-Client Synchronization

There is an increasing demand on the transmission of media streams over networks, particularly on the Internet (Garner, et al. 2006). Streaming servers (Abdel-Baki 2003) send the data at a prescribed average data rate. This average data rate is maintained by scheduling algorithms (Jarmasz and Georganas 1997).

Early media players for decoding audio and video typically required that the entire content be downloaded on the local computer before the player starts playing. Recent players began to support streaming capabilities by buffering some data and starting to play before the entire content has arrived. If the data rate of the incoming media stream is not sufficient, the player pauses and continues to play when its buffer is filled again. Buffering also compensates for jitter in the channel.

The main challenge is smooth playback of audio synchronized with video under varying network conditions without buffer overflow or underflow. This is mainly accomplished by adjusting the playback rate according to the server's clock rate (Garner, et al. 2006). The previous methods used to accomplish this task are given in the Appendix A.

The presented streaming media player dynamically changes its playback rate according to varying network conditions for continuous and smooth playback of streaming media (Guo, et al. 2001). The player normally plays at an original data rate defined by stream parameters. When input packets start to arrive faster or slower, the player does not stop to rebuffer, generates a clock state based on the difference between the server's clock value and the player's clock value, and adjusts the player's clock value based on the generated clock state. Audio pitch is unchanged as it is slowed and video frame rate is slowed as necessary.

The proposed streaming media player varies the rate of output of the mentioned media stream from the mentioned output device to ensure smooth playback of audio and video together, based on the clock state value, wherein the speed change varies according to the formula (3.13):

$$s = m\,(SC(t) - PC(t)) / (SC(t) - SC(t\text{-}1)) \qquad (3.13)$$

where s is the speed up or slow down ratio with respect to the original speed; m is the maximum allowable slow down ratio with respect to the original speed; PC(t) is the player's clock value at sampling time t; and SC(t) is the server's clock value at sampling time t.

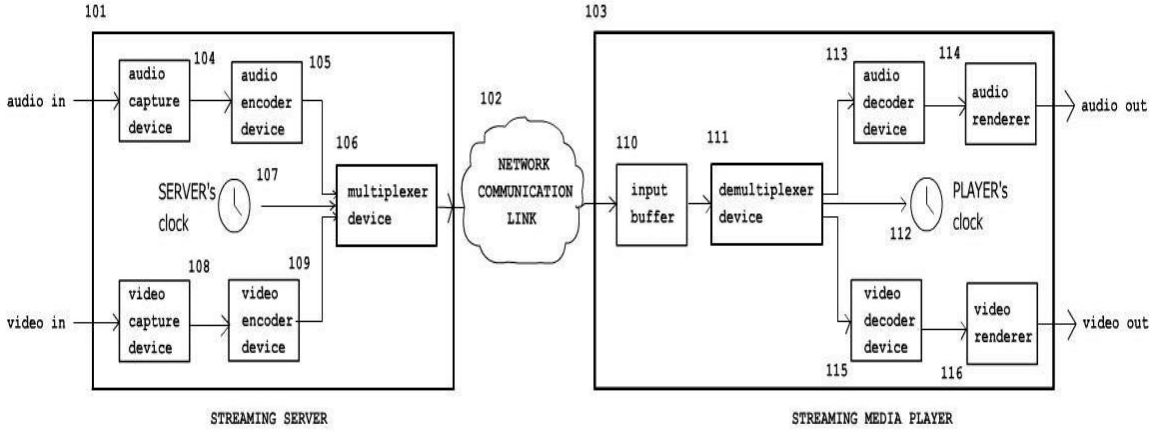


Figure 3.9. Prior art: conventional streaming media player and streaming server

Figure 3.9 describes the prior art, including a standard streaming server 101, a network communication link 102, and a standard streaming media player 103.

In a standard streaming server 101, audio, video and other inputs are input to the streaming server through the capture devices. In the Figure 3.10, audio capture device 104 and video capture device 105 are shown. The possible capture devices which may be present in a streaming server may not be limited to audio and video. In this preferred embodiment, only audio and video are shown but it can be generalized to other capture

devices as well. Audio data captured by 104 is sent to the audio encoder 106, and video data captured by 105 is sent to video encoder 107 for encoding. Encoded media data such as audio and video are multiplexed by the multiplexer device 108 which also gets the server's clock value from the server's clock 109.

The multiplexed media stream output by 108 is sent over the network communication link 102 to a streaming media player 103.

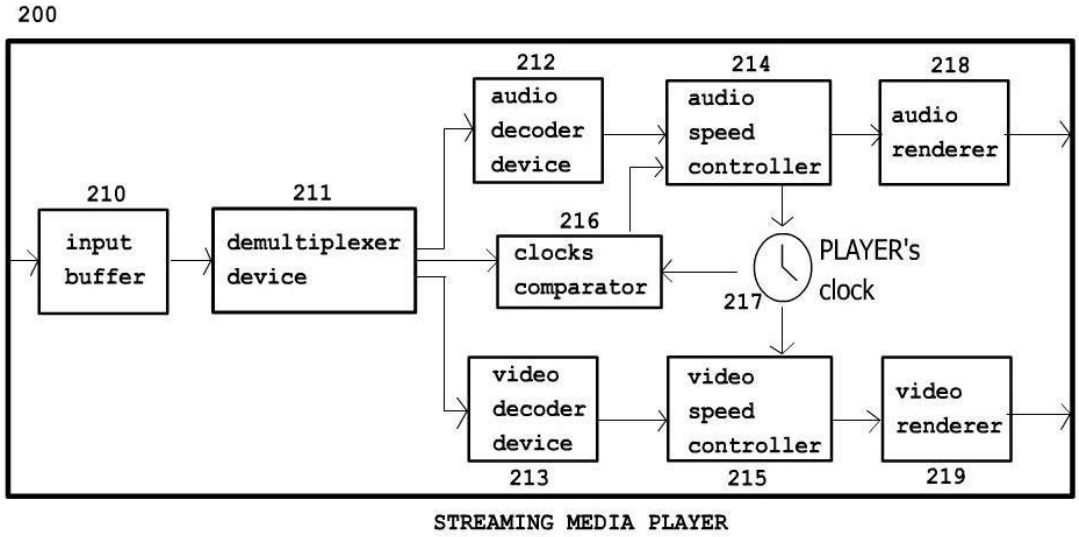


Figure 3.10. Proposed streaming media player

A standard media player buffers the media stream by the input buffer 110 and the media is passed to the demultiplexer device 111. Demultiplexer device 111, demultiplexes the different media streams and reads the server's clock value. Two of the outputs of the demultiplexer device 111, namely audio and video, are illustrated. The demultiplexer updates the player's clock 112 with the server's clock value. Encoded audio and video outputs of the demultiplexer device 111 are input into the audio decoder device 113 and the video decoder device 114. Decoded audio is sent to the audio renderer 115 and decoded video is sent to the video renderer 116.

Figure 3.10 describes the proposed method used in the streaming media player 103. 201 is the clocks comparator device which compares the server's clock value and the player's clock value and generates a clock state value. The amount of data transfered

from audio decoder device 113 to the audio renderer 115 is controlled by the audio speed controller 202. The amount of data transfered from video decoder device 114 to the audio renderer 116 is controlled by the audio speed controller 203.

The clock state value generated by 201 is input to the audio speed controller 202 and the audio speed controller 202 updates the player's clock 112. The video speed controller 203 gets the updated player's clock value as input.

The player's clock speed will be varied according to the clock state generated by the clocks comparator 201. Particularly, the player's clock speed may be varied according to the formula 3.14:

$$s = m\,(SC(t) - PC(t)) / (SC(t) - SC(t-1)) \qquad (3.14)$$

In the formula 3.14, s is the speed up or slow down ratio with respect to the original speed; m is the maximum allowable slow down ratio with respect to the original speed; PC(t) is the player's clock value at sampling time t; and SC(t) is the server's clock value at sampling time t.

CHAPTER 4

THE IMPLEMENTATION

The presented player is implemented on Linux by using C language and compiled with GCC-4.0 compiler. The presented player can be compiled on any Linux distribution installed with required libraries. The required libraries are as the following;

Ffmpeg is an open source codec library (or framework) which includes libavcodec, libavformat, libavutil and libavdevice as sublibraries. This library is used for decoding and demuxing purposes. It has numerous video and audio decoders, and as well as container parsers. Most of the supported codecs by ffmpeg are licensed with LGPL (Lesser General However Public License). However some codecs may require GPL (General Public License). In order to use these codecs, the ffmpeg library must be compiled with GPL licence option enabled. The main difference between GPL and LPGL is with a LPGL licence it is not required to publish the code that uses library unless it modifies the LPGL licenced code. However, the code that uses GPL licenced library must be published even if the GPL licenced library is not modified.

SDL (Simple Direct Media Layer) is also an open source library designed to access low level hardware such as audio, video, keyboard, mouse, etc. This library is used to render and print-to-screen decoded video pictures.

ALSA (Advanced Linux Sound Architecture) provides audio functionality of Linux operating system. This library is used to realize audio rendering and playing.

There are other libraries which can be used as well such as **OSS** for audio rendering, **V4L** for video rendering, **libmpeg2** for MPEG-2 video decoding, **libnemesi** for RTP and RTSP reading, **libcurl** for HTTP reading, **libmms** for MMS reading Modular structure of player makes it possible to use a library instead of other. So, for demuxing, video decoding, audio decoding, video rendering and audio rendering purposes it is possible to integrate any library to player with a little work. The required interfaces that will be used for integration will be given in the further sections.

In order to provide a graphical user interface it is possible to use **GTK-2.0** or **WxWidgets**. The presented player is suitable to use as a standalone application or as a C library after making slight modifications to the original code.

4.1. Implementation Modules

Implementation modules are implemented as separate C files designed to accomplish some peculiar tasks. The implementation of most important modules will be given in the further subsections. The complete list of modules is given in the Table 4.1.

Table 4.1. The implementation modules

Module Name	Module's Task
vesplayer.c	provides access functions of the player.
ui.c	provides a terminal shell as user interface
resource_reader.c	hides details of reading from various resources.
demuxer.c	hides details of parsing various containers
decoderv.c	used for decoding encoded video elementary streams
decodera.c	used for decoding encoded audio elementary streams
outv.h	interface to print decoded pictures to screen
outa.h	interface to play decoded audio samples
synchronizer.c	provides audio-video synchronization
reader_file.c	used by resource_reader to read media stream from file
buffer.c	buffer's common features implemented in this module
reader_http.c	used by resource_reader to read media stream from http
reader_mms.c	used by resource_reader to read media stream from mms
reader_rtsp.c	used by resource_reader to read media stream from rtsp
outv_sdl.c	accessed via outv to print decoded pictures with SDL library
outv_v4l.c	accessed via outv to print decoded pictures with V4L library
outa_alsa.c	accessed via outa to play decoded audio samples with ALSA
outa_oss.c	accessed via outa to play decoded audio samples with OSS
decv_avc.c	used by decoderv to decode encoded video streams by ffmpeg
deca_avc.c	used by decodera to decode encoded audio streams by ffmpeg
buf_admx.c	circular buffer between demuxer and audio decoder
buf_vdmx.c	circular buffer between demuxer and video decoder
buf_adec.c	circular buffer between audio decoder and audio renderer
buf_vdec.c	circular buffer between video decoder and video renderer

4.1.1. The UI Module

Table 4.2. The UI module: player's main function

```
01.  int main(int argc, char *argv[])
02.  {
03.      vbool  ret;
04.
05.      // handle the cmd line arguments
06.      ret = ui_handle_cmdline(argc, argv);
07.      if (argc > 1 && ret == vfalse)        return EXIT_FAILURE;
08.
09.      if (!s_mediafile) {
10.          printf("You must specify media filename via -f...\n");
11.          return EXIT_FAILURE;
12.      }
13.
14.      ret = Vesplayer_init();
15.      if (ret != vtrue)  {
16.          printf("Failed to initialize vesplayer!\n");
17.          return EXIT_FAILURE;
18.      }
19.
20.      s_stream.pb_completed = s_handle_playback_finished;   // callback of media-play completed
21.      s_stream.pbc_args     = NULL;                         // args
22.      s_stream.filename     = s_mediafile;                  // mediafile to play
23.      s_stream.subtfile     = s_subtfile;                   // optional subtitle file
24.      s_stream.pipename     = s_pipename;                   // optional pipename
25.      s_stream.layout.x     = s_x_loc;                      // xloc of screen
26.      s_stream.layout.y     = s_y_loc;                      // yloc of screen
27.      s_stream.layout.w     = s_width;                      // intended screen width
28.      s_stream.layout.h     = s_height;                     // intended screen height
29.
30.      ret = Vesplayer_start_playback(&s_stream);
31.      if (ret == vfalse)    {
32.          printf("Failed to play the media...\n");
33.          Vesplayer_terminate();
34.          return EXIT_FAILURE;
35.      }
36.      ui_init();
37.      Vesplayer_wait_playback();
38.      ui_finish();
39.      return EXIT_SUCCESS;    }
```

The ui module is a simple implementation of user interface that allows user to control player via a terminal or console. The main function of the application which is implemented in the UI module is given at Table 4.2. Firstly command line arguments are parsed, user specified options such as filename, screen position, etc. are initialized at line 6.

The player is initialized at line 14. Then the structure that holds stream's features is initialized at lines 20-28. The Stream structure holds all required information about media bitstream during playback. The definition of this structure is given at Table 4.3.

Table 4.3. The stream structure

```
typedef struct
{
   vbool        initialized;
   vbool        include_audio;
   vbool        include_video;
   vbool        include_subtitles;
   vuint16      demuxer_id;
   vuint16      video_PID;
   vuint16      audio_PID;
   vuint32      duration;
   vsint32      extension;
   vsint32      trick_mode;
   vsint64      last_video_pts;
   Audio_Stream  audio;
   Video_Stream  video;
   Layout       layout;
   char         *filename;     // media file
   char         *subtfile;     // optional subtitle file
   char         *pipename;     // optional pipename
   vplaycompleted  pb_completed; // playback completed callback
   void         *pbc_args;     // and its arguments
} Stream;
```

The Stream structure holds Audio_Stream and Video_Stream structures which hold features of the audio and video streams. The parameters defined in Audio Stream and Video Stream structures are initialized when they are detected by the related

modules. For instance, the size of the decoded picture is set by the video decoder module when the first picture is decoded. The definition of Audio_Stream and Video_Stream structures are given at Table 4.4.

Table 4.4. The Audio_Stream and Video_Stream structures

```
typedef struct
{
   vbool    fullscreen;   // layout
   vuint32  xpos;
   vuint32  ypos;
   vuint32  width;
   vuint32  height;
} Layout;

typedef struct
{
   vuint8   layer;
   vuint8   channel_cnt;
   vsint32  codec_id;
   vuint32  sampling_freq;
   vuint32  size;
   vuint32  duration;
   vuint32  bitrate;
   vuint32  microsecs_per_frame;
   float    timebase;
} Audio_Stream;

typedef struct
{
   double  height;
   double  width;
   vsint32  codec_id;
   vuint32  hor_size;
   vuint32  ver_size;
   vuint32  duration;
   vuint32  bitrate;
   vuint32  microsecs_per_frame;
   double   timebase;
   Frame_Aspect_Ratio aspect_ratio;
} Video_Stream;
```

In the main function of the player at Table 4.2, the playback is started in line 30. “Vesplayer_start_playback(&s_stream);” function creates a playback thread as seen in Table 4.6. User interface which allows user to input commands such as pause, resume, stop, etc. is initialized in line 37 at Table 4.2. Then Vesplayer_wait_playback(); function in line 38 causes main function to wait for the termination of playback thread. When the playback ends ui_finish() function is called and the application is finished.

4.1.2. The Vesplayer Module

This module realizes the control mechanism of the player and provides an interface to control player as seen in the Table 4.5.

Table 4.5. Vesplayer: playback control interface

```
vbool Vesplayer_init();
vbool Vesplayer_start_playback(Stream *a_stream);
void Vesplayer_stop_playback();
void Vesplayer_terminate();
void Vesplayer_slow_motion();
void Vesplayer_fast_motion();
void Vesplayer_fast_forward();
void Vesplayer_fast_backward();
void Vesplayer_normal_playback();
void Vesplayer_pause_playback();
void Vesplayer_seek(vsint32 a_seek_step);
void Vesplayer_seek_to_time(vsettotime seek_to_time);
vsint64 Vesplayer_get_current_playtime();
vbool Vesplayer_playback_completed();
vbool Vesplayer_wait_playback();
```

Vesplayer_start_playback() function creates a playback thread and this thread spawns other required threads in order to play stream such as demuxing thread, video thread, audio thread as seen in the following Table 4.6.

Table 4.6. The playback thread

```
Vbool Vesplayer_start_playback(Stream *a_stream)
{
   pthread_create(&s_thr_playback, NULL, s_playback, a_stream);
   return vtrue;
}

static void* s_playback(void *args)
{
   vbool succeed  = vfalse;
   s_stream = (Stream *) args;
   s_terminated = vfalse;

   // init the buffers
   succeed = Buf_Admx_init();   if (!succeed)        return vfalse;
   succeed = Buf_Adec_init();    if (!succeed)       return vfalse;
   succeed = Buf_Vdmx_init();    if (!succeed)        return vfalse;
   succeed = Buf_Vdec_init();    if (!succeed)       return vfalse;

   Demuxer_find_stream_info(s_stream);
   pthread_create(&s_thr_dmxer,      NULL, Demuxer_demux,    s_stream);

   if (s_stream->include_video){
     pthread_create(&s_thr_dec_vid,  NULL, DecoderV_decode, s_stream);
     pthread_create(&s_thr_playerv,  NULL, PlayerV_play,    s_stream);
   }
   if (s_stream->include_audio) {
      pthread_create(&s_thr_dec_aud, NULL, DecoderA_decode, s_stream);
      pthread_create(&s_thr_playera, NULL, PlayerA_play,    s_stream);
   }
   s_stream->initialized = vtrue;

   do{
      usleep(1000);
      succeed = Vesplayer_playback_completed();
      if (succeed == vtrue)          break;

   } while (s_terminated == vfalse);

   s_finish_threads();
   s_stream->initialized = vfalse;

   RReader_free();
   Buf_Input_free();
   Buf_Adec_free();
   Buf_Admx_free();
   Buf_Vdec_free();
   Buf_Vdmx_free();

   // vplaycompleted callback function is called when playback is completed: only if not null!
   if (s_stream->pb_completed)    {
      s_stream->pb_completed(s_stream->pbc_args);
   }
   return 0;
}
```

As seen in the Table 4.6 s_playback is thread function which firstly initializes circular buffers then starts demuxer, decoder and player (renderer) threads. After initializations are completed successfully, the completion of playback is checked in loop every 1 milisecond (1000 microseconds). Vesplayer_playback_completed() function checks the end of playback as seen in the Table 4.7. When the playback is completed, loop is breaked and then all threads are terminated and buffers are released.

Table 4.7. Checking end of playback

```
vbool Vesplayer_playback_completed()
{

   if (s_stream->include_video)
   {
      if (PlayerV_completed())
      {
         if(PlayerA_completed())
            printf("video and audio completed\n");
         else
            printf("video completed\n");

         return vtrue;
      }
   }

   if (s_stream->include_audio)
   {
      return PlayerA_completed();
   }

   // to be on the safe side
   return vfalse;

}
```

If the stream contains video, which is detected with the start of demuxing, then whether the video player (renderer in other words) thread terminated or ended is checked by PlayerV_completed() function as seen in the Table 4.7. Vesplayer_playback_completed function returns true which means playback is completed when audio or video renderer threads are completed. The renderer threads

can be completed because of two reasons; first one is termination of playback by user's stop playback command and the other is reaching to end of stream.

4.1.3. The Buffer Module

The buffer module creates the circular buffers which are used to transfer data from one thread to another. The buffer module has a generic implementation, so all circular buffers use this same implementation with the different parameters. The circular buffer is initialized as seen in the Table 4.8.

Table 4.8. Initialization of circular buffer

```
Vbool Buf_init(VBuffer *buf, vuint32 a_frame_count, vuint32 a_frame_size)
{
   // init flags
   buf->read_terminated  = vfalse;
   buf->write_terminated = vfalse;
   buf->is_protected     = vtrue;
   buf->frame_amount     = a_frame_count;
   buf->frame_size       = a_frame_size;
   buf->look_at_pos      = 0;
   buf->frames           = 0;

   // init the buffer controller
   buf->ctrl.rd_pos   = 0;
   buf->ctrl.wr_pos   = 0;
   buf->ctrl.item_cnt = 0;

   // init the buffer controller semaphores
   if (sem_init(&buf->ctrl.full, 0, 0) < 0)       return vfalse;

   // initially all are empty, thus count is buf->frame_amount
   if (sem_init(&buf->ctrl.empty, 0, buf->frame_amount) < 0)
   {
      return vfalse;
   }

   // allocate buffers if wanted
   if (buf->frame_size > 0)
   {
      buf->frames = (char *) malloc(buf->frame_size * buf->frame_amount);

      if (buf->frames == 0)      {
         VERR(("can't get memory for buf->frames\n"));
         return vfalse;
      }
   }

   return vtrue;
}
```

Buf_Init function takes three parameters as input; buf parameter is the handler for the buffer, frame count shows the item number that the buffer holds and frame size shows the size of each item.

Each buffer basicly has two ports: one for reading and the other for writing. Because each circular buffer conveys data between two threads. In all cases one of these two threads is producer and the other is consumer.

For instance if this circular buffer is between video decoder and video renderer threads, video decoder thread works as producer and puts decoded video pictures to circular buffer, and video renderer module works as consumer and gets the decoded picture from circular buffer to print on screen.

So, producer thread always writes to circular buffer and consumer thread always read from it. Each circular buffer has two flags that shows the status of its producer and consumer threads. The write_terminated flag of the buffer show if the producer thread that write the data is still active or not. The read_terminated flag shows the status of the reader thread.

These flags are very important in providing synchronization between threads. Because, for example if the write_terminated flag is true, that means producer is ended because of end of stream or an error and will not write to buffer anymore. So when the all of items in the circular buffer is consumed by producer, it will check whether the procuder thread is active by using write_terminated flag. If the writer thread is terminated, then there is no need for the produced thread to wait anymore for the new data and it can terminate itself.

The threaded modules: demuxer thread, audio decoder thread, video decoder thread, audio player thread and video player thread always read from one buffer and write to another one. So, there is no direct communication between a thread and the other. This provides abstraction between threads and it is possible to change a writer (or reader) thread with another if it produces same kind of data.

For instance, MPEG-2 video decoder thread is a writer thread for the decoded pictures circular buffer and it can be changed with MPEG-4 video decoder thread without need of any change. Because both decoders write same kind of data (decoded pictures) to the decoded pictures circular buffer. And so for the video player thread which reads from this circular buffer, the identity of the writer thread (video decoder) is unimportant. The read and write functions of the buffer is given in Table 4.9.

Table 4.9. Read and Write functions of the circular buffers

```
Vbool Buf_read(VBuffer *buf, char *a_copy_to_mem, vuint32 a_size)
{

   // return false if the read buffer is terminated or (write buffer is terminated and it is empty)
   if (buf->read_terminated || (buf->write_terminated && buf->ctrl.item_cnt == 0))
      return vfalse;

   if (buf->is_protected)
      sem_wait(&buf->ctrl.full);

   // copy the data
   memcpy(a_copy_to_mem, buf->frames + (buf->ctrl.rd_pos * buf->frame_size), a_size);
   buf->len = buf->length[buf->ctrl.rd_pos];

   // update the buffer controller
   buf->ctrl.rd_pos++;
   buf->ctrl.rd_pos %= buf->frame_amount;
   buf->ctrl.item_cnt--;

   if (buf->is_protected)
      sem_post(&buf->ctrl.empty);

   return vtrue;

}

Vbool Buf_write(VBuffer *buf, char *a_write_this, vuint32 a_size)
{

   // if the read buffer is terminated, there is not need to write
   // so check both the read and write buffers if they are terminated
   if (buf->write_terminated || buf->read_terminated)
      return vfalse;

   if (buf->is_protected)
      sem_wait(&buf->ctrl.empty);

   // copy the data
   memcpy(buf->frames + (buf->ctrl.wr_pos * buf->frame_size), a_write_this, a_size);
   buf->length[buf->ctrl.wr_pos] = a_size;

   // update the buffer controller
   buf->ctrl.wr_pos++;
   buf->ctrl.wr_pos %= buf->frame_amount;
   buf->ctrl.item_cnt++;

   if (buf->is_protected)
         sem_post(&buf->ctrl.full);

   return vtrue;
}
```

4.1.4. The Resource Reader Module

The media stream can be stored on a local file system or it can be broadcasted (or unicasted) over a private network or over Internet via protocols such as udp, rtp, rtsp, http, mms, etc.

The resource reader module provides an abstraction between stream resource and the rest of the player's modules. For example, it reads from local file system by using local file system reader module and puts read stream to input circular buffer. If the stream is broadcasted over http, it reads stream by using http reader module, and puts read data to the same input circular buffer. Demuxer module is the reader of the input circular buffer. So, there is no need for demuxer module to know where the stream comes from. The resource reader module is initialized as seen in the Table 4.11.

The resource of the stream can be detected by parsing url, or it can be enforced by using command line arguments. Function pointers are used to bind functions of resource reader to the functions of the specific reader module such as rtsp reader module, http reader module, etc. as seen in the Table 4.11.

The resource reader module is a threaded module, in other words it has its own thread and reading continues in this thread function in a loop until the end of stream is reached or playback is terminated as seen in the Table 4.10.

Table 4.10. The resource reader module's thread function

```
void * RReader_loop(void *a_params)
{
   vbool    succeed = vfalse;
   vuint32  len     = 0;
  do {
      len = Reader_read( (char *)s_pack.data, INPUT_SIZE);
       if (len > 0) {
            s_pack.size = len;
            Buf_Input_write(&s_pack);
        }
   } while (!s_terminated);
   return 0;
}
```

Table 4.11. The initialization of the resource reader module

```
vbool RReader_init(const char *a_name)
{
   if(strstr(a_name, "http://") != 0)    {
      Reader_init         = &ReaderH_init;
      Reader_free         = &ReaderH_free;
      Reader_read         = &ReaderH_read;
      Reader_rewind       = &ReaderH_rewind;
      Reader_get_filesize = &ReaderH_get_filesize;
      Reader_seek         = &ReaderH_seek;
   }
   else if(strstr(a_name, "mms://") != 0)
   {
      Reader_init         = &ReaderM_init;
      Reader_free         = &ReaderM_free;
      Reader_read         = &ReaderM_read;
      Reader_rewind       = &ReaderM_rewind;
      Reader_get_filesize = &ReaderM_get_filesize;
      Reader_seek         = &ReaderM_seek;
   }
   else if(strstr(a_name, "rtsp://") != 0)
   {
        Reader_init         = &ReaderR_init;
      Reader_free         = &ReaderR_free;
      Reader_read         = &ReaderR_read;
      Reader_rewind       = &ReaderR_rewind;
      Reader_get_filesize = &ReaderR_get_filesize;
      Reader_seek         = &ReaderR_seek;
   }
   else   {
       Reader_init         = &ReaderF_init;
      Reader_free         = &ReaderF_free;
      Reader_read         = &ReaderF_read;
      Reader_rewind       = &ReaderF_rewind;
      Reader_get_filesize = &ReaderF_get_filesize;
      Reader_seek         = &ReaderF_seek;
   }
   return Reader_init(a_name);
}
```

4.1.5. The Demuxer Module

The demuxer module is used to parse media stream container and extract elementary audio and video streams. The extracted audio and video streams are written to encoded audio and encoded video circular buffers. There are numerous container types such avi, asf, MPEG program stream, MPEG transport stream, etc. So firstly container type of the stream must be detected by searching specific patterns that defines the container type. For example, each MPEG transport stream pack is 188 bytes in length and start with the hex code 0x47. So, if this header is detected with 188 bytes intervals, that means container type is MPEG transport stream.

When the container type is detected, the function pointers of the demuxer module are binded to the functions of the demuxer of detected container type as in Resource Reader Module.

The Demuxer Module has its own thread and demuxing continues in a loop until the end of stream is reached as seen in the Table 4.12.

Table 4.12. The thread function of the demuxer module

```
void* Demuxer_demux( )
{
   vsint32  ret = 0;
   vbool succeed = vfalse;
   Packet   *pkt;
  do{
      ret = read_demuxed_packet(pkt);
      if (ret < 0)         break;
      s_pack.pts  = pkt->pts;
      s_pack.size = pkt->size;
      memcpy(s_pack.data,  pkt->data, pkt->size);
      if (pkt->type == AUDIO_ELEMENTARY_STREAM)
            Buf_Admx_write(&s_pack);
      else  if (pkt->type == VIDEO_ELEMENTARY_STREAM)
           Buf_Vdmx_write(&s_pack);
   }   while( !end_of_buffer && !s_terminated );
}
```

4.1.6. The Video Decoder Module

The video decoder module is responsible for providing an interface to the numerous video codecs. For instance, the video decoder library such as MPEG-2 decoder of the ffmpeg library is binded to this module when the codec type is detected as MPEG-2. It is assumed that the video decoder library operates on a frame-by-frame basis, which is the most common case, since this method simplifies the required interaction between player and the video decoder library and improves the usability of the library. The video decoder module must be initialized before decoding. So, firstly the type of the video codec must be found. This information can be extracted during parsing of the stream's container type or it can be detected from the headers of the video elementary stream.

The video decoder module has its own thread, and the decoding continues in a loop until the end of the stream is reached or playback terminated as seen in the Table 4.13. Firstly, demuxed video data is read from the demuxed video circular buffer with the function "Buf_Vdmx_read(&s_demuxed)" as seen in the line 14. The "s_demuxed" is a structure that is composed of an one dimensionel unsigned char array which holds the encoded video data just read from the circular buffer and an unsigned int variable that shows the size of the array. The "demuxed_video_start" is an unsigned char pointer that shows the start of the demuxed video data chunk as seen in the line 17. The size of the demuxed video chunk is assigned to a temp variable "demuxed_video_size" as seen in the line 18. Then in a loop actual video decoding is implemented as seen in the lines 18-28. This loop continues until all of the demuxed video data is consumed by the decoder. Each demuxed video chunk does not hold a fixed number of encoded video frame. For instance, when a demuxed video chunk is processed by the decoder, one or more video frame can be decoded from that chunk.

The actual video decoding is realized by the function Dec_Video_decode as seen in the line 24. It is a function pointer which is binded to the actual video decoder's decode function when the codec type is detected. The Dec_Video_decode takes the start address of the demuxed video data and its size. Then process the demuxed data until a video frame is totally decoded or all of the demuxed data chunk is consumed. If a video frame is decoded, the "picture_out" is set as 1 by the actual video decoder and the consumed size is returned.

Table 4.13. The thread function of the video decoder module

```
01.  void* DecoderV_decode(void *a_params)
02. {
03.    vbool       succeed  = vfalse;
04.    vuint8      *demuxed_video_start = 0;
05.    vuint32      demuxed_video_size  = 0;
06.    vuint32      size  = 0;
07.    vsint32      picture_out  = 0;
08.    Video_Frame  s_decoded;
09.
10.    do
11.    {
12.       // read data from the demuxed video buffer
13.       succeed = Buf_Vdmx_read(&s_demuxed);
14.       if (!succeed) break;
15.
16.       size               = 0;
17.       demuxed_video_start = s_demuxed.data;
18.       demuxed_video_size  = s_demuxed.size;
19.
20.       // decode the demuxed video data: demuxed_video_start and demuxed_video_size will be updated
21.        while (demuxed_video_size > 0)
22.        {
23.          // actual decoding
24.          size   =   Dec_Video_decode(&s_video_context,   &s_decoded,   &picture_out,   demuxed_video_start,
demuxed_video_size);
25.
26.             succeed = Buf_Vdec_write(&s_decoded);
27.             if (!succeed)   break;
28.          }
29.
30.         // update the demuxed video start and size
31.          demuxed_video_start += size;
32.          demuxed_video_size  -= size;
33.
34.       } // while
35.
36.    } while (!s_terminated);
37.
38.    s_free();
39.    return 0;
40. }
```

4.1.7. The Audio Decoder Module

The audio decoder module is very similar to video decoder module. It is also responsible for providing an interface to numerous codecs (in this case audio codecs). For instance, the audio decoder library such as MPEG audio decoder of the ffmpeg library is binded to this module when the codec type is detected as MPEG audio. It is assumed that the audio decoder library operates on a frame-by-frame basis, which is the most common case, since this method simplifies the required interaction between player and the audio decoder library and improves the usability of the library. The audio decoder module must be initialized before decoding. So, firstly the type of the audio codec must be found. This information can be extracted during parsing of the stream's container type or it can be detected from the headers of the audio elementary stream.

The audio decoder module has its own thread, and the decoding continues in a loop until the end of the stream is reached or playback terminated as seen in the Table 4.14. Firstly, demuxed video data is read from the demuxed video circular buffer with the function "Buf_Admx_read(&s_demuxed)" as seen in the line 12. The "s_demuxed" is a structure that is composed of an one dimensionel unsigned char array which holds the encoded video data just read from the circular buffer and an unsigned int variable that shows the size of the array. The "demuxed_audio_start" is an unsigned char pointer that shows the start of the demuxed audio data chunk as seen in the line 17. The size of the demuxed audio chunk is assigned to a temp variable "demuxed_audio_size" as seen in the line 18. Then in a loop actual audio decoding is implemented as seen in the lines 21-37. This loop continues until all of the demuxed audio data is consumed by the decoder. Each demuxed audio chunk does not hold a fixed number of encoded audio frame. For instance, when a demuxed audio chunk is processed by the decoder, one or more audio frame can be decoded from that chunk.

The actual audio decoding is realized by the function Dec_Audio_decode as seen in the line 24. It is a function pointer which is binded to the actual audio decoder's decode function when the codec type is detected. The Dec_Audio_decode takes the start address of the demuxed audio data and its size. Then process the demuxed data until an audio frame is totally decoded or all of the demuxed data chunk is consumed. If a audio frame is decoded, the "decoded_audio size" is set to the decoded audio frame's size by the actual audio decoder and the consumed size is returned.

Table 4.14. The thread function of the audio decoder module

```
01. void* DecoderA_decode()
02. {
03.     vbool    succeed  = vfalse;
04.     vsint32  decoded_audio_size ;
05.     vsint32  size   = 0;
06.     vuint32  demuxed_audio_size = 0;
07.      vuint8  *demuxed_audio_start      = 0;
08.
09.     do
10.     {
11.       // read data from the demuxed audio buffer
12.       succeed = Buf_Admx_read(&s_demuxed);
13.       if (!succeed)      break;
14.
15.       // init
16.       size               = 0;
17.       demuxed_audio_start = s_demuxed.data;
18.       demuxed_audio_size  = s_demuxed.size;
19.
20.       // decode the demuxed audio data: demuxed_audio_start and demuxed_audio_size will be updated
21.       while (demuxed_audio_size > 0)
22.       {
23.          // actual decoding
24.         size = Dec_Audio_decode(&s_audio_context, &s_decoded, &decoded_audio_size, demuxed_audio_start,
demuxed_audio_size);
25.          if (size < 0)    break;
26.          if (decoded_audio_size > 0)  {
27.             s_decoded.size = decoded_audio_size;
28.             // write data to the decoded audio buffer
29.             succeed = Buf_Adec_write(&s_decoded);
30.             if (!succeed)   break;
31.          }
32.
33.          // update the demuxed audio start and size
34.          demuxed_audio_start += size;
35.          demuxed_audio_size  -= size;
36.       }
37.    } while (!s_terminated);
38.
39.    s_free();
40.    return 0;
41. }
```

4.1.8. The Video Renderer Module

The video renderer module is used to print decoded frames to the screen via a graphic library such as SDL, directfb, etc. It is also possible to integrate any graphic library to video renderer module. Because video renderer module defines a common interface to use video graphic libraries. So, writing a wrapper code to the graphic library which is suitable with the video renderer module's defined interface is enough.

Firstly, video renderer module must initialize the video output window. This is accomplished by the "Video_init" pointer function which is binded to the actual graphic library's init function. This init function takes width, height, x and y positions of the video output window as seen at the line 9 in the Table 4.15. The width and height of the output window is set to the decoded video frame's width and height unless user enforces some other values. In the case of using enforced parameters, the video picture must be scaled automatically by the graphic library to fit the picture to the output window. The x and y positions show the coordinates of the upper left corner of the output window. If there is no user defined parameters, the video output window is positioned to the center of the screen.

The video renderer module is also a threaded module and it has a loop which prints decoded pictures to the screen according to the presentation time stamps defined by the encoder. The video rendering loop is as seen at the lines 15-38 in the Table 4.15 continues until the playback is terminated or there is no more decoded video pictures in the circular buffer. Video rendering thread sleeps 1 milisecond at the start of each iteration as seen in the line 17. Then if the boolean variable "s_get_new_frame" is true, a decoded video frame is read from the circular buffer and s_get_new_frame is set as false as seen in the lines 19-24. This boolean variable remains as false until the read frame is printed to the screen. At each iteration, current time and the frame show time is read from the synchronizer module as seen in the lines 26-27. If the current time passes show time of the lastly read decoded video frame (line 29) then it is printed to the screen with the function Video_show_picture() as seen in the line 36. And s_get_new_frame is set as true to read new frame from circular buffer at next iteration. The Video_show_picture function takes the "s_frame" structure as input which holds the decoded YUV picture (as unsigned char buffers for Y, U and V components) and its format(such as YUV420, YUV422 etc).

Table 4.15. The thread function of the video renderer module

```
01. void* PlayerV_play(void *a_params)
02. {
03.     vbool    succeed = vfalse;
04.     vsint64  current_time;
05.     vsint64  frame_show_time;
06.     layout = (Layout *) a_params;
07.
08.     // init the video library
09.     if ( !Video_init(layout->width, layout->height, layout->xpos,  layout->ypos)  )
10.         return 0;
11.
12.     s_curr_playtime = 0;
13.     s_get_new_frame = vtrue;
14.     // start main loop
15.     do
16.     {
17.           usleep(1000);
18.
19.          if (s_get_new_frame) {
20.              // get decoded video data
21.              succeed = Buf_Vdec_read(&s_frame);
22.              if (!succeed)     s_terminated = vtrue;
23.              s_get_new_frame = vfalse;
24.          }
25.
26.          current_time    = Synchronizer_read_system_clock();
27.          frame_show_time = Synchronizer_read_player_clock(s_frame.pts);
28.
29.          if (current_time > frame_show_time)
30.          {
31.                Video_show_picture(&s_frame);
32.                s_curr_playtime = s_frame.pts - s_initial_pts;
33.                if (!s_initial_pts) {
34.                    s_initial_pts = s_frame.pts;
35.                }
36.                s_get_new_frame = vtrue;
37.           }
38.     } while (!s_terminated);
39.
40.     s_free();
41.     return 0;
42. }
```

4.1.9. The Audio Renderer Module

The audio renderer module is responsible for playing decoded audio frames by using libraries such as OSS, ALSA, etc. to access audio drivers.

Firstly audio driver must be initialized according to the audio stream's features such as channel_cnt, sampling_frequency, etc. as seen in the Table 4.4. This initialization is realized by the function "Audio_init" as seen in Table 4.16.

The Audio renderer module is also a threaded module and it has a loop which reads decoded audio frame from circular buffer and writes it to the audio driver's buffer in blocking mode. The blocking mode does not allow writing new data to the audio driver's buffer until all of the current data in the buffer is consumed. Before audio frame is written to the audio driver's buffer, its presentation time stamp is used to set the player's master clock by the function "Synchronizer_update_master_clock" as seen in the Table 4.16.

Table 4.16. The thread function of the audio renderer module

```
void* PlayerA_play(void *a_params)
{
   vbool succeed     = vfalse;
   audio_params = (AudioStream *) a_params;

   // init the audio library
   if (!Audio_init(&audio_params))
            return 0;
   do
   {
         succeed = Buf_Adec_read(&s_frame);
         if (!succeed)        break;
         Synchronizer_update_master_clock(s_frame.pts);
         Audio_write(s_frame.data, s_frame.size);
   } while (!s_terminated);

   s_free();
}
```

4.1.10. The Audio-Video Synchronizer Module

This module is responsible for providing audio-video synchronization according to the proposed method explained in detail at chapter 3 and section 3.3.1. The proposed method uses audio presentation time stamps to update player's master clock. The audio renderer module calls the function Synchronizer_update_master_clock to update the master clock as seen in the Table 4.17.

The other elementary streams such as video, subtitle, etc. uses the updated master clock to compare against their presentation time stamps. This is accomplished by the function Synchronizer_read_player_clock as seen in the Table 4.17.

Table 4.17. The audio-video synchronizer module

```
void Synchronizer_update_master_clock(vsint64 a_pts)
{
   vsint64         current_time;
   struct  timeval  system_clock;
   gettimeofday(&system_clock, NULL);
   if (s_initial_pts_set){    // should set the initial or current time?
      current_time  = system_clock.tv_sec * 1000000 + system_clock.tv_usec;
      s_alpha       = (float)(current_time - s_initial_time) / (float)(a_pts - s_initial_pts);       // set the current time
      s_current_time = current_time;
      s_current_pts  = a_pts;
   } else{
      s_initial_time    = system_clock.tv_sec * 1000000 + system_clock.tv_usec; // set the initial time
      s_initial_pts     = a_pts;
      s_initial_pts_set = vtrue;       // update the flag
      s_current_time = s_initial_time;       // set the current time
      s_current_pts  = s_initial_pts;
      s_alpha       = 1;
   }
}
vsint64 Synchronizer_read_player_clock(vsint64 a_pts)
{
   vsint64 master_time;
   master_time = s_current_time + s_alpha * (float)(a_pts - s_current_pts);
   return master_time;
}
```

CHAPTER 5

CONCLUSION

Streaming media players have great importance for the most of digital entertainment systems such as DVB, DVD players, various handheld devices or even for PCs. So, quality of a streaming media player directly affects Quality of Service or Quality of User Experience. Robustness, support for the most of stream formats, smooth playback, scalability, economic usage of system resources and customizable user interface are important features that define the quality of a streaming media player.

The proposed streaming media player in this thesis has a modular design which makes it more scalable and maintainable. It is scalable, because new features can be supported by adding new modules. And it is maintainable because abstraction that comes with modularity makes it easier to find and fix any bugs or defects.

Smooth playback requires precise audio and video synchronization. The proposed design employs a precise audio and video synchronization scheme. This scheme utilizes audio presentation time stamps to update player's master clock, resulting in a smoother and inter-synchronized playback.

Server client synchronization is also important to provide a smooth playback. The proposed player changes its playback rate by playing the audio stream with a proper sampling rate. Because, the sampling rate shows how many audio samples must be played in a second and hence defines the playback rate of audio stream whose timestamps are used to update the master clock.

Rapidly inspection of stream type is also important, because it enables quicker start of selected digital content so user will not wait too much after selected a stream to open. Conventional media players comprises a dedicated stream inspector module and when a stream is selected to play, firstly this module work once and gathers required information to play the stream. The proposed media player does not include a separate stream inspector module. Instead the type of stream is inspected as the stream goes through the modules which makes stream inspection faster than using a separate module.

REFERENCES

Abdel-Baki, N., E. Perez-Soler, B. Aumann, and H.P. Grossmann. 2003. A simplified design and implementation of a multimedia streaming system. *Telecommunications, ICT 2003. 10th International Conference* 2:1470 – 1474.

Basith, S. 1996. MPEG : standarts, technology and applications. http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol2/sab/article2.html (accessed June 24, 2008).

Blakowski, G., and R. Steinmets. 1996. A media synchronization survey: reference model, specifications, and case studies. *IEEE Journal on Selected Areas in Communications* 14:5-35.

Chan, M., Y. Yu, and A. Constantinides. 1990. Variable size block matching motion compensation with applications to video coding. *IEE Proc. On Communication, Speech and Vision*. 70-89.

Compression. 2002. VirtualDub MSU Motion Compensation Filter http://compression.ru/video/motion_estimation/index_en.html (accessed July 26, 2008)

Conklin, G.J., G.S. Greenbaum, K.O. Lillevold, A.F. Lippman, and Y.A. Reznik. 2001. Video Coding for Streaming Media Delivery on the Internet. *IEEE Trans. on Circuits and Systems for Video Technology*. 11:1.

Dapeng W., Y.T. Hou, W. Zhu, Y.Q. Zhang, and J.M. Peha. 2001. Streaming video over the Internet: approaches and directions. *Circuits and Systems for Video Technology, IEEE Transactions on Publication* 11:282-300.

Gall, D.L. 1991. MPEG: a video compression standart for multimedia applications. *Trans ACM*.

Garner, G.M., F.F. Feng, E.H.S. Ryu, and K. den Hollander. 2006. Timing and synchronization for audio/video applications in a converged residential ethernet network. *Consumer Communications and Networking Conference*. 2:883 – 887

Georganas, N.D. 1996. Synchronization issues in multimedia presentational and conversational applications. in *Proc. Pacific Workshop on Distributed Multimedia Systems* (DMS'96).

Guo, Q., Q. Zhang, W. Zhu, and Y.Q. Zhang. 2001. Sender-adaptive and receiver-driven video multicasting. *IEEE International Symposium on Circuits and Systems,* Sydney, Australia.

Haskell, B.G., A. Puri, and A.N. Netravali. 1997. Digital video: an introduction to MPEG-2. *Chapman&Hall, ISBN 0-412-08411-2.*

Hemy, M., U. Hengartner, P. Steenkiste, T. Gross. 1999. MPEG system streams in best-effort networks. *In Proc. IEEE Packet Video'99.*

Jack, K. 2005. Video demystified: a handbook for the digital engineer. *Newnes, NewYork.* 662-680.

Jamkar, S., S. Belhe, S. Dravid, and M.S. Sutaone. 2002. A comparison of block matching search algorithms in motion estimation. *Proceedings of the 15th International Conference on Computer Communication.* 730-739.

Jarmasz, J.P. and N.D. Georganas. 1997. Designing a distributed multimedia synchronization scheduler. *Proc. IEEE International Conference on Multimedia Computing and Systems* 451-457.

ISO, International Organization for Standarts. 1994. Generic coding of moving pictures and associated audio: systems. *Draft of:1540 ISO/IEC 13818-1*.

Kleidermacher, D.N. 2004. Linux for embedded systems? http://www.cotsjournalonline.com/home/article.php?id=100129 (accessed July 10, 2008).

Lehrbaum, R. 2001. What is so good about open source and Linux in embedded? http://linuxdevices.com/articles/AT8151978006.html (accessed June 14, 2008).

Liu, C. 1999. Multimedia over IP: RSVP, RTP, RTCP, RTSP. *Handbook of Communication Technologies: The Next Decade* 29-46.

Liu, C. 2001. Streaming video profile in MPEG-4. *IEEE Trans. on Circuits and Systems for Video Technology* 11:1.

Linux Devices. 2007. Snapshots of embedded Linux market. http://linuxdevices.com/articles/AT7065740528.html (accessed June 12, 2008).

Mikaeli, N. and W. Ying. 2004. QoS of digital video in mobile environment. http://mikaeli.mikkeliamk.fi/mikaeli/info/tutkimukset/wang/ (accessed June 4, 2008).

Netravali, A.N. and B.G. Haskell. 1988. Digital pictures, representation and compression. *Plenum Press*.

Olshausen, B. A. 2000. Aliasing. *PSC 129 – Sensory Processes* 3 – 4.

Puri, A. 1993. Video coding using the MPEG-2 compression standart. *Proc SPIE Visual Communications and Image Proc* '93.

Sethuraman, S., S. Parameswaran, D. Tamia, A. Kulkarni, and M. Singhal. 2005. Multi-format media player/recorder software design methodology for programmable processors with hardware accelerators. *Consumer Electronics. ICCE. Digest of Technical Papers. International Conference on Publication.*137-138.

Shibata, Y., N. Seta, and S. Shimizu. 1995. Media synchronization protocols for packet audio-video system on multimedia information networks. *System Sciences Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference* 2: 594-601

Smith, S.W. 1997. The scientist and engineer's guide to digital signal processing. *California Technical Publishing* 373-377.

Stiller, C. and J. Konrad. 1999. Estimating motion in image sequences. *IEEE Singal Processing Magazine*. 70-89.

Suárez, F.J., J.C. Granda, J. Molleda, and D.F. García. 2005. Linux based embedded node for capturing, compression and streaming of digital audio and video. *IEC (Prague)* pp. 403-408

Tryfonas, C., and A. Varma. 1999. Timestamping schemes for MPEG-2 systems layer and their Effect on receiver clock recovery. *Multimedia, IEEE Transactions* 1 (3) : 251 – 263

Turaga, D. and M. Alkanhal. 1998. Search algorithms for block-matching in motion estimation. http://www.ece.cmu.edu/~ee899 (accessed July 12, 2008).

Viscito, E. and C. Gonzales. 1991. A video compression algorithm with adaptive bit allocation and quantization. *Proc SPIE Visual Communications and Image* .

Zimmermann, R. 2003. Streaming of DivX AVI movies. *Proceedings of the 2003 ACM symposium on Applied computing*. Session: Multimedia and Visualization, 979 -982

APPENDIX A

RELATED PATENTS

The previous methods used to provide server-client and audio-video synchronization are given in the following US patents:

A.1. US 5,583,652

The title of this patent is “Synchronized, variable-speed playback of digitally recorded audio and video.”

The abstract of it is: “Method and system for providing user-controlled, continuous, synchronized variable-speed playback of a previously recorded digital audio/video presentation. The user directly controls the rate of playback and the audio and video remains synchronized. The audio is expanded or compressed using the time domain harmonic scaling method so that the pitch of the audio remains undistorted. Synchronization is maintained by allowing one clock to serve as the master time clock for the system. The clocks which can serve as the master time clock include the audio decoder clock, video decoder clock and the system clock. The invention is particularly useful in multimedia display systems designed to display MPEG data.”

A.2. US 5664,044

The title of this patent is “Synchronized, variable-speed playback of digitally recorded audio and video (continuation of US 5,583,652)”

The abstract is “Method and system for providing user-controlled, continuous, synchronized variable-speed playback of a previously recorded digital audio/video presentation. The user directly controls the rate of playback and the audio and video remains synchronized. The audio is expanded or compressed using the time domain harmonic scaling method so that the pitch of the audio remains undistorted. Synchronization is maintained by allowing one clock to serve as the master time clock

for the system. The clocks which can serve as the master time clock include the audio decoder clock, video decoder clock and the system clock. The invention is particularly useful in multimedia display systems designed to display MPEG data."

A.3. US 6,665,751

The title of this patent is "Streaming media player varying a play speed from an original to a maximum allowable slowdown proportionally in accordance with a buffer state."

The abstract of it is: "A media player for playing streaming media is capable of dynamically changing its play rate according to the network conditions, so as to compensate for delay packets. The player nominally plays at the prescribed data rate. When packets are delayed, instead of stopping to rebuffer, the player plays the stream slower. Audio pitch is unchanged as it is slowed and video frame rate is slowed as necessary. A threshold is set so that slowing down beyond the threshold is not allowed. Should the buffer contents fall below a prescribed minimum, the player will then stop and rebuffer."