

UNDERSTANDING FFMPEG WITH SOURCE CODE

FFMPEG FUNDAMENTALS

FFMPEG Data Structure

Various Image and Video Filters

Video basics

Audio Basics

Interlaced Video

I-B-P Frame and Timebase

Codecs

Encoding

Decoding

Multiplexing

Demultiplexing

Remultiplexing

Transcoding

Streaming



BY AKIN YAPRAKGÜL

UNDERSTANDING FFMPEG WITH SOURCE CODE

Copyright © 2020 Akin Yaprakgül

All rights reserved

ACKNOWLEDGEMENTS

I thank all who in one way or another contributed in the completion of this book. First, I give thanks to God for protection and ability to do work

The Greatest thanks belongs to the developers of FFmpeg libraries. The project documentation and Chinese articles from Communication University of China were the main source for the book. Another great source was the Wikipedia.

I give deep thanks to Easy Media Suite family and my boss Osman Yel. My special and heartily thanks to my wife Betül Yaprakgül who encouraged and directed me.

Thank you to my son Görkem Yaprakgül who always cheered up his father.

Thank you very much and best wishes.

TABLE OF CONTENTS

[Introduction](#)

[1. FFmpeg Fundamentals](#)

[FFmpeg Data Structure](#)

[AVCodecContext](#)

[AVStream](#)

[AVFormatContext](#)

[AVPacket](#)

[AVIOContext](#)

[URLContext](#)

[URLProtocol](#)

[Time information](#)

[Time information acquisition](#)

[APIs](#)

[Reading series](#)

[Codec series](#)

[Write series](#)

[Other](#)

[Several libavutil tools](#)

[AVOption \(AVClass\)](#)

[AVDictionary](#)

[ParseUtil](#)

[2. Video basics](#)

[Video](#)

[Raw Video](#)

[FOURCC](#)

[FOURCC Identifier](#)

[YUV Pixel Formats](#)

[Packed YUV Formats](#)

[Planar YUV Formats](#)

[RGB Pixel Formats](#)

[Bayer Data Formats](#)

[Generate Dummy Video](#)

[Convert Packet Format to Planar Format](#)

[Generate Various Pictures](#)

[Convert Multiple Image Format and Resolution](#)

[Separate Y, U, V components in YUV420P pixel data](#)

[Separate Y, U, V components in YUV444P pixel data](#)

[Remove the Color of YUV420P Pixel Data](#)

[Reduce the Brightness of YUV420P Pixel Data by](#)

[Half](#)

[Add a Frame Around the YUV420P Pixel Data](#)

[Generate Gray Scale Test Chart in YUV420P Format](#)

[Calculate the PSNR of two YUV420P Pixel Data](#)

[Separate R, G, B Components in RGB24 Pixel Data](#)

[Encapsulate RGB24 Format Pixel Data into BMP](#)

[Image](#)

[Convert RGB24 Format Pixel Data to YUV420P Format Pixel Data](#)

[Generate Color Bar Test Chart in RGB24 Format](#)

[Image Scale](#)

[Filter Chain](#)

[AVFilter](#)

[AVFilterPad](#)

[Picture Buffers](#)

[Reference Counting](#)

[Permissions](#)

[Filter Links](#)

[Usage of Avfilter](#)

[Color Change Rectangular Area](#)

[Watermark](#)

[Color Effects](#)

[Crop Video](#)

[Draw Box on Video](#)

[Remove Logo From Video](#)

[Draw Grid on Video](#)

[Fade Effect on Video](#)

[Horizontal Flip Video](#)

[Add Noise on Video](#)

[Rotate Video](#)

[Trim Video](#)

[All Other Filters In Sample](#)

[Write Your Own Filter \(WYOF\)](#)

[WYOF Filter Structure](#)

[WYOF filter_frame \(\) call flow](#)

[WYOF decode_video //ffmpeg.c](#)

[WYOF av_buffersrc_add_frame_flags//buffersrc.c](#)

[WYOF av_buffersrc_add_frame_internal
//buffersrc.c](#)

[WYOF request_frame //buffersrc.c](#)

[WYOF ff_filter_frame // avfilter.c](#)

[WYOF ff_filter_frame_framed //avfilter.c](#)

[WYOF Extract the Core Code.](#)

[WYOF filter_frame //vf_transform.c](#)

[WYOF Walk into ff_filter_frame again // avfilter.c](#)

[WYOF default_filter_frame //avfilter.c](#)

[WYOF Third time into ff_filter_frame // avfilter.c](#)

[WYOF filter_frame //buffersink.c](#)

[WYOF How to encode ffmpeg after filter](#)

[WYOF do_video_out //ffmpeg.c](#)

[WYOF Function Flowchart](#)

[Metadata](#)

[Read Metadata Sample](#)

[Write Metadata Sample](#)

[3. Audio Basics](#)

[Audio introduction](#)

[Mono](#)

[Stereo](#)

[3D Surround](#)

[Number of Samples](#)

[Frequency of Sampling](#)

[Bit Speed](#)

[VBR \(Variable Bitrate\)](#)

[CBR \(Constant Bitrate\)](#)

[MP3](#)

[Description of Audio Formats](#)

[PCM data format](#)

[PCM stream](#)

[PCM audio processing](#)

[1. Separate left and right channel audio](#)

[2. Audio pcm resampling 48000 to 44100](#)

[3. Using ffmpeg to construct silent frames](#)

[4. Halve the volume of the left channel in the
PCM16LE two-channel audio sample data](#)

[5. Double the sound speed of PCM16LE two-
channel audio sampling data](#)

[6. Convert PCM16LE two-channel audio sampling
data to PCM8 audio sampling data](#)

[7. A part of data will be intercepted from PCM16LE
mono audio sampling data](#)

[8. Convert PCM16LE two-channel audio sampling
data to WAVE format audio data](#)

[Basic Audio Effects-Volume Control](#)

[How to change PCM Sample Rate](#)

[How to change the bit depth](#)

[4. Interlaced Video](#)

[Why TV Interlaced ?](#)

[Development and Evolution of Display Technology](#)

2x Field Frequency Display

2x Horizontal Frequency Display

Progressive Display

Multi-format Receiving Single Format Display

Flat Display Device

Psf-Progressive Segmentation

Progressive Format for Broadcast

Progressive Scan Format for Production and Exchange

Progressive Scan Format for Distribution

Digital Video Interface

Digital Display Interface

DVI

HDMI

Analog Interface

Progressive Scan DVD

Interconversion of Progressive and Interlaced Signals

Digital TV Trends-Progressive, Interlaced or Progressive

HD Production Choice: 50i or 25P?

Can I Use 50i Interlaced Scanning Equipment to Copy and Make 25Psf Progressive Scanning

Programs?

Deinterlacing

5. I-B-P Frame and Timebase

I-Frame

P-Frame

B-frame

DTS and PTS

Time Base and Timestamp in FFmpeg

1. The Concept of Time Base and Timestamp

2. Three Time Bases tbr, tbn and tbc

3. Internal Time Base AV_TIME_BASE

4. Time Value Form Conversion

5 Time Base Conversion Function

6 Time-Based Conversion During the Encapsulation Process

7 Time Base Conversion During Transcoding

7.1 Video Stream

7.2 Audio Stream

6. Codecs

Video Codec

List of Open-source Codecs

Video Codec List

[Audio Codec List](#)

[FFMPEG libavcodec](#)

[7. Encoding](#)

[Video Encoding](#)

[1. FFMpeg Required Structure for Video Encoding](#)

[2. The Main Steps of FFMpeg Encoding](#)

[\(1\) Enter the Encoding Parameters](#)

[\(2\) Initialize the Required FFMpeg Structure as Required](#)

[\(3\), Encoding Loop Body](#)

[\(4\) Finishing Treatment](#)

[H.264 Encoding](#)

[H265 Encoding](#)

[VP8 Encoding](#)

[Audio Encoding](#)

[AAC Encoding](#)

[Picture Encoding](#)

[8. Decoding](#)

[1. Connect and Open the Video Stream](#)

[2. Locate Video Stream Data](#)

[3. Prepare the Decoder Codec](#)

[4. Decode](#)

[Simple Decoder Based on FFmpeg.](#)

[Simple Video Player Based on FFMPEG + SDL](#)

[Simple Audio Player Based on FFMPEG + SDL](#)

[9. Multiplexing](#)

[10. Demultiplexing](#)

[Demultiplexing to H264 and Mp3](#)

[11. Remultiplexing](#)

[12. Transcoding](#)

[Whole process](#)

[Demultiplexing](#)

[Decoding](#)

[Filter](#)

[Encoding](#)

[Reuse](#)

[Time Stamp Processing During Transcoding](#)

[13. Streaming](#)

[1. RTMP](#)

[1.1. General Introduction](#)

[1.2. Handshake](#)

[1.3 RTMP Chunk Stream](#)

[1.4. Different Types of RTMP Message](#)

[1.5 Push RTMP Streamer Sample](#)

[1.6 Save RTMP Streaming Media as a Local FLV File](#)

[2. UDP \(User Data Protocol\)](#)

[2.1. Multicasting \(Multicasting\)](#)

[2.2. UDP Broadcast](#)

[2.3. UDP Multicast](#)

[2.4. UDP Broadcast and Unicast](#)

[2.5. Pull RTMP Stream to UDP Output Playback](#)

[2.6. UDP Receiving TS Stream](#)

[3. RTP \(Real-Time Transport Protocol\)](#)

[3.1. RTP Working Mechanism](#)

[4. RTCP \(RTP Control Protocol\)](#)

[4.1. RTCP Working Mechanism](#)

[5. RTSP \(RealTime Streaming Potocol\)](#)

[Analysis of MPEG2 Transport Stream \(MPEG2 TS\)](#)

[14. Devices](#)

[Read camera](#)

[Screen capture](#)

[References](#)

[About The Author](#)

INTRODUCTION

Dear reader,

FFmpeg is the most complete set of multimedia support library in free software. It implements almost all the common data encapsulation formats, multimedia transmission protocols and audio and video codecs. It is the Swiss army knife of the multimedia industry. Today, youtube, facebook, google and many broadcasters and television channels use the ffmpeg library.

In this book, we will review the ffmpeg source code and learn about ffmpeg's capabilities. In the first part, we will consider ffmpeg in general terms. In the second part, we will discuss the ffmpeg methods related to video. In this section, we will also examine the ffmpeg filters closely.

In the third part, we will learn the details and methods of audio. In the fourth and fifth part, we will learn about Interlaced video and I-B-P frames. In the sixth

part, we will learn what is codec and how does ffmpeg handle it. In the seventh part, we will try to explain how video encoding and audio encoding take place in ffmpeg, with sample codes. In the eighth part, we will try to explain how decoding works in fmmpg and how to make a sample player with the help of SDL. In parts 9, 10 and 11, we will show the video and audio multiplexing methods. In the twelfth part, how to convert a video file into another format. We will learn all the transcoding steps. In the thirteenth part, we will learn all streaming protocols. In the last part, we will learn how to manage video and audio devices with ffmpeg.

1. FFmpeg Fundamentals

FFmpeg is the most complete set of multimedia support library in free software. It implements almost all the common data encapsulation formats, multimedia transmission protocols and audio and video codecs. It is the Swiss army knife of the multimedia industry. Therefore, for engineers engaged in multimedia technology development, in-depth study of FFmpeg becomes an indispensable job. It can be said that FFmpeg is as important to multimedia development engineers as kernel is to embedded system engineers.

FFmpeg is functionally divided into several modules, namely core tools (libutils), media formats (libavformat), codecs (libavcodec), devices (libavdevice) and post-processing (libavfilter, libswscale, libpostproc), respectively responsible for providing public function, realize the reading and writing of multimedia files, complete the coding and decoding of audio and video, manage the operation of

audio and video equipment, and perform audio and video post-processing.

FFMPEG DATA STRUCTURE

Codecs, data frames, media streams and containers are the four basic concepts of digital media processing systems.

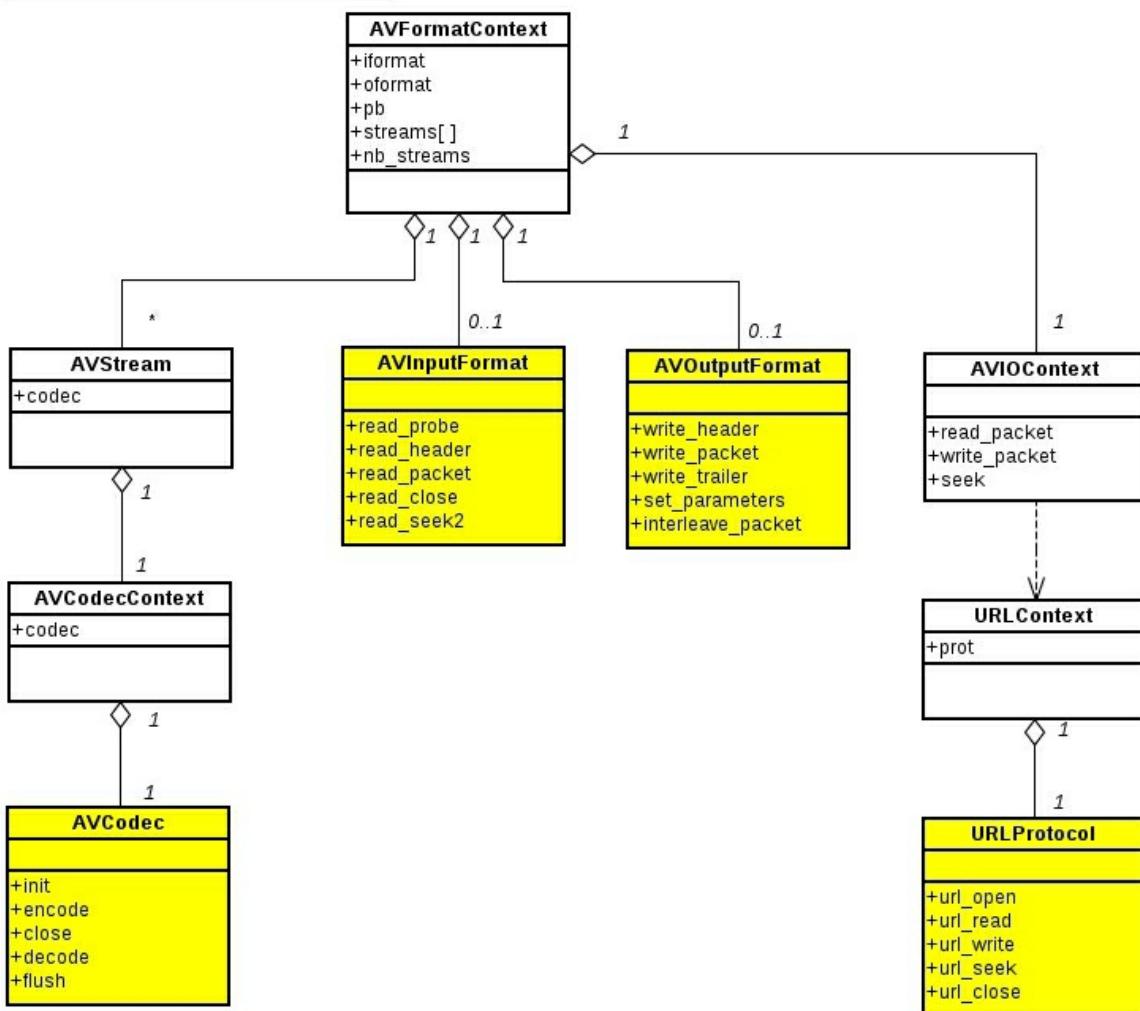
First, the terminology needs to be unified:

- Container / File : It is a multimedia file in a specific format.
- Media stream : refers to a piece of continuous data on the time axis, such as a piece of sound data, a piece of video data or a piece of subtitle data, which can be compressed or uncompressed. Compressed data needs to be associated with a specific codec.
- Frame / Packet: Generally, a media stream consists of a large number of data frames. For compressed data, the frame corresponds to the smallest processing unit of the codec. Generally, data frames belonging to different media streams are interleaved and reused in containers
- Codec: The codec converts compressed data to

original data in units of frames.

In FFmpeg, structures such as AVFormatContext, AVStream, AVCodecContext, AVCodec, and AVPacket are used to abstract these basic elements. Their relationship is shown in the following figure:

FFMpeg Overall Architecture



AVCODECCONTEXT

This is a data structure that describes the context of the codec. It contains the parameter information required by many codecs.

```
typedef struct AVCodecContext {  
    ...  
    /**  
     * some codecs need / can use extradata like Huffman tables.  
     * mjpeg: Huffman tables  
     * rv10: additional flags  
     * mpeg4: global headers (they can be in the bitstream or here)  
     * The allocated memory should be  
     FF_INPUT_BUFFER_PADDING_SIZE bytes larger  
     * than extradata_size to avoid problems if it is read with the bitstream  
     reader.  
     * The bytewise contents of extradata must not depend on the  
     architecture or CPU endianness.  
     *-encoding: Set / allocated / freed by libavcodec.  
     *-decoding: Set / allocated / freed by user.  
     */  
    uint8_t *extradata ;  
    int extradata_size ;  
    /**  
     * This is the fundamental unit of time (in seconds) in terms  
     * of which frame timestamps are represented. For fixed-fps content,  
     * timebase should be 1 / framerate and timestamp increments should  
     be  
     * identically 1.  
    */
```

```
*-encoding: MUST be set by user.  
*-decoding: Set by libavcodec.  
*/  
AVRational time_base ;  
  
/* video only */  
/**  
 * picture width / height.  
 *-encoding: MUST be set by user.  
 *-decoding: Set by libavcodec.  
 * Note: For compatibility it is possible to set this instead of  
 * coded_width / height before decoding.  
 */  
int width , height ;  
  
...  
  
/* audio only */  
int sample_rate ; //<> samples per second  
int channels ; //<number of audio channels  
  
/**  
 * audio sample format  
 *-encoding: Set by user.  
 *-decoding: Set by libavcodec.  
 */  
enum SampleFormat sample_fmt ; //<sample format  
  
/* The following data should not be initialized. */  
/**  
 * Samples per packet, initialized when calling 'init'.  
 */  
int frame_size ;  
int frame_number ; //<audio or video frame number  
  
...  
  
char codec_name [ 32 ] ;  
enum AVMediaType codec_type ; /* see AVMEDIA_TYPE_xxx */
```

```

enum CodecID codec_id ; /* see CODEC_ID_xxx */

/*
 * fourcc (LSB first, so "ABCD"-> ('D' << 24) + ('C' << 16) + ('B' << 8) + 'A').
 * This is used to work around some encoder bugs.
 * A demuxer should set this to what is stored in the field used to identify the codec.
 * If there are multiple such fields in a container then the demuxer should choose the one
 * which maximizes the information about the used codec.
 * If the codec tag field in a container is larger then 32 bits then the demuxer should
 * remap the longer ID to 32 bits with a table or other structure. Alternatively a new
 * extra_codec_tag + size could be added but for this a clear advantage must be demonstrated
 * first.
 *-encoding: Set by user, if not then the default based on codec_id will be used.
 *-decoding: Set by user, will be converted to uppercase by libavcodec during init.
 */

unsigned int codec_tag ;

...
```

```

/*
 * Size of the frame reordering buffer in the decoder.
 * For MPEG-2 it is 1 IPB or 0 low delay IP.
 *-encoding: Set by libavcodec.
 *-decoding: Set by libavcodec.
 */

int has_b_frames ;

/*
 * number of bytes per packet if constant and known or 0
 * Used by some WAV based audio codecs.
 */

int block_align ;

...
```

```

/*
 * bits per sample / pixel from the demuxer (needed for huffyuv).
 *-encoding: Set by libavcodec.
 *-decoding: Set by user.
 */

int bits_per_coded_sample ;

...
```

```
} AVCodecContext ;
```

If you are using libavcodec only, this part of the information needs to be initialized by the caller; if you are using the entire FFmpeg library, this part of the information is initialized according to the file header information and the header information in the media stream during the `avformat_open_input` and `avformat_find_stream_info` process. The definitions of several main fields are as follows:

1. `extradata / extradata_size`: This buffer stores extra information that may be used by the decoder and is filled in `av_read_frame`. Generally speaking, first, a certain format of demuxer will fill `extradata` when reading the format header information. Second, if the demuxer does not do this, for example, there may be no relevant codec information at the head, then the corresponding. The parser will continue to search from the media streams that have been demultiplexed. If no additional information is found, the buffer pointer is empty.
2. `time_base`: The time base of the codec, which is actually the frame rate (or field rate) of the video.
3. `width / height`: The width and height of the

video.

4. sample_rate / channels: Audio sampling rate and number of channels.
5. sample_fmt: The original sampling format of the audio.
6. codec_name / codec_type / codec_id / codec_tag: Codec information.

AVSTREAM

This structure describes a media stream and is defined as follows:

```
typedef struct AVStream {
    int index ;    /** <stream index in AVFormatContext*/
    int id ;       /** <format-specific stream ID */
    AVCodecContext * codec ; /** <codec context */
    /**
     * Real base framerate of the stream.
     * This is the lowest framerate with which all timestamps can be
     * represented accurately (it is the least common multiple of all
     * framerates in the stream). Note, this value is just a guess!
     * For example, if the time base is 1/90000 and all frames have either
     * Approximately 3600 or 1800 timer ticks, then r_frame_rate will be
50/1.
    */
    AVRational r_frame_rate ;

    ...
    /**
     * This is the fundamental unit of time (in seconds) in terms
     * of which frame timestamps are represented. For fixed-fps content,
     * time base should be 1 / framerate and timestamp increments should
be 1.
    */
    AVRational time_base ;

    ...
    /**

```

```

* Decoding: pts of the first frame of the stream, in stream time base.
* Only set this if you are absolutely 100% sure that the value you set
* it to really is the pts of the first frame.
* This may be undefined (AV_NOPTS_VALUE).
* @note The ASF header does NOT contain a correct start_time the
ASF
    * demuxer must NOT set this.
    */
int64_t start_time ;
/***
    * Decoding: duration of the stream, in stream time base.
    * If a source file does not specify a duration, but does specify
    * a bitrate, this value will be estimated from bitrate and file size.
    */
int64_t duration ;

#if LIBAVFORMAT_VERSION_INT <(53 << 16)
    char language [ 4 ] ; /* ISO 639-2 / B 3-letter language code (empty
string if undefined) */
#endif

/* av_read_frame () support */
enum AVStreamParseType need_parsing ;
struct AVCodecParserContext * parser ;

...
/* av_seek_frame () support */
AVIndexEntry * index_entries ; /* <Only used if the format does not
support seeking natively. */
int nb_index_entries ;
unsigned int index_entries_allocated_size ;

    int64_t nb_frames ;      /// <number of frames in this stream if known
or 0

...
/***

```

```
* Average framerate  
*/  
AVRational avg_frame_rate ;  
...  
} AVStream ;
```

The interpretation of the main fields is as follows. The value of most fields can be determined by `avformat_open_input` based on the information in the file header. The missing information needs to be further obtained by calling `avformat_find_stream_info` to read the frame and soft decode:

1. `index / id`: Index corresponds to the index of the stream. This number is automatically generated. According to the index, the stream can be indexed from the `AVFormatContext streams` table; and `id` is the identifier of the stream, depending on the specific container format. For example, for MPEG TS format, `id` is `pid`.
2. `time_base`: The time base of the stream is a real number, and the `pts` and `dts` of the media data in the stream will use this time base as the granularity. Generally, using `av_rescale` / `av_rescale_q` can realize the conversion of different time bases.
3. `start_time`: The start time of the stream, in units of the time base of the stream, usually the `pts` of

- the first frame in the stream.
- 4. duration: The total time of the stream, in units of the time base of the stream.
- 5. need_parsing: The control domain for the parsing process of this stream.
- 6. nb_frames: The number of frames in the stream.
- 7. r_frame_rate / framerate / avg_frame_rate: Frame rate related.
- 8. codec: Points to the AVCodecContext structure corresponding to the stream, and is generated when avformat_open_input is called.
- 9. parser: Points to the AVCodecParserContext structure corresponding to the stream, generated when avformat_find_stream_info is called.

AVFORMATCONTEXT

This structure describes the composition and basic information of a media file or media stream, and is defined as follows:

```
typedef struct AVFormatContext {
    const AVClass * av_class ; /** <Set by avformat_alloc_context. */
    /* Can only be iformat or oformat, not both at the same time. */
    struct AVInputFormat * iformat ;
    struct AVOutputFormat * oformat ;
    void * priv_data ;
    ByteIOContext * pb ;
    unsigned int nb_streams ;
    AVStream * streams [ MAX_STREAMS ] ;
    char filename [ 1024 ] ; /** <input or output filename */
    /* stream info */
    int64_t timestamp ;
#if LIBAVFORMAT_VERSION_INT <(53 << 16)
    char title [ 512 ] ;
    char author [ 512 ] ;
    char copyright [ 512 ] ;
    char comment [ 512 ] ;
    char album [ 512 ] ;
    int year ; /** <ID3 year, 0 if none */
    inttrack ; /** <track number, 0 if none */
    char genre [ 32 ] ; /** <ID3 genre */
#endif

    int ctx_flags ; /** <Format-specific flags, see AVFMTCTX_xx */
    /* private data for pts handling (do not modify directly). */
```

```

/** This buffer is only needed when packets were already buffered but
not decoded, for example to get the codec parameters in MPEG
streams.*/
struct AVPacketList * packet_buffer ;

/** Decoding: position of the first frame of the component, in
AV_TIME_BASE fractional seconds. NEVER set this value directly:
. IT IS deduced from the AVStream The values */
an int64_t START_TIME ;
/** Decoding: The DURATION of Stream, in fractional
AV_TIME_BASE
seconds. Only set this value if you know none of the individual
stream
durations and also dont set any of them. This is deduced from the
AVStream values if not set.*/
Int64_t duration ;
/** decoding: total file size, 0 if unknown */
int64_t file_size ;
/** Decoding: total stream bitrate in bit / s, 0 if not
available. Never set it directly if the file_size and the
duration are known as FFmpeg can compute it automatically.*/
int bit_rate ;

/* av_read_frame () support */
AVStream * cur_st ;
#if LIBAVFORMAT_VERSION_INT <(53 << 16)
const uint8_t * cur_ptr_DEPRECATED ;
int cur_len_DEPRECATED ;
AVPacket cur_pkt_DEPRECATED ;
#endif

/* av_seek_frame () support */
int64_t data_offset ; /** offset of the first packet */
int index_built ;

int mux_rate ;
unsigned int packet_size ;
int preload ;

```

```

int max_delay ;

#define AVFMT_NOOUTPUTLOOP -1
#define AVFMT_INFINITEOUTPUTLOOP 0
/** number of times to loop output in formats that support it */
int loop_output ;

int flags ;
#define AVFMT_FLAG_GENPTS 0x0001 /// <Generate missing pts even
if it requires parsing future frames.
#define AVFMT_FLAG_IGNIDX 0x0002 /// <Ignore index.
#define AVFMT_FLAG_NONBLOCK 0x0004 /// <Do not block when
reading packets from input.
#define AVFMT_FLAG_IGNDTS 0x0008 /// <Ignore DTS on frames that
contain both DTS & PTS
#define AVFMT_FLAG_NOFILLIN 0x0010 /// <Do not infer any values
from other values, just return what is stored in the container
#define AVFMT_FLAG_NOPARSE 0x0020 /// <Do not use AVParsers,
you also must set AVFMT_FLAG_NOFILLIN as the fillin code works on
frames and no parsing-> no frames.Also seeking to frames can not work if
parsing to find frame boundaries has been disabled
#define AVFMT_FLAG_RTP_HINT 0x0040 /// <Add RTP hinting to the
output file

int loop_input ;
/** decoding: size of data to probe; encoding: unused. */
unsigned int probesize ;

/**
 * Maximum time (in AV_TIME_BASE units) during which the input
should
 * be analyzed in avformat_find_stream_info().
*/
int max_analyze_duration ;

const uint8_t * key ;
int keylen ;

unsigned int nb_programs ;

```

```
AVProgram ** programs ;  
  
/**  
 * Forced video codec_id.  
 * Demuxing: Set by user.  
 */  
enum CodecID video_codec_id ;  
/**  
 * Forced audio codec_id.  
 * Demuxing: Set by user.  
 */  
enum CodecID audio_codec_id ;  
/**  
 * Forced subtitle codec_id.  
 * Demuxing: Set by user.  
 */  
enum CodecID subtitle_codec_id ;  
  
/**  
 * Maximum amount of memory in bytes to use for the index of each  
 stream.  
 * If the index exceeds this size, entries will be discarded as  
 * needed to maintain a smaller size. This can lead to slower or less  
 * accurate seeking (depends on demuxer).  
 * Demuxers for which a full in-memory index is mandatory will ignore  
 * this.  
 * muxing: unused  
 * demuxing: set by user  
 */  
unsigned int max_index_size ;  
  
/**  
 * Maximum amount of memory in bytes to use for buffering frames  
 * obtained from realtime capture devices.  
 */  
unsigned int max_picture_buffer ;  
unsigned int nb_chapters ;
```

```

AVChapter ** chapters ;

/**
 * Flags to enable debugging.
 */
int debug ;
#define FF_FDEBUG_TS 0x0001

/**
 * Raw packets from the demuxer, prior to parsing and decoding.
 * This buffer is used for buffering packets until the codec can
 * be identified, as parsing cannot be done without knowing the
 * codec.
 */
struct AVPacketList * raw_packet_buffer ;
struct AVPacketList * raw_packet_buffer_end ;
struct AVPacketList * packet_buffer_end ;

AVMetadata * metadata ;

/**
 * Remaining size available for raw_packet_buffer, in bytes.
 * NOT PART OF PUBLIC API
 */
#define RAW_PACKET_BUFFER_SIZE 2500000
int raw_packet_buffer_remaining_size ;

/**
 * Start time of the stream in real world time, in microseconds
 * since the unix epoch (00:00 1st January 1970). That is, pts = 0
 * in the stream was captured at this real world time.
 *-encoding: Set by user.
 *-decoding: Unused.
 */
int64_t start_time_realtime ;
} AVFormatContext ;

```

This is the most basic structure in FFMpeg, the

root of all other structures, and the fundamental abstraction of a multimedia file or stream. Among them:

- The AVStream structure pointer array represented by nb_streams and streams contains the description of all embedded media streams;
- iformat and oformat point to the corresponding demuxer and muxer pointers;
- pb points to a ByteIOContext structure that controls the reading and writing of the underlying data.
- start_time and duration are the starting time and length of the multimedia file inferred from each AVStream in the streams array, in subtle units.

Usually, this structure is created internally by avformat_open_input and initializes some members with default values. However, if the caller wants to create the structure by himself, he needs to explicitly set default values for some members of the structure-if there is no default value, it will cause an exception in the subsequent actions. The following members need to be followed:

- probesize
- mux_rate

- packet_size
- flags
- max_analyze_duration
- key
- max_index_size
- max_picture_buffer
- max_delay

AVPACKET

AVPacket is defined in avcodec.h as follows:

```
typedef struct AVPacket {
    /**
     * Presentation timestamp in AVStream-> time_base units; the time at
     * which
     *   * the decompressed packet will be presented to the user.
     *   * Can be AV_NOPTS_VALUE if it is not stored in the file.
     *   * pts MUST be larger or equal to dts as presentation cannot happen
     * before
     *   * decompression, unless one wants to view hex dumps. Some formats
     * misuse
     *   * the terms dts and pts / cts to mean something different. Such
     * timestamps
     *   * must be converted to true pts / dts before they are stored in
     * AVPacket.
     */
    int64_t pts ;
    /**
     * Decompression timestamp in AVStream-> time_base units; the time
     * at which
     *   * the packet is decompressed.
     *   * Can be AV_NOPTS_VALUE if it is not stored in the file.
     */
    int64_t dts ;
    uint8_t * data ;
    int size ;
    int stream_index ;
    int flags ;
    /**

```

```

 * Duration of this packet in AVStream-> time_base units, 0 if
unknown.
 * Equals next_pts-this_pts in presentation order.
 */
int duration ;
void (*destruct)( struct AVPacket * ) ;
void *priv ;
int64_t pos ; //<byte position in stream, -1 if unknown

/**
 * Time difference in AVStream-> time_base units from the pts of this
 * packet to the point at which the output from the decoder has
converged
 * independent from the availability of previous frames. That is, the
 * frames are virtually identical no matter if decoding started from
 * the very first frame or from this keyframe.
 * Is AV_NOPTS_VALUE if unknown.
 * This field is not the display duration of the current packet.
 *
 * The purpose of this field is to allow seeking in streams that have no
 * keyframes in the conventional sense. It corresponds to the
 * recovery point SEI in H.264 and match_time_delta in NUT. It is also
 * essential for some types of subtitle streams to ensure that all
 * subtitles are correctly displayed after seeking.
 */
int64_t convergence_duration ;
} AVPacket ;

```

FFMPEG uses AVPacket to temporarily store media data packets and additional information (decoding timestamp, display timestamp, duration, etc.). Such media data packets often carry audio and video data not in the original format, but are encoded in some way. The data and encoding information are

given by the corresponding media stream structure AVStream. AVPacket contains the following data fields:

- dts represents the decoding timestamp, pts represents the display timestamp, and their unit is the time reference of the media stream to which they belong.
- stream_index gives the index of the media stream to which it belongs;
- data is the data buffer pointer and size is the length;
- duration is the duration of the data, and is also based on the time basis of the media stream to which it belongs;
- pos indicates the byte offset of the data in the media stream;
- destruct is a function pointer used to release the data buffer;
- Flags are flag fields, where the lowest setting is 1 to indicate that the data is a key frame.

The AVPacket structure itself is just a container, it uses the data member to refer to the actual data buffer. There are two ways to manage this buffer, one is to directly create a buffer by calling av_new_packet,

and the other is to reference an existing buffer. The buffer is released by calling `av_free_packet`, and its internal implementation also uses two different release methods. The first method is to call the `destruct` function of `AVPacket`. This `destruct` function may be the default `av_destruct_packet`, corresponding to the buffer created by `av_new_packet` or `av_dup_packet`. The area may also be a custom release function, indicating that the provider of the buffer hopes that the user will release it in the way the provider expects when the buffer is ended. The second way is to only clear the `data` and `size` to 0. In this case, it often refers to an existing buffer, and the `destruct` pointer of `AVPacket` is empty.

When using `AVPacket`, for the provider of the buffer, you must pay attention to specify the correct release method by setting the `destruct` function pointer. If the buffer provider intends to release the buffer by itself, remember to set `destruct` to empty; and for the user of the buffer, be sure to call `av_free_packet` at the end of the use to release the buffer (although the release operation may just be a fake action). If a user intends to occupy an `AVPacket` for a long time—for example, do not plan to release it before the function returns it. It is best to call `av_dup_packet` to clone the

buffer and convert it into its own allocated buffer to avoid buffering. The improper occupation of the area caused an abnormal error. `av_dup_packet` will create a new buffer for `AVPacket` where the `destruct` pointer is empty, then copy the data from the original buffer to the new buffer, set the value of data to the address of the new buffer, and set the `destruct` pointer to `av_destruct_packet`.

The above media structure can be visually displayed by the `av_dump_format` method provided by FFMPEG. The following example takes an MPEG-TS file as input and the display result is:

```
Input # 0, mpegt, from  
'/Videos/suite/ts/H.264_High_L3.1_720x480_23.976fps_AAC-LC.ts':
```

```
Duration: 00: 01: 43.29, start: 599.958300, bitrate: 20934 kb / s
```

```
Program 1
```

```
Stream # 0.0 [0x1011]: Video: h264 (High), yuv420p, 720x480 [PAR  
32:27 DAR 16: 9], 23.98 fps, 23.98 tbr, 90k tbn, 47.95 tbc
```

```
Stream # 0.1 [0x1100]: Audio: aac, 48000 Hz, stereo, s16, 159 kb / s
```

You can find the format, path, duration, start time, and global bit rate of the media. In addition, a program included in the media is listed, consisting of two media streams, the first media stream is a video stream with an id of 0x1011; the second is an audio stream with an id of 0x1100. The video stream is High

Profile encoding of h264, the color space is 420p, the size is 720×480 , the average frame rate is 23.98, the reference frame rate is 23.98, the stream time reference is 90000, the encoding time reference is 47.95; the audio stream is aac encoding, 48kHz Sampling, stereo, 16 bits per sample, the bit rate of the stream is 159kbps.

AVIOCONTEXT

AVIOContext mainly uses logic: we have a large video file buffer, and then ffmpeg accesses this piece of data with the help of the io context, which maintains a small buffer on its own, and then ffmpeg internally decapsulates and decodes. You need all the data you need. When the data buffer of this small buffer in the io context is insufficient, you will constantly add data from the large buffer of our video. AVIOContext is defined in avio.h as follows:

```
typedef struct AVIOContext {
    /**
     * A class for private options.
     *
     * If this AVIOContext is created by avio_open2(), av_class is set and
     * passes the options down to protocols.
     *
     * If this AVIOContext is manually allocated, then av_class may be set
     * by
     * the caller.
     *
     * warning -- this field can be NULL, be sure to not pass this
     * AVIOContext
     * to any av_opt_* functions in that case.
     */
    const AVClass *av_class;
    unsigned char *buffer; /**< Start of the buffer. */
```



```
 * A combination of AVIO_SEEKABLE_ flags or 0 when the stream is
not seekable.
 */
int seekable;

/***
 * max filesize, used to limit allocations
 * This field is internal to libavformat and access from outside is not
allowed.
 */
int64_t maxsize;

/***
 * avio_read and avio_write should if possible be satisfied directly
 * instead of going through a buffer, and avio_seek will always
 * call the underlying seek function directly.
 */
int direct;

/***
 * Bytes read statistic
 * This field is internal to libavformat and access from outside is not
allowed.
 */
int64_t bytes_read;

/***
 * seek statistic
 * This field is internal to libavformat and access from outside is not
allowed.
 */
int seek_count;

/***
 * writeout statistic
 * This field is internal to libavformat and access from outside is not
allowed.
 */
int writeout_count;
```

```
/**  
 * Original buffer size  
 * used internally after probing and ensure seekback to reset the buffer  
size  
 * This field is internal to libavformat and access from outside is not  
allowed.  
 */  
int orig_buffer_size;  
} AVIOContext;
```

URLCONTEXT

The opaque of AVIOContext actually points to a URLContext object, which encapsulates the protocol object and protocol operation object, where prot points to a specific protocol operation object, and priv_data points to a specific protocol object.

```
typedef struct URLContext {
    const AVClass *av_class;    /**< information for av_log(). Set by
url_open(). */
    struct URLProtocol *prot;
    void *priv_data;
    char *filename;           /**< specified URL */
    int flags;
    int max_packet_size;      /**< if non zero, the stream is packetized
with this max packet size */
    int is_streamed;          /**< true if streamed (no seek possible), default
= false */
    int is_connected;
    AVIOInterruptCB interrupt_callback;
    int64_t rw_timeout;       /**< maximum time to wait for (network)
read/write operation completion, in mcs */
} URLContext;
```

URLPROTOCOL

URLProtocol is the structure of FFmpeg operation files (including files, network data streams, etc.), including open, close, read, write, seek and other operations. In the av_register_all () function, by calling the REGISTER_PROTOCOL () macro, all URLProtocols are saved in the linked list with first_protocol as the linked list header. The definition of URLProtocol structure (simplified version, not all members are completely listed) is:

```
typedef struct URLProtocol {
    const char *name;
    int (*url_open)( URLContext *h, const char *url, int flags);
    int (*url_read)( URLContext *h, unsigned char *buf, int size);
    int (*url_write)(URLContext *h, const unsigned char *buf, int size);
    int64_t (*url_seek)( URLContext *h, int64_t pos, int whence);
    int (*url_close)(URLContext *h);
        int (*url_get_file_handle)(URLContext *h);
    struct URLProtocol *next;
    int priv_data_size;
    const AVClass *priv_data_class;
} URLProtocol;
```

TIME INFORMATION

Time information is used to achieve multimedia synchronization. The purpose of synchronization is to maintain the inherent time relationship between media objects when displaying multimedia information. There are two types of synchronization, one is intra-stream synchronization, its main task is to ensure the time relationship within a single media stream to meet the perception requirements, such as playing a video at a specified frame rate; the other is inter-stream synchronization, the main task is to ensure the time relationship between different media streams, such as the relationship between audio and video (lipsync).

For fixed rate media, such as fixed frame rate video or fixed bit rate audio, you can put the time information (frame rate or bit rate) in the file header (header), such as AVI's hdrl List, MP4's moov box , and There is a relatively complex scheme that embeds time information inside the media stream, such as MPEG TS and Real video. This scheme can handle variable-rate media and can also effectively avoid time

drift during synchronization. FFmpeg will time stamp each data packet to more effectively support the synchronization mechanism of upper-layer applications. There are two types of time stamps, one is DTS, called decoding time stamp, and the other is PTS, called display time stamp. For sound, the two time stamps are the same, but for some video encoding formats, due to the use of bidirectional prediction technology, it will cause inconsistency between DTS and PTS.

When there is no bidirectional prediction frame:

Image type: IPPPPP...IPP
DTS: 0 1 2 3 4 5 6 ... 100 101 102
PTS: 0 1 2 3 4 5 6 ... 100 101 102

THERE ARE TWO-WAY PREDICTION FRAMES:

Image type: IPBBPBB ... IPB
DTS: 0 1 2 3 4 5 6 ... 100 101 102
PTS: 0 3 1 2 6 4 5 ... 100 104 102

For the case where there are bidirectionally predicted frames, the decoder is usually required to reorder the images to ensure that the output image order is the display order:

Decoder input: IPBBPBB
(DTS) 0 1 2 3 4 5 6
(PTS) 0 3 1 2 6 4 5
Decoder output: XIBBPBBP
(PTS) X 0 1 2 3 4 5 6

TIME INFORMATION ACQUISITION

By calling `avformat_find_stream_info`, multimedia applications can get the time information of the media file from the `AVFormatContext` object: mainly the total time length and start time, in addition to the bit rate and file size related to the time information. The unit of time information is `AV_TIME_BASE`: microseconds.

```
typedef struct AVFormatContext {  
    ...  
    /** Decoding: position of the first frame of the component, in  
     * AV_TIME_BASE fractional seconds. NEVER set this value directly:  
     * . IT IS deduced from the AVStream The values */  
    an int64_t START_TIME ;  
    /** Decoding: The DURATION of Stream, in fractional  
     * AV_TIME_BASE  
     * seconds. Only set this value if you know none of the individual  
     * stream  
     * durations and also dont set any of them. This is deduced from the  
     * AVStream values if not set. */  
    Int64_t duration ;  
    /** decoding: total file size, 0 if unknown */  
    int64_t file_size ;  
    /** Decoding: total stream bitrate in bit / s, 0 if not
```

```
available. Never set it directly if the file_size and the  
duration are known as FFmpeg can compute it automatically. */  
int bit_rate ;  
...  
} AVFormatContext ;
```

The above 4 member variables are all read-only. FFMpeg-based middleware needs to be encapsulated into an interface, such as:

```
LONG GetDuration ( IntfX * ) ;  
LONG GetStartTime ( IntfX * ) ;  
LONG GetFileSize ( IntfX * ) ;  
LONG GetBitRate ( IntfX * ) ;
```

APIS

Most of FFmpeg's APIs use 0 as the successful return value and a negative number as the error code.

READING SERIES

The main function of the Reading Series API is to obtain media packets based on a specified source. This source can be a local file, an RTSP or HTTP source, a camera driver, or others.

avformat_open_input

```
int avformat_open_input( AVFormatContext ** ic_ptr , const char * filename , AVInputFormat * fmt , AVDictionary ** options ) ;
```

avformat_open_input completes two tasks:

1. Open a file or URL, and the underlying input module based on the byte stream is initialized.
2. Parse the header information of the multimedia file or multimedia stream, create the AVFormatContext structure and fill in the key fields, and in turn establish the AVStream structure for each original stream.

The relationship between a multimedia file or multimedia stream and the original stream it contains is as follows:

Multimedia files / multimedia streaming (movie.mkv)

Original Stream 1 (h.264 video)

Original Stream 2 (aac audio for Turkish)

Original Stream 3 (aac audio for english)

Original Stream 4 (Turkish Subtitle)

Original Stream 5 (English Subtitle) ...

About input parameters:

- ic_ptr, which is a pointer to a pointer, used to return an AVFormatContext structure internally constructed by avformat_open_input.
- filename, specify the file name.
- fmt is used to explicitly specify the format of the input file. If it is set to null, the input format is automatically determined.
- options

This function can obtain enough information about files, streams and codecs by parsing the header information and other auxiliary data of multimedia files or streams, but the information provided by any multimedia format is limited and different Multimedia content creation software has different settings for header information. In addition, these software will inevitably introduce some errors when generating multimedia content. Therefore, this function does not guarantee that all required information can be obtained.

In this case, you need to consider another Function:
`avformat_find_stream_info`.

avformat_find_stream_info

```
int avformat_find_stream_info ( AVFormatContext * ic , AVDictionary **  
options ) ;
```

This function is mainly used to obtain the necessary codec parameters, set to `ic` → streams [i] → codec.

First, you must get the type and id of the codec corresponding to each media stream. These are the two enumerations defined in `avutil.h` and `avcodec.h`:

```
enum AVMediaType {  
    AVMEDIA_TYPE_UNKNOWN = -1,  
    AVMEDIA_TYPE_VIDEO,  
    AVMEDIA_TYPE_AUDIO,  
    AVMEDIA_TYPE_DATA,  
    AVMEDIA_TYPE_SUBTITLE,  
    AVMEDIA_TYPE_ATTACHMENT,  
    AVMEDIA_TYPE_NB  
};  
enum CodecID {  
    CODEC_ID_NONE,  
    /* video codecs */  
    CODEC_ID_MPEG1VIDEO,  
    CODEC_ID_MPEG2VIDEO, //<preferred ID for MPEG-1 / 2 video  
decoding  
    CODEC_ID_MPEG2VIDEO_XVMC,  
    CODEC_ID_H261,  
    CODEC_ID_H263,
```

```
}; ...
```

Generally, if a certain media format has complete and correct header information, you can get these two parameters by calling `avformat_open_input`, but if `avformat_open_input` cannot obtain them for some reason, this task will be completed by `avformat_find_stream_info`.

Secondly, it is necessary to obtain the time reference of the codec corresponding to each media stream. In addition, for audio codecs, you also need to get:

1. Sampling Rate,
2. Number of channels,
3. Bit width,
4. Frame length (necessary for some codecs),

For the video codec, it is:

1. Image size,
2. Color space and format,

av_read_frame

```
int av_read_frame ( AVFormatContext * s , AVPacket * pkt ) ;
```

This function is used to read media data from multimedia files or multimedia streams. The acquired

data is stored by the AVPacket structure pkt. For audio data, if it is a fixed bit rate, one or more audio frames are loaded in pkt; if it is a variable bit rate, one audio frame is loaded in pkt. For video data, a video frame is loaded in pkt. It should be noted that: before calling this function again, you must use av_free_packet to release the resources occupied by pkt.

The type of acquired media data can be found through `pkt → stream_index`, so that the data is sent to the corresponding decoder for subsequent processing.

av_seek_frame

```
int av_seek_frame ( AVFormatContext * s , int stream_index , int64_t  
timestamp , int flags ) ;
```

This function achieves random access to media files by changing the read and write pointers of the media files, and supports the following three methods:

- Random access based on time: Specifically, the media file read and write pointer is positioned at a given point in time, and then the media data with the time tag equal to the given point in time can be read when `av_read_frame` is called, which is usually used to implement Media player's fast forward and rewind functions.

- Random access based on file offset: equivalent to the seek function of ordinary files, timestamp also becomes the offset of the file.
- Random access based on frame number: timestamp is the frame number of the media data to be accessed.

About parameters:

- s: is an AVFormatContext pointer, which is the structure returned by `avformat_open_input`.
- stream_index: Specifies the media stream. If it is a random access based on time, the third parameter timestamp will use the time base of the media stream as a unit; if it is set to a negative number, it is equivalent to not specifying a specific media stream. FFMPEG will follow the specific The algorithm searches for the default media stream. At this time, the unit of timestamp is `AV_TIME_BASE` (microseconds).
- timestamp: Time label, the unit depends on other parameters.
- flags: positioning mode,
`AVSEEK_FLAG_BYTE` means based on byte offset, `AVSEEK_FLAG_FRAME` means based on frame number, other means based on time.

av_close_input_file

```
void av_close_input_file ( AVFormatContext * s ) ;
```

Close a media file: release resources and close physical IO.

CODEC SERIES

The codec API is responsible for encoding the original media data and decoding the media compressed data.

avcodec_find_decoder

```
AVCodec * avcodec_find_decoder( enum CodecID id ) ;  
AVCodec * avcodec_find_decoder_by_name( const char * name ) ;
```

According to the given codec id or decoder name, the system searches and returns a pointer to the AVCodec structure.

avcodec_open2

```
int avcodec_open2( AVCodecContext * avctx , const AVCodec * codec ,  
AVDictionary ** options ) ;
```

This function specifies the AVCodecContext structure based on the input AVCodec pointer. Before calling this function, you need to call avcodec_alloc_context to allocate an AVCodecContext structure, or call avformat_open_input to get the AVCodecContext

structure of the corresponding media stream in the media file; in addition, you need to get the AVCodec structure through `avcodec_find_decoder`. This function will also initialize the corresponding decoder.

`avcodec_decode_video2`

```
int avcodec_decode_video2 ( AVCodecContext * avctx , AVFrame *  
picture , int * got_picture_ptr , AVPacket * avpkt ) ;
```

Decode a video frame. `got_picture_ptr` indicates whether there is decoded data output. The input data is in the `AVPacket` structure, and the output data is in the `AVFrame` structure. `AVFrame` is a data structure defined in `avcodec.h`:

```
typedef struct AVFrame {  
    FF_COMMON_FRAME  
} AVFrame ;
```

`FF_COMMON_FRAME` defines many data fields, most of which are used internally by FFMpeg. For users, the more important ones include:

```
#define FF_COMMON_FRAME \  
    uint8_t * data [ 4 ] ; \  
    int linesize [ 4 ] ; \  
    int key_frame ; \  
    int pict_type ; \  
}
```

```
int64_t pts ; \
int reference ; \
```

FFMpeg internally stores the original image data in a planar manner, which divides the image pixels into multiple planes (R / G / B or Y / U / V), and the pointers in the data array point to the starting positions of the four pixel planes, linesize The array stores the line width of each buffer that stores each plane:

```
+++++ data [0]-> ##### picture data ##### ++++++
+++++ #+++++ picture data ##### ++++++
+++++ #+++++ picture data ##### ++++++
+++++ #+++++ picture data ##### ++++++
+++++ .....
```

In addition, key_frame identifies whether the image is a key frame; pict_type indicates the encoding type of the image: I (1) / P (2) / B (3) ...; pts is a time label in units of time_base, for some decoders Such as H.261, H.263 and MPEG4, can be obtained from the header information; reference indicates whether the image is used as a reference.

avcodec_decode_audio4

```
int avcodec_decode_audio4( AVCodecContext * avctx , AVFrame * frame , int * got_frame_ptr , AVPacket * apk ) ;
```

Decode an audio frame. The input data is in the AVPacket structure, the output data is in the frame, and got_frame_ptr indicates whether there is data output.

avcodec_close

```
int avcodec_close ( AVCodecContext * avctx ) ;
```

Close the decoder and release the resources allocated in avcodec_open.

WRITE SERIES

The Write Series API is responsible for distributing media data in the form of packets to a specified target. This target can be a local file, an RTSP or HTTP stream, or others.

avformat_alloc_output_context2

```
int avformat_alloc_output_context2 ( AVFormatContext ** ctx ,  
AVOutputFormat * oformat , const char * format_name , const  
char * filename ) ;
```

This function is responsible for allocating an AVFormatContext for output purposes. The output format is determined by oformat, format_name, and filename. Oformat has the highest priority. If oformat is empty, the format_name is used. If format_name is empty, the file name is used.

avformat_write_header

```
int avformat_write_header ( AVFormatContext * s , AVDictionary **  
options ) ;
```

This function is responsible for generating a file header and writing it to the output target. Before calling, the oformat and pb in the AVFormatContext structure must be correctly set, and the streams list cannot be empty. The codec parameters of each stream in the list need to be configured correctly.

av_write_frame

```
int av_write_frame ( AVFormatContext * s , AVPacket * pkt ) ;
```

This function is responsible for outputting a media package. The AVFormatContext parameter gives the output target and the format setting of the media stream. “pkt” is a structure that contains media data, and its internal members such as dts / pts and stream_index must be correctly set.

av_interleaved_write_frame

```
int av_interleaved_write_frame ( AVFormatContext * s , AVPacket * pkt ) ;
```

This function is responsible for interleaving a media packet. If the caller cannot guarantee that the packets from each media stream are interleaved correctly, it is best to call this function to output the

media packet, otherwise, you can call `av_write_frame` to improve performance.

`av_write_trailer`

```
int av_write_trailer ( AVFormatContext * s );
```

OTHER

avformat_new_stream

```
AVStream * avformat_new_stream ( AVFormatContext * s , AVCodec * c  
);
```

This function is responsible for creating an AVStream structure and adding it to the specified AVFormatContext. At this time, most fields of AVStream are in an illegal state. If the input parameter c is not empty, the codec field of AVStream is initialized according to c.

SEVERAL LIBAVUTIL TOOLS

AVLog is a log output tool for FFmpeg. In FFmpeg, all log output is not through the printf () function but through the av_log () function. av_log () will eventually call the fprintf (stderr, ...) function to output the log content to the command line interface. But in some non-command line programs (MFC programs, Android programs, etc.), the fprintf (stderr, ...) called by av_log () cannot display the log content. For this situation, FFmpeg provides a log callback function av_log_set_callback (). This function can specify a custom log output function to output the log to a specified location.

The following custom function custom_output () outputs the log to the text "simplest_ffmpeg_log.txt".

```
void custom_output(void* ptr, int level, const char* fmt,va_list vl){  
    FILE *fp = fopen("simplest_ffmpeg_log.txt","a+");  
    if(fp){  
        vfprintf(fp,fmt,vl);  
        fflush(fp);  
        fclose(fp);  
    }  
}
```

```
}
```

Call `av_log_set_callback()` in the main function to set this function, as shown below.

```
int main(int argc, char* argv[])
{
    av_log_set_callback(custom_output);
    return 0;
}
```

In addition, the log information is divided into Panic, Fatal, Error, Warning, Info, Verbose, Debug from heavy to light. The following function outputs several different levels of logs.

```
void test_log(){
    av_register_all();
    AVFormatContext *obj=NULL;
    obj=avformat_alloc_context();
    av_log(obj,AV_LOG_PANIC,"Panic: Something went really wrong
and we will crash now.\n");
    av_log(obj,AV_LOG_FATAL,"Fatal: Something went wrong and
recovery is not possible.\n");
    av_log(obj,AV_LOG_ERROR,"Error: Something went wrong and
cannot losslessly be recovered.\n");
    av_log(obj,AV_LOG_WARNING,"Warning: This may or may not
lead to problems.\n");
    av_log(obj,AV_LOG_INFO,"Info: Standard information.\n");
    av_log(obj,AV_LOG_VERBOSE,"Verbose: Detailed
information.\n");
    av_log(obj,AV_LOG_DEBUG,"Debug: Stuff which is only useful
for libav* developers.\n");
```

```
    printf("=====\\n");
    avformat_free_context(obj);
}
```

AVOPTION (AVCLASS)

AVOption is FFmpeg's option setting tool. The most relevant option setting function related to AVOption is `av_opt_set()`. The core concept of AVOption is "operating the attribute value of a structure according to a string". For example, the function of the code between "#if" and "#else" in the following code is the same as the function of the code between "#else" and "#endif".

```
#if TEST_OPT
    av_opt_set(pCodecCtx,"b","400000",0);          //bitrate
    //Another method
    //av_opt_set_int(pCodecCtx,"b",400000,0);      //bitrate

    av_opt_set(pCodecCtx,"time_base","1/25",0);    //time_base
    av_opt_set(pCodecCtx,"bf","5",0);              //max b frame
    av_opt_set(pCodecCtx,"g","25",0);              //gop
    av_opt_set(pCodecCtx,"qmin","10",0);            //qmin/qmax
    av_opt_set(pCodecCtx,"qmax","51",0);

#else
    pCodecCtx->time_base.num = 1;
    pCodecCtx->time_base.den = 25;
    pCodecCtx->max_b_frames=5;
    pCodecCtx->bit_rate = 400000;
    pCodecCtx->gop_size=25;
    pCodecCtx->qmin = 10;
    pCodecCtx->qmax = 51;
```

```
#endif
```

Similarly, `av_opt_get()` can return the attribute value of the structure as a string. For example, the following code verifies the role of `av_opt_get()`:

```
char *val_str=(char *)av_malloc(50);

//preset: ultrafast, superfast, veryfast, faster, fast,
//medium, slow, slower, veryslow, placebo
av_opt_set(pCodecCtx->priv_data,"preset","slow",0);
//tune: film, animation, grain, stillimage, psnr,
//ssim, fastdecode, zerolatency
av_opt_set(pCodecCtx->priv_data,"tune","zerolatency",0);
//profile: baseline, main, high, high10, high422, high444
av_opt_set(pCodecCtx->priv_data,"profile","main",0);

//print
av_opt_get(pCodecCtx->priv_data,"preset",0,(uint8_t **)&val_str);
printf("preset val: %s\n",val_str);
av_opt_get(pCodecCtx->priv_data,"tune",0,(uint8_t **)&val_str);
printf("tune val: %s\n",val_str);
av_opt_get(pCodecCtx->priv_data,"profile",0,(uint8_t **)&val_str);
printf("profile val: %s\n",val_str);
av_free(val_str);
```

You can get the AVOption structure of any option in the structure through `av_opt_find()`. Write a simple function to read the values of some fields in the structure.

```
void print_opt(const AVOption *opt_test){

    printf("=====\\n");
```

```
printf("Option Information:\n");
printf("[name]%s\n",opt_test->name);
printf("[help]%s\n",opt_test->help);
printf("[offset]%"PRIu64"\n",opt_test->offset);

switch(opt_test->type){
case AV_OPT_TYPE_INT:{
printf("[type]int\n[default]"PRId64"\n",opt_test->default_val.i64);
break; }
case AV_OPT_TYPE_INT64:{
printf("[type]int64\n[default]"PRId64"\n",opt_test->default_val.i64);
break; }

case AV_OPT_TYPE_FLOAT:{
printf("[type]float\n[default]"PRIf64"\n",opt_test->default_val.dbl);
break; }

case AV_OPT_TYPE_STRING:{

printf("[type]string\n[default]"PRIs8"\n",opt_test->default_val.str);
break; }

case AV_OPT_TYPE_RATIONAL:{

printf("[type]rational\n[default]"PRId64"/%"PRId64"\n",opt_test-
>default_val.q.num,opt_test->default_val.q.den);
break; }
default:{

printf("[type]others\n");
break; }
}

printf("[max val]"PRIf64"\n",opt_test->max);
printf("[min val]"PRIf64"\n",opt_test->min);

if(opt_test->flags&AV_OPT_FLAG_ENCODING_PARAM){
printf("Encoding param.\n");
}
if(opt_test->flags&AV_OPT_FLAG_DECODING_PARAM){
printf("Decoding param.\n");
}
```

```

if(opt_test->flags&AV_OPT_FLAG_AUDIO_PARAM){
printf("Audio param.\n");
}
if(opt_test->flags&AV_OPT_FLAG_VIDEO_PARAM){
printf("Video param.\n");
}
if(opt_test->unit!=NULL)
printf("Unit belong to:%s\n",opt_test->unit);

printf("=====\\n");
}

```

Use the following code to call the above function to print out the value of each field of the AVOption structure.

```

const AVOption *opt=NULL;
opt=av_opt_find(pCodecCtx, "b", NULL, 0, 0);
print_opt(opt);
opt=av_opt_find(pCodecCtx, "g", NULL, 0, 0);
print_opt(opt);
opt=av_opt_find(pCodecCtx, "time_base", NULL, 0, 0);
print_opt(opt);

```

The following code can print out all options of the structure that supports AVOption (that is, including AVClass):

```

void list_obj_opt(void *obj){
printf("Output some option info about object:\n");
printf("Object name:%s\\n",(*(AVClass **) obj)->class_name);
printf("=====\\n");
printf("Video param:\\n");

```

```
av_opt_show2(obj,stderr,AV_OPT_FLAG_VIDEO_PARAM,NULL);
printf("Audio param:\n");
av_opt_show2(obj,stderr,AV_OPT_FLAG_AUDIO_PARAM,NULL);
printf("Decoding param:\n");
av_opt_show2(obj,stderr,AV_OPT_FLAG_DECODING_PARAM,NULL);
printf("Encoding param:\n");
av_opt_show2(obj,stderr,AV_OPT_FLAG_ENCODING_PARAM,NULL);
printf("=====\n");
```

The following code calls the above function to print out all the options in AVFormatContext.

```
void test_opt(){
    av_register_all();
    AVFormatContext *obj=NULL;
    obj=avformat_alloc_context();
    list_obj_opt(obj);
    avformat_free_context(obj);
}
```

AVDICTIONARY

AVDictionary is a key-value storage tool for FFmpeg. FFmpeg often uses AVDictionary to set / read internal parameters. The following code records the use of AVDictionary.

```
void test_avdictionary(){

    AVDictionary *d = NULL;
    AVDictionaryEntry *t = NULL;
    av_dict_set(&d, "name", "akin", 0);
    av_dict_set(&d, "email", "akin@devtek.com.tr", 0);
    av_dict_set(&d, "school", "ktu", 0);
    av_dict_set(&d, "gender", "man", 0);
    av_dict_set(&d, "website", "https://www.easymediasuite.com/", 0);
    //av_strdup()
    char *k = av_strdup("location");
    char *v = av_strdup("istanbul-Turkey");
    av_dict_set(&d, k, v, AV_DICT_DONT_STRDUP_KEY |
    AV_DICT_DONT_STRDUP_VAL);
    printf("=====\n");
    int dict_cnt= av_dict_count(d);
    printf("dict_count:%d\n",dict_cnt);
    printf("dict_element:\n");
    while (t = av_dict_get(d, "", t, AV_DICT_IGNORE_SUFFIX)) {
        printf("key:%10s | value:%s\n",t->key,t->value);
    }
    t = av_dict_get(d, "email", t, AV_DICT_IGNORE_SUFFIX);
    printf("email is %s\n",t->value);
    av_dict_free(&d);
```

}

PARSEUTIL

ParseUtil is a string parsing tool for FFmpeg. Its resolution parsing function `av_parse_video_size()` can parse an image with a width of 1920 and a height of 1080 from a string like "1920x1080"; its frame rate function `av_parse_video_rate()` can parse out the frame rate information; its time parsing function can parse the time in milliseconds from a string of the form "00:01:01". The following code records how to use ParseUtil.

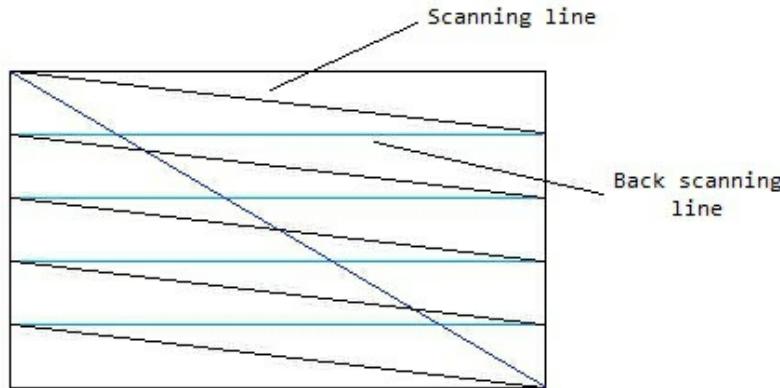
```
void test_parseutil(){
    char input_str[100]={0};
    printf("===== Parse Video Size =====\n");
    int output_w=0;
    int output_h=0;
    strcpy(input_str,"1920x1080");
    av_parse_video_size(&output_w,&output_h,input_str);
    printf("w:%4d | h:%4d\n",output_w,output_h);
    strcpy(input_str,"vga");
    //strcpy(input_str,"hd1080");
    //strcpy(input_str,"ntsc");
    av_parse_video_size(&output_w,&output_h,input_str);
    printf("w:%4d | h:%4d\n",output_w,output_h);
    printf("===== Parse Frame Rate =====\n");
    AVRational output_rational={0,0};
```

```
strcpy(input_str,"15/1");
av_parse_video_rate(&output_rational,input_str);
printf("framerate:%d/%d\n",output_rational.num,output_rational.den);
strcpy(input_str,"pal");
av_parse_video_rate(&output_rational,input_str);
printf("framerate:%d/%d\n",output_rational.num,output_rational.den);
printf("===== Parse Time =====\n");
int64_t output_timeval;
strcpy(input_str,"00:01:01");
av_parse_time(&output_timeval,input_str,1);
printf("microseconds:%lld\n",output_timeval);
printf("=====-----\n");
}
```

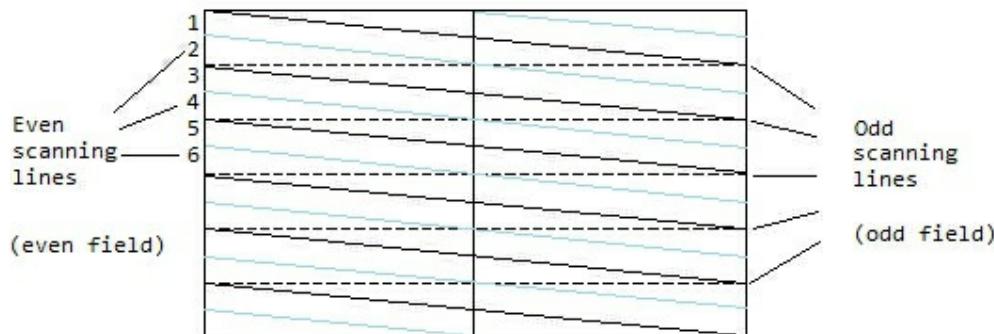
2. VIDEO AND IMAGE BASICS

The image signal is the transformation of any image into electromagnetic energy for transmission or storage. This signal is commonly called video. Video signal, multiple produced in different formats. The image on the screen is obtained by scanning an electrical signal on the horizontal and vertical axis. Frame is created by scanning the screen once time from top to bottom.

A frame contains lines of spatial information of a video signal. For progressive video, these lines contain samples starting from one time instant and continuing through successive lines to the bottom of the frame.



For interlaced video, a frame consists of two fields, a top field and a bottom field. One of these fields will commence one field later than the other.



One of the NTSC and PAL video formats widely used in vision systems. Interlaced scanning is used in both formats.

RAW FORMAT

The RAW format is also called digital negative.
The raw image recorded in the sensor of the camera
can also be called raw.

All raw formats are defined in “libavutil \ pixfmt.h”
under “AVPixelFormat” enumeration.

AV_PIX_FMT_YUV420P	AV_PIX_FMT_RGB48BE	AV_PIX_I
AV_PIX_FMT_RGB24	AV_PIX_FMT_RGB48LE	AV_PIX_I
AV_PIX_FMT_YUYV422	AV_PIX_FMT_RGB565BE	AV_PIX_I
AV_PIX_FMT_BGR24	AV_PIX_FMT_RGB565LE	AV_PIX_I
AV_PIX_FMT_YUV422P	AV_PIX_FMT_RGB555BE	AV_PIX_I
AV_PIX_FMT_YUV444P	AV_PIX_FMT_RGB555LE	AV_PIX_I
AV_PIX_FMT_YUV410P	AV_PIX_FMT_BGR565BE	AV_PIX_I
AV_PIX_FMT_YUV411P	AV_PIX_FMT_BGR565LE	AV_PIX_I
AV_PIX_FMT_GRAY8	AV_PIX_FMT_BGR555BE	AV_PIX_I
AV_PIX_FMT_MONOWHITE	AV_PIX_FMT_BGR555LE	AV_PIX_I
AV_PIX_FMT_MONOBLACK	AV_PIX_FMT_YUV420P16LE	AV_PIX_I
AV_PIX_FMT_PAL8	AV_PIX_FMT_YUV420P16BE	AV_PIX_I
AV_PIX_FMT_YUVJ420P	AV_PIX_FMT_YUV422P16LE	AV_PIX_I
AV_PIX_FMT_YUVJ422P	AV_PIX_FMT_YUV422P16BE	AV_PIX_I
AV_PIX_FMT_YUVJ444P	AV_PIX_FMT_YUV444P16LE	AV_PIX_I
AV_PIX_FMT_YVYU422	AV_PIX_FMT_YUV444P16BE	AV_PIX_I
AV_PIX_FMT_RGB0	AV_PIX_FMT_RGB444LE	AV_PIX_I
AV_PIX_FMT_XVMC	AV_PIX_FMT_RGB444BE	AV_PIX_I
AV_PIX_FMT_UYVY422	AV_PIX_FMT_BGR444LE	AV_PIX_I
AV_PIX_FMT_UYYVYY411	AV_PIX_FMT_BGR444BE	AV_PIX_I

AV_PIX_FMT_BGR8	AV_PIX_FMT_YA8	AV_PIX_I
AV_PIX_FMT_BGR4	AV_PIX_FMT_Y400A	AV_PIX_I
AV_PIX_FMT_BGR4_BYTE	AV_PIX_FMT_GRAY8A	AV_PIX_I
AV_PIX_FMT_RGB8	AV_PIX_FMT_BGR48BE	AV_PIX_I
AV_PIX_FMT_RGB4	AV_PIX_FMT_BGR48LE	AV_PIX_I
AV_PIX_FMT_RGB4_BYTE	AV_PIX_FMT_YUV420P9BE	AV_PIX_I
AV_PIX_FMT_NV12	AV_PIX_FMT_YUV420P9LE	AV_PIX_I
AV_PIX_FMT_ARGB	AV_PIX_FMT_YUV420P10BE	AV_PIX_I
AV_PIX_FMT_RGBA	AV_PIX_FMT_YUV420P10LE	AV_PIX_I
AV_PIX_FMT_ABGR	AV_PIX_FMT_YUV422P10BE	AV_PIX_I
AV_PIX_FMT_BGRA	AV_PIX_FMT_YUV422P10LE	AV_PIX_I
AV_PIX_FMT_GRAY16BE	AV_PIX_FMT_YUV444P9BE	AV_PIX_I
AV_PIX_FMT_GRAY16LE	AV_PIX_FMT_YUV444P9LE	AV_PIX_I
AV_PIX_FMT_YUV440P	AV_PIX_FMT_YUV444P10BE	AV_PIX_I
AV_PIX_FMT_YUVJ440P	AV_PIX_FMT_YUV444P10LE	AV_PIX_I
AV_PIX_FMT_YUVA420P	AV_PIX_FMT_YUV422P9BE	AV_PIX_I

Raw video formats are verified with FOURCC identifier.

FOURCC

FOURCC is short for "four character code" - an identifier for a video codec, compression format, color or pixel format used in media files.

A character in this context is a 1 byte/8 bit value, thus a FOURCC always takes up exactly 32 bits/4 bytes in a file. Another way to write FOURCC is 4CC (4 as in "four" character code).

The four characters in a FOURCC is generally limited to be within the human readable characters in the ASCII table. Because of this it is easy to convey and communicate what the FOURCCs are within a media file.

AVI files is the most widespread, or the first widely used media file format, to use FOURCC identifiers for the codecs used to compress the various video/audio streams within the files.

FOURCC IDENTIFIER

A FOURCC identifier allows you to determine which FOURCCs and audio tags are used within a media file. Knowing this allows you to identify precisely which video codecs and audio codecs that you will need in order to be able to playback the media file. In this section we list, and have available as direct downloads, several free applications that will inspect your media file and tell you what you need to know.

YUV PIXEL FORMATS

YUV formats fall into two distinct groups, the packed formats where Y, U (Cb) and V (Cr) samples are packed together into macropixels which are stored in a single array, and the planar formats where each component is stored as a separate array, the final image being a fusing of the three separate planes.

In the diagrams below, the numerical suffix attached to each Y, U or V sample indicates the sampling position across the image line, so, for example, V0 indicates the leftmost V sample and Y_n indicates the Y sample at the (n+1)th pixel from the left. Subsampling intervals in the horizontal and vertical directions may merit some explanation. The horizontal subsampling interval describes how frequently across a line a sample of that component is taken while the vertical interval describes on which lines samples are taken. For example, UYVYY format has a horizontal subsampling period of 2 for both the U and V components indicating that U and V samples are taken for every second pixel across a line. Their

vertical subsampling period is 1 indicating that U and V samples are taken on each line of the image. For YVU9, though, the vertical subsampling interval is 4. This indicates that U and V samples are only taken on every fourth line of the original image. Since the horizontal sampling period is also 4, a single U and a single V sample are taken for each square block of 16 image pixels. Also, if you are interested in YCrCb to RGB conversion, you may find this table helpful.

PACKED YUV FORMATS

Label	FOURCC in Hex	Bits per pixel	Description
AYUV	0x56555941	32	Combined YUV and alpha
CLJR	0x524A4C43	8	Cirrus Logic format with 4 pixels packed into a u_int32. A form of YUV 4:1:1 with less than 8 bits per Y, U and V sample.
CYUV	0x76757963	16	Essentially a copy of UYVY except that the sense of the height is reversed - the image is upside down with respect to the UYVY version.
GREY (Y800)	0x59455247	8	Apparently a duplicate of Y800 (and also, presumably, "Y8")
HDYC (UYVY)	0x43594448	16	YUV 4:2:2 (Y sample at every pixel, U and V sampled at every second pixel horizontally on each line). A macropixel contains 2 pixels in 1 u_int32. This is a suplicate of UYVY except that the color components use the BT709 color space (as used in HD video).
IRAW	0x57615349	?	Intel uncompressed YUV. I have no information on this format - can you help?

IUYV (UYVY)	0x56595549	16	Interlaced version of UYVY (line order 0, 2, 4,...,1, 3, 5....) registered by Silviu Brinzei of LEAD Technologies.
IY41	0x31555949	12	Interlaced version of Y41P (line order 0, 2, 4,...,1, 3, 5....) registered by Silviu Brinzei of LEAD Technologies.
IYU1	0x31555949	12	12 bit format used in mode 2 of the IEEE 1394 Digital Camera 1.04 spec. This is equivalent to Y411
IYU2	0x32555949	24	24 bit format used in mode 0 of the IEEE 1394 Digital Camera 1.04 spec
UYNV (UYVY)	0x564E5955	16	A direct copy of UYVY registered by NVidia to work around problems in some old codecs which did not like hardware which offered more than 2 UYVY surfaces.
UYVP	0x50565955	24?	YCbCr 4:2:2 extended precision 10-bits per component in U0Y0V0Y1 order. Registered by Rich Ehlers of Evans & Sutherland.
UYVY	0x59565955	16	YUV 4:2:2 (Y sample at every pixel, U and V sampled at every second pixel horizontally on each line). A macropixel contains 2 pixels in 1 u_int32.
V210	0x30313256	32	10-bit 4:2:2 YCrCb equivalent to the Quicktime format of the same name.
V422	0x32323456	16	I am told that this is an upside down version of UYVY.

V655	0x35353656	16?	16 bit YUV 4:2:2 format registered by Vitec Multimedia. I have no information on the component ordering or packing.
VYUY	0x59555956	?	ATI Packed YUV Data (format unknown)
Y16	0x20363159	16	16-bit uncompressed greyscale image.
Y211	0x31313259	8	Packed YUV format with Y sampled at every second pixel across each line and U and V sampled at every fourth pixel.
Y411	0x31313459	12	YUV 4:1:1 with a packed, 6 byte/4 pixel macroblock structure.
Y41P	0x50313459	12	YUV 4:1:1 (Y sample at every pixel, U and V sampled at every fourth pixel horizontally on each line). A macropixel contains 8 pixels in 3 u_int32s.
Y41T	0x54313459	12	Format as for Y41P but the lsb of each Y component is used to signal pixel transparency .
Y422 (UYVY)	0x32323459	16	Direct copy of UYVY as used by ADS Technologies Pyro WebCam firewire camera.
Y42T	0x54323459	16	Format as for UYVY but the lsb of each Y component is used to signal pixel transparency .
Y8 (Y800)	0x20203859	8	Duplicate of Y800 as far as I can see.
			Simple, single Y plane for

Y800	0x30303859	8	monochrome images.
YUNV (YUY2)	0x564E5559	16	A direct copy of YUY2 registered by NVidia to work around problems in some old codecs which did not like hardware which offered more than 2 YUY2 surfaces.
YUVP	0x50565559	24?	YCbCr 4:2:2 extended precision 10-bits per component in Y0U0Y1V0 order. Registered by Rich Ehlers of Evans & Sutherland.
YUY2	0x32595559	16	YUV 4:2:2 as for UYVY but with different component ordering within the u_int32 macropixel.
YUYV (YUY2)	0x56595559	16	Duplicate of YUY2
YVYU	0x55595659	16	YUV 4:2:2 as for UYVY but with different component ordering within the u_int32 macropixel.

PLANAR YUV FORMATS

Label	FOURCC in Hex	Bits per pixel	Description
CLPL	0x4C504C43	12	Format similar to YV12 but including a level of indirection.
CXY1	0x31595843	12	Planar YUV 4:1:1 format registered by Conexant.
CXY2	0x32595842	16	Planar YUV 4:2:2 format registered by Conexant.
I420	0x30323449	12	8 bit Y plane followed by 8 bit 2x2 subsampled U and V planes.
IF09	0x39304649	9.5	As YVU9 but an additional 4x4 subsampled plane is appended containing delta information relative to the last frame. (Bpp is reported as 9)
IMC1	0x31434D49	12	As YV12 except the U and V planes each have the same stride as the Y plane
IMC2	0x32434D49	12	Similar to IMC1 except that the U and V lines are interleaved at half stride boundaries
IMC3 (IMC1)	0x33434D49	12	As IMC1 except that U and V are swapped
IMC4 (IMC2)	0x34434D49	12	As IMC2 except that U and V are swapped

IYUV (I420)	0x56555949	12	Duplicate FOURCC, identical to I420.
NV12	0x3231564E	12	8-bit Y plane followed by an interleaved U/V plane with 2x2 subsampling
NV21	0x3132564E	12	As NV12 with U and V reversed in the interleaved plane
Y41B	0x42313459	12?	Weitek format listed as "YUV 4:1:1 planar". I have no other information on this format.
Y42B	0x42323459	16?	Weitek format listed as "YUV 4:2:2 planar". I have no other information on this format.
Y8 (Y800)	0x20203859	8	Duplicate of Y800 as far as I can see.
Y800	0x30303859	8	Simple, single Y plane for monochrome images.
YUV9	0x39565559	9?	Registered by Intel., this is the format used internally by Indeo video code
YV12	0x32315659	12	8 bit Y plane followed by 8 bit 2x2 subsampled V and U planes.
YV16	0x36315659	16	8 bit Y plane followed by 8 bit 2x1 subsampled V and U planes.
YVU9	0x39555659	9	8 bit Y plane followed by 8 bit 4x4 subsampled V and U planes. Registered by Intel.

RGB PIXEL FORMATS

These formats are defined below

Label	FOURCC in Hex	Bits per pixel	Description
BI_BITFIELDS	0x00000003	16,24,32	Raw RGB with arbitrary sample packing within a pixel. Packing and precision of R, G and B components is determined by bit masks for each.
BI_RGB	0x00000000	1,4,8,16,24,32	Basic Windows bitmap format. 1, 4 and 8 bpp versions are palettised. 16, 24 and 32bpp contain raw RGB samples.
BI_RLE4	0x00000002	4	Run length encoded 4bpp RGB image.
BI_RLE8	0x00000001	8	Run length encoded 8bpp RGB image.
raw	0x32776173	?	Apparently "raw, uncompressed RGB bitmaps" but I have no idea how many bits per pixel (uses bpp field in header?)
RGB (BI_RGB)	0x32424752	1,4,8,16,24,32	Alias for BI_RGB
RGBA	0x41424752	16,32	Raw RGB with alpha. Sample precision and packing is arbitrary and determined using bit masks for each component, as for BI_BITFIELDS.

RGBT	0x54424752	16,32	Raw RGB with a transparency field. Layout is as for BI_RGB at 16 and 32 bits per pixel but the msb in each pixel indicates whether the pixel is transparent or not.
RLE (BI_RLE4)	0x34454C52	4	Alias for BI_RLE4
RLE8 (BI_RLE8)	0x38454C52	8	Alias for BI_RLE8

BAYER DATA FORMATS

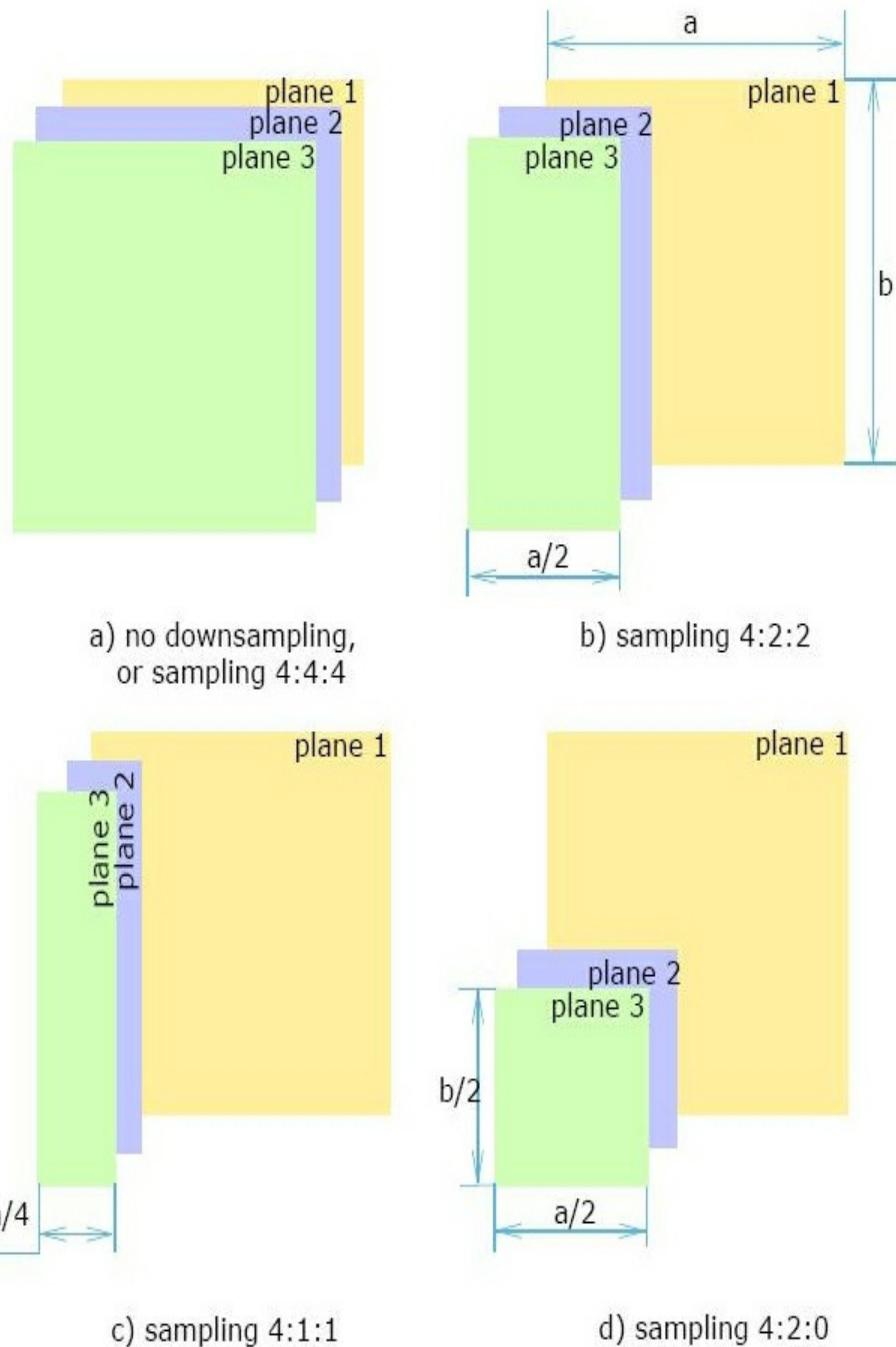
Recently, a collection of "Bayer" data formats have been registered. These encode images with only a single color sample at each pixel position and are frequently used in acquisition of digital images where the image is captured through a color filter array. It is not clear to me how these formats define the layout of the filter array used but it may be fair to assume that they relate to a commonly used Bayer pattern originally developed by Kodak which has builds an image out of 2x2 blocks containing green samples in the top left and bottom right positions, red in the top right and blue in the bottom left.

Label	FOURCC in Hex	Bits per pixel	Description
BA81	0x	8	Raw Bayer data with 8 bits per sample
BYR1	0x	8	Raw Bayer data with 8 bits per sample. May be a duplicate of BA81?
BYR2	0x	16	Raw Bayer data with 12 bit precision samples stored in 16 bit words.

CREATE SAMPLE VIDEO

Data storage for an image can be pixel-oriented(packet) or planar-oriented (planar). For images in pixel order, all channel values for each pixel are clustered and stored consecutively. Their layout depends on the color model and downsampling scheme.

Planar formats; Planar (or sometimes "triplanar") formats use separate matrices for each of the 3 color components. In other words, there is one table of luminance pixel values, and two separate tables for the chrominance components.



In package format; pixel data is sorted in a single array. FFmpeg supports planar format. AVFrame has three data blocks for “Y,Cb, Cr”. In sample this macroblocks are filled with organized

piksel data. AVFrame raw image format is set as “AV_PIX_FMT_YUV420P” and resolution 1280x720.

```
#ifndef __STDC_CONSTANT_MACROS
#define __STDC_CONSTANT_MACROS
#endif
#ifndef D__STDC_FORMAT_MACROS
#define __STDC_FORMAT_MACROS
#endif
extern "C" {
    #include "libavcodec/avcodec.h"
    #include "libavformat/avformat.h"
    #include "libswscale/swscale.h"
    #include "libavutil/channel_layout.h"
    #include "libavutil/common.h"
    #include "libavutil/imgutils.h"
    #include "libavutil/mathematics.h"
    #include "libavutil/samplefmt.h"
    #include "libavfilter/buffersink.h"
    #include "libavfilter/buffersrc.h"
    #include "libavutil/opt.h"
    #include "libavutil/pixdesc.h"
    #include "libavutil/time.h"
    #include "libavutil/timestamp.h"
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <math.h>
};

int main()
{
    AVFrame *frame;
    int i, x, y, ret;
    FILE *f;
    const char *filename;
    filename ="output_planar_yuv420.yuv";
```

```

frame = av_frame_alloc();
if (!frame) { fprintf(stderr, "Could not allocate video frame\n"); exit(1);
}
frame->format = AV_PIX_FMT_YUV420P;
frame->width = 1280;
frame->height = 720;
ret = av_frame_get_buffer(frame, 32);
if (ret < 0) {fprintf(stderr, "Could not allocate the video frame data\n");
exit(1);
f = fopen(filename, "a+");
for (i = 0; i < 1; i++) {
fflush(stdout);
ret = av_frame_make_writable(frame);
if (ret < 0) exit(1);
/* prepare a dummy image */
/* Y */
for (y = 0; y < frame->height; y++) {
for (x = 0; x < frame->width; x++) {
frame->data[0][y * frame->linesize[0] + x] = x + y + i * 3; }
}
/* Cb and Cr */
for (y = 0; y < frame->height/2; y++) {
for (x = 0; x < frame->width/2; x++) {
frame->data[1][y * frame->linesize[1] + x] = 128 + y + i * 2;
frame->data[2][y * frame->linesize[2] + x] = 64 + x + i * 5;
}
}
for (int y = 0; y < frame->height; y++) //plane 0: same width and
height
fwrite(frame->data[0] + y * frame->linesize[0], 1, frame-
>width, f);

for (int y = 0; y < frame->height/4; y++) //plane1: Same width and
quarter hgt.
fwrite(frame->data[1] + y * frame->linesize[1], 1, frame-
>width, f);

for (int y = 0; y < frame->height/4; y++) //plane2: Same width and
quarter hgt.
fwrite(frame->data[2] + y * frame->linesize[2], 1, frame-

```

```
>width, f);  
    frame->pts = i;  
}  
}
```

CONVERT PACKET FORMAT TO PLANAR FORMAT

AVFrame structure describes decoded (raw) audio or video data. AVFrame must be allocated using `av_frame_alloc()`. Note that this only allocates the AVFrame itself, the buffers for the data must be managed through other means (see below). AVFrame must be freed with `av_frame_free()`. AVFrame is typically allocated once and then reused multiple times to hold different data (e.g. a single AVFrame to hold frames received from a decoder). In such a case, `av_frame_unref()` will free any references held by the frame and reset it to its original clean state before it is reused again.

In Example source data “AV_PIX_FMT_UYVY422” is packet format and destination data “AV_PIX_FMT_YUV422P” is planar format. Firstly we create AVFrame structure with dimensions and pixel format. Than we call “`av_register_all()`” to Initialize libavformat and register

all the muxers, demuxers and protocols. We need to create context for Image converting as SwsContext structure. We allocate memory for SwsContext with “sws_alloc_context” .We use “av_image_alloc” to allocate source and destination images. We set parameters with different ways in ffmpeg. These ones

```
int av_opt_set      (void *obj, const char *name, const
char *val, int search_flags);
int av_opt_set_int   (void *obj, const char *name,
int64_t   val, int search_flags);
int av_opt_set_double (void *obj, const char *name,
double    val, int search_flags);
int av_opt_set_q     (void *obj, const char *name,
AVRational val, int search_flags);
int av_opt_set_bin    (void *obj, const char *name,
const uint8_t *val, int size, int search_flags);
int av_opt_set_image_size(void *obj, const char
*name, int w, int h, int search_flags);
int av_opt_set_pixel_fmt (void *obj, const char *name,
enum AVPixelFormat fmt, int search_flags);
int av_opt_set_sample_fmt(void *obj, const char
*name, enum AVSampleFormat fmt, int search_flags);
int av_opt_set_video_rate(void *obj, const char *name,
AVRational val, int search_flags);
int av_opt_set_channel_layout(void *obj, const char
```

```
*name, int64_t ch_layout, int search_flags);
```

we will use “av_opt_set_int” method for set integer parameter.

“sws_flags” first parameter is flag for Interpolation algorithm. The bicubic algorithm is frequently used for scaling images and video for display. The other algorithms are

SWS_FAST_BILINEAR

SWS_BILINEAR

SWS_BICUBIC

“srcw” source image width

“srch” source image height

“src_format” source image pixel format

“src_range” source image range

“dstw” destination image width

“dsth” destination image height

“dst_format” destination image pixel format

“dst_range” destination image range

After set parameters to initialize “img_convert_ctx” we use “sws_init_context” method.

```
struct SwsContext *img_convert_ctx;  
AVPixelFormat src_pixfmt;  
AVPixelFormat dst_pixfmt;  
uint8_t *src_data[4];  
int src_linesize[4];
```

```

    uint8_t *dst_data[4];
    int dst_linesize[4];
    int p_in_w;
    int p_in_h;

void init_colorspace()
{
    //YUV420p I420
    src_pixfmt=AV_PIX_FMT_UYVY422;
    dst_pixfmt=AV_PIX_FMT_YUV422P;
    int src_bpp=av_get_bits_per_pixel(av_pix_fmt_desc_get(src_pixfmt));
    int dst_bpp=av_get_bits_per_pixel(av_pix_fmt_desc_get(dst_pixfmt));

    int ret = av_image_alloc(src_data, src_linesize,p_in_w, p_in_h, src_pixfn
        if (ret< 0) {
            printf( "Could not allocate source image\n");
            return;
        }
    ret = av_image_alloc(dst_data, dst_linesize,p_in_w, p_in_h, dst_pixfmt, );
    if (ret< 0) {
        printf( "Could not allocate destination image\n");
        return;
    }

//Show AVOption
av_opt_show2(img_convert_ctx,stdout,AV_OPT_FLAG_VIDEO_PARAM);

//Set Value
av_opt_set_int(img_convert_ctx,"sws_flags",SWS_BICUBIC|SWS_PRI);
av_opt_set_int(img_convert_ctx,"srcw",p_in_w,0);
av_opt_set_int(img_convert_ctx,"srch",p_in_h,0);
av_opt_set_int(img_convert_ctx,"src_format",src_pixfmt,0);
av_opt_set_int(img_convert_ctx,"src_range",1,0);
av_opt_set_int(img_convert_ctx,"dstw",p_in_w,0);
av_opt_set_int(img_convert_ctx,"dsth",p_in_h,0);
av_opt_set_int(img_convert_ctx,"dst_format",dst_pixfmt,0);
av_opt_set_int(img_convert_ctx,"dst_range",1,0);
sws_init_context(img_convert_ctx,NULL,NULL);
}

int main()
{
    AVFrame* frame;

```

```
frame = av_frame_alloc();
if (!frame) {
    printf("Could not allocate video frame\n");
    return -1; }

p_in_w=1920;
p_in_h=1080;
frame->format = AV_PIX_FMT_YUV422P;
frame->width = p_in_w;
frame->height = p_in_h;
uint8_t *src_data[4];
av_register_all();
img_convert_ctx =sws_alloc_context();
init_colorspace();
uint8_t* packetData;

packetData=new uint8_t[frame->width * frame->height * 2];
src_data[0]=new uint8_t[frame->width * frame->height * 2];

for(int i=0;i<frame->width * frame->height * 2;i++)
    packetData[i]=255;

//AV_PIX_FMT_UYVY422
memcpy(src_data[0], packetData,frame->width * frame->height * 2); //P
sws_scale(img_convert_ctx, src_data, src_linesize, 0, p_in_h,
dst_data, dst_linesize);

frame->data[0]=dst_data[0];
frame->data[1]=dst_data[1];
frame->data[2]=dst_data[2];
return 0;
}
```

IMAGE SCALE

The scale filter forces the output display aspect ratio to be the same of the input, by changing the output sample aspect ratio. Input image format can be different output image format. Image scaler supports different scaling algorithms, these are

fast_bilinear : Select fast bilinear scaling algorithm.

bilinear: Select bilinear scaling algorithm.

bicubic: Select bicubic scaling algorithm.

experimental: Select experimental scaling algorithm.

neighbor: Select nearest neighbor rescaling algorithm.

area: Select averaging area rescaling algorithm.

bicublin: Select bicubic scaling algorithm for the luma component, bilinear for chroma components.

gauss: Select Gaussian rescaling algorithm.

sinc: Select sinc rescaling algorithm.

lanczos: Select Lanczos rescaling algorithm. The default width (alpha) is 3

spline: Select natural bicubic spline rescaling algorithm.

Scaling algorithms can be set with “sws_flags” flags.

In sample input resolution 1920x1080 and output resolution is 1280x720.

```
void Scale()
{
    p_in_w=1920;
    p_in_h=1080;
    p_out_w =1280;
    p_out_h =720;

    src_pixfmt=AV_PIX_FMT_UYVY422;
    dst_pixfmt=AV_PIX_FMT_YUV422P;
    int src_bpp=av_get_bits_per_pixel(av_pix_fmt_desc_get(src_pixfmt));
    int dst_bpp=av_get_bits_per_pixel(av_pix_fmt_desc_get(dst_pixfmt));
    int ret = av_image_alloc(src_data, src_linesize,p_in_w, p_in_h, src_pixfn
    if (ret< 0) {printf( "Could not allocate source image\n");return;}
    ret = av_image_alloc(dst_data, dst_linesize,p_in_w, p_in_h, dst_pixfm
    if (ret< 0) { printf( "Could not allocate destination image\n");return;}
    av_opt_show2(img_convert_ctx,stdout,AV_OPT_FLAG_VIDEO_PA]
    av_opt_set_int(img_convert_ctx,"sws_flags",SWS_BICUBIC|SWS_PL
    av_opt_set_int(img_convert_ctx,"srcw",p_in_w,0);
    av_opt_set_int(img_convert_ctx,"srch",p_in_h,0);
    av_opt_set_int(img_convert_ctx,"src_format",src_pixfmt,0);
    av_opt_set_int(img_convert_ctx,"src_range",1,0);
    av_opt_set_int(img_convert_ctx,"dstw",p_out_w,0);
    av_opt_set_int(img_convert_ctx,"dsth",p_out_h,0);
    av_opt_set_int(img_convert_ctx,"dst_format",dst_pixfmt,0);
    av_opt_set_int(img_convert_ctx,"dst_range",1,0);
    sws_init_context(img_convert_ctx,NULL,NULL);
}
```

Converting multiple image format and resolution sample; This sample uses libswscale to scale and convert pixels.

```
/**  
* This software uses libswscale to scale / convert pixels.  
* It convert YUV420P format to RGB24 format,  
* and changes resolution from 480x272 to 1280x720.  
* It's the simplest tutorial about libswscale.  
*/  
  
#include <stdio.h>  
  
#define __STDC_CONSTANT_MACROS  
  
#ifdef __WIN32  
//Windows  
extern "C"  
{  
#include "libswscale/swscale.h"  
#include "libavutil/opt.h"  
#include "libavutil/imgutils.h"  
};  
#else  
//Linux...  
#ifdef __cplusplus  
extern "C"  
{  
#endif  
#include <libswscale/swscale.h>  
#include <libavutil/opt.h>  
#include <libavutil/imgutils.h>  
#ifdef __cplusplus  
};  
#endif  
#endif  
  
int main(int argc, char* argv[])  
{  
    //Parameters  
    FILE *src_file =fopen("sintel_480x272_yuv420p.yuv", "rb");  
    const int src_w=480,src_h=272;
```

```

AVPixelFormat src_pixfmt=AV_PIX_FMT_YUV420P;

int src_bpp=av_get_bits_per_pixel(av_pix_fmt_desc_get(src_pixfmt));

FILE *dst_file = fopen("sintel_1280x720_rgb24.rgb", "wb");
const int dst_w=1280,dst_h=720;
AVPixelFormat dst_pixfmt=AV_PIX_FMT_RGB24;
int dst_bpp=av_get_bits_per_pixel(av_pix_fmt_desc_get(dst_pixfmt));

//Structures
uint8_t *src_data[4];
int src_linesize[4];

uint8_t *dst_data[4];
int dst_linesize[4];

int rescale_method=SWS_BICUBIC;
struct SwsContext *img_convert_ctx;
uint8_t *temp_buffer=(uint8_t *)malloc(src_w*src_h*src_bpp/8);
int frame_idx=0;
int ret=0;

ret= av_image_alloc(src_data, src_linesize,src_w, src_h, src_pixfmt, 1)
if (ret< 0) {
printf( "Could not allocate source image\n");
return -1;
}
ret = av_image_alloc(dst_data, dst_linesize,dst_w, dst_h, dst_pixfmt, 1
if (ret< 0) {
printf( "Could not allocate destination image\n");
return -1;
}
//-----
//Init Method 1
img_convert_ctx =sws_alloc_context();
//Show AVOption
av_opt_show2(img_convert_ctx,stdout,AV_OPT_FLAG_VIDEO_PARAM);
//Set Value
av_opt_set_int(img_convert_ctx,"sws_flags",SWS_BICUBIC|SWS_PAL);
av_opt_set_int(img_convert_ctx,"srcw",src_w,0);

```

```

av_opt_set_int(img_convert_ctx,"srch",src_h,0);
av_opt_set_int(img_convert_ctx,"src_format",src_pixfmt,0);
//'0' for MPEG (Y:0-235);'1' for JPEG (Y:0-255)
av_opt_set_int(img_convert_ctx,"src_range",1,0);
av_opt_set_int(img_convert_ctx,"dstw",dst_w,0);
av_opt_set_int(img_convert_ctx,"dsth",dst_h,0);
av_opt_set_int(img_convert_ctx,"dst_format",dst_pixfmt,0);
av_opt_set_int(img_convert_ctx,"dst_range",1,0);
sws_init_context(img_convert_ctx,NULL,NULL);

//Init Method 2
//img_convert_ctx = sws_getContext(src_w, src_h,src_pixfmt, dst_w, c
//      rescale_method, NULL, NULL, NULL);
//-----
/*
//Colorspace
ret=sws_setColorspaceDetails(img_convert_ctx,sws_getCoefficients(SWS_C
    sws_getCoefficients(SWS_CS_ITU709),0,
    0, 1 << 16, 1 << 16);
if (ret== -1) {
printf( "Colorspace not support.\n");
return -1;
}
*/
while(1)
{
if (fread(temp_buffer, 1, src_w*src_h*src_bpp/8, src_file) != src_w*sr
break;
}
switch(src_pixfmt){
case AV_PIX_FMT_GRAY8:{
memcpy(src_data[0],temp_buffer,src_w*src_h);
break;
}
case AV_PIX_FMT_YUV420P:{

memcpy(src_data[0],temp_buffer,src_w*src_h); //Y
memcpy(src_data[1],temp_buffer+src_w*src_h,src_w*src_h/4); //U
}
}
}

```

```

break;
}
case AV_PIX_FMT_YUV422P:{
memcpy(src_data[0],temp_buffer,src_w*src_h); //Y
memcpy(src_data[1],temp_buffer+src_w*src_h,src_w*src_h/2); //U
break;
}
case AV_PIX_FMT_YUV444P:{
memcpy(src_data[0],temp_buffer,src_w*src_h); //Y
memcpy(src_data[1],temp_buffer+src_w*src_h,src_w*src_h); //U
memcpy(src_data[2],temp_buffer+src_w*src_h*2,src_w*src_h); //V
break;
}
case AV_PIX_FMT_YUYV422:{
memcpy(src_data[0],temp_buffer,src_w*src_h*2); //Packed
break;
}
case AV_PIX_FMT_RGB24:{
memcpy(src_data[0],temp_buffer,src_w*src_h*3); //Packed
break;
}
default:{
printf("Not Support Input Pixel Format.\n");
break;
}
}
sws_scale(img_convert_ctx, src_data, src_linesize, 0, src_h, dst_data, dst_linesize);
printf("Finish process frame %5d\n",frame_idx);
frame_idx++;

switch(dst_pixfmt){
case AV_PIX_FMT_GRAY8:{
fwrite(dst_data[0],1,dst_w*dst_h,dst_file);
break;
}
case AV_PIX_FMT_YUV420P:{
fwrite(dst_data[0],1,dst_w*dst_h,dst_file); //Y
}
}
}

```

```
fwrite(dst_data[1],1,dst_w*dst_h/4,dst_file);           //U
fwrite(dst_data[2],1,dst_w*dst_h/4,dst_file);           //V
break;
}
case AV_PIX_FMT_YUV422P:{
fwrite(dst_data[0],1,dst_w*dst_h,dst_file); //Y
fwrite(dst_data[1],1,dst_w*dst_h/2,dst_file); //U
fwrite(dst_data[2],1,dst_w*dst_h/2,dst_file); //V
break;
}
case AV_PIX_FMT_YUV444P:{
fwrite(dst_data[0],1,dst_w*dst_h,dst_file);           //Y
fwrite(dst_data[1],1,dst_w*dst_h,dst_file);           //U
fwrite(dst_data[2],1,dst_w*dst_h,dst_file);           //V
break;
}
case AV_PIX_FMT_YUYV422:{
fwrite(dst_data[0],1,dst_w*dst_h*2,dst_file); //Packed
break;
}
case AV_PIX_FMT_RGB24:{
fwrite(dst_data[0],1,dst_w*dst_h*3,dst_file); //Packed
break;
}
default:{
printf("Not Support Output Pixel Format.\n");
break;
}
}
}

sws_freeContext(img_convert_ctx);

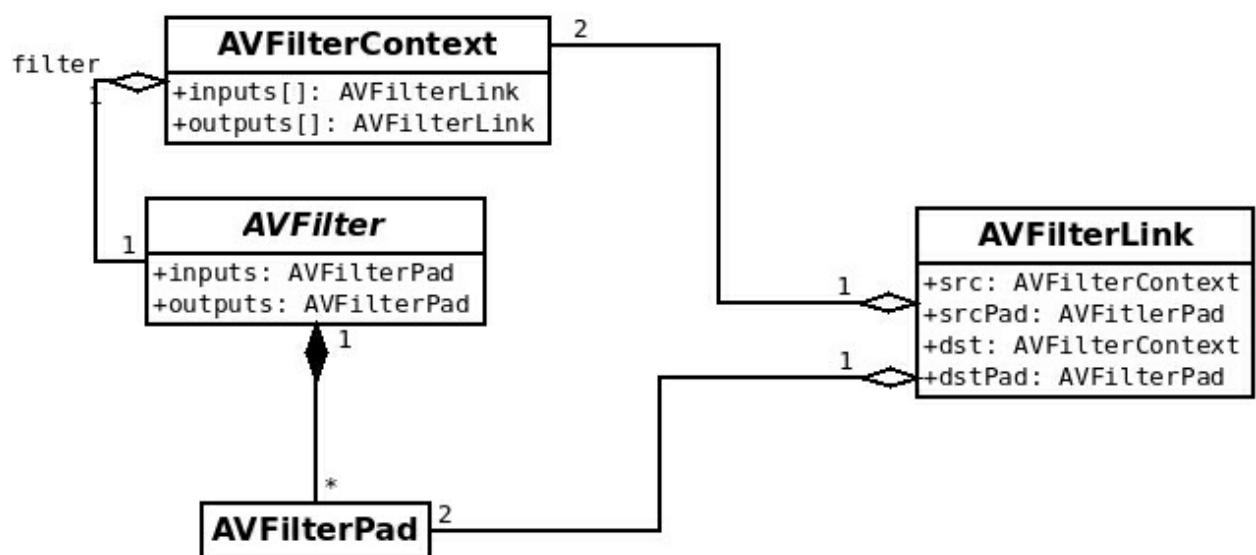
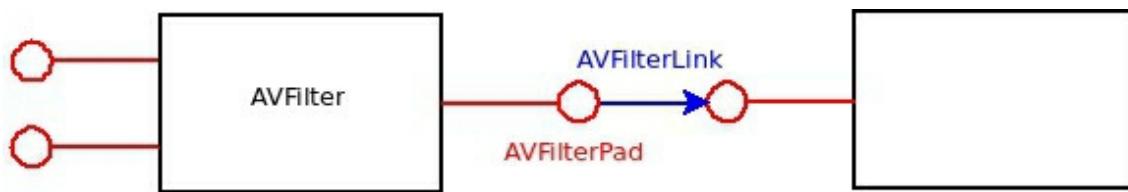
free(temp_buffer);
fclose(dst_file);
av_freep(&src_data[0]);
av_freep(&dst_data[0]);
```

```
    return 0;  
}
```

FILTER CHAIN

FFMPEG supports a filter chain composed of multiple filter structures to implement subsequent processing of video frames and audio sample data, such as image scaling, image enhancement, and sound resampling.

Data structure:



The four main data structures used by

FFMPEG to construct the filter chain are shown in the figure above. Among them, AVFilter is an abstraction of a filter, it has a number of AVFilterPad, respectively realize the input and output of filter data. The AVFilterLink represents the connection between various filters. The implementation of this connection is based on AVFilterPad. Each connection has an AVFilterPad for input and an AVFilterPad for output, as well as corresponding source and destination filters. In addition, there are two data structures AVFilterBuffer and AVFilterBufferRef to represent the input and output data of the filter, where AVFilterBufferRef is a reference to AVFilterBuffer, the purpose of this design is to avoid unnecessary data copying. Through `avfilter_get_buffer_ref_from_frame`, you can get a reference to the AVFrame structure, so that the filter can process the image raw data or sound sampling sequence from the AVFrame.

AVFILTER

All filters are described by an AVFilter structure. This structure gives information needed to initialize the filter, and information on the entry points into the filter code. This structure is declared in libavfilter/avfilter.h:

```
typedef struct
{
    char *name;      ///< filter name
    int priv_size;   ///< size of private data to allocate for the filter
    int (*init)(AVFilterContext *ctx, const char *args, void *opaque);
    void (*uninit)(AVFilterContext *ctx);
    int (*query_formats)(AVFilterContext *ctx);
    const AVFilterPad *inputs;
    const AVFilterPad *outputs
} AVFilter;
```

The query_formats function sets the in_formats member of connected **output** links, and the out_formats member of connected **input** links, described under AVFilterLink.

AVFILTERPAD

Let's take a quick look at the AVFilterPad structure, which is used to describe the inputs and outputs of the filter. This is also defined in libavfilter/avfilter.h:

```
typedef struct AVFilterPad
{
    char *name;
    int type;
    int min_perms;
    int rej_perms;

    void (*start_frame)(AVFilterLink *link, AVFilterPicRef *picref);
    AVFilterPicRef *(*get_video_buffer)(AVFilterLink *link, int perms);
    void (*end_frame)(AVFilterLink *link);
    void (*draw_slice)(AVFilterLink *link, int y, int height);
    int (*request_frame)(AVFilterLink *link);
    int (*config_props)(AVFilterLink *link);
} AVFilterPad;
```

Fields and their descriptions

Field	Description
Name	Name of the pad. No two inputs should have the same name, and no two outputs should have the same name.
Type	Only AV_PAD_VIDEO currently.

<code>config_props</code>	Handles configuration of the link connected to the pad
---------------------------	--

Fields only relevant to input pads are:

Field	Description
<code>min_perms</code>	Minimum permissions required to a picture received as input.
<code>rej_perms</code>	Permissions not accepted on pictures received as input.
<code>start_frame</code>	Called when a frame is about to be given as input.
<code>draw_slice</code>	Called when a slice of frame data has been given as input.
<code>end_frame</code>	Called when the input frame has been completely sent.
<code>get_video_buffer</code>	Called by the previous filter to request memory for a picture.

Fields only relevant to output pads are:

Field	Description
<code>request_frame</code>	Requests that the filter output a frame.

PICTURE BUFFERS

Reference Counting

All pictures in the filter system are reference counted. This means that there is a picture buffer with memory allocated for the image data, and various filters can own a reference to the buffer. When a reference is no longer needed, its owner frees the reference. When the last reference to a picture buffer is freed, the filter system automatically frees the picture buffer.

PERMISSIONS

The upshot of multiple filters having references to a single picture is that they will all want some level of access to the image data. It should be obvious that if one filter expects to be able to read the image data without it changing that no other filter should write to the image data. The permissions system handles this.

In most cases, when a filter prepares to output a frame, it will request a buffer from the filter to which it will be outputting. It specifies the minimum permissions it needs to the buffer, though it may be given a buffer with more permissions than the minimum it requested. When it wants to pass this buffer to another filter as output, it creates a new reference to the picture, possibly with a reduced set of permissions. This new reference will be owned by the filter receiving it. So, for example, for a filter which drops frames if they are similar to the last frame it output, it would want to keep its own reference to a picture after outputting it, and make sure that no other filter modified the buffer either. It would do this by requesting the permissions

AV_PERM_READ|AV_PERM_WRITE|AV_PERM_P for itself, and removing the AV_PERM_WRITE permission from any references it gave to other filters. The available permissions are:

Permission	Description
AV_PERM_READ	Can read the image data.
AV_PERM_WRITE	Can write to the image data.
AV_PERM_PRESERVE	Can assume that the image data will not be modified by other filters. This means that no other filters should have the AV_PERM_WRITE permission.
AV_PERM_REUSE	The filter may output the same buffer multiple times, but the image data may not be changed for the different outputs.
AV_PERM_REUSE2	The filter may output the same buffer multiple times, and may modify the image data between outputs.

FILTER LINKS

A filter's inputs and outputs are connected to those of another filter through the AVFilterLink structure:

```
typedef struct AVFilterLink
{
    AVFilterContext *src; //< source filter
    unsigned int srcpad; //< index of the output pad on the source filter
    AVFilterContext *dst; //< dest filter
    unsigned int dstpad; //< index of the input pad on the dest filter
    int w; //< agreed upon image width
    int h; //< agreed upon image height
    enum PixelFormat format; //< agreed upon image colorspace
    AVFilterFormats *in_formats; //< formats supported by source filter
    AVFilterFormats *out_formats; //< formats supported by dest. filter
    AVFilterPicRef *srcpic;
    AVFilterPicRef *cur_pic;
    AVFilterPicRef *outpic;
};
```

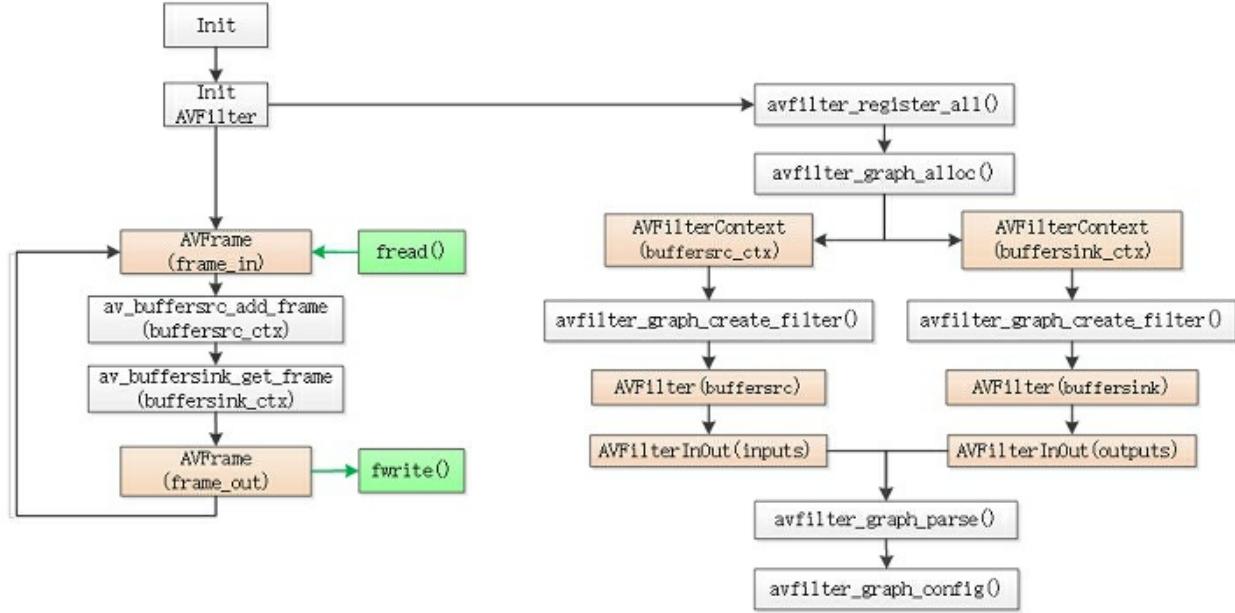
The src and dst members indicate the filters at the source and destination ends of the link, respectively. The srcpad indicates the index of the output pad on the source filter to which the link is connected. Likewise, the dstpad indicates the index of the input pad on the destination filter. The in_formats

member points to a list of formats supported by the source filter, while the `out_formats` member points to a list of formats supported by the destination filter. The `AVFilterFormats` structure used to store the lists is reference counted, and in fact tracks its references. The upshot is that if a filter provides pointers to the same list on multiple input/output links, it means that those links will be forced to use the same format as each other.

When two filters are connected, they need to agree upon the dimensions of the image data they'll be working with, and the format that data is in. Once this has been agreed upon, these parameters are stored in the link structure.

The `srcpic` member is used internally by the filter system, and should not be accessed directly. The `cur_pic` member is for the use of the destination filter. When a frame is currently being sent over the link (ie. starting from the call to `start_frame()` and ending with the call to `end_frame()`), this contains the reference to the frame which is owned by the destination filter.

See below filter chain flowchart:



USAGE OF AVFILTER

We can test many filters with the sample code.
In this example we will use SDL library to show output frames.

Conceptually SDL library is very simple to use:

- 1- Call `SDL_CreateTexture()` with the `SDL_TEXTUREACCESS_STREAMING` access type
- 2- Call `SDL_LockTexture()` to get raw access to the pixels
- 3- Do any pixel manipulation you want
- 4- Call `SDL_UnlockTexture()`
- 5- Use the texture in rendering normally.

We create “AVFrame” structure for decoded frame and output frame, “AVPacket” for decoded frame data.

```
AVPacket packet;  
AVFrame *pFrame;  
AVFrame *pFrame_out;
```

Initialize libavformat and register all the muxers, demuxers and protocols with “`av_register_all()`” function. Initialize the filter system and register all builtin filters with “`avfilter_register_all()`” function.

Than Open an input stream with “avformat_open_input()”, Read packets of a media file to get stream information with “avformat_find_stream_info()”, Find the "best" stream in the file with “av_find_best_stream()” and load pFormatCtx.

Set AVCodecContext with pFormatCtx’s video stream.

```
pCodecCtx = pFormatCtx->streams[video_stream_index]->codec;
```

Initialize the AVCodecContext to use the given AVCodec with “avcodec_open2()”.

Next step is initialization of filters with “init_filters()”.

Define filters and create input and outputs pins

```
AVFilter *buffersrc = avfilter_get_by_name("buffer");
AVFilter *buffersink = avfilter_get_by_name("ffbuffersink");
AVFilterInOut *outputs = avfilter_inout_alloc();
AVFilterInOut *inputs = avfilter_inout_alloc();
enum AVPixelFormat pix_fmts[] = { AV_PIX_FMT_YUV420P,
AV_PIX_FMT_NONE };
AVBufferSinkParams *buffersink_params;
```

Search filter with name and allocate with “avfilter_get_by_name()” and allocate filter pins with “avfilter_inout_alloc()”

Create a new filter instance in filter graph with “avfilter_graph_alloc()”. Create and add a filter instance into an existing graph with

“avfilter_graph_create_filter()” Set Endpoints for the filter graph

```
outputs->name      = av_strdup("in");
outputs->filter_ctx = buffersrc_ctx;
outputs->pad_idx    = 0;
outputs->next       = NULL;

inputs->name      = av_strdup("out");
inputs->filter_ctx = buffersink_ctx;
inputs->pad_idx    = 0;
inputs->next       = NULL;
```

Add a graph described by a string to a graph with “avfilter_graph_parse_ptr()”. It gets filter description as parameter than check validity and configure all the links and formats in the graph with “avfilter_graph_config()”.

Allocate input and output frame;

```
pFrame=av_frame_alloc();
pFrame_out=av_frame_alloc();
```

Push the decoded frame into the filtergraph with “av_buffersrc_add_frame()” and pull filtered pictures from the filtergraph with “av_buffersink_get_frame()”. In sample write output to file or show with SDL on window.

```
#include <stdio.h>
#define __STDC_CONSTANT_MACROS
```

```
#ifdef _WIN32
#define snprintf _snprintf
//Windows
extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libavfilter/avfiltergraph.h"
#include "libavfilter/buffersink.h"
#include "libavfilter/buffersrc.h"
#include "libavutil/avutil.h"
#include "libswscale/swscale.h"
#include "SDL/SDL.h"
};

#else
//Linux...
#endif __cplusplus
extern "C"
{
#endif
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavfilter/avfiltergraph.h>
#include <libavfilter/buffersink.h>
#include <libavfilter/buffersrc.h>
#include <libavutil/avutil.h>
#include <libswscale/swscale.h>
#include <SDL/SDL.h>
#endif __cplusplus
};

#endif
#endif
//Enable SDL?
#define ENABLE	SDL 1
//Output YUV data?
#define ENABLE_YUVFILE 1
```

```
static AVFormatContext *pFormatCtx;
static AVCodecContext *pCodecCtx;
AVFilterContext *buffersink_ctx;
AVFilterContext *buffersrc_ctx;
AVFilterGraph *filter_graph;
static int video_stream_index = -1;

static int open_input_file(const char *filename)
{
    int ret;
    AVCodec *dec;

    if ((ret = avformat_open_input(&pFormatCtx, filename, NULL,
NULL)) < 0) {
        printf( "Cannot open input file\n");
        return ret;
    }

    if ((ret = avformat_find_stream_info(pFormatCtx, NULL)) < 0) {
        printf( "Cannot find stream information\n");
        return ret;
    }

    /* select the video stream */
    ret = av_find_best_stream(pFormatCtx, AVMEDIA_TYPE_VIDEO,
-1, -1, &dec, 0);
    if (ret < 0) {
        printf( "Cannot find a video stream in the input file\n");
        return ret;
    }
    video_stream_index = ret;
    pCodecCtx = pFormatCtx->streams[video_stream_index]->codec;

    /* init the video decoder */
    if ((ret = avcodec_open2(pCodecCtx, dec, NULL)) < 0) {
        printf( "Cannot open video decoder\n");
        return ret;
    }
```

```

    return 0;
}

static int init_filters(char *filters_descr)
{
    char args[512];
    int ret;
    AVFilter *buffersrc = avfilter_get_by_name("buffer");
    AVFilter *buffersink = avfilter_get_by_name("ffbuffersink");
    AVFilterInOut *outputs = avfilter_inout_alloc();
    AVFilterInOut *inputs = avfilter_inout_alloc();
    enum AVPixelFormat pix_fmts[] = { AV_PIX_FMT_YUV420P,
AV_PIX_FMT_NONE };
    AVBufferSinkParams *buffersink_params;

    filter_graph = avfilter_graph_alloc();

/* buffer video source: the decoded frames from the decoder will be
inserted here.*/
    snprintf(args, sizeof(args),
            "video_size=%dx%d:pix_fmt=%d:time_base=%d/%d:pixel_aspec
            pCodecCtx->width, pCodecCtx->height, pCodecCtx->pix_fmt,
            pCodecCtx->time_base.num, pCodecCtx->time_base.den,
            pCodecCtx->sample_aspect_ratio.num, pCodecCtx-
            >sample_aspect_ratio.den);

    ret = avfilter_graph_create_filter(&buffersrc_ctx, buffersrc, "in",
                                      args, NULL, filter_graph);
    if (ret < 0) {
        printf("Cannot create buffer source\n");
        return ret;
    }

/* buffer video sink: to terminate the filter chain. */
    buffersink_params = av_buffersink_params_alloc();
    buffersink_params->pixel_fmts = pix_fmts;
    ret = avfilter_graph_create_filter(&buffersink_ctx, buffersink, "out",
                                      NULL, buffersink_params, filter_graph);
    av_free(buffersink_params);
}

```

```

if (ret < 0) {
    printf("Cannot create buffer sink\n");
    return ret;
}

/* Endpoints for the filter graph. */
outputs->name      = av_strdup("in");
outputs->filter_ctx = buffersrc_ctx;
outputs->pad_idx   = 0;
outputs->next      = NULL;

inputs->name      = av_strdup("out");
inputs->filter_ctx = buffersink_ctx;
inputs->pad_idx   = 0;
inputs->next      = NULL;

if ((ret = avfilter_graph_parse_ptr(filter_graph, filters_descr,
                                    &inputs, &outputs, NULL)) < 0)
    return ret;

if ((ret = avfilter_graph_config(filter_graph, NULL)) < 0)
    return ret;
return 0;
}

int main(int argc, char* argv[])
{
    int ret;
    AVPacket packet;
    AVFrame *pFrame;
    AVFrame *pFrame_out;

    int got_frame;

    av_register_all();
    avfilter_register_all();

    char *filter_descr;
    filter_descr = "movie=my_logo.png[wm];[in][wm]overlay=5:5[out]";
}

```

```
//filter_descr = "color=red@0.2:qcif:10 [color]; [in][color] overlay  
[out]";  
//filter_descr = "boxblur";  
//filter_descr = "crop=2/3*in_w:2/3*in_h";  
//filter_descr =  
"drawbox=x=100:y=100:w=100:h=100:color=pink@0.5";  
//filter_descr = "delogo=x=0:y=0:w=100:h=77:band=10";  
//filter_descr =  
"drawgrid=width=100:height=100:thickness=2:color=red@0.5";  
//filter_descr = "drawgrid=w=iw/3:h=ih/3:t=2:c=white@0.5";  
//filter_descr = "fade=in:0:60";  
//filter_descr = "format=pix_fmts=yuv420p";  
//filter_descr = "format=pix_fmts=yuv420p|yuv444p|yuv410p";  
//filter_descr = "gradfun=3.5:8";  
//filter_descr = "hflip";  
//filter_descr = "hue=h=90:s=1";  
//filter_descr =  
"kerndeint=thresh=10:map=1:order=0:sharp=0:twoway=0";  
//filter_descr = "lutyuv=y=gammaval(0.5)";  
//filter_descr = "lutyuv=y='bitand(val, 128+64+32)"';  
//filter_descr = "lutyuv=u='(val-maxval/2)*2+maxval/2':v='(val-  
maxval/2)*2+maxval/2"';  
//filter_descr = "format=yuv420p,mergeplanes=0x000201:yuv420p";  
//filter_descr = "noise=alls=20:allf=t+u";  
//filter_descr = "pixdesctest";  
//filter_descr = "pp=hb/vb/dr/al";  
//filter_descr = "pp=de/-al";  
//filter_descr = "pp=default/tmpnoise|1|2|3";  
//filter_descr = "pp=hb|y/vb|a";  
//filter_descr = "pullup";  
//filter_descr = "rotate=PI/6";  
//filter_descr = "scale=w=200:h=100";  
//filter_descr = "setdar=dar=1.77777";  
//filter_descr = "setsar=sar=10/11";  
//filter_descr = "setdar=ratio=16/9:max=1000";  
//filter_descr = "shuffleplanes=0:2:1:3";  
//filter_descr = "stereo3d=sbsl:aybd";
```

```

//filter_descr = "stereo3d=abl:sbsr";
/////////////////////////////filter_descr = "subtitles=filename=sample.srt";
//filter_descr = "thumbnail=50";
//filter_descr = "transpose=dir=1:passthrough=portrait";
//filter_descr = "trim=60:120";
//filter_descr = "trim=duration=1";
//filter_descr =
"unsharp=luma_msize_x=7:luma_msize_y=7:luma_amount=2.5";
//filter_descr = "unsharp=7:7:-2:7:7:-2";
//filter_descr = "vidstabdetect";
//filter_descr = "vidstabdetect=show=1";
//filter_descr = "vidstabdetect=shakiness=5:show=1";
//filter_descr = "vidstabtransform,unsharp=5:5:0.8:3:3:0.4";
//filter_descr = "vidstabtransform=smoothing=30";
//filter_descr = "vignette=PI/4";
//filter_descr = "vignette='PI/4+random(1)*PI/50':eval=frame";
//filter_descr = "blackframe";
//filter_descr = "blackframe=amount=2:threshold=4";
//filter_descr = "fps=10:start_time=100";
//filter_descr = "hqdn3d";
//filter_descr = "lutyuv='y=maxval+minval-val:u=maxval+minval-
val:v=maxval+minval-val'";
//filter_descr = "negate";
//filter_descr = "pad='max(iw\,ih):ow:(ow-iw)/2:(oh-ih)/2'";
//filter_descr = "pad='ih*16/9:ih:(ow-iw)/2:(oh-ih)/2'";
//filter_descr = "select='expr=eq(pict_type\,I)'";
//filter_descr = "setdar=dar=16/9";
//filter_descr = "setpts=expr=PTS-STARTPTS";
//filter_descr = "setsar=sar=10/11";
//filter_descr = "settb=expr=1/5";
//filter_descr = "shuffleplanes=0:2:1:3";

if ((ret = open_input_file("test.mp4")) < 0)
    goto end;
if ((ret = init_filters(filter_descr)) < 0)
    goto end;

#if ENABLE_YUVFILE

```

```

FILE *fp_yuv=fopen("test.yuv","wb+");
#endif
#if ENABLE SDL
    SDL_Surface *screen;
    SDL_Overlay *bmp;
    SDL_Rect rect;
    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO |
SDL_INIT_TIMER)) {
        printf( "Could not initialize SDL - %s\n", SDL_GetError());
        return -1;
    }
    screen = SDL_SetVideoMode(pCodecCtx->width, pCodecCtx-
>height, 0, 0);
    if(!screen) {
        printf("SDL: could not set video mode - exiting\n");
        return -1;
    }
    bmp = SDL_CreateYUVOverlay(pCodecCtx->width, pCodecCtx-
>height,SDL_YV12_OVERLAY, screen);

    SDL_WM_SetCaption("Simplest FFmpeg Video Filter",NULL);
#endif

pFrame=av_frame_alloc();
pFrame_out=av_frame_alloc();

/* read all packets */
while (1) {

    ret = av_read_frame(pFormatCtx, &packet);
    if (ret< 0)
        break;

    if (packet.stream_index == video_stream_index) {
        got_frame = 0;
        ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_frame,
&packet);
        if (ret < 0) {
            printf( "Error decoding video\n");

```

```

        break;
    }

    if (got_frame) {
        pFrame->pts =
av_frame_get_best_effort_timestamp(pFrame);
        /* push the decoded frame into the filtergraph */
        if (av_buffersrc_add_frame(buffersrc_ctx, pFrame) < 0) {
            printf( "Error while feeding the filtergraph\n");
            break;
        }

        /* pull filtered pictures from the filtergraph */
        while (1) {
            ret = av_buffersink_get_frame(buffersink_ctx, pFrame_out);
            if (ret < 0) break;
            printf("Process 1 frame!\n");

            if (pFrame_out->format==AV_PIX_FMT_YUV420P) {
#ifndef ENABLE_YUVFILE
                //Y, U, V
                for(int i=0;i<pFrame_out->height;i++){
fwrite(pFrame_out->data[0]+pFrame_out->linesize[0]*i,1,pFrame_out->width,fp_yuv);
                }
                for(int i=0;i<pFrame_out->height/2;i++){
fwrite(pFrame_out->data[1]+pFrame_out->linesize[1]*i,1,pFrame_out->width/2,fp_yuv);
                }
                for(int i=0;i<pFrame_out->height/2;i++){
fwrite(pFrame_out->data[2]+pFrame_out->linesize[2]*i,1,pFrame_out->width/2,fp_yuv);
                }
#endif
#ifndef ENABLE SDL
                SDL_LockYUVOverlay(bmp);
                int y_size=pFrame_out->width*pFrame_out->height;
                memcpy(bmp->pixels[0],pFrame_out->data[0],y_size); //Y

```

```

        memcpy(bmp->pixels[2],pFrame_out->data[1],y_size/4); //U
        memcpy(bmp->pixels[1],pFrame_out->data[2],y_size/4); //V
        bmp->pitches[0]=pFrame_out->linesize[0];
        bmp->pitches[2]=pFrame_out->linesize[1];
        bmp->pitches[1]=pFrame_out->linesize[2];
        SDL_UnlockYUVOverlay(bmp);
        rect.x = 0;
        rect.y = 0;
        rect.w = pFrame_out->width;
        rect.h = pFrame_out->height;
        SDL_DisplayYUVOverlay(bmp, &rect);
        //Delay 40ms
        SDL_Delay(40);
#endif
    }
    av_frame_unref(pFrame_out);
}
av_frame_unref(pFrame);
}
av_free_packet(&packet);
}

#endif ENABLE_YUVFILE
fclose(fp_yuv);
#endif

end:
avfilter_graph_free(&filter_graph);
if (pCodecCtx)
    avcodec_close(pCodecCtx);
avformat_close_input(&pFormatCtx);

if (ret < 0 && ret != AVERROR_EOF) {
    char buf[1024];
    av_strerror(ret, buf, sizeof(buf));
    printf("Error occurred: %s\n", buf);
    return -1;
}

```

```
    return 0;  
}
```

COLOR CHANGE RECTANGULAR AREA

Provide an uniformly colored input.

It accepts the following parameters:

'color'

Specify the color of the source. It can be the name of a color (case insensitive match) or a 0xRRGGBB[AA] sequence, possibly followed by an alpha specifier. The default value is "black".

'size'

Specify the size of the sourced video, it may be a string of the form widthxheight, or the name of a size abbreviation. The default value is "320x240".

'framerate'

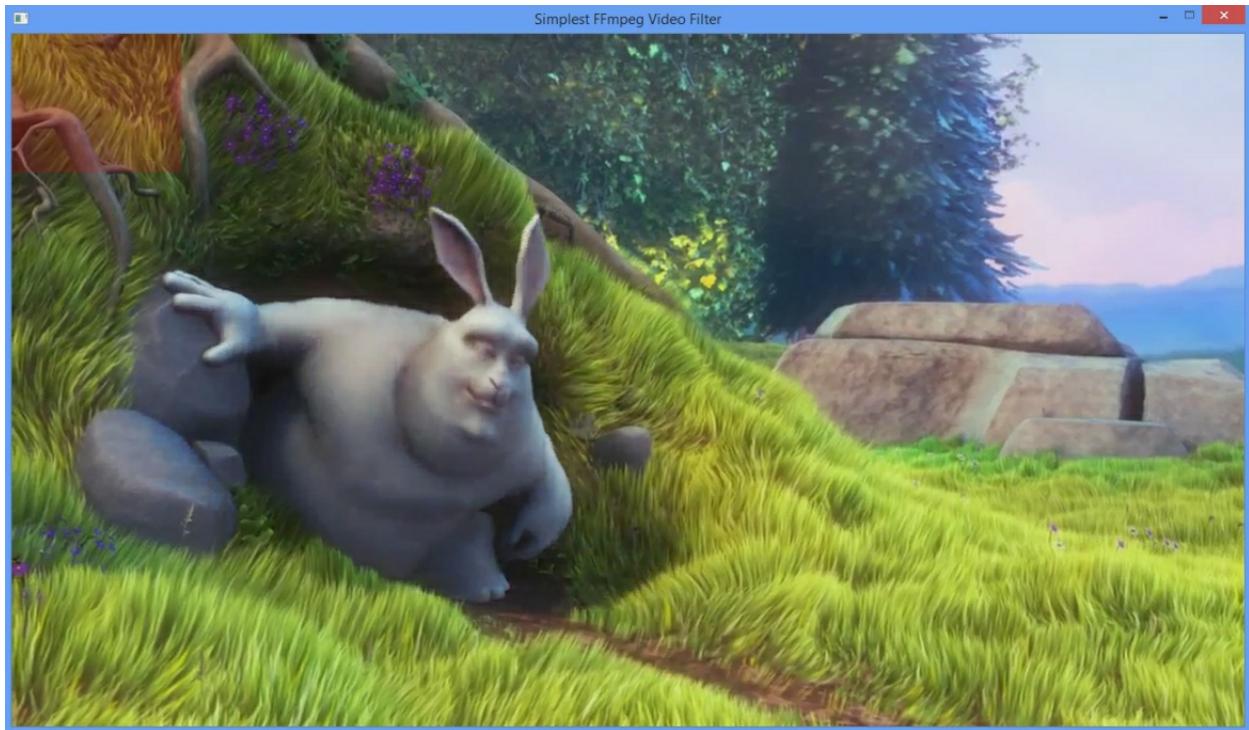
Specify the frame rate of the sourced video, as the number of frames generated per second. It has to be a string in the format frame_rate_num/frame_rate_den, an integer number, a floating point number or a valid video frame rate abbreviation. The default value is "25".

The following graph description will generate a red source with an opacity of 0.2, with size "qcif" and a

frame rate of 10 frames per second, which will be overlaid over the source connected to the pad with identifier "in":

```
filter_descr = "color=red@0.2:qcif:10 [color]; [in][color] overlay [out]";
```

OUTPUT:



WATERMARK

Read a video stream from a movie container.

Note that this source is a hack that bypasses the standard input path. It can be useful in applications that do not support arbitrary filter graphs, but its use is discouraged in those that do. It should never be used with avconv; the ‘-filter_complex’ option fully replaces it. It accepts the following parameters:

‘filename’

The name of the resource to read (not necessarily a file; it can also be a device or a stream accessed through some protocol).

‘format_name, f’

Specifies the format assumed for the movie to read, and can be either the name of a container or an input device. If not specified, the format is guessed from movie_name or by probing.

‘seek_point, sp’

Specifies the seek point in seconds. The frames will be output starting from this seek point. The parameter is evaluated with av_strtod, so the numerical value may

be suffixed by an IS postfix. The default value is "0".

'stream_index, si'

Specifies the index of the video stream to read. If the value is -1, the most suitable video stream will be automatically selected. The default value is "-1".

It allows overlaying a second video on top of the main input of a filtergraph, as shown in this graph:

INPUT ----->
DELTAPTS0 --> OVERLAY
--> OUTPUT
 ^
 |

**MOVIE --> SCALE-->
DELTAPTS1 -----+**

```
filter_descr = "movie=my_logo.png[wm];[in][wm]overlay=5:5[out]";
```

OUTPUT:



COLOR EFFECTS

BOXBLUR

Apply a boxblur algorithm to the input video.

It accepts the following parameters:

- ‘luma_radius’
- ‘luma_power’
- ‘chroma_radius’
- ‘chroma_power’
- ‘alpha_radius’
- ‘alpha_power’

The chroma and alpha parameters are optional. If not specified, they default to the corresponding values set for luma_radius and luma_power.

luma_radius, chroma_radius, and alpha_radius represent the radius in pixels of the box used for blurring the corresponding input plane. They are expressions, and can contain the following constants:

- ‘w, h’

The input width and height in pixels.

- ‘cw, ch’

The input chroma image width and height in pixels.

- ‘hsub, vsub’

The horizontal and vertical chroma subsample values. For example, for the pixel format "yuv422p", hsub is 2 and vsub is 1.

The radius must be a non-negative number, and must not be greater than the value of the expression $\min(w,h)/2$ for the luma and alpha planes, and of $\min(cw,ch)/2$ for the chroma planes.

luma_power, chroma_power, and alpha_power represent how many times the boxblur filter is applied to the corresponding plane.

```
filter_descr = "boxblur";
```

HUE

Modify the hue and/or the saturation of the input.

It accepts the following parameters:

'h'

Specify the hue angle as a number of degrees. It accepts an expression, and defaults to "0".

's'

Specify the saturation in the [-10,10] range. It accepts an expression and defaults to "1".

'H'

Specify the hue angle as a number of radians. It accepts an expression, and defaults to "0".

'b'

Specify the brightness in the [-10,10] range. It accepts an expression and defaults to "0".

'h' and 'H' are mutually exclusive, and can't be specified at the same time.

The 'b', 'h', 'H' and 's' option values are expressions containing the following constants:

'n'

frame count of the input frame starting from 0

‘pts’

presentation timestamp of the input frame expressed in time base units

‘r’

frame rate of the input video, NAN if the input frame rate is unknown

‘t’

timestamp expressed in seconds, NAN if the input timestamp is unknown

‘tb’

time base of the input video

```
filter_descr = "hue=h=90:s=1";
```

OUTPUT:



LUT, LUTRGB, LUTYUV

Compute a look-up table for binding each pixel component input value to an output value, and apply it to the input video. lutyuv applies a lookup table to a YUV input video, lutrgb to an RGB input video.

These filters accept the following parameters:

- ‘c0 (first pixel component)’
- ‘c1 (second pixel component)’
- ‘c2 (third pixel component)’
- ‘c3 (fourth pixel component, corresponds to the alpha component)’
- ‘r (red component)’
- ‘g (green component)’
- ‘b (blue component)’
- ‘a (alpha component)’
- ‘y (Y/luminance component)’
- ‘u (U/Cb component)’
- ‘v (V/Cr component)’

Each of them specifies the expression to use for computing the lookup table for the corresponding pixel component values.

The exact component associated to each of the c* options depends on the format in input.

The lut filter requires either YUV or RGB pixel formats in input, lutrgb requires RGB pixel formats in input, and lutyuv requires YUV.

The expressions can contain the following constants and functions:

‘E, PI, PHI’

These are approximated values for the mathematical constants e (Euler's number), pi (Greek pi), and phi (the golden ratio).

‘w, h’

The input width and height.

‘val’

The input value for the pixel component.

‘clipval’

The input value, clipped to the minval-maxval range.

‘maxval’

The maximum value for the pixel component.

‘minval’

The minimum value for the pixel component.

‘negval’

The negated value for the pixel component value, clipped to the minval-maxval range; it corresponds to the expression "maxval-clipval+minval".

‘clip(val)’

The computed value in val, clipped to the minval-maxval range.

‘gammaval(gamma)’

The computed gamma correction value of the pixel component value, clipped to the minval-maxval range. It corresponds to the expression "pow((clipval-minval)/(maxval-minval)\,gamma)*(maxval-minval)+minval"

All expressions default to "val".

```
filter_descr = "lutyuv=y=gammaval(0.5);
```

CROP VIDEO

Crop the input video to given dimensions.

It accepts the following parameters:

‘out_w’

The width of the output video.

‘out_h’

The height of the output video.

‘x’

The horizontal position, in the input video, of the left edge of the output video.

‘y’

The vertical position, in the input video, of the top edge of the output video.

The parameters are expressions containing the following constants:

‘E, PI, PHI’

These are approximated values for the mathematical constants e (Euler’s number), pi (Greek pi), and phi (the golden ratio).

‘x, y’

The computed values for x and y. They are evaluated for each new frame.

'in_w, in_h'

The input width and height.

'iw, ih'

These are the same as in_w and in_h.

'out_w, out_h'

The output (cropped) width and height.

'ow, oh'

These are the same as out_w and out_h.

'n'

The number of the input frame, starting from 0.

't'

The timestamp expressed in seconds. It's NAN if the input timestamp is unknown.

The out_w and out_h parameters specify the expressions for the width and height of the output (cropped) video. They are only evaluated during the configuration of the filter.

The default value of out_w is "in_w", and the default value of out_h is "in_h".

The expression for out_w may depend on the value of out_h, and the expression for out_h may depend on out_w, but they cannot depend on x and y, as x and y are evaluated after out_w and out_h.

The x and y parameters specify the expressions for the position of the top-left corner of the output (non-cropped) area. They are evaluated for each frame. If the

evaluated value is not valid, it is approximated to the nearest valid value.

The default value of x is " $(in_w-out_w)/2$ ", and the default value for y is " $(in_h-out_h)/2$ ", which set the cropped area at the center of the input image.

The expression for x may depend on y, and the expression for y may depend on x.

```
filter_descr = "crop=2/3*in_w:2/3*in_h";
```

DRAW BOX ON VIDEO

Draw a colored box on the input image. It accepts the following parameters:

‘x, y’

Specify the top left corner coordinates of the box. It defaults to 0.

‘width, height’

Specify the width and height of the box; if 0 they are interpreted as the input width and height. It defaults to 0.

‘color’

Specify the color of the box to write. It can be the name of a color (case insensitive match) or a 0xRRGGBB[AA] sequence.

```
filter_descr = "drawbox=x=100:y=100:w=100:h=100:color=pink@0.5";
```

OUTPUT:



REMOVE LOGO FROM VIDEO

SUPPRESS A TV STATION LOGO BY A SIMPLE INTERPOLATION OF THE SURROUNDING PIXELS. JUST SET A RECTANGLE COVERING THE LOGO AND WATCH IT DISAPPEAR (AND SOMETIMES SOMETHING EVEN UGLIER APPEAR - YOUR MILEAGE MAY VARY).

IT ACCEPTS THE FOLLOWING PARAMETERS:

‘x, y’

SPECIFY THE TOP LEFT CORNER COORDINATES OF THE LOGO.
THEY MUST BE SPECIFIED.

‘w, h’

SPECIFY THE WIDTH AND HEIGHT OF THE LOGO TO CLEAR. THEY
MUST BE SPECIFIED.

‘band, t’

SPECIFY THE THICKNESS OF THE FUZZY EDGE OF THE RECTANGLE
(ADDED TO w AND h). THE DEFAULT VALUE IS 4.

‘show’

WHEN SET TO 1, A GREEN RECTANGLE IS DRAWN ON THE SCREEN
TO SIMPLIFY FINDING THE RIGHT x, y, w, h PARAMETERS, AND

band is set to 4. The default value is 0.

```
filter_descr = "delogo=x=0:y=0:w=100:h=77:band=10";
```

OUTPUT:



DRAW GRID ON VIDEO

Draw a grid on the input image. It accepts the following parameters: ‘x’, ‘y’ The expressions which specify the coordinates of some point of grid intersection (meant to configure offset). Both default to 0.

‘width, height’

The expressions which specify the width and height of the grid cell, if 0 they are interpreted as the input width and height, respectively, minus thickness, so image gets framed. Default to 0.

‘color’

Specify the color of the grid. For the general syntax of this option, check the (ffmpeg-utils)"Color" section in the ffmpeg-utils manual. If the special value invert is used, the grid color is the same as the video with inverted luma.

‘thickness’

The expression which sets the thickness of the grid line. Default value is 1.

See below for the list of accepted constants.

‘replace’

Applicable if the input has alpha. With 1 the pixels of the painted grid will overwrite the video’s color and alpha pixels. Default is 0, which composites the grid onto the input, leaving the video’s alpha intact.

The parameters for x, y, w and h and t are expressions containing the following constants:

‘dar’

The input display aspect ratio, it is the same as (w / h) * sar.

‘hsub,vsub’

horizontal and vertical chroma subsample values. For example for the pixel format "yuv422p" hsub is 2 and vsub is 1.

‘in_h, ih, in_w, iw’

The input grid cell width and height.

‘sar’

The input sample aspect ratio.

‘x, y’

The x and y coordinates of some point of grid intersection (meant to configure offset).

‘w,h’

The width and height of the drawn cell.

‘t’

The thickness of the drawn cell. These constants allow the x, y, w, h and t expressions to refer to each other, so you may for example specify y=x/dar or h=w/dar.

```
filter_descr =  
"drawgrid=width=100:height=100:thickness=2:color=red@0.5";
```

OUTPUT:



FADE EFFECT ON VIDEO

Apply a fade-in/out effect to the input video. It accepts the following parameters:

‘type’

The effect type can be either "in" for a fade-in, or "out" for a fade-out effect.

‘start_frame’

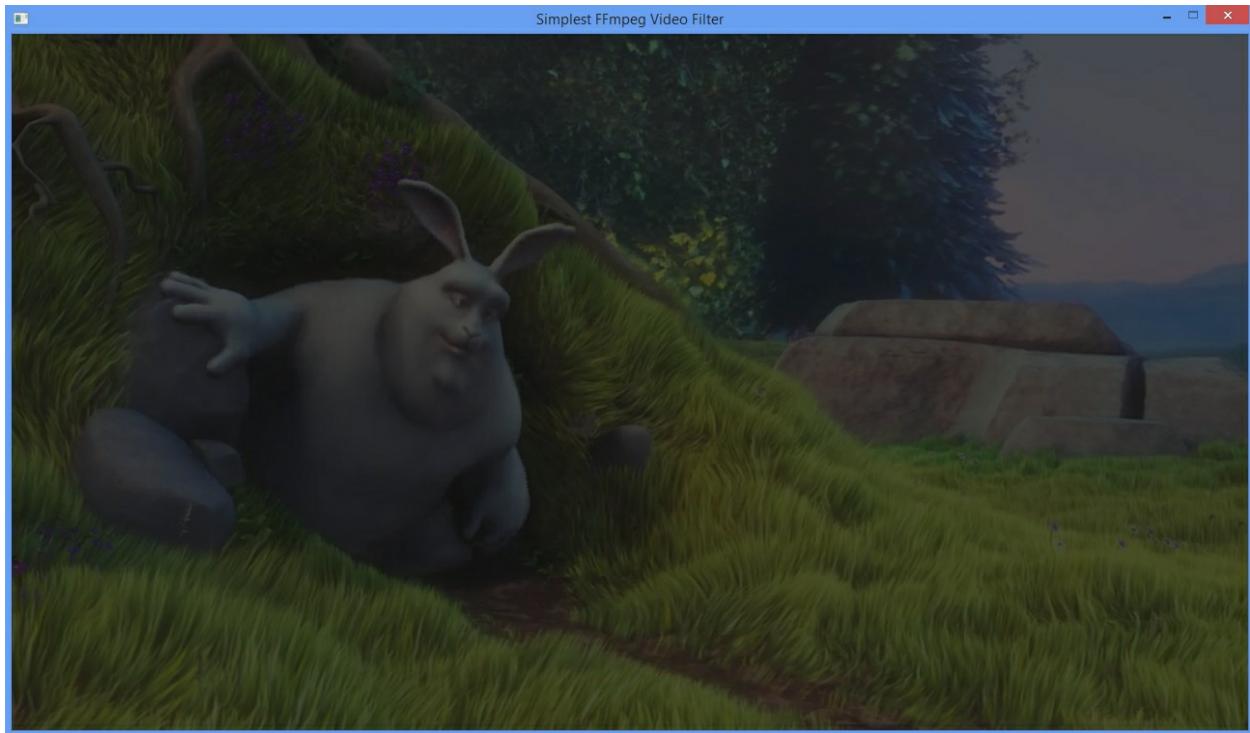
The number of the frame to start applying the fade effect at.

‘nb_frames’

The number of frames that the fade effect lasts. At the end of the fade-in effect, the output video will have the same intensity as the input video. At the end of the fade-out transition, the output video will be completely black.

```
filter_descr = "fade=in:0:60";
```

OUTPUT:

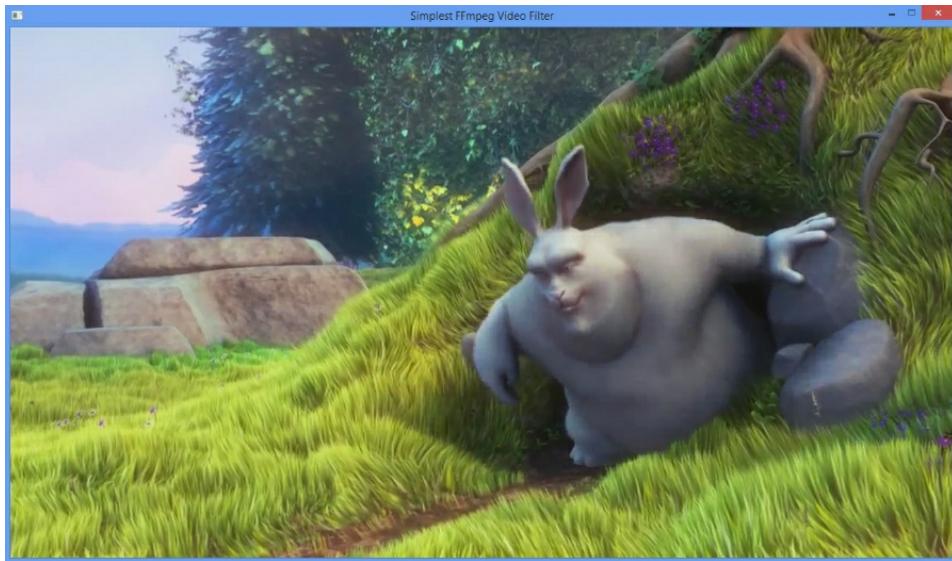


HORIZONTAL FLIP VIDEO

Flip the input video horizontally.

```
filter_descr = "hflip";
```

OUTPUT:



ADD NOISE ON VIDEO

Add noise on video input frame. The filter accepts the following options:

all_seed

c0_seed

c1_seed

c2_seed

c3_seed

Set noise seed for specific pixel component or all pixel components in case of all_seed. Default value is 123457.

all_strength, alls

c0_strength, c0s

c1_strength, c1s

c2_strength, c2s

c3_strength, c3s

Set noise strength for specific pixel component or all pixel components in case all_strength. Default value is 0. Allowed range is [0, 100].

all_flags, allf

c0_flags, c0f

c1_flags, c1f

c2_flags, c2f

c3_flags, c3f

Set pixel component flags or set flags for all components if all_flags. Available values for component flags are:

‘a’ averaged temporal noise (smoother)

‘p’ mix random noise with a (semi)regular pattern

‘t’ temporal noise (noise pattern changes between frames)

‘u’ uniform noise (gaussian otherwise)

```
filter_descr = "noise=alls=20:allf=t+u";
```

ROTATE VIDEO

Rotate video by an arbitrary angle expressed in radians.
The filter accepts the following options:

A description of the optional parameters follows.

`angle, a`

Set an expression for the angle by which to rotate the input video clockwise, expressed as a number of radians. A negative value will result in a counter-clockwise rotation. By default it is set to "0". This expression is evaluated for each frame.

`out_w, ow`

Set the output width expression, default value is "iw".
This expression is evaluated just once during configuration.

`out_h, oh`

Set the output height expression, default value is "ih".
This expression is evaluated just once during configuration.

`bilinear`

Enable bilinear interpolation if set to 1, a value of 0

disables it. Default value is 1.

fillcolor, c

Set the color used to fill the output area not covered by the rotated image. For the general syntax of this option, check the (ffmpeg-utils)"Color" section in the ffmpeg-utils manual. If the special value "none" is selected then no background is printed (useful for example if the background is never shown).

Default value is "black".

The expressions for the angle and the output size can contain the following constants and functions:

‘n’ sequential number of the input frame, starting from 0. It is always NAN before the first frame is filtered.

‘t’ time in seconds of the input frame, it is set to 0 when the filter is configured. It is always NAN before the first frame is filtered.

hsub

vsub

horizontal and vertical chroma subsample values. For example for the pixel format "yuv422p" hsub is 2 and vsub is 1.

in_w, iw

in_h, ih

the input video width and height

out_w, ow

out_h, oh

the output width and height, that is the size of the padded area as specified by the width and height expressions

rotw(a)

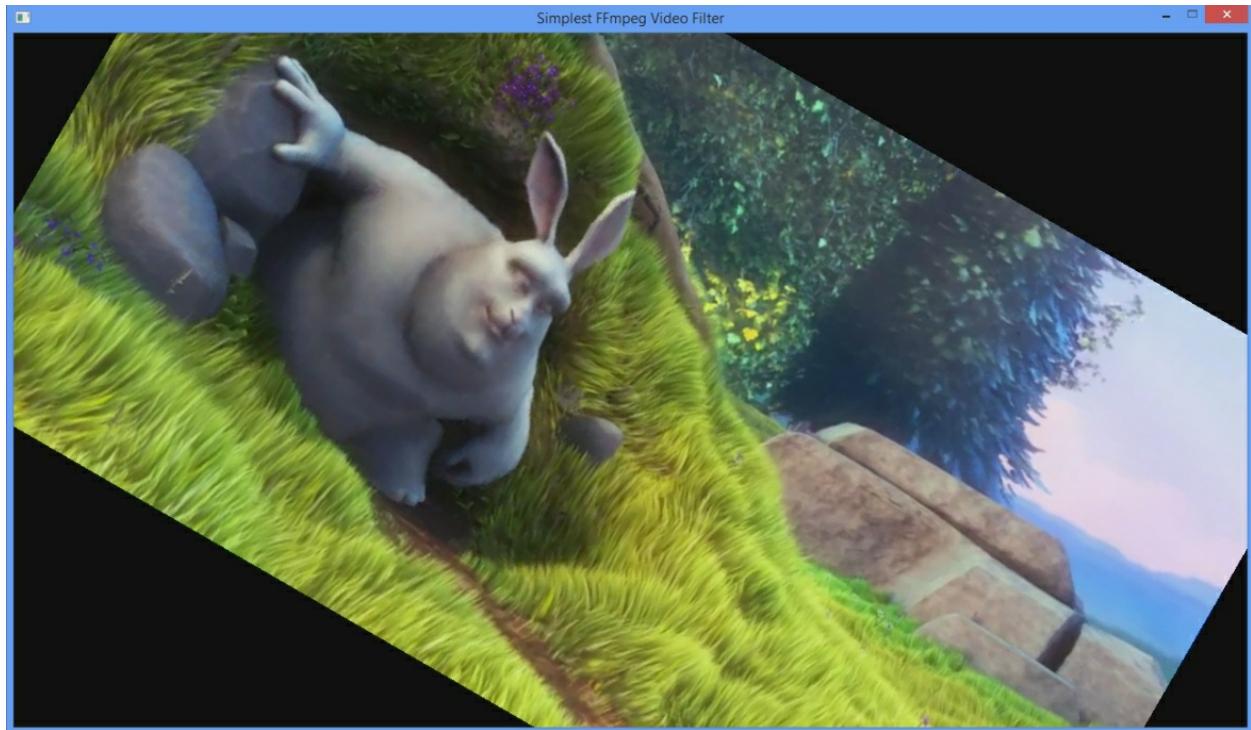
roth(a)

the minimal width/height required for completely containing the input video rotated by a radians.

These are only available when computing the out_w and out_h expressions.

```
filter_descr = "rotate=PI/6";
```

OUTPUT:



TRIM VIDEO

Trim the input so that the output contains one continuous subpart of the input. It accepts the following parameters:

‘start’

The timestamp (in seconds) of the start of the kept section. The frame with the timestamp start will be the first frame in the output.

‘end’

The timestamp (in seconds) of the first frame that will be dropped. The frame immediately preceding the one with the timestamp end will be the last frame in the output.

‘start_pts’

This is the same as start, except this option sets the start timestamp in timebase units instead of seconds.

‘end_pts’

This is the same as end, except this option sets the end timestamp in timebase units instead of seconds.

‘duration’

The maximum duration of the output in seconds.

‘start_frame’

The number of the first frame that should be passed to the output.

‘end_frame’

The number of the first frame that should be dropped.

Note that the first two sets of the start/end options and the ‘duration’ option look at the frame timestamp, while the _frame variants simply count the frames that pass through the filter. Also note that this filter does not modify the timestamps. If you wish for the output timestamps to start at zero, insert a setpts filter after the trim filter.

If multiple start or end options are set, this filter tries to be greedy and keep all the frames that match at least one of the specified constraints. To keep only the part that matches all the constraints at once, chain multiple trim filters.

The defaults are such that all the input is kept. So it is possible to set e.g. just the end values to keep everything before the specified time.

```
filter_descr = "trim=60:120";
```

ALL OTHER FILTERS IN SAMPLE

Convert the input video to one of the specified pixel formats	filter_descr = "format=pix_fmts=yuv420p";
Fix the banding artifacts that are sometimes introduced into nearly flat regions by truncation to 8-bit color depth. Interpolate the gradients that should go where the bands are, and dither them.	filter_descr = "gradfun=3.5:8";
Deinterlace input video by applying Donald Graft's adaptive kernel deinterlacing. Work on	filter_descr = "kerndeint thresh=10:map=1:order=(

interlaced parts of a video to produce progressive frames.	
Pixel format descriptor test filter, mainly useful for internal testing. The output video should be equal to the input video.	filter_descr = "pixdesctest";
Enable the specified chain of postprocessing subfilters using libpostproc. This library should be automatically selected with a GPL build (--enable-gpl). Subfilters must be separated by '/' and can be disabled by prepending a '-'. Each subfilter and some options have a short and a long	filter_descr = "pp=hb/vb/dr/al";

<p>name that can be used interchangeably, i.e. dr/dering are the same.</p>	
<p>Pulldown reversal (inverse telecine) filter, capable of handling mixed hard-telecine, 24000/1001 fps progressive, and 30000/1001 fps progressive content.</p>	<p>filter_descr = "pullup";</p>
<p>Scale the input video and/or convert the image format.</p>	<p>filter_descr = "scale=w=200:h=100";</p>
<p>Set the Display Aspect Ratio for the filter output video.</p>	<p>filter_descr = "setdar=dar=1.77777";</p>
<p>Set the Sample (aka Pixel) Aspect Ratio for the filter output video.</p>	<p>filter_descr = "setsar=sar=10/11";</p>
<p>Reorder and/or duplicate video planes.</p>	<p>filter_descr = "shuffleplanes=0:2:1:3"</p>

Convert between different stereoscopic image formats.	filter_descr = "stereo3d=sbsl:aybd";
Sharpen or blur the input video.	filter_descr = "unsharp=7:7:-2:7:7:-2"
Analyze video stabilization/deshaking.	filter_descr = "vidstabdetect";
Video stabilization/deshaking	filter_descr = "vidstabtransform,unsharp=5:5:0.8:3";
Make or reverse a natural vignetting effect.	filter_descr = "vignette=PI/4";
Detect frames that are (almost) completely black.	filter_descr = "blackframe";
Convert the video to specified constant frame rate by duplicating or dropping frames as necessary.	filter_descr = "fps=10:start_time=100"
This is a high precision/quality 3d denoise filter.	filter_descr = "hqdn3d";
Negate (invert) the	filter_descr = "negate";

input video.	
Add paddings to the input image, and place the original input at the provided x, y coordinates.	filter_descr = "pad='max(iw\,ih):ow:(ih)/2"';
Select frames to pass in output.	filter_descr = "select='expr=eq(pict_t"
Change the PTS (presentation timestamp) of the input frames.	filter_descr = "setpts=expr=PTS-STA
Set the timebase to use for the output frames timestamps.	filter_descr = "settb=expr=1/5";

WRITE YOUR OWN FILTER (WYOF)

how do you implement one yourself?

It's simple, do 3 things:

a). Write a XXX.c file, such as vf_transform.c, in the libavfilter directory. Code can refer to other filters;

b) Add a line in libavfilter / allfilters.c:

REGISTER_FILTER (TRANSFORM, transform, vf);

c) Modify libavfilter / Makefile and add a line:

OBJS-\$ (CONFIG_TRANSFORM_FILTER) +=
vf_transform.o

Know the steps, now do the first step, start coding a C file, the name is vf_transform.c, the code is shown below.

```
#include "libavutil/opt.h"
#include "libavutil/imgutils.h"
#include "libavutil/avassert.h"
#include "avfilter.h"
#include "formats.h"
#include "internal.h"
#include "video.h"
```

```

typedef struct TransformContext {
    const AVClass *class;
    int backUp;
    //add some private data if you want
} TransformContext;

typedef struct ThreadData {
    AVFrame *in, *out;
} ThreadData;

static void image_copy_plane(uint8_t *dst, int dst_linesize,
                           const uint8_t *src, int src_linesize,
                           int bytewidth, int height)
{
    if (!dst || !src)
        return;
    av_assert0(abs(src_linesize) >= bytewidth);
    av_assert0(abs(dst_linesize) >= bytewidth);
    for (;height > 0; height--) {
        memcpy(dst, src, bytewidth);
        dst += dst_linesize;
        src += src_linesize;
    }
}

//for YUV data, frame->data[0] save Y, frame->data[1] save U, frame->data[2] save V
static int frame_copy_video(AVFrame *dst, const AVFrame *src)
{
    int i, planes;

    if (dst->width > src->width ||
        dst->height > src->height)
        return AVERROR(EINVAL);

    planes = av_pix_fmt_count_planes(dst->format);
    //make sure data is valid
    for (i = 0; i < planes; i++)
        if (!dst->data[i] || !src->data[i])
            return AVERROR(EINVAL);
}

```

```

const AVPixFmtDescriptor *desc = av_pix_fmt_desc_get(dst->format);
int planes_nb = 0;
for (i = 0; i < desc->nb_components; i++)
    planes_nb = FFMAX(planes_nb, desc->comp[i].plane + 1);

for (i = 0; i < planes_nb; i++) {
    int h = dst->height;
    int bwidth = av_image_get_linesize(dst->format, dst->width, i);
    if (bwidth < 0) {
        av_log(NULL, AV_LOG_ERROR, "av_image_get_linesize fail");
        return;
    }
    if (i == 1 || i == 2) {
        h = AV_CEIL_RSHIFT(dst->height, desc->log2_chroma_h);
    }
    image_copy_plane(dst->data[i], dst->linesize[i],
                    src->data[i], src->linesize[i],
                    bwidth, h);
}
return 0;
}

*****
* you can modify this function, do what you want here. use src frame, and bl
frame.
* for this demo, we just copy some part of src frame to dst frame(out_w = in_
in_h/2)
*****

static int do_conversion(AVFilterContext *ctx, void *arg, int jobnr,
                        int nb_jobs)
{
    TransformContext *privCtx = ctx->priv;
    ThreadData *td = arg;
    AVFrame *dst = td->out;
    AVFrame *src = td->in;

    frame_copy_video(dst, src);
    return 0;
}

```

```

}

static int filter_frame(AVFilterLink *link, AVFrame *in)
{
    av_log(NULL, AV_LOG_WARNING, "### chenxf filter_frame, link %s\n",
    link, in);
    AVFilterContext *avctx = link->dst;
    AVFilterLink *outlink = avctx->outputs[0];
    AVFrame *out;

    //allocate a new buffer, data is null
    out = ff_get_video_buffer(outlink, outlink->w, outlink->h);
    if (!out) {
        av_frame_free(&in);
        return AVERRORE(ENOMEM);
    }

    //the new output frame, property is the same as input frame, only width/height
    //different
    av_frame_copy_props(out, in);
    out->width = outlink->w;
    out->height = outlink->h;

    ThreadData td;
    td.in = in;
    td.out = out;
    int res;
    if(res = avctx->internal->execute(avctx, do_conversion, &td, NULL, FFMIN(
    >h, avctx->graph->nb_threads))) {
        return res;
    }

    av_frame_free(&in);

    return ff_filter_frame(outlink, out);
}

static av_cold int config_output(AVFilterLink *outlink)
{

```

```

AVFilterContext *ctx = outlink->src;
TransformContext *privCtx = ctx->priv;

//you can modify output width/height here
outlink->w = ctx->inputs[0]->w/2;
outlink->h = ctx->inputs[0]->h/2;
av_log(NULL, AV_LOG_DEBUG, "configure output, w h = (%d %d), fo
outlink->w, outlink->h, outlink->format);

    return 0;
}

static av_cold int init(AVFilterContext *ctx)
{
    av_log(NULL, AV_LOG_DEBUG, "init \n");
    TransformContext *privCtx = ctx->priv;
    //init something here if you want
    return 0;
}

static av_cold void uninit(AVFilterContext *ctx)
{
    av_log(NULL, AV_LOG_DEBUG, "uninit \n");
    TransformContext *privCtx = ctx->priv;
    //uninit something here if you want
}

//currently we just support the most common YUV420, can add more if need
static int query_formats(AVFilterContext *ctx)
{
    static const enum AVPixelFormat pix_fmts[] = {
        AV_PIX_FMT_YUV420P,
        AV_PIX_FMT_NONE
    };
    AVFilterFormats *fmts_list = ff_make_format_list(pix_fmts);
    if (!fmts_list)
        return AVERRORE(ENOMEM);
    return ff_set_common_formats(ctx, fmts_list);
}

```

```

//*****
#define OFFSET(x) offsetof(TransformContext, x)
#define FLAGS
AV_OPT_FLAG_VIDEO_PARAM|AV_OPT_FLAG_FILTERING_PARAM

static const AVOption transform_options[] = {
{ "backUp",      "a backup parameters, NOT use so
far",      OFFSET(backUp),  AV_OPT_TYPE_STRING, {.str = "0"}, CH
CHAR_MAX, FLAGS },
{ NULL }

};// TODO: add something if needed

static const AVClass transform_class = {
.class_name    = "transform",
.item_name     = av_default_item_name,
.option        = transform_options,
.version       = LIBAVUTIL_VERSION_INT,
.category      = AV_CLASS_CATEGORY_FILTER,
};

static const AVFilterPad avfilter_vf_transform_inputs[] = {
{
.name      = "transform_inputpad",
.type      = AVMEDIA_TYPE_VIDEO,
.filter_frame = filter_frame,
},
{ NULL }
};

static const AVFilterPad avfilter_vf_transform_outputs[] = {
{
.name = "transform_outputpad",
.type = AVMEDIA_TYPE_VIDEO,
.config_props = config_output,
},
{ NULL }
};

```

```
AVFilter ff_vf_transform = {
    .name      = "transform",
    .description = NULL_IF_CONFIG_SMALL("cut a part of video"),
    .priv_size   = sizeof(TransformContext),
    .priv_class   = &transform_class,
    .init        = init,
    .uninit       = uninit,
    .query_formats = query_formats,
    .inputs       = avfilter_vf_transform_inputs,
    .outputs      = avfilter_vf_transform_outputs,
};
```

To write a filter, basically follow the template above. The most critical function is `filter_frame`. You can modify the `filter_frame` to do the transform you want. Of course, I still want to introduce the code above. From the bottom up, the first thing we want to write is `AVFilter`, where the name is the externally declared name, which is the same as the "`-vf transform`" to be used on the command line.

Private size initializes the `TransformContext`. This is the private context of the filter you wrote yourself. You can put all the local global variables you need here, which is fine.

Then there are `init` and `uninit`. This depends on the situation. If your private context has something to initialize, put it in `init`. If not, you can delete these two

sentences. The init / uninit function does not need to be written.

Next is query_formats, which is a frame that declares what formats your filter supports. This example only writes YUV420, of course you can add support as needed.

Next is AVFilterPad inputs / outputs, which you can think of as the bridge between the filter and the outside.

For example, AVFilterPad avfilter_vf_transform_inputs, declares the structure function filter_frame, which will point to the filter_frame (...) function of this file. At this time, other filters can indirectly call filter_frame (...) through this function pointer.

Similarly, AVFilterPad avfilter_vf_transform_outputs declares the structure function pointer config_props, which will point to the config_output (...) function of this file. At this time, other filters can indirectly call config_output (...) through this function pointer.

filter_frame (...) is the most critical function, and the transformation we want to do must be implemented in this function.

Why use config_output (...)? It is used to configure the

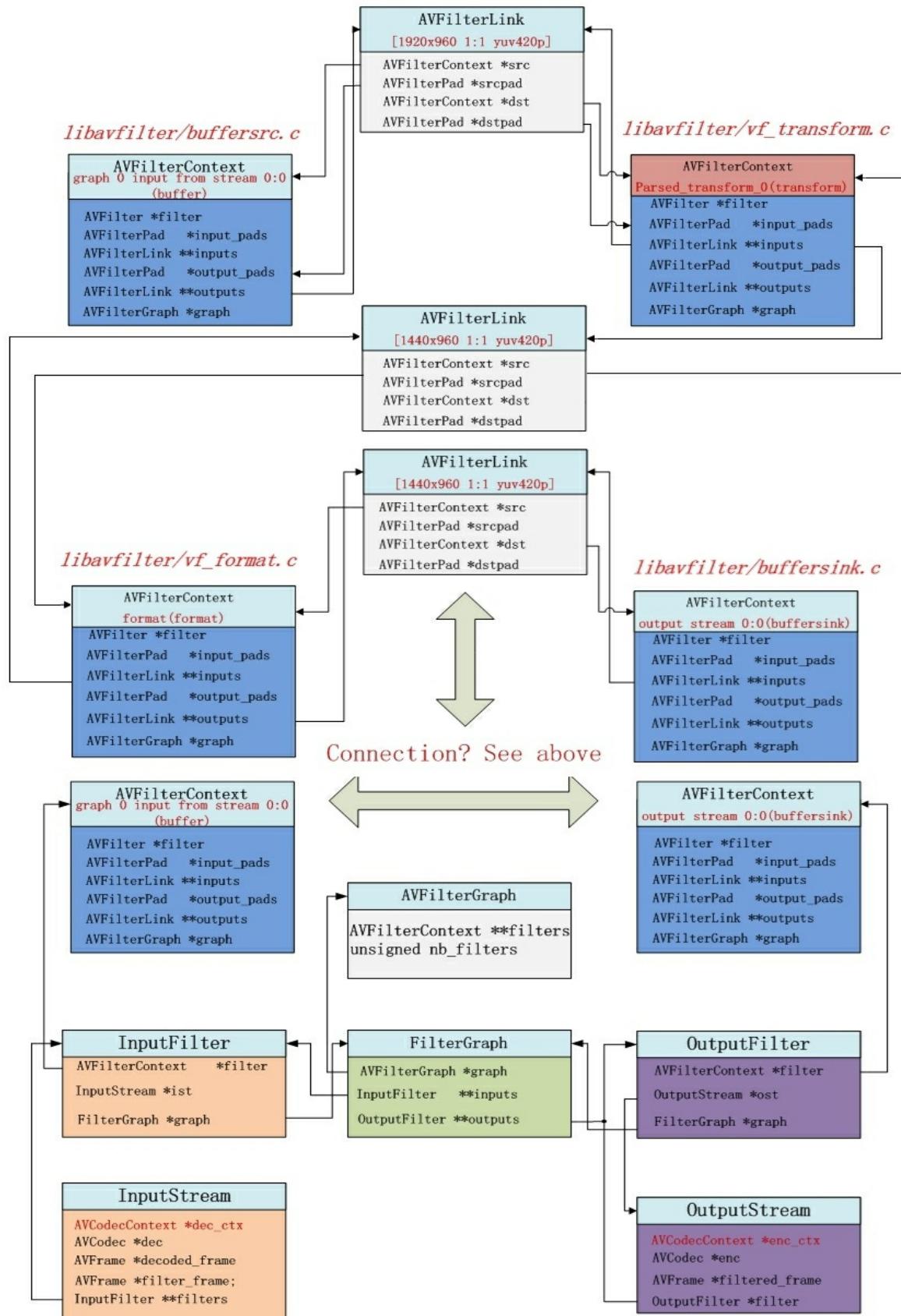
size of the output frame. For example, to input a 1920x1080 frame, we want to transform it and output it at 960x540, then this 960x540 must be set in this function.

Having said that, it seems that you basically understand. In this example, in the filter_frame function, the input frame, the upper left part, the cut dst frame, and then output. OK, recompile ffmpeg, and then you can run.

WYOF FILTER STRUCTURE

The structures involved in filter mainly include:
InputStream, OutputStream, FilterGraph,
AVFilterGraph, AVFilterContext, AVFilterLink,
AVFilterPad.

To clarify the intricate relationship between them, it is difficult to memorize the code alone. For this reason, Look picture, as shown below. (Take the example above as background)



The above example uses the command:

```
./ffmpeg -loglevel warning -i input.mp4 -vf transform  
output.mp4
```

That is, we use the transform filter we wrote. Suppose the source video input.mp4 has one video and one audio. Then, audio and video have 1 InputStream and 1 OutputStream, respectively.

Take video as an example, a total of InputStream & OutputStream. Then, the structure involved in video is as shown in the figure above. Generally, one InputStream corresponds to one Inputfilter and one OutputStream corresponds to one OutputFilter.

FilterGraph manages Inputfilter and OutputFilter (of course, the pointer * graph of Inputfilter and OutputFilter can find the manager FilterGraph). In addition, FilterGraph manages an AVFilterGraph.

What does AVFilterGraph do? It has a double pointer inside, ** filters, which is obviously an array of pointers, which stores a bunch of AVFilterContext pointers.

What does AVFilterContext correspond to? It actually corresponds to a filter. In other words, the context of a filter is AVFilterContext. So for the above picture, the ** filters of AVFilterGraph actually point to 4

AVFilterContexts.

Do you have a question, why did we write a filter ourselves, how could there be 4 filters involved?

In fact, ffmpeg has 3 filters by default! The names are "buffer", "format", "buffersink", which are the first, third, and fourth AVFilterContext in the upper part of the figure above.

What does AVFilterLink do? It is to establish the connection between AVFilterContext. Therefore, if there are 4 AVFilterContexts, then 3 AVFilterLinks are required.

The src pointer of AVFilterLink points to the previous AVFilterContext, and the dst pointer points to the next AVFilterContext.

What does AVFilterPad do? It is used for callbacks between AVFilterContexts.

How is the callback method?

Quite simply, the outputs [0] pointer of the first AVFilterContext points to the first AVFilterLink, and the dst pointer of this AVFilterLink points to the second AVFilterContext.

If I call in the previous AVFilterContext
outputs [0]-> dstpad-> filter_frame (Frame *

`input_frame1`), that actually means that the first filter can pass a processed frame (named `input_frame1`) to this second pass to the second `Filter_frame` function of the `input_pads` of each filter. The `vf_transform.c` we implemented is the second filter I said, which implements `filter_frame ()`.

WYOF FILTER_FRAME() CALL FLOW

Since filter_frame is the most critical function, it is also the function we must customize to write the filter ourselves, so let's take a look at where this function comes from and where it will go!

WYOF DECODE_VIDEO //FFMPEG.C

The original source was the decode_video function of ffmpeg.c.

Extract the core code as follows:

```
static int decode_video(InputStream *ist, AVPacket *pkt, int *got_output)
{
    AVFrame* decoded_frame, f;
    ret = avcodec_decode_video2(ist->dec_ctx, decoded_frame,
                               got_output, pkt);

    for (i = 0; i < ist->nb_filters; i++) {
        f = decodec_frame;
        ret = av_buffersrc_add_frame_flags(ist->filters[i]->filter, f,
AV_BUFFERSRC_FLAG_PUSH);
    }
}
```

It can be seen that the most important thing is to do two things, one decoding and one for the filter.

Which filter is given to? The nb_filters of InputStream * ist is 1, which actually points to the filter named "buffer" (source file: buffersrc.c). This filter is different from other filters in that it is the first entry for all

filters. After decoding, give it first, and then pass it to the next. Why give it to him first? Quite simply, it is a FIFO for buffering data.

WYOF

AV_BUFFERSRC_ADD_FRAME

This function goes directly to
av_buffersrc_add_frame_internal //buffersrc.c

WYOF

AV_BUFFERSRC_ADD_FRAME

//BUFFERSRC.C

```
static int av_buffersrc_add_frame_internal(AVFilterContext *ctx,
                                         AVFrame *frame, int flags)
{
    //FIFO
    av_fifo_generic_write(s->fifo, &copy, sizeof(copy), NULL);
    if ((flags & AV_BUFFERSRC_FLAG_PUSH))
        if ((ret = ctx->output_pads[0].request_frame(ctx->outputs[0])) < 0)
            return ret;
    return 0;
}
```

After extracting the core code, it can be seen that the frame is written to the FIFO, and then the request_frame of its output_pads [0] is adjusted.

WYOF REQUEST_FRAME //BUFFERSRC.C

```
static int request_frame(AVFilterLink *link)
{
    BufferSourceContext *c = link->src->priv;
    AVFrame *frame;
    int ret;
    av_fifo_generic_read(c->fifo, &frame, sizeof(frame), NULL);
    av_log(NULL, AV_LOG_WARNING, "request_frame, frame-pts %lld
\n", frame->pts);
    ret = ff_filter_frame(link, frame);

    return ret;
}
```

Extract the core code, it can be seen that it reads a frame of data from the FIFO. Then call ff_filter_frame. The link entered at this time is the first AVFilterLink.

WYOF FF_FILTER_FRAME // AVFILTER.C

The function does some basic checks and goes to
`ff_filter_frame_framed`

WYOF

FF_FILTER_FRAME_FRAMED

//AVFILTER.C

```
static int ff_filter_frame_framed(AVFilterLink *link, AVFrame *frame)
{
    //Define a function pointer filter_frame. The function pointed to, the
    //parameters are AVFilterLink *, AVFrame *, and the return value is int
    int (*filter_frame)(AVFilterLink *, AVFrame *);
    AVFilterContext *dstctx = link->dst;
    //The next AVFilterContext, for this example, is the transform filter
    //we wrote ourselves, the source code is in vf_transform.c
    AVFilterPad *dst = link->dstpad;
    AVFrame *out = NULL;
    int ret;

    if (!(filter_frame = dst->filter_frame))
        //Function pointer filter_frame, link-> dstpad is actually dstctx->
        //input_pads, which is defined by the transform filter
        filter_frame = default_filter_frame;
    //300 words omitted
    ret = filter_frame(link, out);
    link->frame_count++;
    ff_update_link_current_pts(link, pts);
    return ret;
}
```

WYOF EXTRACT THE CORE CODE.

Define a function pointer filter_frame. The function pointed to must be AVFilterLink, AVFrame, and the return value is int

filter_frame = dst-> filter_frame

dst = link-> dstpad, and link-> dstpad is actually dstctx-> input_pads, which is the input_pads defined by the transform filter

```
static const AVFilterPad avfilter_vf_transform_inputs[] = {
    {
        .name      = "default",
        .type      = AVMEDIA_TYPE_VIDEO,
        .filter_frame = filter_frame,
    },
    { NULL }
};
```

Therefore, the filter_frame function pointer points to the filter_frame function implemented by vf_transform.c.

WYOF FILTER_FRAME //VF_TRANSFORM.C

various filters defined by ffmpeg, such as
vf_colorbalance.c, vf_scale.c, etc., also have this
function, the same process

```
static int filter_frame(AVFilterLink *link, AVFrame *in)
{
    AVFilterContext *avctx = link->dst;//The first dst AVFilterContext of
    //the link is actually the AVFilterContext of the current filter
    AVFilterLink *outlink = avctx->outputs[0];//The current
    //AVFilterContext, outputs [0] points to the second AVFilterLink
    AVFrame *out;

    //Allocates an empty AVFrame.
    out = ff_get_video_buffer(outlink, outlink->w, outlink->h);
    if (!out) {
        av_frame_free(&in);
        return AVERRORE(ENOMEM);
    }

    //The parameters of the allocated empty buffer are basically the same as
    //the previous one, but the width and height are modified. Of course, if you
    //do n't change the width and height, you do n't need the following two
    //sentences.
    av_frame_copy_props(out, in);
    out->width = outlink->w;
    out->height = outlink->h;
    out->format = outlink->format;
```

```

ThreadData td;
td.in = in;
td.out = out;
int res;
if(res = avctx->internal->execute(avctx, do_conversion, &td, NULL,
FFMIN(outlink->h, avctx->graph->nb_threads))) {
    return res;
} //Enable a child thread to perform a time-consuming transformation.
do_conversion is the transformation we want to do.

av_frame_free(&in);

return ff_filter_frame(outlink, out); //At this time, the input parameters
of ff_filter_frame are different from those previously called by buffersrc.c.
outlink is the second AVFilterLink, and the buffer is a new buffer that has
been transformed.
}

```

Extract the key code, abstract it, and allocate an empty buffer through `ff_get_video_buffer`. This buffer is used to store the result of the transformation and will be passed to the next filter through `ff_filter_frame`. `do_conversion` is a function that actually does the conversion, but in fact, if the processing to be done is not time-consuming, it is not necessary to use another thread for processing. You can do it directly in the `filter_frame`.

The processed new data is placed in `out`, `ff_filter_frame` is called, and passed to the next filter. Note that the oulink of `ff_filter_frame` corresponds to the second AVFilterLink in the figure above.

WYOF WALK INTO FF_FILTER_FRAME AGAIN // AVFILTER.C

As known above, ff_filter_frame does only a few basic checks, and goes to ff_filter_frame_framed. So we look directly at ff_filter_frame_framed

```
static int ff_filter_frame_framed(AVFilterLink *link, AVFrame *frame)
{
    //Define a function pointer filter_frame. The function pointed to, the
    //parameters are AVFilterLink *, AVFrame *, and the return value is int
    int (*filter_frame)(AVFilterLink *, AVFrame *);

    AVFilterContext *dstctx = link->dst;//The next AVFilterContext, for
    //this example, is the system's default third filter, named "format", and the
    //source code is in vf_format.c

    AVFilterPad *dst = link->dstpad;
    AVFrame *out = NULL;
    int ret;

    if (!(filter_frame = dst->filter_frame))//vf_format.c does not implement
    //the filter function because the return is empty
        filter_frame = default_filter_frame;//So the function will go here,
    //the function pointer filter_frame will point to default_filter_frame
    //300 words omitted

    ret = filter_frame(link, out);
    link->frame_count++;
    ff_update_link_current_pts(link, pts);
    return ret;
```

```
}
```

As stated in the comment, because vf_format.c does not implement the filter function, the filter_frame pointer at this time points to defalut_filter_frame.

WYOF default_filter_frame //avfilter.c

```
static int default_filter_frame(AVFilterLink *link, AVFrame *frame)
{
    //This function did nothing, and called ff_filter_frame again. The first
    parameter was replaced by the third AVFilterLink, the second parameter
    was unchanged, and the frame was silently passed out.
    return ff_filter_frame(link->dst->outputs[0], frame);
}
```

At this time link-> dst-> outputs [0] corresponds to the third AVFilterLink in the figure above

WYOF THIRD TIME INTO FF_FILTER_FRAME // AVFILTER.C

```
static int ff_filter_frame_framed(AVFilterLink *link, AVFrame *frame)
{
    //Define a function pointer filter_frame. The function pointed to, the
    parameters are AVFilterLink *, AVFrame *, and the return value is int
    int (*filter_frame)(AVFilterLink *, AVFrame *);

    AVFilterContext *dstctx = link->dst;//The next AVFilterContext, for
    this example, is the last default filter of the system, named "buffersink",
    and the source code is in buffersink.c

    AVFilterPad *dst = link->dstpad;
    AVFrame *out = NULL;
    int ret;
    if (!(filter_frame = dst->filter_frame))//Point to the filter_frame function
    implemented by buffersink.c
        filter_frame = default_filter_frame;
    ret = filter_frame(link, out);
    link->frame_count++;
    ff_update_link_current_pts(link, pts);
    return ret;
}
```

The filter_frame pointer at this time points to the filter_frame function implemented by buffersink.c

WYOF filter_frame //buffersink.c

```
static int add_buffer_ref(AVFilterContext *ctx, AVFrame *ref)
{
    BufferSinkContext *buf = ctx->priv;
    /* cache frame */
    //Store buffer in FIFO
    av_fifo_generic_write(buf->fifo, &ref, FIFO_INIT_ELEMENT_SIZE,
NULL);
    return 0;
}
```

Extract the key code. It is clear that the buffer is actually stored in the FIFO. At this point, clear the ins and outs of filter_frame!

WYOF HOW TO ENCODE FFMPEG AFTER FILTER

When we wrote a filter and processed the video, how did ffmpeg encode it?

Through research, it is found that the source function of the encoding is reap_filters (...), which will be called by the transcode_step (...) function.

```
static int reap_filters(int flush)
{
    AVFrame *filtered_frame = NULL;//This pointer will store a buffer
    processed by the filter and send it to the encoder
    int i;
    /* Reap all buffers present in the buffer sinks */
    for (i = 0; i < nb_output_streams; i++) {//All the way to video, all the
    way to audio, then nb_output_streams = 2
        OutputStream *ost = output_streams[i];
        OutputFile  *of = output_files[ost->file_index];
        AVFilterContext *filter;
        AVCodecContext *enc = ost->enc_ctx;
        int ret = 0;
        if (!ost->filter) continue;
        filter = ost->filter->filter;//OutputStream's filter pointer points to the
        AVFilterContext defined by buffersink.c.
        if (!ost->filtered_frame && !(ost->filtered_frame =
        av_frame_alloc())))
            return AVERROR(ENOMEM);
    }
```

```

filtered_frame = ost->filtered_frame;

while (1) {
    double float_pts = AV_NOPTS_VALUE; // this is identical to
filtered_frame.pts but with higher precision
    //av_buffersink_get_frame_flags is defined in buffersink.c and is
used to read a frame from the FIFO
    ret = av_buffersink_get_frame_flags(filter, filtered_frame,
                                         AV_BUFFERSINK_FLAG_NO_RE
    if (ret < 0) {
        //Omitted, check ret
        //If ret <0, it is not another error, then it is considered that
there is no data, and the loop is out
        break;
    }
    switch (filter->inputs[0]->type) {
        case AVMEDIA_TYPE_VIDEO:
            //do_video_out function will do video encoding
            do_video_out(of->ctx, ost, filtered_frame, float_pts);
            break;
        case AVMEDIA_TYPE_AUDIO:
            //do_audio_out function will do audio encoding
            do_audio_out(of->ctx, ost, filtered_frame);
            break;
        default:
            // TODO support subtitle filters
            av_assert0(0);
    }
    av_frame_unref(filtered_frame);
}
}
return 0;
}

```

As mentioned in the previous section, the final result of filter_frame (...) is that the buffer is stored in the FIFO

of buffersink.c.

Well, this section is actually a process of reading data from the buffersink's FIFO and encoding it.

As can be seen from the above, the `av_buffersink_get_frame_flags` function reads one frame of data from buffersink and puts it into `filtered_frame`.

WYOF DO_VIDEO_OUT //FFMPEG.C

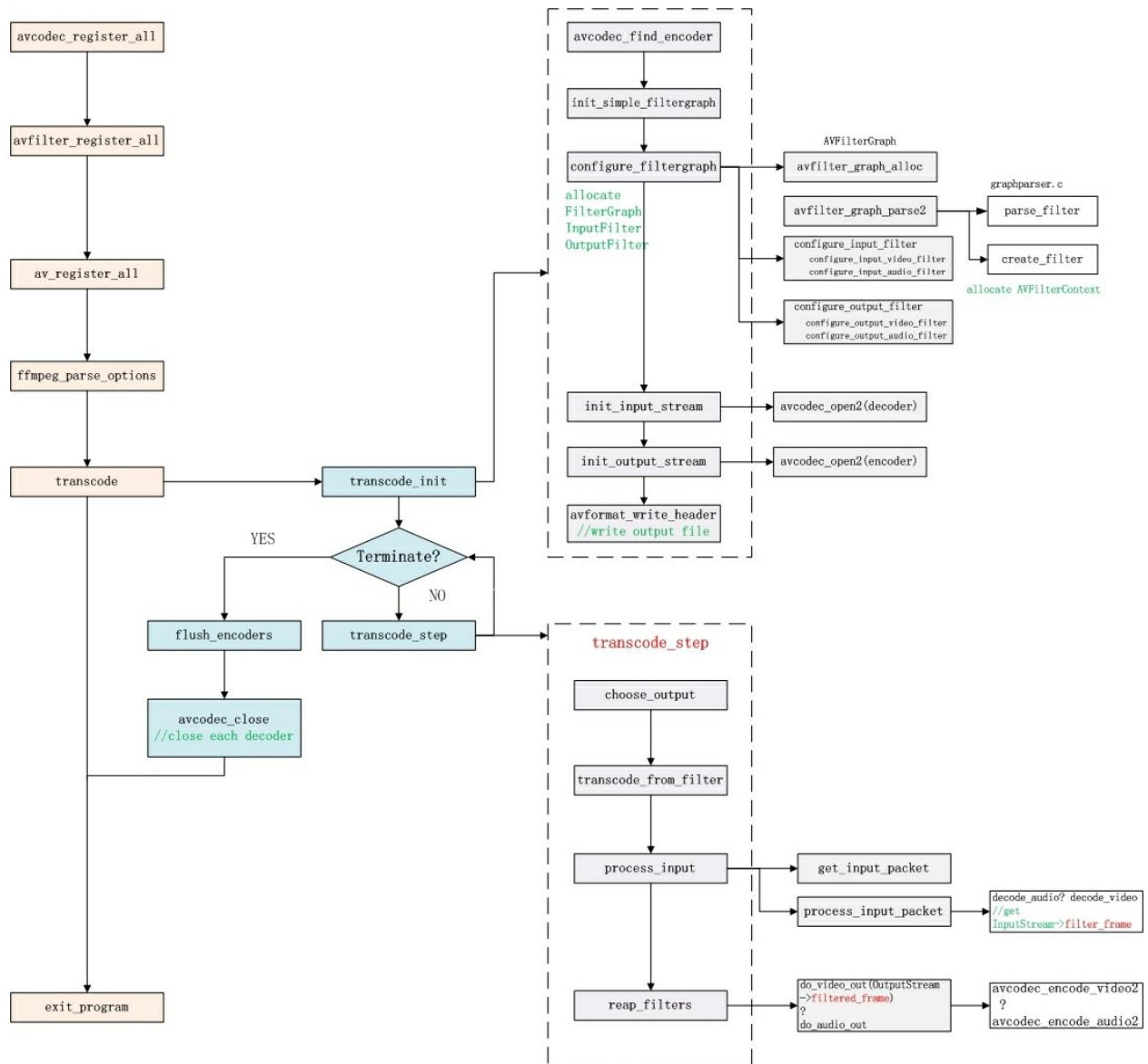
```
static void do_video_out(AVFormatContext *s,
                        OutputStream *ost,
                        AVFrame *next_picture,
                        double sync_ipts)
{
    int ret;
    AVCodecContext *enc = ost->enc_ctx;
    int nb_frames, nb0_frames, i;

    for (i = 0; i < nb_frames; i++) {
        AVFrame *in_picture;
        if (i < nb0_frames && ost->last_frame) {
            in_picture = ost->last_frame;
        } else
            in_picture = next_picture;
        ost->frames_encoded++;
        //Start coding
        ret = avcodec_encode_video2(enc, &pkt, in_picture, &got_packet);
    }
}
```

This function is long and does a lot of chores, but the key code is to call the encoding function
avcodec_encode_video2

WYOF FUNCTION FLOWCHART

After so long talking, it's time for a big move! The following is the function flowchart of ffmpeg when using filter, which mainly pulls out the functions related to filter!



`transcode_init()` is mainly used to initialize the various structures mentioned above. `Transcode_step` mainly works:

Decoding-> send filter-> encoding-> continue decoding .

METADATA

Metadata is data about data. Every single digital artifact has it. It describes the who, what, when, where, how, and sometimes even, why, for any document, video, photo, or sound clip. This information comes in handy sometimes, like when you're flipping through old pictures by date, or by location. But in the wrong hands, this same information could be damaging.

Metadata exists in the parts of images, videos, or music that we can't experience as humans. But if you pry into any digital artifact, you can see metadata as a list of keys (or tags) and their corresponding values. One of the simplest tags is "Creation Date," which naturally points to the time when its creator pushed the shutter button, or pressed record. Other interesting tags include the "Make" and "Model" tags, which can tell you what type of camera or computer was used to create the media. There are dozens of such tags, and each one can help tell a very distinct story; this is why understanding how rich metadata is can better help protect the identities of sources who have shared their digital

media with you.

Example metadata:

```
ffmpeg -i track05.wav \
-metadata title="This is the title" \
-metadata author="Made by Me" \
-metadata copyright="Copyright 2009 Me"
-metadata comment="An exercise in Realmedia metadata" \
-y track05.rm
```

This is what the start of the file looks like in a hex editor:

```
0040 00 01 00 03 43 4F 4E 54 00 00 00 5F 00 00 00 11 ....CONT..._....
0050 54 68 69 73 20 69 73 20 74 68 65 20 74 69 74 6C This is the titl
0060 65 00 0A 4D 61 64 65 20 62 79 20 4D 65 00 11 43 e..Made by
Me..C
0070 6F 70 79 72 69 67 68 74 20 32 30 30 39 20 4D 65 opyright 2009
Me
0080 00 21 41 6E 20 65 78 65 72 63 69 73 65 20 69 6E .!An exercise
in
0090 20 52 65 61 6C 6D 65 64 69 61 20 6D 65 74 61 64 Realmedia
metad
00A0 61 74 61 4D 44 50 52 00 00 00 9B 00 00 00 00
ataMDPR.....
```

READ METADATA SAMPLE

```
#include <stdio.h>
#include <libavformat/avformat.h>
#include <libavutil/dict.h>
int main (int argc, char **argv)
{
AVFormatContext *fmt_ctx = NULL;
AVDictionaryEntry *tag = NULL;
int ret;
if (argc != 2) {
printf("usage: %s <input_file>\n"
"example program to demonstrate the use of the libavformat metadata
API.\n"
"\n", argv[0]);
return 1;
}
av_register_all();
if ((ret = avformat_open_input(&fmt_ctx, argv[1], NULL, NULL)))
return ret;
while ((tag = av_dict_get(fmt_ctx->metadata, "", tag,
AV_DICT_IGNORE_SUFFIX)))
printf("%s=%s\n", tag->key, tag->value);
avformat_close_input(&fmt_ctx);
return 0;
}
```

WRITE METADATA SAMPLE

```
#include <iostream>
#ifndef __cplusplus
extern "C" {
#endif

#include <libavformat/avformat.h>

#ifndef __cplusplus
}
#endif

int main(int argc, char **argv) {

    av_register_all();
    avcodec_register_all();
    AVFormatContext* ctx;
    std::string path("./input.mp3");

    ctx = avformat_alloc_context();

    if (avformat_open_input(&ctx, path.c_str(), 0, 0) < 0)
        std::cout << "error1" << std::endl;

    if (avformat_find_stream_info(ctx, 0) < 0)
        std::cout << "error2" << std::endl;

    AVDictionaryEntry *tag = nullptr;
    tag = av_dict_get(ctx->metadata, "artist", tag,
                      AV_DICT_IGNORE_SUFFIX);
    std::cout << tag->key << " : " << tag->value << std::endl;
```

```

av_dict_set(&ctx->metadata, "TPFL", "testtest", 0);
std::cout << "test!" << std::endl;
int status;
AVFormatContext* ofmt_ctx;
AVOutputFormat* ofmt = av_guess_format("mp3", "./out.mp3",
NULL);
status = avformat_alloc_output_context2(&ofmt_ctx, ofmt, "mp3",
"./out.mp3");
if (status < 0) {
    std::cerr << "could not allocate output format" << std::endl;
    return 0;
}
int audio_stream_index = 0;
for (unsigned i = 0; i < ctx->nb_streams; i++) {
    if (ctx->streams[i]->codecpar->codec_type ==
AVMEDIA_TYPE_AUDIO) {
        audio_stream_index = i;
        const AVCodec *c = avcodec_find_encoder(ctx->streams[i]-
>codecpar->codec_id);
        if (c) {
            AVStream *ostream = avformat_new_stream(ofmt_ctx, c);
            avcodec_parameters_copy(ostream->codecpar, ctx->streams[i]->codecpar);
            ostream->codecpar->codec_tag = 0;
        }
        break;
    }
}
av_dict_set(&ofmt_ctx->metadata, "TPFL", "testtest", 0);
av_dump_format(ofmt_ctx, 0, "./out.mp3", 1);

if (!(ofmt_ctx->oformat->flags & AVFMT_NOFILE)) {
    avio_open(&ofmt_ctx->pb, "./out.mp3", AVIO_FLAG_WRITE);
}

if (avformat_init_output(ofmt_ctx, NULL) ==
AVSTREAM_INIT_IN_WRITE_HEADER) {
    status = avformat_write_header(ofmt_ctx, NULL);
}

```

```
}

AVPacket *pkt = av_packet_alloc();
av_init_packet(pkt);
pkt->data = NULL;
pkt->size = 0;

while (av_read_frame(ctx, pkt) == 0) {
    if (pkt->stream_index == audio_stream_index) {
        // this is optional, we are copying the stream
        av_packet_rescale_ts(pkt, ctx->streams[audio_stream_index]-
>time_base,
                           ofmt_ctx->streams[audio_stream_index]->time_base);
        av_write_frame(ofmt_ctx, pkt);
    }
}

av_packet_free(&pkt);
av_write_trailer(ofmt_ctx);
avformat_close_input(&ctx);
avformat_free_context(ofmt_ctx);
avformat_free_context(ctx);
if(status == 0)
    std::cout << "test1" << std::endl;
return 0;
}
```

3. AUDIO BASICS

AUDIO INTRODUCTION

The digital audio system reproduces the original sound by converting the sound wave waveform into a series of binary data. The equipment used to implement this step is an analog-to-digital converter (A / D), which samples sound waves at a rate of tens of thousands of times per second.

Each sample records the state of the original analog sound wave at a certain time, which is called the sample. A series of samples can be connected to describe a section of sound waves. The number of samples per second is called the sampling frequency or recovery rate, and the unit is HZ (Hertz). The higher the sampling frequency, the higher the sonic frequency that can be described.

The sampling rate determines the range of sound frequencies (equivalent to pitch) and can be represented by digital waveforms. The frequency range represented by the waveform is often called the bandwidth. To correctly understand audio sampling can be divided into the number of samples and the frequency of sampling.

MONO

The so-called mono is that the sound is produced by only one speaker. The listener can clearly hear that the source of the sound is the position of the speaker. The performance of the sound is relatively dull. When it comes to information, we can clearly feel that the sound is transmitted to our ears from the middle of the two speakers.

Mono to stereo sample:

```
unsigned char *mono2stereo(unsigned char *buf_mono, int mono_size, int
bit, int *stereo_size)
{
    int i, j;
    unsigned char *pszBufStereo = NULL;
    *stereo_size = mono_size * 2;
    pszBufStereo = ortp_malloc(*stereo_size);
    if (pszBufStereo == NULL)
    {
        printf("mono2stereo(): Allocate stereo buffer error, size %d\n",
*stereo_size);
        return NULL;
    }
    i = 0;
    j = 0;
    switch (bit>>3)
    {
        case 1:
```

```
for (; mono_size > 0; mono_size--)
{
    pszBufStereo[i++] = buf_mono[j];
    pszBufStereo[i++] = buf_mono[j];
    j++;
}
break;

case 2:
for (; mono_size > 0; mono_size -= 2)
{
    pszBufStereo[i++] = buf_mono[j];
    pszBufStereo[i++] = buf_mono[j+1];
    pszBufStereo[i++] = buf_mono[j];
    pszBufStereo[i++] = buf_mono[j+1];
    j += 2;
}
break;

case 3:
for (; mono_size > 0; mono_size -= 3)
{
    pszBufStereo[i++] = buf_mono[j];
    pszBufStereo[i++] = buf_mono[j+1];
    pszBufStereo[i++] = buf_mono[j+2];
    pszBufStereo[i++] = buf_mono[j];
    pszBufStereo[i++] = buf_mono[j+1];
    pszBufStereo[i++] = buf_mono[j+2];
    j += 3;
}
break;

case 4:
for (; mono_size > 0; mono_size -= 4)
{
    pszBufStereo[i++] = buf_mono[j];
    pszBufStereo[i++] = buf_mono[j+1];
    pszBufStereo[i++] = buf_mono[j+2];
```

```
    pszBufStereo[i++] = buf_mono[j+3];
    pszBufStereo[i++] = buf_mono[j];
    pszBufStereo[i++] = buf_mono[j+1];
    pszBufStereo[i++] = buf_mono[j+2];
    pszBufStereo[i++] = buf_mono[j+3];
    j += 4;
}
break;

default:
printf("mono2stereo(): Unsupport bits-per-sample: %i\n", bit);
ortp_free(pszBufStereo);
return NULL;
}
return pszBufStereo;
}
```

STEREO

It uses two independent channels for recording, and the whole process does not add any sound processing. The reproduction of a stereo system requires a pair of speakers. By adjusting the size of the sound emitted by the two speakers in the system, we mistakenly believe that the sound source comes from any position in the straight line between the two speakers. Especially when using headphones, because the crosstalk of the left and right sides of the sound rarely occurs, the positioning of the sound is more accurate; coupled with the more realistic sound field feel, its expressiveness is much more realistic than mono. However , the shortcomings of Stereo are also very obvious. The most obvious is that the position of the speakers is required. The poor positioning will directly affect the sound expression.

Stereo to mono sample:

```
int stereo2mono(unsigned char *buf_stereo, int stereo_size, int bit)
{
    int i = 0;
    int j = 0;
    switch (bit>>3){
```

```
case 1:  
    for (; j < stereo_size; j += 2) {  
        buf_stereo[i++] = buf_stereo[j];  
    }  
    break;  
case 2:  
    for (; j < stereo_size; j += 4) {  
        buf_stereo[i++] = buf_stereo[j];  
        buf_stereo[i++] = buf_stereo[j+1];  
    }  
    break;  
case 3:  
    for (; j < stereo_size; j += 6) {  
        buf_stereo[i++] = buf_stereo[j];  
        buf_stereo[i++] = buf_stereo[j+1];  
        buf_stereo[i++] = buf_stereo[j+2];  
    }  
    break;  
case 4:  
    for (; j < stereo_size; j += 8) {  
        buf_stereo[i++] = buf_stereo[j];  
        buf_stereo[i++] = buf_stereo[j+1];  
        buf_stereo[i++] = buf_stereo[j+2];  
        buf_stereo[i++] = buf_stereo[j+3];  
    }  
    break;  
default:  
    printf("stereo2mono(): Unsupport bits-per-sample: %i\n",  
bit);  
    return 0;  
}  
return stereo_size >> 1;  
}
```

3D SURROUND

Sometimes called 3D enhanced stereo (3D Enhancement). It is an analog surround sound system. The stereo signals of the left and right channels are processed by digital signals to generate a three-dimensional surround sound field effect through the left and right speakers. It uses general two-channel to create a surround sound field with a three-dimensional feel. It is better than stereo, but it is quite different from the coded surround system.

NUMBER OF SAMPLES

The number of sampling bits can be understood as the resolution of the sound processed by the acquisition card. The larger the value, the higher the resolution and the more realistic the recorded and played back sound. We first need to know: the sound file in the computer is represented by the numbers 0 and 1. Continuous analog signals (windowed and truncated) are sampled by digital pulses at a certain sampling frequency, and each discrete pulse signal is quantized into a series of binary coded streams with a certain quantization accuracy. The number of bits in this string of encoded streams is the number of samples, also known as the quantization accuracy. The relationship between the bit rate and the number of sampling bits can be clearly seen from the bit rate calculation formula:

Bit rate = sampling frequency × quantization accuracy
× number of channels.

FREQUENCY OF SAMPLING

Sampling frequency refers to the number of times a sound device samples a sound signal in one second. The higher the sampling frequency, the more realistic and natural the sound is.

In today's mainstream capture cards, the sampling frequency is generally divided into three levels: 22.05KHz, 44.1KHz, and 48KHz.

22.05 KHz can only reach the sound quality of FM broadcasting,

44.1KHz is the theoretical CD sound quality limit.

48KHz is more accurate.

The human ear cannot discern the sampling frequency higher than 48KHz, so there is not much use value on the computer.

BIT SPEED

Bit rate refers to the amount of information that can be passed per second in a data stream. You may have seen situations where audio files were described using "128–Kbps MP3" or "64–Kbps WMA". Kbps stands for "kilobits per second", so a larger value means more data:

A 128–Kbps MP3 audio file contains twice as much data and takes up twice as much space as a 64–Kbps WMA file.

(But in this case, the two files sound no different. what is the reason? Some file formats use data more efficiently than others, and the sound quality of 64–Kbps WMA files is the same as the sound quality of 128–Kbps MP3.)

The important point to understand is that the higher the bit rate, the greater the amount of information, the greater the amount of processing needed to decode this information, and the more space the file will occupy. Choosing the appropriate bit rate for your project depends on the playback target: If you want to play the produced VCD on a DVD player, the video must be

1150 Kbps and the audio must be 224 Kbps. A typical 206 MHz Pocket PC supports up to 400 Kbps of MPEG video. The exception occurs when playing beyond this limit.

VBR (VARIABLE BITRATE)

That is, there is no fixed bit rate. The compression software determines what bit rate to use in real time based on the audio data during compression. This is an algorithm developed by Xing. They encode the complex parts of a song with a high bitrate and the simple parts with a low bitrate.

Although the idea is good, unfortunately the VBR algorithm of the Xing encoder is very poor, and the sound quality is far from CBR. Fortunately, Lame perfectly optimized the VBR algorithm to make it the best encoding mode for MP3. This is a way of considering file size with quality as the prerequisite, and recommends the encoding mode. ABR (Average Bitrate) is an interpolation parameter of VBR. Lame created this encoding mode based on the characteristics of CBR's poor file volume ratio and VBR's variable file size. ABR is also called "Safe VBR", it is within a specified average Bitrate, with a period of every 50 frames (30 frames about 1 second). Low frequency and insensitive frequencies use relatively low traffic, high

frequency and large dynamic performance. When using high traffic. For example, when specifying a 192kbps ABR to encode a wav file, Lame will fixedly encode 85% of the file with 192kbps, and then dynamically optimize the remaining 15%: The complex part is encoded above 192kbps, and the simple part is encoded below 192kbps. Compared with 192kbps CBR, 192kbps ABR is similar in file size, but the sound quality is improved a lot. ABR coding is 2 to 3 times faster than VBR coding, and the quality is better than CBR in the range of 128-256kbps. Can be used as a compromise between VBR and CBR.

CBR (CONSTANT BITRATE)

CBR (Constant Bitrate) is a constant bit rate, which means that the file is a bit rate from beginning to end. Compared to VBR and ABR, the compressed file is very large, but the sound quality will not be significantly improved. Bitrate is the most important factor for MP3. It is used to indicate how many bits (bit per second, referred to as bps) are occupied by audio data per second. The higher this value, the better the sound quality.

MP3

The full name of MP3 should be MPEG1 Layer-3 audio file, MPEG (Moving Picture Experts Group) is translated into Chinese as a moving picture expert group, specifically referring to the standard of moving video compression.

MPEG audio file is the sound part of MPEG1 standard, also called MPEG audio layer,

It is divided into three layers according to compression quality and coding complexity, namely Layer-1, Layer2, Layer3, And corresponding to three audio files MP1, MP2, MP3, and use different levels of encoding according to different purposes.

The higher the level of MPEG audio coding, the more complicated the encoder, and the higher the compression ratio. The compression ratios of MP1 and MP2 are 4: 1 and 6: 1-8: 1, respectively.

The compression rate of MP3 is as high as 10: 1-12: 1, which means that one minute of CD-quality music requires 10MB of storage space without compression.

After MP3 compression encoding, it is only about 1MB. However, MP3 uses lossy compression for audio signals. In order to reduce sound distortion, MP3 adopts "sensory coding technology", that is, spectrum analysis is performed on audio files first, Then use a filter to filter out the noise level, and then quantize the remaining bits and arrange them.

Finally, an MP3 file with a higher compression ratio is formed, and the compressed file can achieve a sound effect closer to the original sound source during playback. (Another MP3PRO: The mp3PRO encoder divides the audio recording into two parts: the mp3 part and the PRO part. The mp3 part analyzes the Low Frequency Band information and encodes it into a normal mp3 file data stream. This enables the encoder to focus less useful information and obtain better quality encoding results. At the same time, this also ensures the compatibility of mp3PRO files with older mp3 players. The PRO part analyzes the High Frequency Band information and encodes it as part of the mp3 data stream. These are usually ignored in older mp3 decoders. The new mp3PRO decoder will effectively use this part of the data stream, combining the two segments (high-band and low-band) to produce a complete audio band, to enhance the sound quality.

DESCRIPTION OF AUDIO FORMATS

It is often described like this: 44100HZ 16bit stereo or 22050HZ 8bit mono and so on.

44100HZ 16bit stereo: 44100 samples per second, sampled data is recorded with 16 bits (2 bytes), two channels (stereo);

22050HZ 8bit mono: 22050 samples per second, sampled data is recorded with 8 bits (1 byte), mono; Of course, there can also be 16-bit mono or 8-bit stereo, and so on.

Sampling rate: refers to the number of times a sound signal is sampled in a unit time during the "Analog-to-Digital" conversion.

Sampling value: It means the integrated value of the sound analog signal in each sampling period.

For a mono sound file, the sample data is an eight-bit short integer (short int 00H-FFH); For a two-channel stereo sound file, each sample data is a 16-bit integer (int), and the upper eight bits (left channel) and the lower eight bits (right channel) respectively represent two channels. The frequency recognition range of

humans is 20HZ-20000HZ. If the sound can be 20,000 samples per second, it can meet the needs of the human ear during playback. Therefore, the sampling frequency of 22050 is commonly used, 44100 is already CD sound quality, and more than 48000 sampling has no meaning to the human ear. This is similar to the movie's 24 frames per second picture.

Each sample data records the amplitude, and the sampling accuracy depends on the size of the storage space:

1 byte (that is, 8bit) can only record 256 numbers, that is, it can only divide the amplitude into 256 levels;

2 bytes (that is, 16bit) can be as thin as 65536, which is already a CD standard;

4 bytes (i.e. 32bit) can subdivide the amplitude to 4294967296 levels, which is really unnecessary.

If it is stereo, the samples are duplicated and the file is almost double the size.

In this way, we can estimate the playback length of a wav file based on the size, sampling frequency, and sampling size of a wav file. For example,

The file length of "Windows XP Startup.wav" is 424,644 bytes, and it is in "22050HZ / 16bit / stereo" format (this can be seen in its "Properties->Summary"),

Then its transmission rate per second (bit rate, also called bit rate, sampling rate) is $22050 * 16 * 2 = 705600$ (bit / s),

When converted into bytes, it is $705600/8 = 88200$ (bytes / second).

Playing time: 424644 (total bytes) / 88200 (bytes per second) ≈ 4.8145578 (seconds).

But this is not precise enough,

WAVE files (*.wav) in the standard PCM format are packed with at least 42 bytes of header information, which should be removed when calculating the playback time. So there is: $(424644-42) / (22050 * 16 * 2/8) \approx 4.8140816$ (seconds). This is more accurate.

There is another concept about sound files: "bit rate", also called bit rate, sampling rate, For example, the bit rate of the above file is 705.6kbps or 705600bps, where b is bit and ps means per second; Compressed audio files are often expressed in bit rate. For example, MP3 that achieves CD sound quality is: 128kbps / 44100HZ.

PCM DATA FORMAT

PCM (Pulse-code-modulation), also known as pulse code modulation. It is an expression after the analog signal is converted into a digital signal at a fixed sampling frequency. The sound data in the PCM is not compressed. Generally, five data are used to describe a PCM data. These five data are needed to play a PCM file.

Sample Rate:

The sampling frequency unit is: Hz. The higher the sampling frequency, the better the audio quality and the larger the footprint.

Sign:

Whether the audio data is signed. It is usually signed. If you use signed data as unsigned data, the sound will be very stingy.

Sample Size:

Represents the size of each sampled data. This value is usually 16-bit.

Byte Ordering:

Endianness refers to little-endian or big-endian. Represents the storage byte order of audio

data. It is usually little-endian.

Number of Channels:

Identifies whether the audio is mono (1 channel) or stereo (2 channels).

Storage of PCM data, as shown in the figure:

<i>8-bit Mono</i>	Sample 1	Sample 2	Sample 3	Sample 4
--------------------------	----------	----------	----------	----------

<i>8-bit Stereo</i>	<i>Left Channel</i>	<i>Right Channel</i>	<i>Left Channel</i>	<i>Right Channel</i>
	Sample 1	Sample 1	Sample 2	Sample 2

<i>16-bit Mono</i>	<i>LSB</i>	<i>MSB</i>	<i>LSB</i>	<i>MSB</i>
	Sample 1		Sample 2	

<i>16-bit Stereo</i>	<i>Left Channel</i>		<i>Right Channel</i>	
	<i>LSB</i>	<i>MSB</i>	<i>LSB</i>	<i>MSB</i>
	Sample 1		Sample 1	

PCM STREAM

Mono:

```
+ ----- + ----- + ----- + ----- + ----- + ----- +  
----- + ----- +  
| 500 | 300 | -100 | -20 | -300 | 900 | -200 | -50 | 250 |  
+ ----- + ----- + ----- + ----- + ----- + ----- +  
----- + ----- +
```

Each integer occupies 2 bytes (16-bit), 9 samples is 18 bytes of data.

The minimum integer size of each sample is -32768 and the maximum is 32768.

If you draw a graph based on the position and value of the sampled data, you will get a wave graph like that on the player.

We can read the data into a C language array like the following pseudo code example:

```
FILE * pcmfile  
int16_t * pcmdat;  
pcmfile = fopen (your pcm data file);  
pcmdat = malloc (size of the file);  
fread (pcmdat, sizeof (int16_t), size of file / sizeof  
(int16_t), pcmfile);
```

If we send this data to the sound card, we can hear the sound.

Of course we need to tell the sound card the sampling rate of these data.

If we tell the sound card that the sample rate is greater than the data itself, the playback speed of these data will be higher than its original speed. It's a fast-release function.

stereo:

```
+ ----- + ----- + ----- + ----- + ----- - + ----- + -  
----- + ----- + ----- +  
LFrame1 | RFrame1 | LFrame2 | RFrame2 | LFrame3 | RFrame3 |  
LFrame4 | RFrame4 | LFrame5 |  
+ ----- + ----- + ----- + ----- + ----- - + ----- + -  
----- + ----- + ----- +
```

Each frame is a 16-bit sampling point. The data of the left and right channels are cross-stored.

PCM AUDIO PROCESSING

1. SEPARATE LEFT AND RIGHT CHANNEL AUDIO

This sample introduces the separation of the left and right channels in the PCM16LE two-channel data into two files. The PCM audio data can be imported and viewed using audio editing software. Audacity, a free and open source audio editing software is recommended.

The sampling values of the left and right channels in the PCM16LE two-channel data are stored at intervals. The sampling frequency of the sound samples in this sample is always 44100Hz, and the sampling format is always 16LE. "16" indicates that the number of sampling bits is 16 bits. Since 1Byte = 8bit, one sample value of one channel occupies 2Byte. "LE" stands for Little Endian, and the storage method of the 2 Byte sample value is stored in the high address. The following figure is the waveform diagram of the input two-channel PCM data. The upper waveform is the left channel, and the lower waveform is the right

```

#define FALSE 0
#define TRUE 1
#define OLD_FILE_PATH "old.pcm"
#define LEFT_FILE_PATH "left.pcm"
#define RIGHT_FILE_PATH "right.pcm"

void pcm_lr_separate(void)
{
    char buf[16] = {0};
    static int leftFlag = FALSE;
    int size = 0;

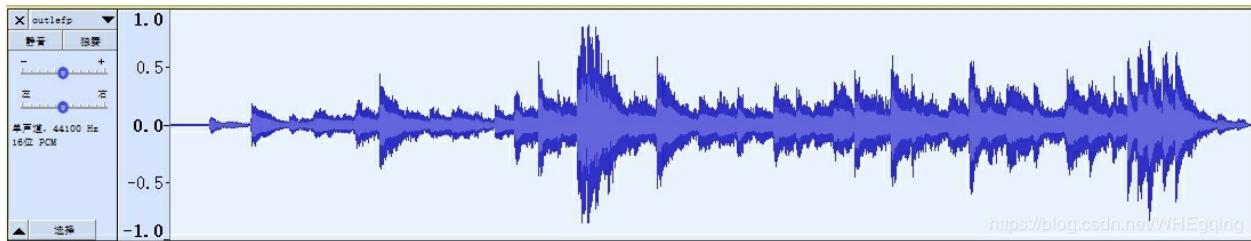
    FILE *fp = fopen(OLD_FILE_PATH, "rb+");
    FILE *fp_l = fopen(LEFT_FILE_PATH, "wb+");
    FILE *fp_r = fopen(RIGHT_FILE_PATH, "wb+");

    while(!feof(fp))
    {
        size = fread(buf, 1, 2, fp);
        if( (size>0) && (leftFlag == FALSE) )
        {
            leftFlag = TRUE;
            fwrite(buf, 1, size, fp_l);
        }
        else if( (size>0) && (leftFlag == TRUE) )
        {
            leftFlag = FALSE;
            fwrite(buf, 1, size, fp_r);
        }
    }

    fclose(fp);
    fclose(fp_l);
    fclose(fp_r);
}

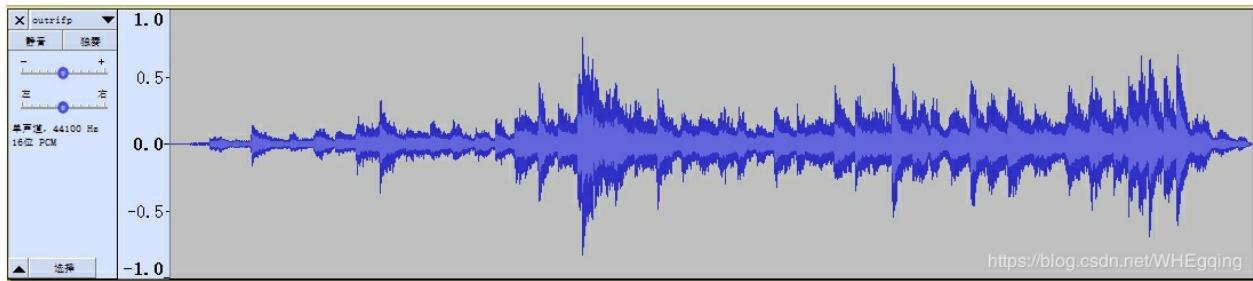
```

The audio waveform of the left channel is as follows:



Left channel waveform chart

THE AUDIO WAVEFORM
OF THE RIGHT CHANNEL
IS AS FOLLOWS:



Right
channel waveform

2. AUDIO PCM RESAMPLING 48000 TO 44100

Some work needs to be saved as an FLV file. During the saving process, the 48000 sampling rate does not meet the FLV packaging standard (up to 44100), so here we call ffmpeg to resample pcm and save the file.

```
extern "C"
{
#include <stdio.h>
#include <libavutil/opt.h>
#include <libavutil/channel_layout.h>
#include <libavutil/samplefmt.h>
#include <libswresample/swresample.h>

static int get_format_from_sample_fmt(const char **fmt,
                                      enum AVSampleFormat sample_fmt)
{
    int i;
    struct sample_fmt_entry {
        enum AVSampleFormat sample_fmt; const char *fmt_be, *fmt_le;
    } sample_fmt_entries[] = {
        { AV_SAMPLE_FMT_U8, "u8", "u8" },
        { AV_SAMPLE_FMT_S16, "s16be", "s16le" },
        { AV_SAMPLE_FMT_S32, "s32be", "s32le" },
    };
}
```

```

    { AV_SAMPLE_FMT_FLT, "f32be", "f32le" },
    { AV_SAMPLE_FMT_DBL, "f64be", "f64le" },
};

*fmt = NULL;

for (i = 0; i < FF_ARRAY_ELEMS(sample_fmt_entries); i++) {
    struct sample_fmt_entry *entry = &sample_fmt_entries[i];
    if (sample_fmt == entry->sample_fmt) {
        *fmt = AV_NE(entry->fmt_be, entry->fmt_le);
        return 0;
    }
}

fprintf(stderr,
        "Sample format %s not supported as output format\n",
        av_get_sample_fmt_name(sample_fmt));
return AVERROR(EINVAL);
}

/***
* Fill dst buffer with nb_samples, generated starting from t.
*/
static void fill_samples(double *dst, int nb_samples, int nb_channels, int
sample_rate, double *t)
{
    int i, j;
    double tincr = 1.0 / sample_rate, *dstp = dst;
    const double c = 2 * M_PI * 440.0;

    /* generate sin tone with 440Hz frequency and duplicated channels */
    for (i = 0; i < nb_samples; i++) {
        *dstp = sin(c * *t);
        for (j = 1; j < nb_channels; j++)
            dstp[j] = dstp[0];
        dstp += nb_channels;
        *t += tincr;
    }
}

```

```
int main(int argc, char **argv)
{
    FILE *pInputFile = fopen("huangdun_r48000_FMT_S16_c2.pcm",
"rb");
    int64_t src_ch_layout = AV_CH_LAYOUT_STEREO, dst_ch_layout =
AV_CH_LAYOUT_STEREO;//AV_CH_LAYOUT_SURROUND;
    int src_rate = 48000, dst_rate = 44100;
    uint8_t **src_data = NULL, **dst_data = NULL;
    int src_nb_channels = 0, dst_nb_channels = 0;
    int src_linesize, dst_linesize;
    int src_nb_samples = 1024, dst_nb_samples, max_dst_nb_samples;
    enum AVSampleFormat src_sample_fmt = AV_SAMPLE_FMT_S16,
dst_sample_fmt = AV_SAMPLE_FMT_S16;
    const char *dst_filename = NULL;
    FILE *dst_file;
    int dst_bufsize;
    const char *fmt;
    struct SwrContext *swr_ctx;
    double t;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s output_file\n"
"API example program to show how to resample an audio stream with
libswresample.\n"
"This program generates a series of audio frames, resamples them to a
specified "
        "output format and rate and saves them to an output file named
        output_file.\n",
        argv[0]);
        exit(1);
    }
    dst_filename = argv[1];

    dst_file = fopen(dst_filename, "wb");
    if (!dst_file) {
        fprintf(stderr, "Could not open destination file %s\n", dst_filename);
        exit(1);
    }
```

```

}

/* create resampler context */
swr_ctx = swr_alloc();
if (!swr_ctx) {
    fprintf(stderr, "Could not allocate resampler context\n");
    ret = AVERROR(ENOMEM);
    goto end;
}

/* set options */
av_opt_set_int(swr_ctx, "in_channel_layout", src_ch_layout, 0);
av_opt_set_int(swr_ctx, "in_sample_rate", src_rate, 0);
av_opt_set_sample_fmt(swr_ctx, "in_sample_fmt", src_sample_fmt, 0);

av_opt_set_int(swr_ctx, "out_channel_layout", dst_ch_layout, 0);
av_opt_set_int(swr_ctx, "out_sample_rate", dst_rate, 0);
av_opt_set_sample_fmt(swr_ctx, "out_sample_fmt", dst_sample_fmt,
0);

/* initialize the resampling context */
if ((ret = swr_init(swr_ctx)) < 0) {
    fprintf(stderr, "Failed to initialize the resampling context\n");
    goto end;
}

/* allocate source and destination samples buffers */

src_nb_channels =
av_get_channel_layout_nb_channels(src_ch_layout);
ret = av_samples_alloc_array_and_samples(&src_data, &src_linesize,
src_nb_channels, src_nb_samples, src_sample_fmt, 0);
if (ret < 0) {
    fprintf(stderr, "Could not allocate source samples\n");
    goto end;
}

/* compute the number of converted samples: buffering is avoided
 * ensuring that the output buffer will contain at least all the

```

```

/* converted input samples */
max_dst_nb_samples = dst_nb_samples =
    av_rescale_rnd(src_nb_samples, dst_rate, src_rate,
AV_ROUND_UP);
/* buffer is going to be directly written to a rawaudio file, no alignment
*/
dst_nb_channels =
av_get_channel_layout_nb_channels(dst_ch_layout);
ret = av_samples_alloc_array_and_samples(&dst_data, &dst_linesize,
dst_nb_channels, dst_nb_samples, dst_sample_fmt, 0);
if (ret < 0) {
    fprintf(stderr, "Could not allocate destination samples\n");
    goto end;
}
t = 0;
int iRealRead;
do {
    /* generate synthetic audio */
    //fill_samples((double *)src_data[0], src_nb_samples,
src_nb_channels, src_rate, &t);
    iRealRead = fread((double*)src_data[0], 1, 4096, pInputFile);

    /* compute destination number of samples */
    dst_nb_samples = av_rescale_rnd(swr_get_delay(swr_ctx, src_rate)
+
src_nb_samples, dst_rate, src_rate,
AV_ROUND_UP);
    if (dst_nb_samples > max_dst_nb_samples) {
        av_freep(&dst_data[0]);
        ret = av_samples_alloc(dst_data, &dst_linesize,
dst_nb_channels,
dst_nb_samples, dst_sample_fmt, 1);
        if (ret < 0)
            break;
        max_dst_nb_samples = dst_nb_samples;
    }
}

```

```

    /* convert to destination format */
printf("src_nb_samples:%d, dst_nb_samples:%d\n",src_nb_samples,
dst_nb_samples);
ret = swr_convert(swr_ctx, dst_data, dst_nb_samples, (const uint8_t
**)src_data, src_nb_samples);
    if (ret < 0) {
        fprintf(stderr, "Error while converting\n");
        goto end;
    }
    dst_bufsize = av_samples_get_buffer_size(&dst_linesize,
dst_nb_channels,
                                         ret, dst_sample_fmt, 1);
    if (dst_bufsize < 0) {
        fprintf(stderr, "Could not get sample buffer size\n");
        goto end;
    }
printf("t:%f in:%d out:%d ,dst_bufsize:%d\n", t, src_nb_samples, ret,
dst_bufsize);
    fwrite(dst_data[0], 1, dst_bufsize, dst_file);
} while (iRealRead>0);

if ((ret = get_format_from_sample_fmt(&fmt, dst_sample_fmt)) < 0)
    goto end;
fprintf(stderr, "Resampling succeeded. Play the output file with the
command:\n"
            "ffplay -f %s -channel_layout %"PRIId64" -channels %d -ar %d
%s\n",
            fmt, dst_ch_layout, dst_nb_channels, dst_rate, dst_filename);

end:
fclose(dst_file);

if (src_data)
    av_freep(&src_data[0]);
av_freep(&src_data);

if (dst_data)
    av_freep(&dst_data[0]);

```

```
    av_freep(&dst_data);

    swr_free(&swr_ctx);
    return ret < 0;
}
}
```

It should be noted that the original PCM is AV_SAMPLE_FMT_S16, which is also 16-bit interleaved storage, and the same is true for resampling. If you want to convert to AV_SAMPLE_FMT_S16P, you need to pay attention that when you get the PCM after resampling, not only Need to get the data of dst_data [0], and also need to get the data of dst_data [1] and store it in stack. If it is S16, you only need to store dst_data [0] (such as code), because the 16P data is stored in the array in parallel Of the two elements.

3. USING FFMPEG TO CONSTRUCT SILENT FRAMES

When processing audio in a project, sometimes there is no data from the audio source, but you cannot interrupt the input of audio data to the encoder, otherwise a serious problem that the sound and the picture are not synchronized may occur.

```
AVFrame *alloc_silence_frame(int channels, int samplerate, int format)
{
    AVFrame *frame;
    int32_t ret;
    frame = av_frame_alloc();
    if(!frame)
    {
        return NULL;
    }

    frame->sample_rate = samplerate;
    frame->format = format;
    frame->channel_layout = av_get_default_channel_layout(channels);
    frame->channels = channels;
    frame->nb_samples = AAC_NBSAMPLE;
    ret = av_frame_get_buffer(frame, 0);
    if(ret < 0)
    {
```

```
    av_frame_free(&frame);
    return NULL;
}

av_samples_set_silence(frame->data, 0, frame->nb_samples, frame-
>channels,frame->format);
return frame;
}
```

4. HALVE THE VOLUME OF THE LEFT CHANNEL IN THE PCM16LE TWO- CHANNEL AUDIO SAMPLE DATA

The function in this program can reduce the volume of the left channel in the PCM16LE two-channel data by half. The code of the function is shown below.

```
int simplest_pcm16le_halfvolumeleft(char *url){
    FILE *fp=fopen(url,"rb+");
    FILE *fp1=fopen("output_halfleft.pcm","wb+");
    int cnt=0;
    unsigned char *sample=(unsigned char *)malloc(4);

    while(!feof(fp)){
        short *samplenum=NULL;
        fread(sample,1,4,fp);

        samplenum=(short *)sample;
        *samplenum=*samplenum/2; //L
        fwrite(sample,1,2,fp1); //R
        fwrite(sample+2,1,2,fp1);
        cnt++;
    }
}
```

```

    }
    printf("Sample Cnt:%d\n",cnt);

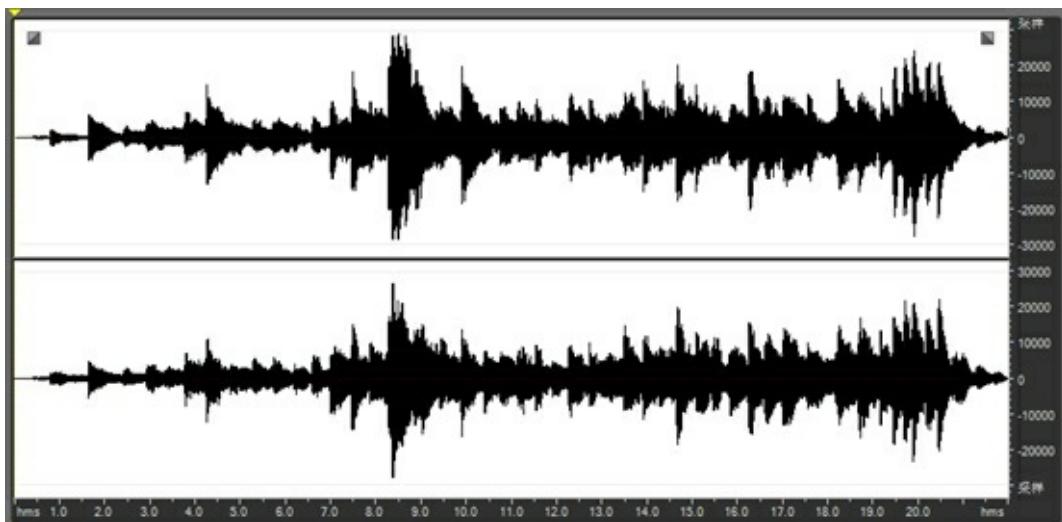
    free(sample);
    fclose(fp);
    fclose(fp1);
    return 0;
}

```

The method to call the above function is shown below.

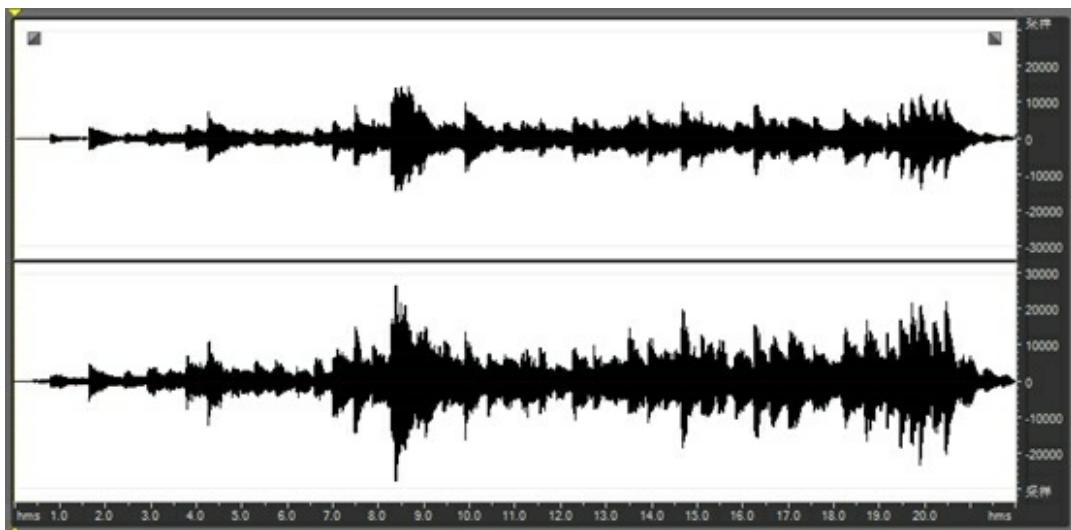
```
simplest_pcm16le_halfvolumeleft("NocturneNo2inEfla
```

As can be seen from the source code, after reading the 2 Byte sample value of the left channel, this program treats it as a short variable in C language. After dividing this value by 2, it was written back to the PCM file. The following figure is the waveform diagram of input PCM two-channel audio sampling data.



The figure below shows the processed waveform of the left channel. It can be seen that the waveform

amplitude of the left channel has been reduced by half.



5. DOUBLE THE SOUND SPEED OF PCM16LE TWO-CHANNEL AUDIO SAMPLING DATA

The functions in this program can double the speed of PCM16LE two-channel data in an abstract way. The code of the function is shown below.

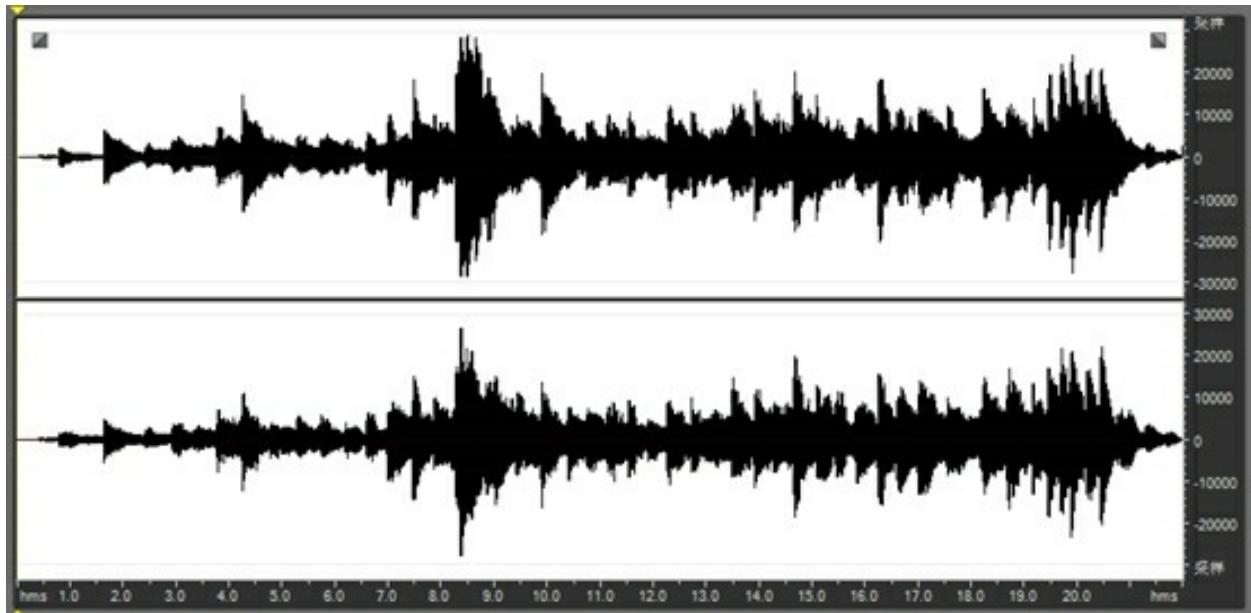
```
/**  
 * Re-sample to double the speed of 16LE PCM file  
 * @param url Location of PCM file.  
 */  
int simplest_pcm16le_doublespeed(char *url){  
    FILE *fp=fopen(url,"rb+");  
    FILE *fp1=fopen("output_doublespeed.pcm","wb+");  
  
    int cnt=0;  
    unsigned char *sample=(unsigned char *)malloc(4);  
  
    while(!feof(fp)){  
        fread(sample,1,4,fp);  
  
        if(cnt%2!=0){  
            fwrite(sample,1,2,fp1); //L  
            fwrite(sample+2,1,2,fp1); //R  
        }  
        cnt++;
```

```
    }
    printf("Sample Cnt:%d\n",cnt);

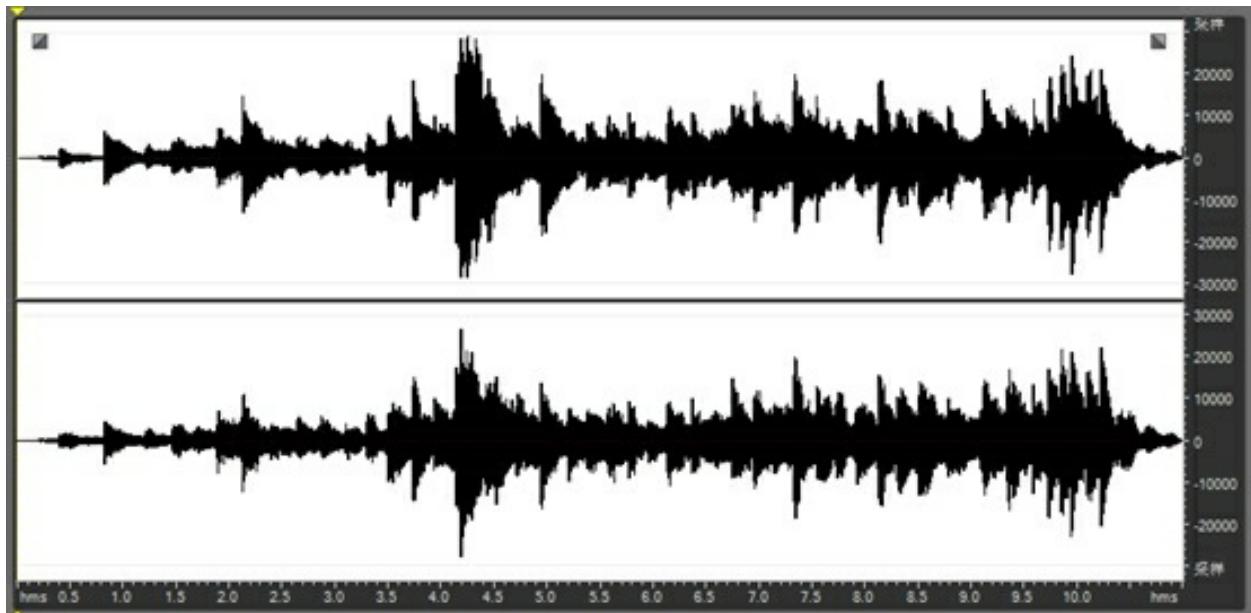
    free(sample);
    fclose(fp);
    fclose(fp1);
    return 0;
}
```

The method to call the above function is shown below.
simplest_pcm16le_doublespeed("NocturneNo2inEflat_

As can be seen from the source code, this program only samples the odd points of each channel. After the processing was completed, the original audio of about 22 seconds became about 11 seconds. The audio playback speed has been increased by 2 times, and the pitch of the audio has also become much higher. The following figure is the waveform diagram of input PCM two-channel audio sampling data.



The following figure is the waveform diagram of the output PCM two-channel audio sampling data. The timeline shows that the audio is much shorter.



6. CONVERT PCM16LE TWO-CHANNEL AUDIO SAMPLING DATA TO PCM8 AUDIO SAMPLING DATA

The functions in this program can convert the 16-bit sampling number of PCM16LE dual-channel data to 8-bit by calculating. The code of the function is shown below.

```
/**  
 * Convert PCM-16 data to PCM-8 data.  
 * @param url Location of PCM file.  
 */  
int simplest_pcm16le_to_pcm8(char *url){  
    FILE *fp=fopen(url,"rb+");  
    FILE *fp1=fopen("output_8.pcm","wb+");  
  
    int cnt=0;  
  
    unsigned char *sample=(unsigned char *)malloc(4);  
  
    while(!feof(fp)){  
  
        short *samplenumber16=NULL;  
        char samplenumber8=0;  
        unsigned char samplenumber8_u=0;
```

```

fread(sample,1,4,fp);
//(-32768-32767)
samplenumber16=(short *)sample;
samplenumber8=(*samplenumber16)>>8;
//(0-255)
samplenumber8_u=samplenumber8+128;
//L
fwrite(&samplenumber8_u,1,1,fp1);

samplenumber16=(short *)(sample+2);
samplenumber8=(*samplenumber16)>>8;
samplenumber8_u=samplenumber8+128;
//R
fwrite(&samplenumber8_u,1,1,fp1);
cnt++;
}
printf("Sample Cnt:%d\n",cnt);

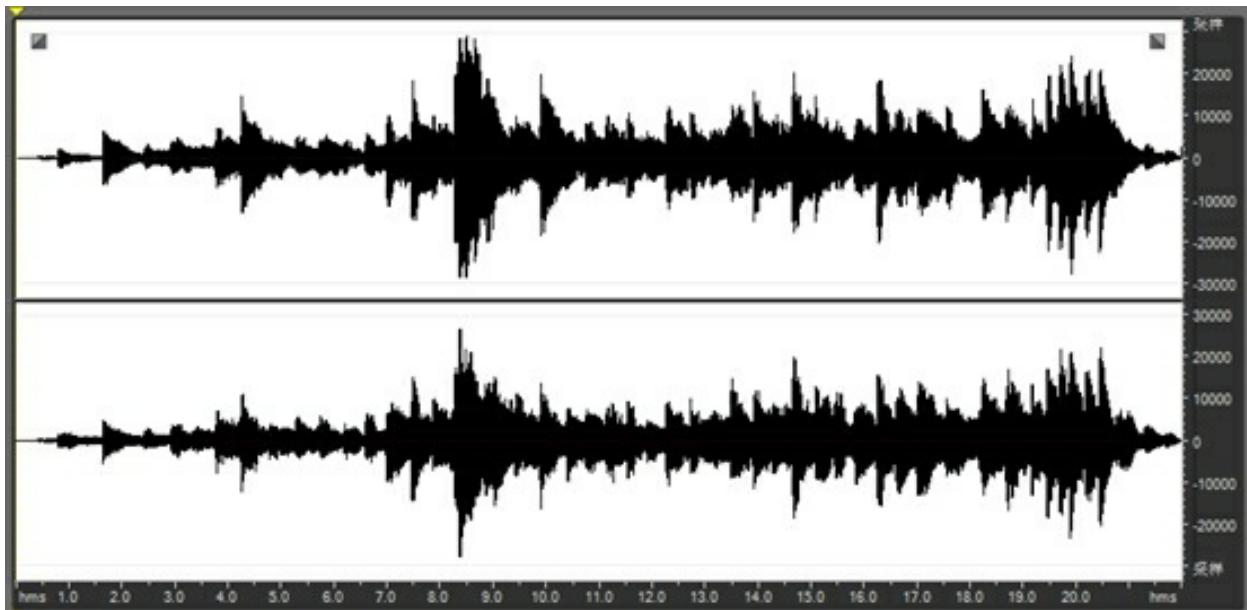
free(sample);
fclose(fp);
fclose(fp1);
return 0;
}

```

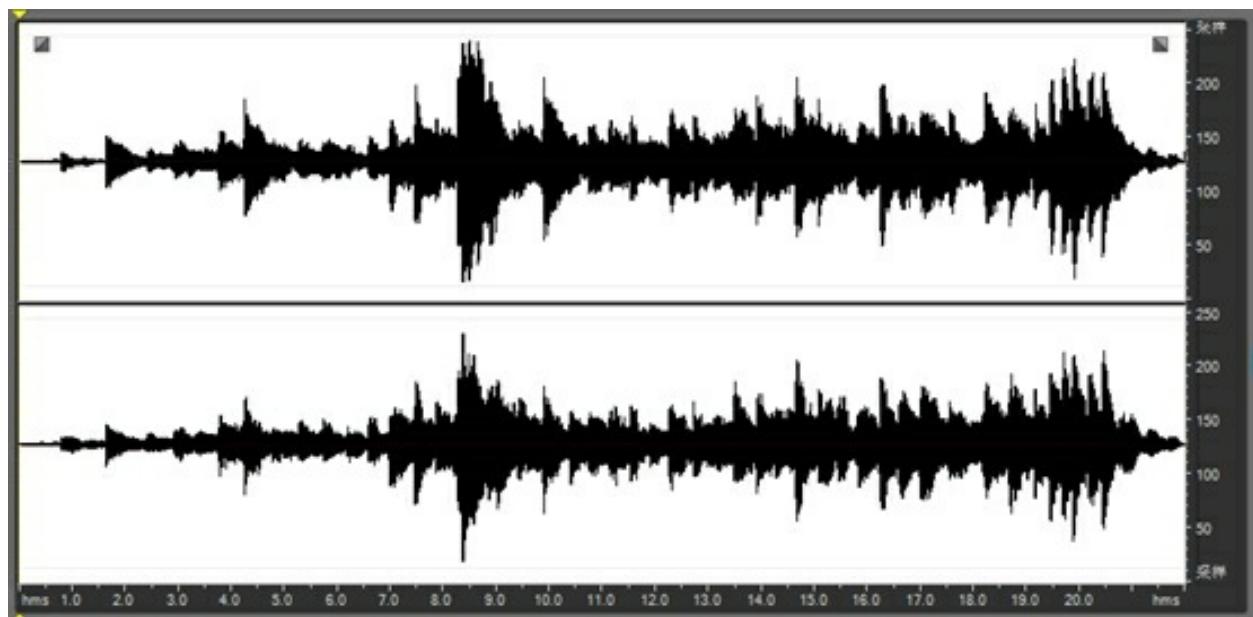
The method to call the above function is shown below.
simplest_pcm16le_to_pcm8("NocturneNo2inEflat_44.1

PCM16LE format sample data ranges from -32768 to 32767, while PCM8 format sample data ranges from 0 to 255. So the conversion from PCM16LE to PCM8 requires two steps: the first step is to convert the 16-bit signed value from -32768 to 32767 to the 8-bit signed value from -128 to 127, and

the second step is to convert the 8-bit signed value from -128 to 127. The value is converted to an 8-bit unsigned value from 0 to 255. In this program, the 16-bit sample data is stored by a short variable, and the 8-bit sample data is stored by an unsigned char type. The following figure is the waveform diagram of the input 16bit PCM two-channel audio sampling data.



The following figure shows the waveform of the output 8-bit PCM two-channel audio sampling data. Note that the range of ordinate values in the observation chart has changed from 0 to 255. If you listen carefully, you will find that the sound quality of 8-bit PCM is obviously inferior to that of 16-bit PCM.



7. A PART OF DATA WILL BE INTERCEPTED FROM PCM16LE MONO AUDIO SAMPLING DATA

The function in this program can intercept a piece of data from the PCM16LE mono data and output the sample value of the intercepted data. The code of the function is shown below.

```
/**  
 * Cut a 16LE PCM single channel file.  
 * @param url      Location of PCM file.  
 * @param start_num start point  
 * @param dur_num   how much point to cut  
 */  
int simplest_pcm16le_cut_singlechannel(char *url,int start_num,int  
dur_num){  
    FILE *fp=fopen(url,"rb+");  
    FILE *fp1=fopen("output_cut.pcm","wb+");  
    FILE *fp_stat=fopen("output_cut.txt","wb+");  
  
    unsigned char *sample=(unsigned char *)malloc(2);  
  
    int cnt=0;  
    while(!feof(fp)){  
        fread(sample,1,2,fp);  
        if(cnt==start_num){  
            fwrite(sample,1,2,fp1);  
        }  
        if(cnt==start_num+dur_num-1){  
            break;  
        }  
        cnt++;  
    }  
    free(sample);  
}
```

```

if(cnt>start_num&&cnt<=(start_num+dur_num)){
    fwrite(sample,1,2,fp1);

    short samplenumber=sample[1];
    samplenumber=samplenumber*256;
    samplenumber=samplenumber+sample[0];

    fprintf(fp_stat,"%6d,",samplenumber);
    if(cnt%10==0)
        fprintf(fp_stat,"\n",samplenumber);
    }
    cnt++;
}

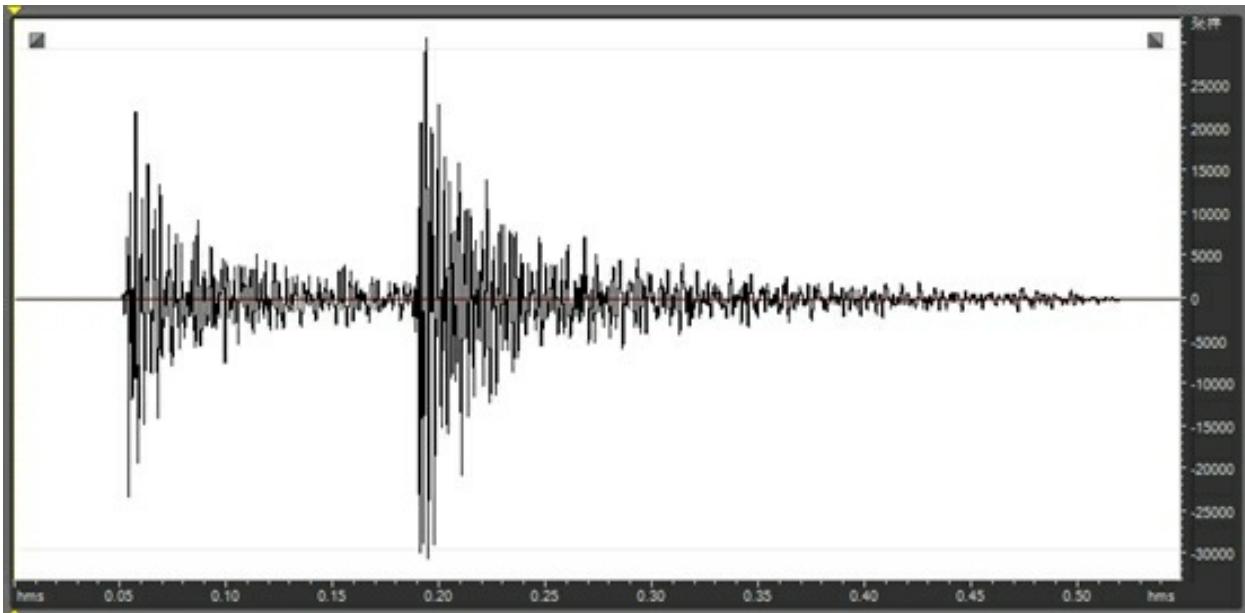
free(sample);
fclose(fp);
fclose(fp1);
fclose(fp_stat);
return 0;
}

```

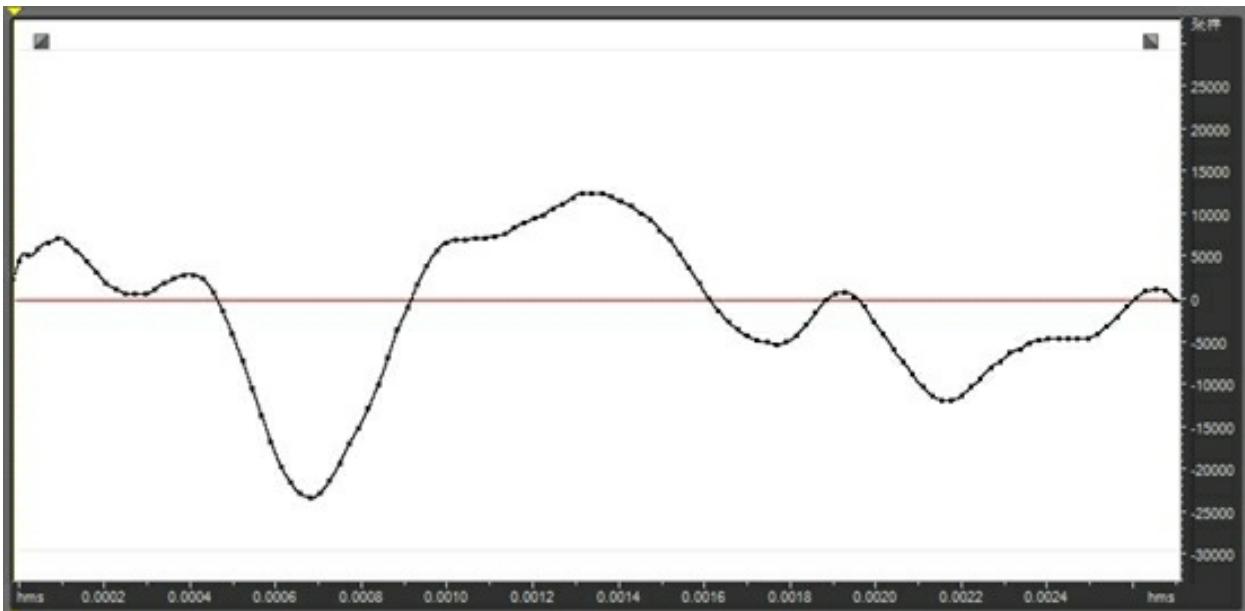
The method to call the above function is shown below.

```
simplest_pcm16le_cut_singlechannel("drum.pcm",2360)
```

This program can select a piece of sampled value from the PCM data and save it, and output the value of these sampled values. After the above code is run, the 120-point data from 2360 points in "drum.pcm" in mono PCM16LE format are saved as output_cut.pcm file. The following figure is the waveform of "drum.pcm". The audio sampling frequency is 44100KHz, the length is 0.5 seconds, and it contains about 22050 sampling points in total.



The figure below shows the data in the output_cut.pcm file that has been extracted.



Sample values for the above data are listed below.

4460, 5192, 5956, 6680, 7199, 6706, 5727, 4481, 3261, 1993,
1264, 747, 767, 752, 1248, 1975, 2473, 2955, 2952, 2447,
974, -1267, -4000,
-6965, -10210, -13414, -16639, -19363, -21329, -22541,

-23028, -22545, -21055, -19067, -16829, -14859, -12596, -9900,
-6684, -3475,
-983, 1733, 3978, 5734, 6720, 6978, 6993, 7223, 7225, 7440
7688, 8431, 8944, 9468, 9947, 10688, 11194, 11946, 12449,
12446,
12456, 11974, 11454, 10952,
10167, 9425, 8153, 6941, 5436, 3716,
1952, 236, -1254, -2463, -3493, -4223, -4695, -4927, -5190,
-4941,
-4188, -2956, -1490, -40, 705, 932, 446, -776, -2512, -3994,
-5723, -7201, -8687, -10157, -11134, -11661, -11642, -11168, -10155,
-9142,
-7888, -7146, -6186, -5694, -4971, -4715, -4498, -4471, -4468,
-4452,
-4452, -3940, -2980, -1984, -752, 257, 1021, 1264, 1032, 31,

8. CONVERT PCM16LE TWO-CHANNEL AUDIO SAMPLING DATA TO WAVE FORMAT AUDIO DATA

WAVE format audio (extension ".wav") is the most common type of audio in Windows. The essence of this format is to add a file header in front of the PCM file. The functions of this program can be packaged as WAVE format audio by adding a WAVE file header in front of the PCM file. The code of the function is shown below.

```
/**  
 * Convert PCM16LE raw data to WAVE format  
 * @param pcmpath    Input PCM file.  
 * @param channels   Channel number of PCM file.  
 * @param sample_rate Sample rate of PCM file.  
 * @param wavepath   Output WAVE file.  
 */  
int simplest_pcm16le_to_wave(const char *pcmpath,int channels,int  
sample_rate,const char *wavepath)  
{
```

```
typedef struct WAVE_HEADER{
    char      fccID[4];
    unsigned long  dwSize;
    char      fccType[4];
}WAVE_HEADER;

typedef struct WAVE_FMT{
    char      fccID[4];
    unsigned long  dwSize;
    unsigned short wFormatTag;
    unsigned short wChannels;
    unsigned long   dwSamplesPerSec;
    unsigned long   dwAvgBytesPerSec;
    unsigned short  wBlockAlign;
    unsigned short  uiBitsPerSample;
}WAVE_FMT;

typedef struct WAVE_DATA{
    char      fccID[4];
    unsigned long dwSize;
}WAVE_DATA;

if(channels==0||sample_rate==0){
    channels = 2;
    sample_rate = 44100;
}
int bits = 16;

WAVE_HEADER  pcmHEADER;
WAVE_FMT    pcmFMT;
WAVE_DATA   pcmDATA;

unsigned short m_pcmData;
FILE *fp,*fout;

fp=fopen(pcopath, "rb");
if(fp == NULL) {
    printf("open pcm file error\n");
    return -1;
```

```

}

    fpout=fopen(wavepath, "wb+");
if(fpout == NULL) {
    printf("create wav file error\n");
    return -1;
}

//WAVE_HEADER
memcpy(pcmHEADER.fccID,"RIFF",strlen("RIFF"));
memcpy(pcmHEADER.fccType,"WAVE",strlen("WAVE"));
fseek(fpout,sizeof(WAVE_HEADER),1);

//WAVE_FMT
pcmFMT.dwSamplesPerSec=sample_rate;
pcmFMT.dwAvgBytesPerSec=pcmFMT.dwSamplesPerSec*sizeof(m_pc
pcmFMT.uiBitsPerSample=bits;
memcpy(pcmFMT.fccID,"fmt ",strlen("fmt "));
pcmFMT.dwSize=16;
pcmFMT.wBlockAlign=2;
pcmFMT.wChannels=channels;
pcmFMT.wFormatTag=1;

fwrite(&pcmFMT,sizeof(WAVE_FMT),1,fpout);

//WAVE_DATA;
memcpy(pcmDATA.fccID,"data",strlen("data"));
pcmDATA.dwSize=0;
fseek(fpout,sizeof(WAVE_DATA),SEEK_CUR);

fread(&m_pcmData,sizeof(unsigned short),1,fp);
while(!feof(fp)){
    pcmDATA.dwSize+=2;
    fwrite(&m_pcmData,sizeof(unsigned short),1,fpout);
    fread(&m_pcmData,sizeof(unsigned short),1,fp);
}

pcmHEADER.dwSize=44+pcmDATA.dwSize;

rewind(fpout);
fwrite(&pcmHEADER,sizeof(WAVE_HEADER),1,fpout);
fseek(fpout,sizeof(WAVE_FMT),SEEK_CUR);

```

```
    fwrite(&pcmDATA,sizeof(WAVE_DATA),1,fpout);
    fclose(fp);
    fclose(fpout);

    return 0;
}
```

The method to call the above function is shown below.

simplest_pcm16le_to_wave("NocturneNo2inEflat_44.1")
WAVE file is a RIFF format file. Its basic block name
is "WAVE", which contains two sub-blocks "fmt" and
"data". From a programming perspective, it is simply
composed of 4 parts: WAVE_HEADER,
WAVE_FMT, WAVE_DATA, and sampling data. Its
structure is shown below.

WAVE_HEADER

WAVE_FMT

WAVE_DATA

PCM data

The structure of the first three parts is shown
below. When writing the WAVE file header, assign
appropriate values to each of the fields. But there is
one thing to note: WAVE_HEADER and
WAVE_DATA contain a dwSize field of file length
information, the value of this field must be obtained

after writing the audio sample data. Therefore, these two structures are finally written into the WAVE file.

```
typedef struct WAVE_HEADER{
    char fccID[4];
    unsigned long dwSize;
    char fccType[4];
}WAVE_HEADER;

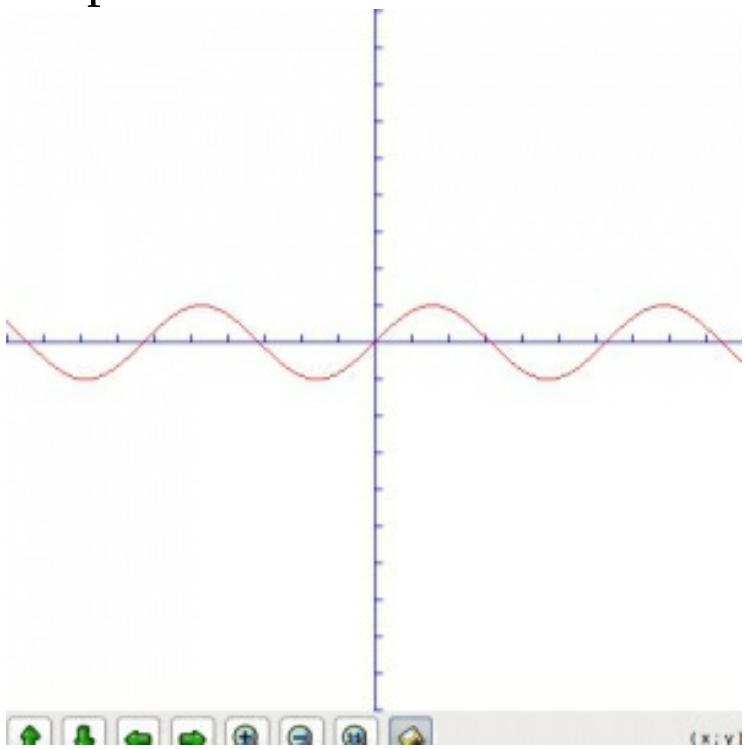
typedef struct WAVE_FMT{
    char fccID[4];
    unsigned long dwSize;
    unsigned short wFormatTag;
    unsigned short wChannels;
    unsigned long dwSamplesPerSec;
    unsigned long dwAvgBytesPerSec;
    unsigned short wBlockAlign;
    unsigned short uiBitsPerSample;
}WAVE_FMT;

typedef struct WAVE_DATA{
    char fccID[4];
    unsigned long dwSize;
}WAVE_DATA;
```

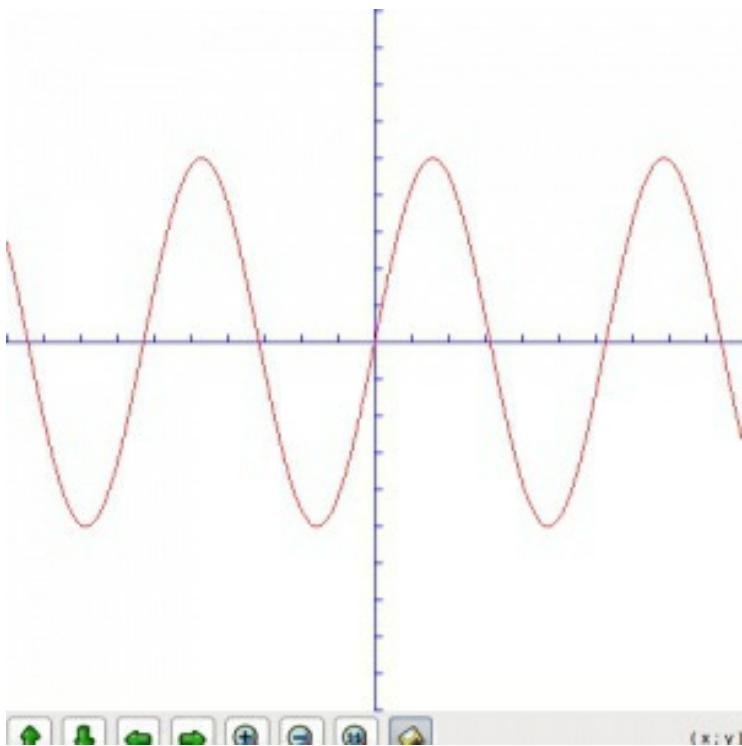
After the function execution of this program is completed, the NocturneNo2inEflat_44.1k_s16le.pcm file can be packaged into the output_nocturne.wav file.

BASIC AUDIO EFFECTS- VOLUME CONTROL

Now let's take a look at some real waveforms. The simplest is a sine wave.



We will increase the amplitude of the waveform by a factor of five, as shown below:



So if you want to increase the volume of PCM data, you just need to multiply each sampled data by a factor.

If our PCM data has 2048 bytes, it contains 1024 samples. We use the following pseudo code to increase the volume:

```
int16_t pcm [1024] = read in some pcm data;  
    for (ctr = 0; ctr < 1024; ctr++) {  
        pcm [ctr] *= 2;  
    }
```

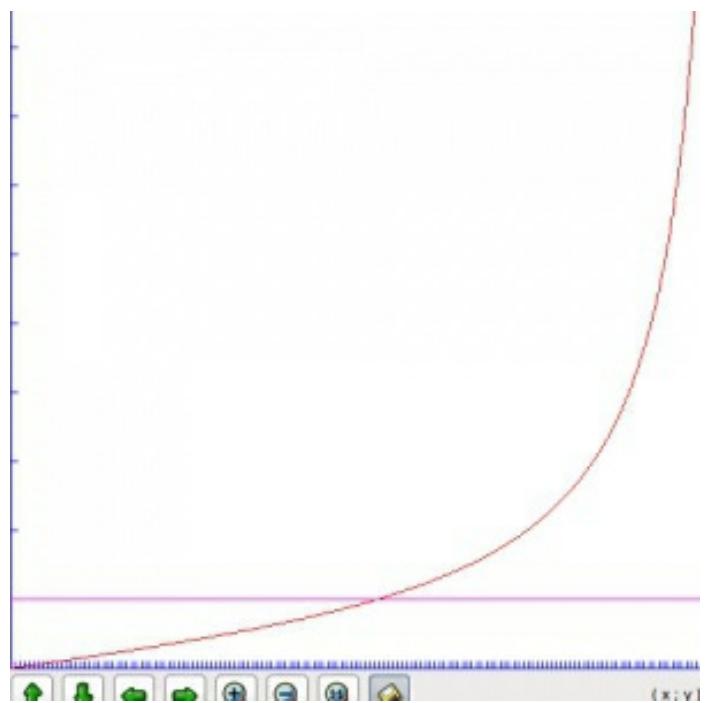
Volume control is as simple as that, but there are two things to note:

If the value of the sampling point multiplied by the expansion factor is less than -32768 or greater than

32768, the value sampled here can only take -32768 or 32768

```
int16_t pcm [1024] = read in some pcm data;  
int32_t pcmval;  
for (ctr = 0; ctr <1024; ctr++) {  
    pcmval = pcm [ctr] * 2;  
    if (pcmval <32767 && pcmval> -32768) {  
        pcm [ctr] = pcmval  
    } else if (pcmval> 32767) {  
        pcm [ctr] = 32767;  
    } else if (pcmval <-32768) {  
        pcm [ctr] = -32768;  
    }  
}
```

Multiplying the data of the sample point by 2 does not mean that we have tripled the volume of the sound, in fact it is true. The relationship between the gain coefficient of the sound volume and the volume is as follows:



HOW TO CHANGE PCM SAMPLE RATE

By definition, Sample Rate means the number of samples per second, So if you want to change the sampling frequency of the audio, we just need to discard or copy the sampling points appropriately. For example: the original audio is opus encoded, mono, the sampling rate is 48kHz, and the sampling point size is 16-bit.

How do I get audio encoded as speex with a sampling rate of 16kHz and a sample size of 16-bit?

We need the following steps:

Step1: Decode opus into PCM format data (called PCM1), and the sampling rate of PCM1 at this time is 48kHz

Step1: Take the $3 * n$ (n is a natural number starting from 0) sampling points in the PCM1 data, and discard the sampling points at $3 * n + 1$ and $3 * n + 2$.

Get PCM2, the PCM2 sampling rate at this time is $48\text{kHz} / 3 = 16\text{kHz}$

Step1: encode PCM2 into speex data

HOW TO CHANGE THE BIT DEPTH

The number of sampling bits can be understood as the resolution of the sound processed by the acquisition card. The larger the value, the higher the resolution and the more realistic the recorded and played back sound. We first need to know: the sound file in the computer is represented by the numbers 0 and 1. Continuous analog signals (windowed and truncated) are sampled by digital pulses at a certain sampling frequency, and each discrete pulse signal is quantized into a series of binary coded streams with a certain quantization accuracy. The number of bits in this string of encoded streams is the number of samples, also known as the quantization accuracy. The relationship between the bit rate and the number of sampling bits can be clearly seen from the bit rate calculation formula: bit rate = sampling frequency × quantization accuracy × number of channels.

The essence of recording on a computer is to convert analog sound signals into digital signals. On the contrary, during playback, digital signals are

restored to analog sound signals and output. The bit of the capture card refers to the binary digits of the digital sound signal used by the capture card when collecting and playing sound files. The position of the acquisition card objectively reflects the accuracy of the digital sound signal's description of the input sound signal. 8 bits represent the power of 2-256, and 16 bits represent the power of 2-64K. By comparison, for a piece of the same music information, a 16-bit sound card can divide it into 64K precision units for processing, while an 8-bit sound card can only handle 256 precision units. The difference between 8-bit samples is the dynamic range (Wikipedia: dynamic range) is the ratio of the maximum and minimum values of a variable signal (such as sound or light). It can also be expressed as a base-10 logarithm (decibel) or a base-2 logarithm.) Wide and wide dynamic range, the volume fluctuation can be recorded more finely, In this way, whether it is a subtle sound or a strong dynamic shock, it can be vividly expressed, and the sampling specification for CD sound quality is the official 16-bit sampling specification.

A sample of audio is represented by several bits, called the sampling accuracy, and also called bit-depth. Our commonly used bit depth is 16bit, which means that 16bit expresses a sample. In this way, the

highest signal-to-noise ratio can be expressed as $20\log(2^{16}) = 96\text{db}$,

With 24bit depth, the highest SNR can reach $20\log(2^{24}) = 144\text{db}$.

Professional digital audio processing software actually uses float to represent a sample, that is, 32bit, then the highest signal-to-noise ratio can reach 193db, which is already very high. We need to achieve the goal, for example, to reach the industry's vivo codec cs4398 signal-to-noise ratio of 120db, which only needs 24bit to theoretically provide feasibility.

For lossy formats such as MP3, regardless of whether the pcm data before compression is 24bit or 16bit, the minimum bit-depth will be used to store the data during the compression process.

Therefore, the bit-depth of decoding is theoretically meaningless. However, in terms of engineering, the accuracy of the addition, subtraction, multiplication and division operations during decoding using mp3 with a 24-bit decoder becomes larger, which can improve some signal-to-noise ratio.

For convenience, 24bit is usually put into a 32bit int, how to put it? It depends on the type of container and the size of the machine. For example, the WAV container is stored in little-endian. In this way, storing the 24-bit sample of 0xAA BB CC needs to be stored

as:

CC BB AA (--- address from low to high --->)

In order to convert to int32_t for processing, you need to fill its high-order bits to the high-order bits of an int32_t type, and pad the lowest 8 bits with zeros, and finally get 0xAABBCC00.

Since arm is also little-endian, the layout in memory should be like this:

00 CC BB AA (--- address from low to high --->)

The change of bit depth is as follows:

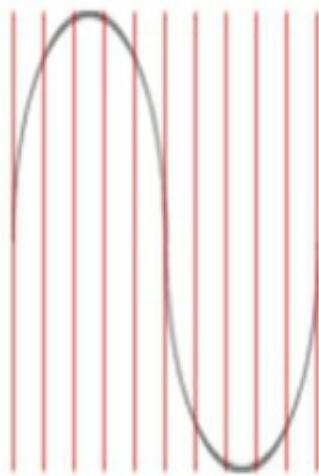


Figure 4 - higher sample rate



Figure 5- resulting waveform

Low sample rate and low bit depth

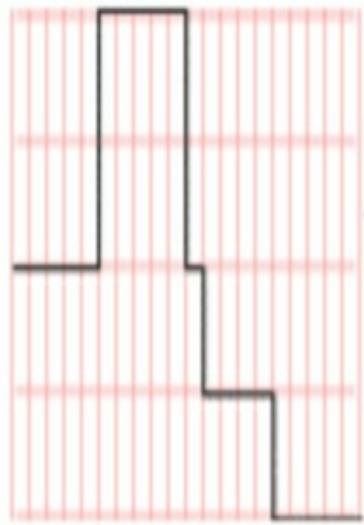


Figure 6 - low bit depth

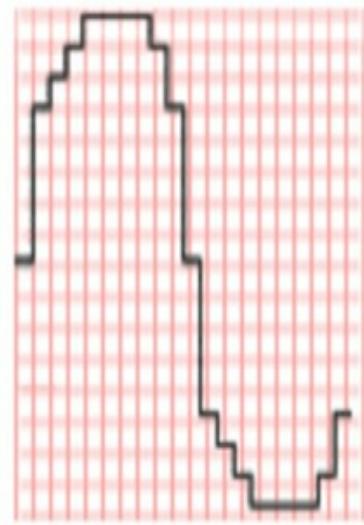


Figure 7 - higher bit depth

High sampling rate and high bit depth

4. INTERLACED VIDEO

TV broadcasts have been using interlaced scanning for nearly 60 years, while computer displays, graphics processing, and digital cinema have used progressive scanning. Progressive scanning is the simplest and best scanning method. However, due to the limitation of frequency band resources and refresh frequency, progressive scanning cannot be applied in analog television broadcasting. In recent years, the development of digital television technology has made progressive scanning in the broadcast television industry. Application becomes possible.

WHY TV INTERLACED ?

The basis for the reproduction of moving images in movies and television is the visual inertia of the human eye (or the visual residual characteristics of the human eye). The threshold of human visual inertia is 24 times per second, that is, continuous display of more than 24 different stills per second. When the picture is taken, the human eye will feel that the images are continuously moving without distinguishing them into still pictures. Therefore, from the perspective of reproducing moving images, the refresh rate of the image must be above 24Hz.

Another important characteristic of human visual inertia is the sensitivity to light source flicker. The threshold value of human eye flicker is about 50 Hz, that is, the discontinuous light source flicker frequency is higher than 50 times per second. The light source emits light continuously rather than intermittently. Therefore, from the perspective of eliminating image flicker, the frequency of image refresh (flicker) must be above 50 Hz.

The television uses a scanning method to

decompose a two-dimensional still picture into a plurality of one-dimensional scanning lines, and all scanning lines forming a complete picture are called television frames. The simplest scanning method is that each frame of the image is scanned sequentially by the electron beam one line after another, that is, progressive scanning. Obviously, in order to meet the requirements for transmitting moving pictures, TV pictures of more than 24 frames must be reproduced per second.

However, the frame frequency above 24Hz can only meet the requirements of moving picture for TV transmission. As mentioned above, the flickering of the image can be eliminated only when the refresh frequency is above 50Hz. It's too wide. For example, in a standard definition analog television system, the allowed image signal bandwidth is 5 to 6 MHz. When the frame rate is doubled, the image signal bandwidth will reach 10 to 12 MHz, which greatly reduces the efficiency of frequency resource utilization.

In order to improve the image refresh rate under the condition of limited bandwidth resources, the interlaced scanning technology was used in the analog TV era. The so-called 2: 1 interlaced scanning is to divide a television frame into two television fields and scan them separately. The field formed by the odd

scanning lines is called the odd field, and the field formed by the even scanning lines is called the even field. The odd and even fields are interlaced to form a television frame. The line scanning frequency of 2: 1 interlaced scanning is half that of progressive scanning, so the spectrum of the television signal and the channel bandwidth for transmitting the signal are also half of the progressive scanning. After adopting 2: 1 interlaced scanning, the refresh frequency is changed from the frame frequency to the field frequency, which is doubled. The channel utilization rate has also doubled when the image quality has not decreased much. Due to historical reasons, most countries and regions with an AC grid frequency of 50Hz have selected a TV field frequency of 50Hz, while countries and regions with a grid frequency of 60Hz have a TV field frequency of 60Hz, which means that in most countries in the world The field frequency of TV signals in Hehe District is the same as the frequency of the local power grid. Compared to progressive scanning, interlaced scanning saves transmission bandwidth but also brings some negative effects. Because a frame is composed of two fields interlaced, the vertical resolution of interlaced scanning is lower than progressive. The Kell Factor of progressive scanning is approximately 0.7, and the interlaced scanning is 0.5, which means that the

vertical resolution of interlaced scanning is only about 70% of that of progressive scanning with the same number of scanning lines. Interlaced scanning also brings more obvious inter-field flicker, which is prone to adverse effects such as jagged edges when shooting moving objects. Compared to progressive, interlaced images are not conducive to computer processing of graphics and images, which reduces compression efficiency. Eye fatigue easily when watching time.

In order to obtain higher quality image quality, progressive scanning has also become the option of digital TV. For example, digital TV broadcast formats in the United States include both interlaced 480 / 60i and 1080 / 60i, as well as progressive 480 / 60P and 720 / 60P. 1080 / 24P, 25P and 30P progressive scanning are also used for the production and exchange of high-definition digital TV programs. Most digital movie programs with DVD as the distribution carrier use 480 / 24P or 576 / 25P progressive scanning.

In essence, interlaced scanning is an analog TV signal bandwidth compression technology. This scanning method effectively solves the contradiction between the refresh rate and the signal bandwidth in the analog TV era, and is also widely used in the digital TV era. At present, television broadcasting generally uses 2: 1 interlaced scanning, that is, dividing a frame

into two fields. In theory, you can also use 3: 1 or even 4: 1 interlaced scanning, which is to divide a frame into three or four fields. Only 2: 1, that is, one-line scanning is widely used, because a higher interlaced ratio can improve the channel utilization rate, but the image quality is poor when displaying a moving image. It can be concluded from the above discussion that progressive scanning can obtain higher image quality than interlaced scanning, but the television industry chooses interlaced scanning based on two very simple considerations, the first is the flicker frequency and the second is the transmission bandwidth. In fact, in the era of analog television, interlaced scanning technology is the best compromise to achieve optimized display effects with limited frequency band resources.

DEVELOPMENT AND EVOLUTION OF DISPLAY TECHNOLOGY

In the era of analog television, the scanning method and refresh rate of television shooting, production and display must be the same, which is one of the reasons why analog television uses interlaced scanning. Since the late 1980s, with the gradual popularization of digital processing technology, especially the use of frame memory in receivers, and the development of new flat-panel display devices such as liquid crystal and plasma since the late 1990s, the scanning method and refresh rate of image display are different from shooting And production also becomes a possible choice, this processing method is impossible to achieve in the era of analog television.

2X FIELD FREQUENCY DISPLAY

In order to eliminate the flickering of the displayed image without changing the scanning method of the transmitted signal, many large-screen receivers use 2x field frequency scanning technology, that is, first store the 50Hz interlaced signal in the frame memory, and then use the 100Hz The interlace scanning method reads out and displays, in fact, each signal is displayed twice, so the refresh rate of the display is doubled compared to the sampling speed when shooting. Although the effective information amount in this unit-of-field-frequency display mode has not increased in unit time, the flickering feeling of the displayed image can be effectively eliminated due to the increase of the refresh rate. The 2x field frequency display is mainly used in countries and regions with a field frequency of 50Hz. The main reason is that 50Hz is the critical point of flicker sensitivity of the human eye. The 50Hz field frequency flicker is much more obvious than the 60Hz field frequency.

2X HORIZONTAL FREQUENCY DISPLAY

The 2x line frequency display is also the double line display, that is, the number of scanning lines displayed is double that of the source. Line doubling is to add a scanning line between every two scanning lines using digital technology in the display terminal. The rougher processing is to simply repeat the previous scanning line. The precise processing uses interpolation to replace adjacent lines. The level and spatial distribution of the new lines are obtained after calculation. The double-line display method can effectively reduce the visibility of the scanning lines on the screen of the large-sized picture tube and increase the delicate feeling of the picture. Similar to the method of doubled field frequency display, although the doubled line frequency display cannot increase the effective information amount per unit time, the purpose of improving the display image quality is achieved by changing the raster structure.

PROGRESSIVE DISPLAY

Progressive display is the digital processing of interlaced TV signals into progressive scan signals to achieve the purpose of improving display quality. A commonly used interlaced to progressive processing method is to convert an interlaced signal of 50 fields per second and 312.5 lines per field into a progressive signal of 50 frames per second and 625 lines per frame. In order to reduce the flickering of the image, some TV sets increase the display refresh rate to 60 frames per second, and some digital TV receivers convert interlaced TV signals to progressive computer display formats such as VGA (640x480, 60Hz or more high frame rate), SVGA (800x600) or XGA (1024x768) display.

The image obtained by converting the signal of interlaced scanning to progressive display on the display end is different from the picture obtained by progressive scanning, because the additional scanning lines after converting one field into one frame are obtained by interpolation. So the effective information amount of this "frame" has not increased compared

with the original one field. However, due to changes in the scanning method, this progressive display processing method can effectively eliminate the inter-line flicker when a large-screen TV displays an interlaced signal.

In fact, the above several display technologies all achieve the purpose of improving the display image quality by changing the scanning method of the display terminal without changing the scanning method of the transmitted signal.

MULTI-FORMAT RECEIVING SINGLE FORMAT DISPLAY

With the coexistence of multiple DTV formats in the United States, a multi-format receiving single format display scheme has also emerged. In order to receive digital TV programs with different definitions and scanning methods, some receivers use multi-format set-top boxes. The set-top box has a built-in scan / definition converter. No matter whether the received signal is interlaced or progressive, HD or SD, the set-top box is built-in. The scan / resolution conversion chip all converts it into a fixed scan / resolution format output display. For example, some set-top boxes use a VGA or DVI display interface, and the output resolution is XGA or WXGA, that is, a 4: 3 screen with a resolution of 1024x768 or a 16: 9 screen with a resolution of 1366x768 at 60 frames per second. The built-in converter will convert the input Different signal formats are converted into XGA or WXGA output to match the corresponding flat-panel display.

FLAT DISPLAY DEVICE

With the expansion of the display screen size in recent years, traditional display devices such as kinescopes are gradually giving way to flat panel display devices such as plasma display panels (PDP) and liquid crystal displays (LCD). Projection display devices using LCD or DLP chips are also available. Made great progress. The grating of the picture tube is formed by electron beam scanning. This flexible scanning addressing method does not need to drive each pixel one by one, so it is suitable for both interlaced and progressive display. The image display principle of PDP, LCD, and DLP is completely different from that of kinescopes. They are area array display devices with limited pixels. This display method using XY addressing requires driving each pixel unit one by one, regardless of the scanning mode. Regardless of their sharpness, they must be converted into sharpness displays with the same number of pixels as the display panel, so this new type of flat panel display device is more suitable for displaying progressive scanning signals. In fact, TVs using PDP,

LCD, DLP and other devices display progressive scan images, and even if the input is an interlaced scan signal, it will be converted to progressive scan in the display drive circuit. Therefore, PDP, LCD, DLP, etc. can also be called progressive scan display devices.

PSF-PROGRESSIVE SEGMENTATION

There are two types of progressive scanning signal transmission, one is to transmit progressive scanning signals frame by frame, and the other is to divide the television signal of progressive scanning into odd and even fields and transmit them separately, namely Progressed Segmented Frame- Segment-by-line transmission, abbreviated Psf. Psf is a transmission method that uses an interlaced interface to transmit progressive signals. Because the signal transmission structure of Psf is the same as interlaced scanning, Psf transmission can make progressive signals and most of the recording and production only support interlaced formats. Compatible with display devices. For example, if images shot with 1080 / 25P are transmitted using progressive 25P transmission, only a few recording, production, and display devices with a 25P progressive transmission interface can receive and process such progressive scanning signals; while using 25Psf transmission, all Interlaced devices such as video recorders, switchers and monitors that support 1080 /

50i can directly record, process and display 25Psf as 50i. If the progressive device at the receiving end also supports the Psf transmission mode, the progressive scanning signal divided into two fields for transmission can be combined into one frame through the Psf interface at the receiving end. Similarly, all interlaced devices supporting 1080 / 60i (1080 / 59.94i) can also be used for the transmission, recording, and display of 1080 / 30Psf (1080 / 29.97Psf) progressive signals.

The difference between Psf and interlaced scanning signals is that progressive Psf is sampled once per frame to obtain a frame picture, and this frame is divided into two fields and transmitted separately. The two fields at the receiving end can still be combined into a complete frame; Interlaced sampling is performed twice per frame, that is, once per field, but these two fields are sampled at different times. When shooting moving images, the two fields cannot be completely spatially due to the different sampling times of the two fields. It is overlapped, so the two fields obtained by interlacing cannot be combined into a complete picture when shooting moving pictures.

A device configured with a Psf interface can use the Psf method to input / output a progressive scan signal, but it cannot receive or transmit non-Psf or non-

segment progressive scan signals; while a device configured with a progressive scan transmission interface can only input / Output non-segment progressive signals, they cannot exchange progressive programs with devices with Psf interface.

Actual progressive format used Although the vast majority of television broadcasts are currently interlaced, progressive scan formats have begun to be used in broadcast, production, exchange, and distribution.

PROGRESSIVE FORMAT FOR BROADCAST

480/60P-720x480@59.94P, one of the digital TV broadcast formats in the United States, is a progressive scanning version of 480 / 60i or NTSC, which is adopted by the American ABC. It should be noted that due to historical reasons, the field frequency of the US NTSC standard is not 60Hz but 60 / 1.001, which is a non-integer that has a frequency reduction of one thousandth. Generally, 60i or 60P is actually 59.94i or 59.94P.

720/60P-1280x720@59.94P, one of the digital TV broadcast formats in the United States, is the highest-definition progressive scan format currently broadcast, and is used by ABC and FOX.

In addition to the above two progressive scanning formats used directly for broadcasting, 50Hz countries and regions (especially in Europe) are also considering the possibility of broadcasting using two progressive scanning formats, 576 / 50P and 720 / 50P.

PROGRESSIVE SCAN FORMAT FOR PRODUCTION AND EXCHANGE

1080 / 24P-1920x1080 @ 24P and 1920x1080@23.976P, referred to as 24P, is one of the digital TV production and exchange formats. Because 24P has a natural affinity with movie film with 24 frames per second, 24P is mainly used for digital movie shooting, production, and program exchange. Digital versions of programs shot at 24P can be converted to film for display in ordinary theaters, and can also be converted to 480 / 24P compressed to progressively scanned MPEG2 files for DVD distribution. In addition, prime-time TV shows and commercials traditionally filmed in North America are increasingly being replaced by 24P digital shooting.

Because the refresh rate of 24 times per second is too low, 24P is only suitable for program production and exchange and cannot be directly used for broadcast. In 60Hz countries and regions, 24P

programs can obtain 60 fields of 1080 / 60i and 480 / 60i (NTSC) interlaced signals through 3-2 pull-down conversion and down-conversion. This pull-down conversion and the conversion of 24 film frames per second the processing is exactly the same for a 60-field television signal. There are actually two versions of 24P, one is the 24P used by the film industry, and the other is 23.976P for TV production. For historical reasons, the 24P frame rate used for TV production in the United States is a non-integer $24 / 1.001 = 23.976$, which has a frequency reduced by one-thousandth. In this way, the field frequency after 3-2 pull-down conversion is 59.94Hz, but due to customary reasons this 23.976P is often referred to as "24P".

There are three methods to display 24P progressive scan signals. One is to use Psf transmission to convert 24 frames of progressive signals into 48 fields of interlaced signal display (ie 24Psf), but because the refresh frequency of 48Hz is lower than the flicker of the human eye. The threshold is 50Hz. When viewing this 48i interlaced image, the flicker is too strong and the eyes are easily fatigued. The second method is to convert 24P into a 60-field interlaced signal display through a 3-2 pull-down conversion; the best method is 3 Double frame rate display, that is, the progressive scanning of each frame is displayed three

times in a dedicated monitor by digital processing. This way, a 72P progressive scan image is displayed on the monitor, which improves the display quality. Flicker is eliminated again.

1080 / 25P-1920x1080 @ 25P, one of the digital TV production and exchange formats. The application range of 25P is almost the same as that of 24P. The difference is that 24P is mainly used in countries and regions at 60Hz and 25P is mainly used in countries and regions at 50Hz. This is because the progressive scanning signal of 1080 / 25P can easily pass Psf. The transmission method is converted into 1080 / 50i (ie 25Psf) HD interlaced signals or down-converted to 576 / 50i (PAL) SD interlaced signals for broadcast. It can also be converted to 576 / 25P compressed into progressive MPEG2 files for DVD release.

There are two kinds of 25P display methods. In addition to the above-mentioned Psf transmission method to convert 25P to 50i (ie 25Psf) interlaced display, a better method is to double the frame rate display, which uses digital processing to scan each frame progressively. The screen is repeatedly displayed 2 or 3 times, so that a 50P or 75P progressive scan image is displayed on the monitor.

1080/30P-1920x1080@29.97P, one of the

digital TV production and exchange formats. 30P is mainly used in the production and exchange of high-definition digital TV programs in 60Hz countries and regions. The completed 1080 / 30P progressive programs can be easily converted into 1080 / 60i (ie 30Psf) high-definition interlaced signals through the Psf transmission method. Or down-convert to 480 / 60i (NTSC) SD interlaced signals for direct broadcast.

There are also two display methods of 30P, that is, 30P is converted to 60i interlaced display using Psf transmission method and digital processing method is used to display 60P progressive scan image at 2x frame rate.

Programs shot and produced with 24P, 25P, and 30P are very similar to the effects of being converted into a television signal after filming, because the number of frames shot per second is 24 to 30, and only the number of interlaced shots with a field frequency of 50 or 60Hz Therefore, when moving objects such as push, pull, pan, and move are not smooth enough, the same jitter phenomenon as in movie film shooting will occur, which is the so-called "movie feeling" or "film feeling". This jitter also exists whether it is a 2x or 3x frame rate display or converted to an interlaced display. This is because no matter how the display mode is changed, the original sampling rate

cannot be changed when shooting.

PROGRESSIVE SCAN FORMAT FOR DISTRIBUTION

480/24P-720x480@23.976P, a DVD-based distribution format, referred to as 480P. Most of the DVD's programs come from motion picture film. The DVD released in countries and regions with a field frequency of 60Hz (ie NTSC regions such as North America and Japan) compresses 24 movie frames per second into MPEG2 files. Progressive scan format 480 / 24P. Since most DVD movies are currently distributed by Hollywood producers, most DVD discs purchased on the market are in 480 / 24P progressive scan format.

576 / 25P-720x576 @ 25P, a DVD-based distribution format, referred to as 576P. Some DVD movies released in countries and regions with a field frequency of 50 Hz (that is, PAL regions in Europe, Asia, etc.) play back film footage shot at 24 frames per second at a rate of 25 frames per second and compress them into 576 / 25P MPEG2 files (Some films in Europe are

shot at 25 frames per second).

It should be noted that some DVD movies do not use the progressive scanning encoding method. In particular, most DVD movies produced by domestic production masters use the same 576 / 50i interlaced scanning compression encoding as television signals.

Progressive scan signal transmission interface

Common digital and analog video interfaces can transmit progressive signals.

DIGITAL VIDEO INTERFACE

At present, the main connection method between high-definition digital TV production equipment is serial digital component interface HD-SDI. According to the SMPTE 292M standard, the code rate of HD-SDI is 1485Mbps. It supports the following types of signal formats with effective video code rates lower than 1485Mbps:

Table: Signal format transmitted by HD-SDI interface (10-bit quantization)

Signal format	transfer method	Effective bit rate
1080 / 60i	1080 / 60i	
1080 / 30P	1080 / 30P	1244.16Mbps
1080 / 30Psf		
1080 / 50i	1080 / 50i	1036.8Mbps

1080 / 25P	1080 / 25P	
1080 / 25Psf		
1080 / 24P	1080 / 24P	995.328Mbps
	1080 / 24Psf	
720 / 60P	720 / 60P	1105.92Mbps

As can be seen from Table 1, HD-SDI can transmit 4 progressive scan formats:

1080 / 30P	1080 / 25P	1080 / 24P	720 / 60P
------------	------------	------------	-----------

Except for 720 / 60P, the other three progressive formats among these four progressive formats can be transmitted in P or Psf. As mentioned earlier, P and Psf are two different transmission methods. When transmitting a progressive scan signal, it must be ensured that the transmission modes at the sending and receiving ends are the same. For example, when the transmitting end uses 1080 / 24Psf transmission, if the interface of the receiving end device only supports 1080 / 24P, it cannot be transmitted correctly.

For 480 / 60P, because its code rate is twice

that of 480 / 60i, the existing SDI digital interface that complies with the SMPTE 292M standard cannot transmit such progressive signals. The 480 / 60P digital interface is based on the SMPTE 294M standard Defined. SDI and HD-SDI serial digital interfaces transmit 4: 2: 2 brightness and color difference component signals Y / RY / BY, and when transmitting 4: 4: 4 RGB signals, SDI and HD-SDI must use dual link mode, which requires two cables.

DIGITAL DISPLAY INTERFACE

It can also use DVI or HDMI to transmit progressive high-resolution digital TV signals when connected to digital display devices such as liquid crystal displays, liquid crystals or DLP projectors.

DVI

The Digital Visual Interface (DVI) standard for display devices is developed by the Digital Display Working Group. Commonly used are pure digital DVI-D and digital / analog compatible DVI-I interfaces. The DVI-D interface has a total of 24 pins, and has two working modes: SingleLink and DualLink. Both single-link and dual-link use only one plug and one cable. For single-link, only half of the 24 pins or 12 pins are used. For dual-link, all 24 pins are used. As can be seen from Table 2, DVI-D can support 720 / 60P (up to 85P) progressive scan display when using a single link, and 1080 / 60P (up to 85P) progressive scan display when using dual link.

Table: Signal format for DVI-D transmission

Link method	Number of pins used	Transmission bit rate	Highest resolution	Comments
	12 pin	3.96Gb / s (clock frequency 165MHz)	Computer display	UXGA (1600 / 60Hz)

	SXGA (1280x1024) / 85Hz			
DVI single link		1080 / 60i (1920x1080 / 60 fields, interlaced)		
	HDTV display	720 / 60P (1280x720 / 60 frames, progressive, up to 85 frames)		
DVI dual link	24 pin	2x3.96Gb / s (clock frequency 165MHz)	Computer display	QXO (2048 / 60Hz)
HDTV display	1080 / 60P (1920x1080 / 60 frames, progressive, up to 85 frames)			

HDMI

High Definition Multimedia Interface HDMI (High Definition Multimedia Interface) is developed on the basis of DVI. Compared with DVI, HDMI adds multi-channel digital audio transmission function, so it is more suitable for high definition home theater equipment. Connection.

The HDMI standard was developed by the HDMI Forum (HDMI Founder). Its plug pins are compatible with DVI but use different package sizes. From Figure 1, you can see that the HDMI plug size is only a little over half that of DVI. Currently, HDMI can transmit the same signal format as DVI.

Unlike video serial digital component interfaces SDI and HD-SDI, digital display interfaces DVI and HDMI transmit RGB signals.

ANALOG INTERFACE

The traditional analog composite video interface cannot transmit progressive scan signals, and the analog component video interface (Y / RY / BY) can support progressive and standard definition and high-definition television signal transmission. In order to facilitate the distinction, the SD analog component interfaces supporting HD analog components and progressive scanning are generally labeled Y / Pr / Pb, and the ordinary SD analog component interfaces are labeled Y / Cr / Cb.

15-pin analog VGA interface (D-Sub) commonly used in the IT industry can also be used to transmit progressive scanning signals when connecting display devices over short distances. The analog 15-pin VGA interface can support VGA, SVGA, XGA, SXGA, UXGA, and QXGA resolution. Progressive scan signal transmission, the refresh rate (frame rate) can reach 60Hz to 80Hz. Like the digital display interface, the analog display interface also transmits RGB signals.

Most progressive scan DVD players currently on the

market use analog component video interfaces or VGA interfaces to output progressive scan signals.

PROGRESSIVE SCAN DVD

The DVD video uses the MPEG2 compression algorithm. According to different sources of the program, the DVD uses two different scanning formats to compress the MPEG2 video file:

◆ The film

NTSC DVDs released by 60Hz country and region (North America, mainly Hollywood) producers compress 24 movie frames per second into 480 / 24P progressive scan MPEG2 files at 24 frames per second. When the DVD player replays the disc, if it detects that the compressed file is in the 24P progressive scan format, it will automatically start the 3-2 pull-down conversion function in the signal processing chip to convert the 480 / 24P progressive scan signal into 480 / 60i (NTSC) Interlace signal output. In addition to motion picture film, in recent years, progressive scan programs shot and produced by 1080 / 24P high-definition television equipment have been converted to 480 / 24P by a down converter and compressed into progressively scanned MPEG2 files to make DVDs.

The PAL version of DVD issued by some European film producers compresses 25 movie frames per second to 25 frames per second at 576 / 25P progressive scan MPEG2 files. If the DVD player detects that the compressed file is 25P progressive when scanning the format, the progressive-to-interlace conversion function in the signal processing chip is automatically activated, that is, a frame of progressively scanned TV signals is divided into odd and even fields, and the Psf is transmitted separately 576 / 50i (PAL) interlaced signal output. Progressive scan programs shot and produced with 1080 / 25P high-definition television equipment can also be converted to 576 / 25P by a downconverter and compressed into progressively scanned MPEG2 files to make DVDs.

When compressing 576 / 50i (PAL) or 480 / 60i (NTSC) interlaced TV signals, the DVD encoding directly uses the same interlaced compression algorithm as the signal source. When the DVD player replays this disc, the decoder will output the corresponding Interlaced signal.

◆ **Progressive Scan DVD Player and Progressive Scan TV**

Compared with ordinary DVD, progressive scan DVD (PDVD) player adds progressive scan component video

or VGA output interface. When replaying 24P movie format DVD disc, progressive scan DVD player can convert 480 / 24P progressive The line scan signal is converted into 480 / 60P progressive scan signal through 2.5 times frame rate processing similar to the 3-2 pull-down conversion. The progressive analog signal is output through the progressive analog component interface; when a 25P movie format DVD disc is played back, the progressive DVD can convert 576 / 25P The progressive signal is converted into a 576 / 50P progressive scan signal through a simple 2x frame rate processing (displayed twice per frame). Because ordinary TVs cannot display such 480P or 576P progressive scan signals, a progressive scan DVD player must be used with a TV with a progressive scan input interface that supports 480P and 576P progressive scan. Some mid- to high-priced large-screen TVs currently on the market are equipped with a progressive scan input interface, which can be used with progressive scan DVDs.

Under the condition that the number of effective scanning lines is the same, the Kell Factor of progressive scanning is 30% higher than that of interlaced scanning. Therefore, progressive DVD and progressive TV are used to display the same

progressive source disc. The vertical sharpness of the image is about 30% higher than that of ordinary interlaced DVD.

In addition, flat-panel display or projection devices such as LCD, PDP, and DLP are progressive scan display in nature. This type of progressive display device cooperates with a progressive DVD player to reproduce progressive discs from shooting, production to display "Progressive", which avoids the loss of image quality caused by the progressive-interlaced-progressive conversion, and can effectively improve the display image quality.

Progressive DVD uses interpolated arithmetic to convert interlaced signals to progressive signals when playing interlaced TV program discs. When playing 480 / 60i (NTSC) interlaced programs, 480 / 60P output by PDVD is actually 480 / 60i after interpolation; 576 / 50P output when playing 576 / 50i (PAL) interlaced programs is 576 after interpolation / 50i, so it does not improve the vertical sharpness of the reproduced image like a progressive disc.

In summary, there are three basic conditions for using a progressive scan DVD to display high-quality images:

- Use progressive scan DVD player
- Playback of DVD movie discs in progressive scan format

-Use a TV or monitor with a progressive scan input interface and support 480P or 576P progressive scan display function

The successful application of the progressive scan DVD mode in the SD television era shows that it is a feasible option to use progressive production and distribution even in an environment where interlaced equipment has an absolute advantage. Programs produced by progressive scanning with appropriate technology can be viewed in an interlaced environment without any obstacles, and when users choose a better device, the same program source can get higher image quality than existing interlaced devices, And this better image quality is achieved under the condition of reducing transmission or distribution resources (code rate), because the compression efficiency of progressive scanning signals is higher than that of interlaced when using compression coding to transmit or distribute programs.

INTERCONVERSION OF PROGRESSIVE AND INTERLACED SIGNALS

In actual production, it is often necessary to convert progressive and interlaced signals to each other. This conversion mainly refers to the following two cases:

1. Interlaced and progressive signals with the same number of effective scanning lines and the same number of scanning frames are converted to each other, such as between 480 / 30P and 480 / 60i.
2. The number of effective scanning lines is the same, and the interlaced and progressive signals with different scanning frames are converted to each other, such as between 480 / 60P and 480 / 60i.

From the perspective of signal processing and sampling, the conversion from progressive to interlaced in either case is the process of splitting and transmitting or reducing the information, so its processing is relatively simple and easy, without causing loss of image quality. The conversion from

interlaced to progressive is an interpolation process that requires additional information, so the result obtained after the conversion is worse than the original image taken progressively.

The conversion between progressive and interlaced signals can be done in real time either by hardware (scan converter or so-called format converter), or by non-real-time precision processing in software in non-linear equipment.

Progressive to interlaced

Take 480 / 30P or 60P to 60i as examples:

◆ 480 / 30P to 60i is a simple piecewise transmission of Psf, that is, a progressive frame is divided into odd and even fields and transmitted separately, which is the 480 / 30Psf transmission method introduced earlier.

The amount of information before and after conversion has not changed. The image quality is the same. The difference between the interlaced signal obtained by Psf and the interlaced image is that the sampling frequency is different. A progressive 30Psf sample is taken once per frame to obtain a frame, and this frame is divided into two fields to be transmitted separately. The interlaced 60i is sampled twice per frame, that is, each field is sampled once, so 30Psf is not as smooth as 60i when displaying a moving picture, and there is a feeling of jitter.

- ◆ For 480 / 60P to 60i, you only need to remove the even lines from the first frame of 60P and keep the odd lines to become the odd field, and remove the odd lines from the second frame and keep the even lines to become the even field. The resulting 60i interlaced signal is exactly the same quality as the image captured with interlaced.

Interlace to progressive

Take 480 / 60i to 60P or 30P as an example:

- ◆ 480 / 60i to 60P needs to double the amount of information, that is, to double the scanning line of a field into a frame by interpolation. Because these "out of nothing" scan lines are calculated by data from adjacent rows rather than captured, their effective information has not increased. In some non-linear editing devices, this interlaced to progressive conversion is called De-Interlace. The image quality of interlaced to progressive conversion depends on the accuracy of the interpolation operation. The more complex the algorithm used, the higher the accuracy and the better the conversion quality. However, complex algorithms can only make the converted image look more "comfortable" but cannot increase the amount of effective information.

- ◆ The processing of 480 / 60i to 30P is exactly the same as that of 480 / 60i to 60P, except that the

converted 60P signal is cut in half, that is, 30P is obtained by removing one frame every other frame. Unlike the conversion where the amount of information does not change when 30P turns to 60i, the effective information amount after 60i turns to 30P is only half of that before conversion, and the image quality obtained is the worst. This is because during interlaced shooting, each frame is sampled twice, that is, each field is sampled once, but the two fields are sampled at different times. When shooting moving images, the two fields are in space because the sampling time of the two fields is different. The above fields cannot be completely overlapped, so the two fields obtained during interlaced shooting of the moving picture cannot be combined into a complete picture.

DIGITAL TV TRENDS- PROGRESSIVE, INTERLACED OR PROGRESSIVE

In the analog era, in order to eliminate the flickering of the display image, the refresh rate of shooting, production and display must be increased at the same time. The development of digital display technology can make the refresh rate of the display end higher than that of shooting and production. Making programs has become one of the choices for digital TV program production. The successful experience of DVD proves the feasibility of using progressive production, interlaced or progressive display mode in an interlaced environment.

Because IT and digital movies use progressive scanning, the use of progressive scanning to produce programs creates conditions for the integration of television and film, and television and IT.

Psf makes the conversion from progressive to interlaced, and the transmission of progressive signals

is very simple. The existing interlaced environment after using Psf can support most progressive programming, without making the cost of progressive production higher than the existing interlaced.

Production.

Countries and regions that have already broadcast DTV, such as the United States, Japan, Europe, and Australia, have paid great attention to progressive production. For example, many 480 / 60i and 1080 / 60i programs broadcast by many DTVs in the United States use 480 / 30P and 1080 / 24P. Or 1080 / 30P production; Australian DTV interlaced scanning formats are 1080 / 50i and 576 / 50i, 1080 / 50i (1080 / 25Psf) and 576 / 50i (576 / 25Psf).

HD PRODUCTION CHOICE: 50I OR 25P?

The GY / T155-2000 high-definition television program production and exchange video parameter values stipulated by the General Administration of Radio, Film and Television. The code rates (occupied production resources) of 1080 / 50i and 1080 / 25P are exactly the same. After using Psf transmission, 1080 / 25Psf is completely compatible with 1080 / 50i, just like many 1080 / 60i broadcasted by American digital TV.

Table : Comparison of 1080 / 50i and 1080 / 25P (25Psf)

1080 / 50i	1080 / 25P (25Psf)
advantage	Disadvantage
The moving picture is smoother than 25P	The moving picture is not as smooth as the 50i
Disadvantage	advantage

Low vertical definition	High vertical definition
Not conducive to the international exchange and future development of the program	Conducive to the international exchange and future development of programs
Low compression efficiency	High compression efficiency
Interlaced to progressive degradation of image quality	Simple and lossless conversion from progressive to interlaced

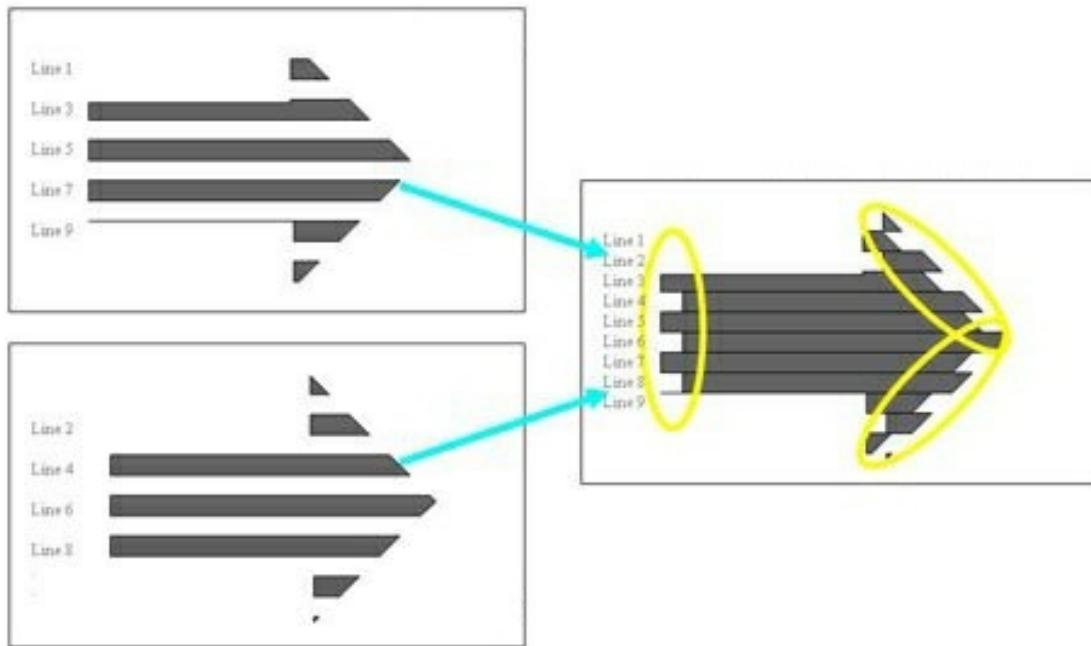
The cost of producing 25P and 50i shows is almost the same, but the production of 25P shows has both 25P and 50i scanning formats. It can be used for progressive or interlaced TV broadcasts as well as digital movies. Can't get high quality 25P.

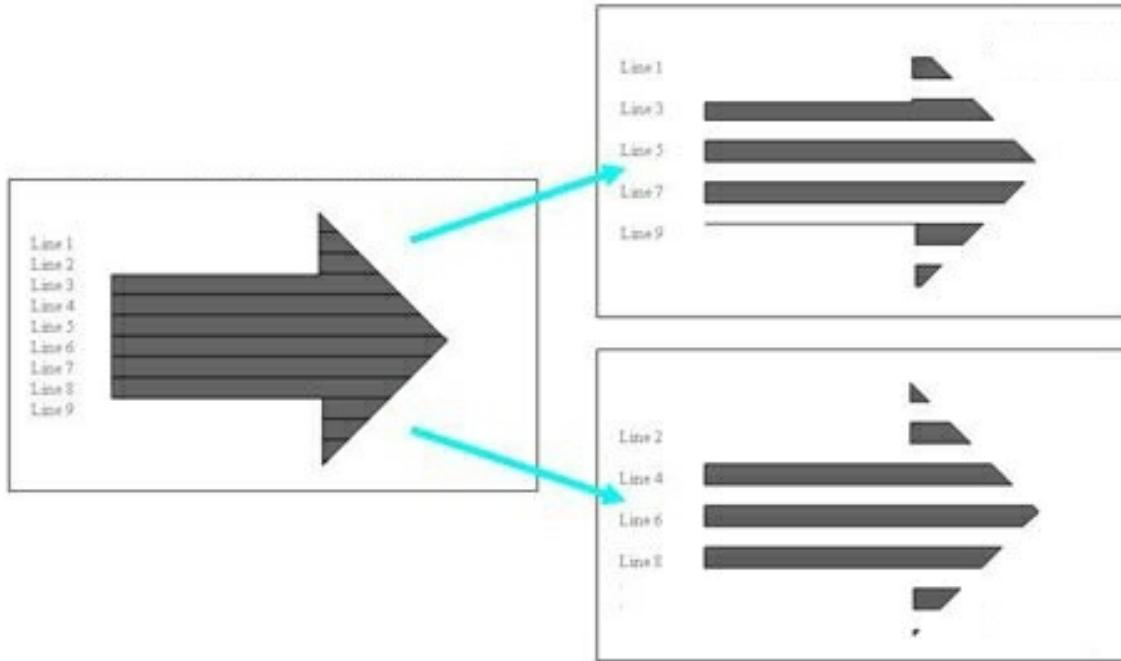
CAN I USE 50I INTERLACED SCANNING EQUIPMENT TO COPY AND MAKE 25PSF PROGRESSIVE SCANNING PROGRAMS?

The data volume of 1080 / 25P and 1080 / 50i is exactly the same. 50i captures 50 fields and 50 samples per second, 25P captures 25 frames and 25 samples per second, but 25Psf splits a frame into two fields and transmits 50 fields per second. The transmission structure is the same as 50i, so all devices supporting 1080 / 50i can record, process and display 1080 / 25Psf signals.

The difference between Psf and interlaced scanning is that progressive Psf is sampled once per frame to obtain a frame, and then this frame is divided into two fields and transmitted separately. The two fields at the receiving end can still be combined into a complete frame; interlaced the scan is sampled twice per frame,

that is, once per field, but the two fields are sampled at different times. When shooting moving images, the two fields cannot completely overlap in space due to the different sampling times of the two fields. Thus live shooting movies cannot get interlaced when the two surfaces into a synthetic complete picture.





The progressive scan signal transmitted by 25Psf is not changed after being recorded with a 50i video recorder. In essence, it is still a progressive 25P scan. However, the oscilloscope and other tools will recognize the output signal of the 50i video recorder as "1080 / 50i" instead of 1080 / 25Psf. The 25P progressive shooting of 25Psf and transmission of the signal recorded by 50i and 25Psf is the 25P signal after re-recording. The two fields transmitted in one shot can still be combined into a complete frame.

After the progressive scanning signal transmitted by

25Psf is edited with 50i equipment, the interlaced linear or non-linear editing equipment processes the signal in units of fields when completing the effect synthesis effect, so the content of the signal after 50i editing has changed. There is no guarantee that the relationship between the two occasions becomes the original shooting frame. Therefore, when making 25P programs, you must make sure that every link of shooting and production uses equipment that supports progressive scanning format.

In short, due to the compatibility of Psf signal transmission with interlaced equipment, a large number of devices that only support 50i interlaced scanning can be used for recording, transmission and display of 25P progressive signals. Simple copying with 50i equipment will not affect Progressive production causes damage, but equipment that only supports 50i interlaced production cannot be used for 25P progressive production.

DEINTERLACING

In ffmpeg, avfilter is used for deinterlacing, that is, image filter. There are many filters in ffmpeg, which are very powerful. The deinterlacing filter is yadif.

The basic filter usage process is:

Decoded picture ---> buffer filter ----> other filter --
--> buffersink filter ---> processed picture

All filters form a filter chain. The two filters that must be used are the buffer filter and the buffersink filter. The former is to load the decoded picture into the filter chain, and the latter is to handle the processing. The picture is read from the filter chain. Then the de-interlaced filter chain should look like this:

buffer filter ---> yadif filter ---> buffersink filter

Filter-related structures:

AVFilterGraph: manage all filter images

AVFilterContext: filter context

AVFilter: filter

Let's see how to create a filter chain:

The first step is to create an AVFilterGraph

```
AVFilterGraph * filter_graph = avfilter_graph_alloc();
```

The second step is to get the filter to be used:

```
AVFilter * filter_buffer = avfilter_get_by_name ("buffer");
AVFilter * filter_yadif = avfilter_get_by_name ("yadif");
AVFilter * filter_buffersink = avfilter_get_by_name ("buffersink");
```

The third step is to create a filter context, namely AVFilterContext :

```
int avfilter_graph_create_filter (AVFilterContext ** filt_ctx, const
AVFilter * filt, const char * name, const char * args, void * opaque,
AVFilterGraph * graph_ctx);
```

Parameter description: filt_ctx is used to save the created filter context, filt is the filter, name is the filter name (should be unique in the filter chain), args is the parameter passed to the filter (each filter is different, you can (Found in the corresponding filter code), opaque is not used in the code, and graph_ctx is the filter image management pointer. For example:

```
AVFilterContext * filter_buffer_ctx, * filter_yadif_ctx, *
filter_buffersink_ctx;
// Create a buffer filter
snprintf (args, sizeof (args),
"video_size=% dx% d: pix_fmt=% d: time_base=% d /% d:
pixel_aspect=% d /% d",
```

```

    dec_ctx-> width, dec_ctx-> height, dec_ctx -> pix_fmt,
    dec_ctx-> time_base.num, dec_ctx-> time_base.den,
    dec_ctx-> sample_aspect_ratio.num, dec_ctx->
sample_aspect_ratio.den);
    avfilter_graph_create_filter (& filter_buffer_ctx,
avfilter_get_by_name ( "Buffer"), "in", args, NULL, filter_graph);
    // create yadif filter
    avfilter_graph_create_filter (& filter_yadif_ctx ,
avfilter_get_by_name ("yadif"), "yadif", "mode = send_frame: parity =
auto: deint = interlaced", NULL, filter_graph);
    // Create a buffersink filter
    enum AVPixelFormat pix_fmts [] = {AV_PIX_FMT_YUV420P,
AV_PIX_FMT_NONE};
    avfilter_graph_create_filter (& filter_buffersink_ctx,
avfilter_get_by_name ("buffersink"), "out",  NULL, NULL, filter_graph);
    av_opt_set_int_list (filter_buffersink_ctx, "pix_fmts", pix_fmts,
AV_PIX_FMT_NONE, AV_OPT_SEARCH_CHILDREN);

```

Step four, connect the filter

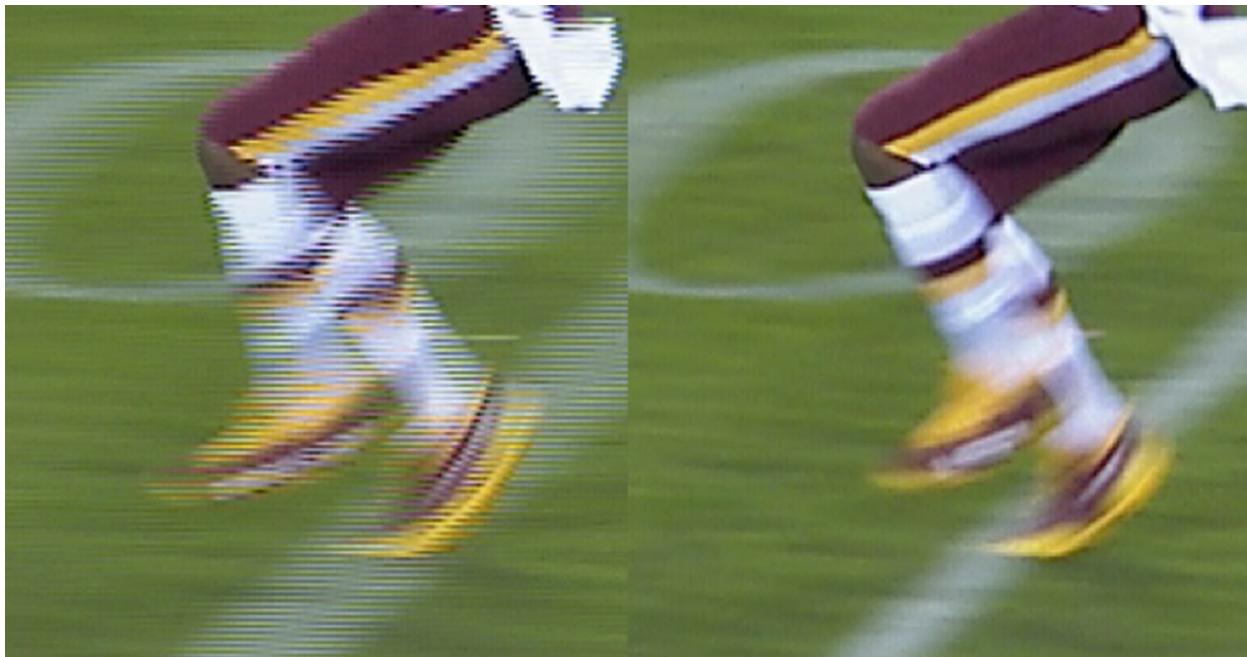
```

    avfilter_link (filter_buffer_ctx, 0, filter_yadif_ctx, 0);
    avfilter_link (filter_yadif_ctx, 0, filter_buffersink_ctx, 0);
The fifth step is to check that all configurations are correct:
    if ((ret = avfilter_graph_config (player-> filter_graph, NULL)) <0)
{
    LOGE (0, "avfilter_graph_config:% d \n", ret);
    goto end;
}
//Note that all the functions above should check the return value. Here is
the //abbreviation. If there is no error, the filter chain is created.
//How to use the filter chain for filtering, mainly using two functions:
    // Push the decoded frame to the filter chain
    int av_buffersrc_add_frame_flags (AVFilterContext * buffer_src,
AVFrame * frame, int flags);
    // Pull out the processed frame:

```

```
int av_buffersink_get_frame (AVFilterContext * ctx, AVFrame *  
frame);  
for example:  
av_buffersrc_add_frame_flags (filter_buffer_ctx, orgin_frame,  
AV_BUFFERSRC_FLAG_KEEP_REF);  
while (1)  
{  
    ret = av_buffersink_get_frame (filter_buffersink_ctx, frame);  
    if (ret == AVERROR (EAGAIN) || ret == AVERROR_EOF) {  
        break;  
    }  
    display (frame);  
};
```

Comparison of the image before deinterlacing and the image after deinterlacing:



Although it is more blurred, the horizontal stripes are indeed removed.

After looking for it, I found that there is also an anti-interlacing filter kerndeint in ffmpeg, which is licensed by the GPL. When using it, you need to configure to open the GPL

After passing in the parameter threshold = 0: map = 0: order = 1: sharp = 0: two way = 0, it seems to be better than yadif, and the effect is as follows:

ffmpeg also has some deinterlace filters. The test found that in terms of filtering time and picture quality, the better is the pp / lb filter.

5. I-B-P FRAME AND TIMEBASE

I-FRAME

I frame (Intra-coded picture, often referred to as key frame) contains a complete image information, which belongs to the intra-coded image, does not include motion vectors, and does not need to refer to other frames when decoding. Therefore, the channel can be switched at the I-frame image without causing the image to be lost or unable to be decoded. I-frame images are used to prevent the accumulation and diffusion of errors. In a closed GOP, the first frame of each GOP must be an I frame, and the data of the current GOP will not refer to the data of the preceding and following GOPs.

P-FRAME

P-frame (Predictive-coded picture, predictive-coded picture frame) is an inter-frame coded frame, using the previous I-frame or P-frame for predictive coding.

B-FRAME

B-frame (Bi-directionally predicted picture, bi-directionally predicted picture frame) is an inter-frame coded frame, and the I-frame or P-frame before and / or after it is used for bidirectional prediction coding. B frames cannot be used as reference frames.

B-frames have higher compression rates, but require more buffering time and higher CPU usage. Therefore, B-frames are suitable for local storage and video on demand, and are not suitable for live broadcast systems with high real-time requirements.

DTS AND PTS

Fortunately, both the audio and video streams have the information about how fast and when you are supposed to play them inside of them. Audio streams have a sample rate, and the video streams have a frames per second value. However, if we simply synced the video by just counting frames and multiplying by frame rate, there is a chance that it will go out of sync with the audio. Instead, packets from the stream might have what is called a decoding time stamp (DTS) and a presentation time stamp (PTS). To understand these two values, you need to know about the way movies are stored. Some formats, like MPEG, use what they call "B" frames (B stands for "bidirectional"). The two other kinds of frames are called "I" frames and "P" frames ("I" for "intra" and "P" for "predicted"). I frames contain a full image. P frames depend upon previous I and P frames and are like diffs or deltas. B frames are the same as P frames, but depend upon information found in frames that are displayed both before and after them! This explains why we might not have a finished frame after we

call avcodec_decode_video2.

So let's say we had a movie, and the frames were displayed like: I B B P. Now, we need to know the information in P before we can display either B frame. Because of this, the frames might be stored like this: I P B B. This is why we have a decoding timestamp and a presentation timestamp on each frame. The decoding timestamp tells us when we need to decode something, and the presentation time stamp tells us when we need to display something. So, in this case, our stream might look like this:

PTS: 1 4 2 3

DTS: 1 2 3 4

Stream: I P B B

TIME BASE AND TIMESTAMP IN FFMPEG

1. THE CONCEPT OF TIME BASE AND TIMESTAMP

In FFmpeg, the time base (`time_base`) is a unit of timestamp (`timestamp`). The timestamp value is multiplied by the time base to obtain the actual time value (in seconds, etc.). For example, if the `dts` of a video frame is 40, the `pts` is 160, and its `time_base` is $1/1000$ second, then the decoding time of this video frame is 40 milliseconds ($40/1000$), and the display time is 160 milliseconds ($160 / 1000$). The type of timestamp (`pts` / `dts`) in FFmpeg is `int64_t`. If a `time_base` is regarded as a clock pulse, then `dts` / `pts` can be regarded as the count of clock pulses.

2. THREE TIME BASES TBR, TBN AND TBC

Different packaging formats have different time bases. Different stages in FFmpeg's processing of audio and video also use different time bases. There are three kinds of time bases in FFmpeg. The printed values of tbr, tbn, and tbc on the command line are the reciprocals of these three kinds of time bases: tbn: the time base in the corresponding container. The value is the inverse tbc of AVStream.time_base: corresponding to the time base in the codec. The value is the inverse tbr of AVCodecContext.time_base guessed from the video stream, it may be the frame rate or field rate (2 times the frame rate). There are three different time bases for time stamps in FFmpeg. The values printed are actually reciprocals of these, i.e. 1/tbr, 1/tbn and 1/tbc. tbn is the time base in AVStream that has come from the container, I think. It is used for all AVStream time stamps. tbc is the time base in AVCodecContext for the codec used for a particular stream. It is used for all AVCodecContext and related time stamps. tbr is guessed from the video stream and is the value users

want to see when they look for the video frame rate, except sometimes it is twice what one would expect because of field rate versus frame rate.

3. INTERNAL TIME BASE AV_TIME_BASE

In addition to the above three time bases, FFmpeg also has an internal time base AV_TIME_BASE (and AV_TIME_BASE_Q in fractional form)

```
// Internal time base represented as integer
#define AV_TIME_BASE      1000000
// Internal time base represented as fractional value
#define AV_TIME_BASE_Q    (AVRational){1, AV_TIME_BASE}
```

AV_TIME_BASE and AV_TIME_BASE_Q are used for FFmpeg internal function processing. The time value calculated using this time base represents microseconds.

4. TIME VALUE FORM CONVERSION

av_q2d () converts time from AVRational form to double form. AVRational is a fraction type, double is a double-precision floating-point number type, and the result unit of the conversion is seconds. The values before and after the conversion are based on the same time base, only the representation of the values is different.

av_q2d () is implemented as follows:

1

```
static inline double av_q2d(AVRational a){  
    return a.num / (double) a.den;  
}
```

av_q2d () is used as follows:

```
AVStream stream;  
AVPacket packet;  
Packet playback time value: timestamp (in seconds) = packet.pts ×  
av_q2d(stream.time_base);  
Packet playback duration value: duration (in seconds) = packet.duration ×  
av_q2d(stream.time_base);
```

5 TIME BASE CONVERSION FUNCTION

av_rescale_q () is used to convert between different time bases. It is used to convert time values from one time base to another.

1

```
int64_t av_rescale_q(int64_t a, AVRational bq, AVRational cq) av_const;
```

av_packet_rescale_ts () is used to convert various time values in AVPacket from one time base to another time base.¹¹²

```
void av_packet_rescale_ts(AVPacket *pkt, AVRational tb_src,  
AVRational tb_dst);
```

6 TIME-BASED CONVERSION DURING THE ENCAPSULATION PROCESS

The time base in the container (AVStream.time_base, tbn) is defined as follows:

1

```
typedef struct AVStream {  
    .....  
    /**/  
     * This is the fundamental unit of time (in seconds) in terms  
     * of which frame timestamps are represented.  
     *  
     * decoding: set by libavformat  
     * encoding: May be set by the caller before avformat_write_header()  
     * to  
     *      provide a hint to the muxer about the desired timebase. In  
     *      avformat_write_header(), the muxer will overwrite this field  
     *      with the timebase that will actually be used for the timestamps  
     *      written into the file (which may or may not be related to the  
     *      user-provided one, depending on the format).  
     */  
    AVRational time_base;  
    .....  
}
```

AVStream.time_base is the time unit of pts and dts in AVPacket. The time_base in the input stream and the output stream is determined as follows:

For the input stream: After opening the input file, call avformat_find_stream_info () to obtain the time_base in each stream.

For the output stream: After opening the output file, call avformat_write_header () to determine the time_base of each stream according to the output file encapsulation format and write it to the output file

Different packaging formats have different time bases. In the process of transpackaging (converting one packaging format to another packaging format), the time-base conversion related code is as follows:

1

```
av_read_frame(ifmt_ctx, &pkt);
pkt.pts = av_rescale_q_rnd(pkt.pts, in_stream->time_base, out_stream-
>time_base, AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX);
pkt.dts = av_rescale_q_rnd(pkt.dts, in_stream->time_base, out_stream-
>time_base, AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX);
pkt.duration = av_rescale_q(pkt.duration, in_stream->time_base,
out_stream->time_base);
```

The following code has the same effect as the above code:

```
// Read the packet from the input file
```

```
av_read_frame(ifmt_ctx, &pkt);
// Convert each time value in the packet from the input stream
encapsulation format time base to the output stream encapsulation format
time base
av_packet_rescale_ts(&pkt, in_stream->time_base, out_stream-
>time_base);
```

The time base `in_stream->time_base` in the stream is the time base `out_stream->time_base` in the container.

For example, the `time_base` of the flv package format is `{1,1000}`, and the `time_base` of the ts package format is `{1,90000}`.

7 TIME BASE CONVERSION DURING TRANSCODING

The time base in the codec
(AVCodecContext.time_base, tbc in section 3.2) is
defined as follows:

1

```
typedef struct AVCodecContext {  
    .....  
    /**/  
     * This is the fundamental unit of time (in seconds) in terms  
     * of which frame timestamps are represented. For fixed-fps content,  
     * timebase should be 1/framerate and timestamp increments should be  
     * identically 1.  
     * This often, but not always is the inverse of the frame rate or field rate  
     * for video. 1/time_base is not the average frame rate if the frame rate  
is not  
     * constant.  
     *  
     * Like containers, elementary streams also can store timestamps,  
1/time_base  
     * is the unit in which these timestamps are specified.  
     * As example of such codec time base see ISO/IEC 14496-2:2001(E)  
     * vop_time_increment_resolution and fixed_vop_rate  
     * (fixed_vop_rate == 0 implies that it is different from the framerate)  
     *  
     * - encoding: MUST be set by user.
```

```
* - decoding: the use of this field for decoding is deprecated.  
*      Use framerate instead.  
*/  
AVRational time_base;  
.....  
}
```

The above comment indicates that AVCodecContext.time_base is the reciprocal of the frame rate (video frame), and each frame timestamp is incremented by 1, then tbc is equal to the frame rate. This parameter should be set by the user during the encoding process. This parameter is obsolete during decoding. It is recommended to use the reciprocal of the frame rate as the time base directly. Here's a question: According to the note here, the video stream with a frame rate of 25 should have a tbc of 25, but the actual value is 50. I don't know what to explain? Is tbc obsolete and not relevant?

According to the suggestions in the comments, in actual use, during the video decoding process, we do not use AVCodecContext.time_base, but use the reciprocal of the frame rate as the time base. During the video encoding process, we set AVCodecContext.time_base to the reciprocal of the frame rate.

7.1 VIDEO STREAM

The video is played frame by frame, so the decoded original video frame time base is $1 / \text{framerate}$.
Time base conversion process in video decoding process:

1

```
AVFormatContext *ifmt_ctx;  
AVStream *in_stream;  
AVCodecContext *dec_ctx;  
AVPacket packet;  
AVFrame *frame;  
  
// Read encoded frames from input file  
av_read_frame(ifmt_ctx, &packet);  
  
// Time base conversion  
int raw_video_time_base = av_inv_q(dec_ctx->framerate);  
av_packet_rescale_ts(packet, in_stream->time_base,  
raw_video_time_base);  
  
// decoding  
avcodec_send_packet(dec_ctx, packet)  
avcodec_receive_frame(dec_ctx, frame);
```

Time base conversion process in video encoding process:

1

```
AVFormatContext *ofmt_ctx;
AVStream *out_stream;
AVCodecContext *dec_ctx;
AVCodecContext *enc_ctx;
AVPacket packet;
AVFrame *frame;
// coding
avcodec_send_frame(enc_ctx, frame);
avcodec_receive_packet(enc_ctx, packet);
// Time base conversion
packet.stream_index = out_stream_idx;
enc_ctx->time_base = av_inv_q(dec_ctx->framerate);
av_packet_rescale_ts(&opacket, enc_ctx->time_base, out_stream-
>time_base);

// Write encoded frames to output media file
av_interleaved_write_frame(o_fmt_ctx, &packet);
```

7.2 AUDIO STREAM

The audio is played at the sampling point, so the decoded original audio frame time base is $1 / \text{sample_rate}$

Time base conversion processing in audio decoding process:

```
AVFormatContext *ifmt_ctx;
AVStream *in_stream;
AVCodecContext *dec_ctx;
AVPacket packet;
AVFrame *frame;

// Read encoded frames from an input file
av_read_frame(ifmt_ctx, &packet);

// Time base conversion
int raw_audio_time_base = av_inv_q(dec_ctx->sample_rate);
av_packet_rescale_ts(packet, in_stream->time_base,
raw_audio_time_base);

// decoding
avcodec_send_packet(dec_ctx, packet)
avcodec_receive_frame(dec_ctx, frame);
```

Time base conversion process in audio encoding process:1,

```
AVFormatContext *ofmt_ctx;
```

```
AVStream *out_stream;
AVCodecContext *dec_ctx;
AVCodecContext *enc_ctx;
AVPacket packet;
AVFrame *frame;

// coding
avcodec_send_frame(enc_ctx, frame);
avcodec_receive_packet(enc_ctx, packet);

// Time base conversion
packet.stream_index = out_stream_idx;
enc_ctx->time_base = av_inv_q(dec_ctx->sample_rate);
av_packet_rescale_ts(&opacket, enc_ctx->time_base, out_stream-
>time_base);
// Write encoded frames to output media file
av_interleaved_write_frame(o_fmt_ctx, &packet);
```

6. CODECS

VIDEO CODEC

A video codec is an electronic circuit or software that compresses or decompresses digital video. It converts uncompressed video to a compressed format or vice versa. In the context of video compression, "codec" is a concatenation of "encoder" and "decoder"—a device that only compresses is typically called an encoder, and one that only decompresses is a decoder.

The compressed data format usually conforms to a standard video compression specification. The compression is typically lossy, meaning that the compressed video lacks some information present in the original video. A consequence of this is that decompressed video has lower quality than the original, uncompressed video because there is insufficient information to accurately reconstruct the original video.

There are complex relationships between the video quality, the amount of data used to represent the video (determined by the bit rate), the complexity of the encoding and decoding algorithms, sensitivity to

data losses and errors, ease of editing, random access, and end-to-end delay (latency).

LIST OF OPEN-SOURCE CODECS

VIDEO CODEC LIST

- x264 – H.264/MPEG-4 AVC implementation.
x264 is not a codec (encoder/decoder); it is just an encoder (it cannot decode video).
- OpenH264 – H.264 baseline profile encoding and decoding
- x265 – An encoder based on the High Efficiency Video Coding (HEVC/H.265) standard.
- Xvid – MPEG-4 Part 2 codec, compatible with DivX
- libvpx – VP8 and VP9 implementation; formerly a proprietary codec developed by On2 Technologies, released by Google under a BSD-like license in May 2010.
- FFmpeg codecs – Codecs in the libavcodec library from the FFmpeg project (FFV1, Snow, MPEG-1, MPEG-2, MPEG-4 part 2, MSMPEG-4, WMV2, SVQ1, MJPEG, HuffYUV and others). Decoders in the libavcodec (H.264, SVQ3, WMV3, VP3, Theora, Indeo, Dirac, Lagarith and others).

- Lagarith – Video codec designed for strong lossless compression in RGB(A) colorspace (similar to ZIP/RAR/etc.)
- libtheora – A reference implementation of the Theora format, based on VP3, part of the Ogg Project
- Dirac as dirac-research, a wavelet based codec created by the BBC Research, and Schrödinger, an implementation developed by David Schleef. [1]
- Huffyuv – Lossless codec from BenRG
- Daala – Experimental Video codec which was under development by the Xiph.Org Foundation and finally merged into AV1.
- Thor – Experimental royalty free video codec which was under development by Cisco Systems, and merged technologies into AV1.
- Turing - A High Efficiency Video Coding (HEVC/H.265) encoder implemented by BBC Research.
- libaom – Reference implementation for the royalty free AV1 video coding format by AOMedia, inheriting technologies from VP9, Daala and Thor.

AUDIO CODEC LIST

- FLAC – Lossless codec developed by Xiph.Org Foundation.
- LAME – Lossy compression (MP3 format).
- TooLAME/TwoLAME – Lossy compression (MP2 format).
- Musepack – Lossy compression; based on MP2 format, with many improvements.
- Speex – Low bitrate compression, primarily voice; developed by Xiph.Org Foundation. Deprecated in favour of Opus according to www.speex.org.
- CELT – Lossy compression for low-latency audio communication
- libopus – A reference implementation of the Opus format, the IETF standards-track successor to CELT. (Opus support is mandatory for WebRTC implementations.)
- libvorbis – Lossy compression, implementation of the Vorbis format; developed by Xiph.Org Foundation.

- iLBC – Low bitrate compression, primarily voice
- iSAC – Low bitrate compression, primarily voice; (free when using the WebRTC codebase)
- TTA – Lossless compression
- WavPack – Hybrid lossy/lossless
- Bonk – Hybrid lossy/lossless; supported by fre:ac (formerly BonkEnc)
- Apple Lossless – Lossless compression (MP4)
- Fraunhofer FDK AAC – Lossy compression (AAC)
- FFmpeg codecs in the libavcodec library, e.g. AC-3, AAC, ADPCM, PCM, Apple Lossless, FLAC, WMA, Vorbis, MP2, etc.
- FAAD2 – open-source decoder for Advanced Audio Coding. There is also FAAC, the same project's encoder, but it is proprietary (but still free of charge).
- libgsm – Lossy compression (GSM 06.10)
- opencore-amr – Lossy compression (AMR and AMR-WB)
- liba52 – a free ATSC A/52 stream decoder (AC-3)
- libdca – a free DTS Coherent Acoustics decoder

- Codec2 - Low bitrate compression, primarily voice

FFMPEG LIBAVCODEC

The first thing to look at is the AVCodec struct.

```
typedef struct AVCodec {
    const char *name;
    enum CodecType type;
    enum CodecID id;
    int priv_data_size;
    int (*init)(AVCodecContext *);
    int (*encode)(AVCodecContext *, uint8_t *buf, int buf_size, void
    *data);
    int (*close)(AVCodecContext *);
    int (*decode)(AVCodecContext *, void *outdata, int *outdata_size,
                  uint8_t *buf, int buf_size);
    int capabilities;
    struct AVCodec *next;
    void (*flush)(AVCodecContext *);
    const AVRational *supported_framerates; //array of supported
    framerates, or NULL if any, array is terminated by {0,0}
    const enum PixelFormat *pix_fmts;      //array of supported pixel
    formats, or NULL if unknown, array is terminated by -1
} AVCodec;
```

Here we can see that we have some elements to name the codec, what type it is (audio/video), the supported pixel formats and some function pointers for init/encode/decode and close. Now lets see how it is used.

LIBAVCODEC/COOK.C

If we look in this file at the bottom we can see this code:

```
AVCodec cook_decoder =  
{  
    .name      = "cook",  
    .type      = CODEC_TYPE_AUDIO,  
    .id        = CODEC_ID_COOK,  
    .priv_data_size = sizeof(COOKContext),  
    .init       = cook_decode_init,  
    .close      = cook_decode_close,  
    .decode     = cook_decode_frame,  
};
```

First we get an AVCodec struct named cook_decoder. And then we set the variables of cook_decoder. Note that we only set the variables that are needed. Currently there is no encoder so we don't set any. If we now look at the id variable we can see that CODEC_ID_COOK isn't defined in libavcodec/cook.c. It is declared in avcodec.h.

LIBAVCODEC/AVCODEC.H

Here we will find the CodecID enumeration.

```
enum CodecID {  
    ...  
    CODEC_ID_GSM,  
    CODEC_ID_QDM2,  
    CODEC_ID_COOK,  
    CODEC_ID_TRUESPEECH,  
    CODEC_ID_TTA,  
    ...  
};
```

CODEC_ID_COOK is there in the list. This is the list of all supported codecs in FFmpeg, the list is fixed and used internally to id every codec. Changing the order would break binary compatibility. This is all enough to declare a codec. Now we must register them for internal use also. This is done at runtime.

LIBAVCODEC/ALLCODECS.C

In this file we have the avcodec_register_all() function, it has entries like this for all codecs.

```
REGISTER_DECODER(COOK, cook);
```

This macro expands to a register_avcodec() call which registers a codec for internal use. Note that register_avcodec() will only be called when CONFIG_COOK_DECODER is defined. This allows to not compile the decoder code for a specific codec.

LIBAVCODEC/COOK.C

INIT

After ffmpeg knows what codec to use, it calls the declared initialization function pointer declared in the codecs AVCodec struct. In cook.c it is called cook_decode_init. Here we setup as much as we can before we start decoding. The following things should be handled in the init, vlc table initialization, table generation, memory allocation and extradata parsing.

LIBAVCODEC/COOK.C

CLOSE

The cook_decode_close function is the codec clean-up call. All memory, vlc tables, etc. should be freed here.

LIBAVCODEC/COOK.C

DECODE

In cook.c the name of the decode call is
cook_decode_frame.

```
static int cook_decode_frame(AVCodecContext *avctx,
    void *data, int *data_size,
    uint8_t *buf, int buf_size) {
...
```

The function has 5 arguments:

- avctx is a pointer to an AVCodecContext
- data is the pointer to the output buffer
- data_size is a variable that should be set to the output buffer size in bytes (this is usually the number of samples decoded * the number of channels * the byte size of a sample)
- buf is the pointer to the input buffer
- buf_size is the byte size of the input buffer

The decode function shall return the number of bytes consumed from the input buffer or -1 in case of an error. If there is no error during decoding, the return value is usually buf_size as buf should only contain

one 'frame' of data. Bitstream parsers to split the bitstream into 'frames' used to be part of the codec so a call to the decode function could have consumed less than buf_size bytes from buf. It is now encouraged that bitstream parsers be separate.

7. ENCODING

VIDEO ENCODING

1. FFmpeg REQUIRED STRUCTURE FOR VIDEO ENCODING

AVCodec : The AVCodec structure holds an instance of a codec to implement the actual encoding function. Usually we define a pointer to AVCodec structure in the program to point to this instance.

AVCodecContext : AVCodecContext represents the context information represented by AVCodec, and saves some parameters required by AVCodec. For implementing the encoding function, we can set the encoding parameters we specify in this structure. Usually also defines a pointer to AVCodecContext.

AVFrame : The AVFrame structure holds the pixel data before encoding and serves as the input data for the encoder. It is also a pointer in the program.

AVPacket : AVPacket represents the stream packet structure and contains the encoded stream data. The structure can be defined as an object without defining a

pointer. In our program, we integrated these structures into one structure:

```
*****
Struct:      CodecCtx
Description: FFMpeg codec context
*****  
typedef struct
{
    AVCodec *codec;//Point to codec example
    AVFrame *frame;//Save pixel data after decoding / before encoding
    AVCodecContext *c; //Codec context, save some parameter settings
    of the codec
    AVPacketpkt; //Code stream packet structure, including coded code
    stream data
} CodecCtx;
```

2. THE MAIN STEPS OF FFMPEG ENCODING

(1) Enter the Encoding Parameters

At this step we can set up a special configuration file, write the parameters into the configuration file according to something, and then parse the configuration file in the program to obtain the encoded parameters. If there are not many parameters, we can directly use the command line to pass encoding parameters.

(2) INITIALIZE THE REQUIRED FFmpeg STRUCTURE AS REQUIRED

First of all, all functions related to codecs must be registered before the audio and video codec can be used. Registering codecs calls the following functions:

```
avcodec_register_all();
```

After the codec is registered, it searches for the specified codec instance according to the specified CODEC_ID. CODEC_ID usually specifies the codec format. Here we use the most widely used H.264 format as an example. The function called by findc is avcodec_find_encoder, and its declaration format is:

```
AVCodec *avcodec_find_encoder(enum AVCodecID id);
```

The input parameter of this function is an enumerated type of AVCodecID, and the return value is a pointer to an AVCodec structure, which is used to

receive the found codec instance. If not found, the function returns a null pointer. The calling method is as follows:

```
/* find the mpeg1 video encoder */
ctx.codec = avcodec_find_encoder(AV_CODEC_ID_H264);      //Find
pointer to codec object instance based on CODEC_ID
if (!ctx.codec)
{
    fprintf(stderr, "Codec not found\n");
    return false;
}
```

After the AVCodec lookup succeeds, the next step is to allocate an AVCodecContext instance. Allocating an AVCodecContext instance requires the AVCodec we found earlier as a parameter, and the avcodec_alloc_context3 function is called. It is declared as:

```
AVCodecContext *avcodec_alloc_context3(const AVCodec *codec);
```

Its characteristics are similar to avcodec_find_encoder, which returns a pointer to an AVCodecContext instance. If the allocation fails, a null pointer is returned. The calling method is:

```
ctx.c = avcodec_alloc_context3(ctx.codec);                  //Assign
AVCodecContext instance
if (!ctx.c)
```

```
{  
    fprintf(stderr, "Could not allocate video codec context\n");  
    return false;  
}
```

It should be noted that after the allocation is successful, the encoded parameter settings should be assigned to the members of the AVCodecContext.

Now, the pointers of AVCodec and AVCodecContext have been allocated, and then open the encoder object with the pointers of these two objects as parameters. The called function is avcodec_open2, which is declared as:

```
int avcodec_open2(AVCodecContext *avctx, const AVCodec *codec,  
AVDictionary **options);
```

The first two parameters of the function are the two objects we just created, and the third parameter is a dictionary type object, which is used to save the AVCodecContext and other private setting options that the function always fails to recognize. The return value of the function indicates whether the encoder was successfully opened. If the encoder returns 0, it returns a negative number. The calling method is:

```
if (avcodec_open2(ctx.c, ctx.codec, NULL) < 0) //Open encoder  
based on encoder context
```

```
{  
    fprintf(stderr, "Could not open codec\n");  
    exit(1);  
}
```

Then we need to process the AVFrame object. AVFrame represents a container of the original pixel data of the video. Processing this type of data requires two steps. One is to allocate an AVFrame object, and the other is to allocate the storage space of the actual pixel data. Allocating object space is similar to the new operator, except that the function av_frame_alloc needs to be called. If it fails, the function returns a null pointer. After the AVFrame object is successfully allocated, you need to set the resolution and pixel format of the image. The actual calling process is as follows:

```
ctx.frame = av_frame_alloc(); //Assign AVFrame object  
if (!ctx.frame)  
{  
    fprintf(stderr, "Could not allocate video frame\n");  
    return false;  
}  
ctx.frame->format = ctx.c->pix_fmt;  
ctx.frame->width = ctx.c->width;  
ctx.frame->height = ctx.c->height;
```

Allocating pixel storage space requires calling the av_image_alloc function, which is declared as:

```
int av_image_alloc(uint8_t *pointers[4], int linesizes[4], int w, int h, enum AVPixelFormat pix_fmt, int align);
```

The four parameters of the function represent the buffer pointer in the AVFrame structure, the width of each color component, the image resolution (width, height), the pixel format, and the size of the memory. This function returns the size of the allocated memory, or a negative value if it fails. The specific calling method is as follows:

```
ret = av_image_alloc(ctx.frame->data, ctx.frame->linesize, ctx.c->width,  
ctx.c->height, ctx.c->pix_fmt, 32);  
if (ret < 0)  
{ fprintf(stderr, "Could not allocate raw picture buffer\n"); return false; }
```

(3), ENCODING LOOP BODY

At this point, our preparations have been roughly completed. Let's start the loop process of actual encoding. The process of roughly expressing the encoding with pseudo code is:

```
while (numCoded < maxNumToCode)
{
    read_yuv_data();
    encode_video_frame();
    write_out_h264();
}
```

Among them, the `read_yuv_data` part can be directly read using the `fread` statement. All you need to know is that the addresses of the three color components Y / U / V are `AVframe :: data [0]`, `AVframe :: data [1]`, and `AVframe:: : data [2]`, the width of the image is `AVframe :: linesize [0]`, `AVframe :: linesize [1]`, and `AVframe :: linesize [2]`. It should be noted that the value in `linesize` usually refers to stride instead of width, that is to say, the pixel storage area may have an invalid edge area with a certain width, and you need to

pay attention when reading data.

Initialize the AVPacket object when you need to complete additional operations before encoding. This object holds the encoded stream data. The operation of initializing it is very simple, just need to call `av_init_packet` and pass in a pointer to the AVPacket object. Then set `AVPacket :: data` to `NULL` and `AVPacket :: size` to `0`.

After the original YUV pixel values are successfully saved in the AVframe structure, the `avcodec_encode_video2` function can be called to perform the actual encoding operation. This function is the core of the entire project, and its declaration method is:

```
int avcodec_encode_video2(AVCodecContext *avctx, AVPacket *avpkt,
                           const AVFrame *frame, int *got_packet_ptr);
```

The meaning of its parameters and return values:

- `avctx`: `AVCodecContext` structure, which specifies some parameters for encoding;
- `avpkt`: a pointer to an `AVPacket` object, used to save the output code stream;
- `frame`: `AVframe` structure, used to pass in the original pixel data;

- `got_packet_ptr`: output parameter, used to identify whether there is already a complete frame in `AVPacket`;
- Return value: Whether the encoding was successful. 0 on success, negative error code on failure

Through the output parameter `* got_packet_ptr`, we can determine whether there should be a complete frame stream packet output. If it is, then the stream data in `AVpacket` can be output. Its address is `AVPacket :: data` and the size is `AVPacket :: size`. The specific calling method is as follows:

```

/* encode the image */
ret = avcodec_encode_video2(ctx.c, &(ctx(pkt), ctx.frame,
&got_output); //Encode the pixel information in AVFrame into the code
stream in AVPacket
if (ret < 0)
{
    fprintf(stderr, "Error encoding frame\n");
    exit(1);
}

if (got_output)
{
    //Get a complete coded frame
    printf("Write frame %3d (size=%5d)\n", frameIdx, ctx(pkt.size));
    fwrite(ctx(pkt).data, 1, ctx(pkt.size, io_param.pFout);
    av_packet_unref(&(ctx(pkt)));
}

```

Therefore, a complete encoding loop can be implemented using the following code:

```
/* encode 1 second of video */
for (frameIdx = 0; frameIdx < io_param.nTotalFrames; frameIdx++)
{
    av_init_packet(&(ctx(pkt));           //Initialize AVPacket instance
    ctx(pkt).data = NULL;                // packet data will be allocated by the
encoder
    ctx(pkt).size = 0;

    fflush(stdout);
    Read_yuv_data(ctx, io_param, 0);      //Y component
    Read_yuv_data(ctx, io_param, 1);      //U component
    Read_yuv_data(ctx, io_param, 2);      //V component

    ctx.frame->pts = frameIdx;

    /* encode the image */
    ret = avcodec_encode_video2(ctx.c, &(ctx(pkt)), ctx.frame,
&got_output); //Encode the pixel information in AVFrame into the code
stream in AVPacket
    if (ret < 0)
    {
        fprintf(stderr, "Error encoding frame\n");
        exit(1);
    }

    if (got_output)
    {
        //Get a complete coded frame
        printf("Write frame %3d (size=%5d)\n", frameIdx, ctx(pkt).size);
        fwrite(ctx(pkt).data, 1, ctx(pkt).size, io_param.pFout);
        av_packet_unref(&(ctx(pkt)));
    }
} //for (frameIdx = 0; frameIdx < io_param.nTotalFrames; frameIdx++)
```

(4) FINISHING TREATMENT

If we end the entire running process of the encoder, we will find that the code stream after encoding is one frame less than the original data. This is because we judge the end of the loop based on the end of reading the original pixel data, so that the last frame remains in the encoder and has not yet been output. So before closing the entire decoding process, we must continue to perform the encoding operation until the last frame is output. Performing this operation still calls the avcodec_encode_video2 function, but only indicates that the parameters of the AVFrame are set to NULL:

```
/* get the delayed frames */
for (got_output = 1; got_output; frameIdx++)
{
    fflush(stdout);

    ret = avcodec_encode_video2(ctx.c, &(ctx.pkt), NULL,
&got_output);                  //Output the remaining code stream in the
encoder
    if (ret < 0)
    {
```

```

fprintf(stderr, "Error encoding frame\n");
exit(1);
}
if (got_output)
{
printf("Write frame %3d (size=%5d)\n", frameIdx, ctx(pkt.size);
fwrite(ctx(pkt).data, 1, ctx(pkt.size, io_param.pFout);
av_packet_unref(&(ctx(pkt)));
}
} //for (got_output = 1; got_output; frameIdx++)

```

After that, we can close the various components of the encoder as planned, and end the entire encoding process. The release process of the encoder component can be analogized to the establishment process, which needs to close AVCoce, release AVCodecContext, release the image cache in AVFrame and the object itself:

```

avcodec_close(ctx.c);
av_free(ctx.c);
av_freep(&(ctx.frame->data[0]));
av_frame_free(&(ctx.frame));

```

The main process of using FFmpeg for video encoding is as follows:

1. First analyze and process the input parameters, such as the parameters of the encoder, the parameters of the image, and the input and output files;

2. Various component tools for building the entire FFmpeg encoder, in that order:
avcodec_register_all-> avcodec_find_encoder->
avcodec_alloc_context3-> avcodec_open2->
av_frame_alloc-> av_image_alloc;
3. Encoding loop: av_init_packet->
avcodec_encode_video2 (twice)->
av_packet_unref

Close the encoder components: avcodec_close,
av_free, av_freep, av_frame_free

H.264 ENCODING

The main functions libx264 video encoding are shown below.

1. `x264_param_default ()`: Set the default value of the parameter set structure `x264_param_t`.
2. `x264_picture_alloc ()`: Allocate memory for the image structure `x264_picture_t`.
3. `x264_encoder_open ()`: Open the encoder.
4. `x264_encoder_encode ()`: encode a frame of image.
5. `x264_encoder_close ()`: Close the encoder.
6. `x264_picture_clean ()`: releases the resources requested by `x264_picture_alloc ()`.
7. `x264_picture_t`: stores pixel data before compression encoding.
8. `x264_nal_t`: stores the stream data after compression encoding.

In addition, the flowchart also includes a "flush_encoder" module, which uses the same functions as the encoding module. The only difference is that video pixel data is no longer entered. Its role is

to output the remaining stream data in the encoder.

```
#include <stdio.h>
#include <stdlib.h>
#include "stdint.h"
#if defined ( __cplusplus)
extern "C"
{
#include "x264.h"
};

#else
#include "x264.h"
#endif

int main(int argc, char** argv)
{

    int ret;
    int y_size;
    int i,j;

    FILE* fp_src = fopen("../cuc_ieschool_640x360_yuv420p.yuv",
"rb");
    FILE* fp_dst = fopen("cuc_ieschool.h264", "wb");
    //Encode 50 frame
    //if set 0, encode all frame
    int frame_num=50;
    int csp=X264_CSP_I420;
    int width=640,height=360;

    int iNal = 0;
    x264_nal_t* pNals = NULL;
    x264_t* pHandle = NULL;
    x264_picture_t* pPic_in =
(x264_picture_t*)malloc(sizeof(x264_picture_t));
    x264_picture_t* pPic_out =
(x264_picture_t*)malloc(sizeof(x264_picture_t));
    x264_param_t* pParam =
```

```

(x264_param_t*)malloc(sizeof(x264_param_t));

//Check
if(fp_src==NULL||fp_dst==NULL){
    printf("Error open files.\n");
    return -1;
}

x264_param_default(pParam);
pParam->i_width = width;
pParam->i_height = height;
/*
//Param
pParam->i_log_level = X264_LOG_DEBUG;
pParam->i_threads = X264_SYNC_LOOKAHEAD_AUTO;
pParam->i_frame_total = 0;
pParam->i_keyint_max = 10;
pParam->i_bframe = 5;
pParam->b_open_gop = 0;
pParam->i_bframe_pyramid = 0;
pParam->rc.i_qp_constant=0;
pParam->rc.i_qp_max=0;
pParam->rc.i_qp_min=0;
pParam->i_bframe_adaptive = X264_B_ADAPT_TRELLIS;
pParam->i_fps_den = 1;
pParam->i_fps_num = 25;
pParam->i_timebase_den = pParam->i_fps_num;
pParam->i_timebase_num = pParam->i_fps_den;
*/
pParam->i_csp=csp;
x264_param_apply_profile(pParam, x264_profile_names[5]);
pHandle = x264_encoder_open(pParam);

x264_picture_init(pPic_out);
x264_picture_alloc(pPic_in, csp, pParam->i_width, pParam-
>i_height);
//ret = x264_encoder_headers(pHandle, &pNals, &iNal);

```

```

y_size = pParam->i_width * pParam->i_height;
//detect frame number
if(frame_num==0){
    fseek(fp_src,0,SEEK_END);
    switch(csp){
        case
X264_CSP_I444:frame_num=ftell(fp_src)/(y_size*3);break;
        case
X264_CSP_I420:frame_num=ftell(fp_src)/(y_size*3/2);break;
        default:printf("Colorspace Not Support.\n");return -1;
    }
    fseek(fp_src,0,SEEK_SET);
}

//Loop to Encode
for( i=0;i<frame_num;i++){
    switch(csp){
        case X264_CSP_I444:{
            fread(pPic_in-
>img.plane[0],y_size,1,fp_src);      //Y
            fread(pPic_in-
>img.plane[1],y_size,1,fp_src);      //U
            fread(pPic_in-
>img.plane[2],y_size,1,fp_src);      //V
            break;}
        case X264_CSP_I420:{
            fread(pPic_in-
>img.plane[0],y_size,1,fp_src);      //Y
            fread(pPic_in-
>img.plane[1],y_size/4,1,fp_src);    //U
            fread(pPic_in-
>img.plane[2],y_size/4,1,fp_src);    //V
            break;}
        default:{
            printf("Colorspace Not Support.\n");
            return -1;}
    }
}

```

```

    pPic_in->i_pts = i;

    ret = x264_encoder_encode(pHandle, &pNals, &iNal,
pPic_in, pPic_out);
    if (ret< 0){
        printf("Error.\n");
        return -1;
    }

    printf("Succeed encode frame: %5d\n",i);

    for ( j = 0; j < iNal; ++j){
        fwrite(pNals[j].p_payload, 1, pNals[j].i_payload,
fp_dst);
    }
}

i=0;
//flush encoder
while(1{
    ret = x264_encoder_encode(pHandle, &pNals, &iNal,
NULL, pPic_out);
    if(ret==0){
        break;
    }
    printf("Flush 1 frame.\n");
    for (j = 0; j < iNal; ++j){
        fwrite(pNals[j].p_payload, 1, pNals[j].i_payload,
fp_dst);
    }
    i++;
}
x264_picture_clean(pPic_in);
x264_encoder_close(pHandle);
pHandle = NULL;

free(pPic_in);
free(pPic_out);
free(pParam);

```

```
fclose(fp_src);
fclose(fp_dst);

return 0;
}
```

H265 ENCODING

The main functions h265 video encoding are shown below.

1. `av_register_all ()`: Register all FFmpeg codecs.
2. `avformat_alloc_output_context2 ()`: Initializes the AVFormatContext of the output stream.
3. `avio_open ()`: Open the output file.
4. `av_new_stream ()`: Create an AVStream for the output stream.
5. `avcodec_find_encoder ()`: find the encoder.
6. `avcodec_open2 ()`: Open the encoder.
7. `avformat_write_header ()`: write the file header (for some packaging formats without a file header, this function is not needed. For example, MPEG2TS).
8. `avcodec_encode_video2 ()`: encode a frame of video. That is, AVFrame (storing YUV pixel data) is encoded into AVPacket (storing stream data in formats such as H.264).
9. `av_write_frame ()`: Write the encoded video stream to a file.
10. `flush_encoder ()`: This function is called after

the input pixel data is read. Used to output the remaining AVPackets in the encoder.

11. `av_write_trailer ()`: write the end of the file (for some packaging formats without a file header, this function is not needed. For example, MPEG2TS)

```
#include <stdio.h>

extern "C"
{
#include "libavutil\opt.h"
#include "libavcodec\avcodec.h"
#include "libavformat\avformat.h"
#include "libswscale\swscale.h"
};

int flush_encoder(AVFormatContext *fmt_ctx,unsigned int stream_index)
{
    int ret;
    int got_frame;
    AVPacket enc_pkt;
    if (!(fmt_ctx->streams[stream_index]->codec->codec->capabilities &
CODEC_CAP_DELAY))
        return 0;
    while (1) {
        printf("Flushing stream #%"PRIu32" encoder\n", stream_index);
        //ret = encode_write_frame(NULL, stream_index, &got_frame);
        enc_pkt.data = NULL;
        enc_pkt.size = 0;
        av_init_packet(&enc_pkt);
        ret = avcodec_encode_video2 (fmt_ctx->streams[stream_index]-
>codec, &enc_pkt,
```

```
    NULL, &got_frame);
    av_frame_free(NULL);
    if (ret < 0)
        break;
    if (!got_frame){
        ret=0;
        break;
    }
    printf("Succeed to encode 1 frame! 编码成功1帧 ! \n");
/* mux encoded frame */
    ret = av_write_frame(fmt_ctx, &enc_pkt);
    if (ret < 0)
        break;
    }
    return ret;
}

int main(int argc, char* argv[])
{
    AVFormatContext* pFormatCtx;
    AVOutputFormat* fmt;
    AVStream* video_st;
    AVCodecContext* pCodecCtx;
    AVCodec* pCodec;

    uint8_t* picture_buf;
    AVFrame* picture;
    int size;

    //FILE *in_file = fopen("src01_480x272.yuv", "rb");      //Input
YUV data video YUV source file
    FILE *in_file = fopen("ds_480x272.yuv", "rb"); //Input YUV data
video YUV source file
    int in_w=480,in_h=272;//Width Height
//Frames to encode
    int framenum=100;
//const char* out_file = "src01.h264"; //Output file path
//const char* out_file = "src01.ts";
```

```
//const char* out_file = "src01.hevc";
const char* out_file = "ds.hevc";

av_register_all();
//Method1 Use several functions in combination
pFormatCtx = avformat_alloc_context();
//Guess Format
fmt = av_guess_format(NULL, out_file, NULL);
pFormatCtx->oformat = fmt;
//Method 2 Be more automated
//avformat_alloc_output_context2(&pFormatCtx, NULL, NULL,
out_file);
//fmt = pFormatCtx->oformat;

//Output Format
if (avio_open(&pFormatCtx->pb,out_file,
AVIO_FLAG_READ_WRITE) < 0)
{
printf("Failed to open output file!");
return -1;
}

video_st = avformat_new_stream(pFormatCtx, 0);
video_st->time_base.num = 1;
video_st->time_base.den = 25;

if (video_st==NULL)
{
return -1;
}
//Param that must set
pCodecCtx = video_st->codec;
//pCodecCtx->codec_id = AV_CODEC_ID_HEVC;
pCodecCtx->codec_id = fmt->video_codec;
pCodecCtx->codec_type = AVMEDIA_TYPE_VIDEO;
pCodecCtx->pix_fmt = PIX_FMT_YUV420P;
pCodecCtx->width = in_w;
pCodecCtx->height = in_h;
```

```

pCodecCtx->time_base.num = 1;
pCodecCtx->time_base.den = 25;
pCodecCtx->bit_rate = 400000;
pCodecCtx->gop_size=250;
//H264
//pCodecCtx->me_range = 16;
//pCodecCtx->max_qdiff = 4;
//pCodecCtx->qcompress = 0.6;
pCodecCtx->qmin = 10;
pCodecCtx->qmax = 51;
//Optional Param
pCodecCtx->max_b_frames=3;

// Set Option
AVDictionary *param = 0;
//H.264
if(pCodecCtx->codec_id == AV_CODEC_ID_H264) {
av_dict_set(?m, "preset", "slow", 0);
av_dict_set(?m, "tune", "zerolatency", 0);
}
//H.265
if(pCodecCtx->codec_id == AV_CODEC_ID_H265){
av_dict_set(?m, "x265-params", "qp=20", 0);
av_dict_set(?m, "preset", "ultrafast", 0);
av_dict_set(?m, "tune", "zero-latency", 0);
}

//Dump Information
av_dump_format(pFormatCtx, 0, out_file, 1);

pCodec = avcodec_find_encoder(pCodecCtx->codec_id);
if (!pCodec){
printf("Can not find encoder!\n");
return -1;
}
if (avcodec_open2(pCodecCtx, pCodec,?m) < 0){
printf("Failed to open encoder!\n");
return -1;
}

```

```

    picture = avcodec_alloc_frame();
    size = avpicture_get_size(pCodecCtx->pix_fmt, pCodecCtx->width,
pCodecCtx->height);
    picture_buf = (uint8_t *)av_malloc(size);
    avpicture_fill((AVPicture *)picture, picture_buf, pCodecCtx-
>pix_fmt, pCodecCtx->width, pCodecCtx->height);

//Write File Header
avformat_write_header(pFormatCtx,NULL);

AVPacket pkt;
int y_size = pCodecCtx->width * pCodecCtx->height;
av_new_packet(&pkt,y_size*3);

for (int i=0; i<framenum; i++){
//Read YUV
if (fread(picture_buf, 1, y_size*3/2, in_file) < 0){
printf("Failed to read YUV data! \n");
return -1;
}else if(feof(in_file)){
break;
}
picture->data[0] = picture_buf; //Y
picture->data[1] = picture_buf+ y_size; // U
picture->data[2] = picture_buf+ y_size*5/4; // V
//PTS
picture->pts=i;
int got_picture=0;
//Encode
int ret = avcodec_encode_video2(pCodecCtx, &pkt,picture,
&got_picture);
if(ret < 0){
printf("Failed to encode! \n");
return -1;
}
if (got_picture==1){

```

```
printf("Succeed to encode 1 frame! \n");
pkt.stream_index = video_st->index;
ret = av_write_frame(pFormatCtx, &pkt);
av_free_packet(&pkt);
}
}

//Flush Encoder
int ret = flush_encoder(pFormatCtx,0);
if (ret < 0) {
printf("Flushing encoder failed\n");
return -1;
}

//Write file trailer
av_write_trailer(pFormatCtx);

//Clean
if (video_st){
avcodec_close(video_st->codec);
av_free(picture);
av_free(picture_buf);
}
avio_close(pFormatCtx->pb);
avformat_free_context(pFormatCtx);

fclose(in_file);

return 0;
}
```

VP8 ENCODING

libvpx in this project can also encode VP9 format video. However, there is a problem with the packaging format that has not yet been resolved, so it does not include the code for encoding VP9 for the time being. The function calls for encoding VP9 and VP8 are identical.

The main functions in vp8 video encoding are shown below.

1. `vpx_img_alloc ()`: Allocate memory for the image structure `vpx_image_t`.
2. `vpx_codec_enc_config_default ()`: Set the default value of the parameter set structure `vpx_codec_enc_cfg_t`.
3. `vpx_codec_enc_init ()`: Open the encoder.
4. `vpx_codec_encode ()`: encode a frame of image.
5. `vpx_codec_get_cx_data ()`: Get one frame of compression-encoded data.
6. `vpx_codec_destroy ()`: Close the encoder.
7. `vpx_image_t`: stores pixel data before

- compression encoding.
8. `vpx_codec_cx_pkt_t`: Stores code stream data after compression encoding.
 9. The functions handled by the IVF package format are shown below.
 10. `write_ivf_file_header()`: write a file header in the IVF package format.
 11. `write_ivf_frame_header()`: write the frame header of each frame of data in the IVF package format.

In addition, the flowchart also includes a "flush_encoder" module, which uses the same functions as the encoding module. The only difference is that video pixel data is no longer entered. Its role is to output the remaining stream data in the encoder.

```
#include <stdio.h>
#include <stdlib.h>

#define VPX_CODEC_DISABLE_COMPAT 1

#include "vpx/vpx_encoder.h"
#include "vpx/vp8cx.h"

#define interface (&vpx_codec_vp8_cx_algo)

#define fourcc 0x30385056

#define IVF_FILE_HDR_SZ (32)
#define IVF_FRAME_HDR_SZ (12)

static void mem_put_le16(char *mem, unsigned int val) {
```

```

    mem[0] = val;
    mem[1] = val>>8;
}

static void mem_put_le32(char *mem, unsigned int val) {
    mem[0] = val;
    mem[1] = val>>8;
    mem[2] = val>>16;
    mem[3] = val>>24;
}

static void write_ivf_file_header(FILE *outfile,
                                  const vpx_codec_enc_cfg_t *cfg,
                                  int frame_cnt) {
    char header[32];

    if(cfg->g_pass != VPX_RC_ONE_PASS && cfg->g_pass != VPX_RC_LAST_PASS)
        return;
    header[0] = 'D';
    header[1] = 'K';
    header[2] = 'T';
    header[3] = 'F';
    mem_put_le16(header+4, 0);           /* version */
    mem_put_le16(header+6, 32);          /* headersize */
    mem_put_le32(header+8, fourcc);      /* headersize */
    mem_put_le16(header+12, cfg->g_w);   /* width */
    mem_put_le16(header+14, cfg->g_h);   /* height */
    mem_put_le32(header+16, cfg->g_timebase.den); /* rate */
    mem_put_le32(header+20, cfg->g_timebase.num); /* scale */
    mem_put_le32(header+24, frame_cnt);    /* length */
    mem_put_le32(header+28, 0);           /* unused */

    fwrite(header, 1, 32, outfile);
}

static void write_ivf_frame_header(FILE *outfile,

```

```

    const vpx_codec_cx_pkt_t *pkt)
{
    char header[12];
    vpx_codec_pts_t pts;

    if(pkt->kind != VPX_CODEC_CX_FRAME_PKT)
        return;

    pts = pkt->data.frame.pts;
    mem_put_le32(header, pkt->data.frame.sz);
    mem_put_le32(header+4, pts&0xFFFFFFFF);
    mem_put_le32(header+8, pts >> 32);

    fwrite(header, 1, 12, outfile);
}

int main(int argc, char **argv) {

    FILE *infile, *outfile;
    vpx_codec_ctx_t codec;
    vpx_codec_enc_cfg_t cfg;
    int frame_cnt = 0;
    unsigned char file_hdr[IVF_FILE_HDR_SZ];
    unsigned char frame_hdr[IVF_FRAME_HDR_SZ];
    vpx_image_t raw;
    vpx_codec_err_t ret;
    int width,height;
        int y_size;
    int frame_avail;
    int got_data;
    int flags = 0;

    width = 640;
    height = 360;

    /* Open input file for this encoding pass */
    infile = fopen("../cuc_ieschool_640x360_yuv420p.yuv", "rb");
    outfile = fopen("cuc_ieschool.ivf", "wb");

    if(infile==NULL||outfile==NULL){

```

```

        printf("Error open files.\n");
        return -1;
    }

    if(!vpx_img_alloc(&raw, VPX_IMG_FMT_I420, width, height, 1)){
        printf("Fail to allocate image\n");
        return -1;
    }

printf("Using %s\n",vpx_codec_iface_name(interface));

/* Populate encoder configuration */
ret = vpx_codec_enc_config_default(interface, &cfg, 0);
if(ret) {
    printf("Failed to get config: %s\n", vpx_codec_err_to_string(ret));
    return -1;
}

/* Update the default configuration with our settings */
cfg.rc_target_bitrate =800;
cfg.g_w = width;
cfg.g_h = height;

write_ivf_file_header(outfile, &cfg, 0);

/* Initialize codec */
if(vpx_codec_enc_init(&codec, interface, &cfg, 0)){
    printf("Failed to initialize encoder\n");
    return -1;
}

frame_avail = 1;
got_data = 0;

y_size=cfg.g_w*cfg.g_h;

while(frame_avail || got_data) {
    vpx_codec_iter_t iter = NULL;
    const vpx_codec_cx_pkt_t *pkt;

```

```

    if(fread(raw.planes[0], 1, y_size*3/2, infile)!=y_size*3/2){
        frame_avail=0;
    }

    if(frame_avail){
        ret=vpx_codec_encode(&codec,&raw,frame_cnt,1,
    }else{
        ret=vpx_codec_encode(&codec,NULL,frame_cnt,1
    }

    if(ret){
        printf("Failed to encode frame\n");
        return -1;
    }
got_data = 0;
while( (pkt = vpx_codec_get_cx_data(&codec, &iter)) ) {
    got_data = 1;
    switch(pkt->kind) {
    case VPX_CODEC_CX_FRAME_PKT:
        write_ivf_frame_header(outfile, pkt);
        fwrite(pkt->data.frame.buf, 1, pkt->data.frame.sz,outfile);
        break;
    default:
        break;
    }
}

printf("Succeed encode frame: %5d\n",frame_cnt);
frame_cnt++;
}

fclose(infile);
vpx_codec_destroy(&codec);

/* Try to rewrite the file header with the actual frame count */
if(!fseek(outfile, 0, SEEK_SET))
    write_ivf_file_header(outfile, &cfg, frame_cnt-1);

fclose(outfile);
return 0;

```

}

AUDIO ENCODING

AAC ENCODING

pcm audio is uncompressed data and occupies space. Generally, lossy compression (such as aac, MP3, etc.) is usually selected when storing or transmitting. pcm audio data is generally stored in planner format when the file is stored, for example:
channel 1 channel 2 channel 3. .channel 1 channel 2
channel 3 ...

1. One encoding format can have multiple encapsulation formats, and one encapsulation format can also be used for multiple encoding formats. To query all the encapsulation formats supported by an encoding format, open the ffmpeg source code to query in the following way
2. When encapsulating with AVFormatContext, ffmpeg must compile the corresponding encapsulation format options. The corresponding encapsulation format option name is .name = corresponding value. That is, add --enable_muxer = this value when compiling

`AVCodec * avcodec_find_encoder (enum AVCodecID`

`id);`

find the corresponding encoder by the specified ID

```
AVCodec * avcodec_find_encoder_by_name (char *  
name);
```

find the corresponding encoder by the specified name

1. The same id may appear in the source code of the ffmpeg library for the corresponding encoding library. For example, `AV_CODEC_ID_AAC` corresponds to the two libraries `libshine` and `libmp3lame`. When looking for the encoder by ID, it means that the encoder before index in `codec_list []` The library will be found first (note here). It is best to search by name.
2. If `AVCodec`'s `wrapper_name` has a value, then the codec is generally an external library or a system library. If it is `NULL`, it means that `ffmpeg`'s own code includes the library.
3. `AVCodecContext * avcodec_alloc_context3
(AVCodec * codec);`
create an encoder context.
4. `int avcodec_open2 (AVCodecContext,
AVCodec codec, AVDictionary ** options);`
initialize the encoder context
5. `int avformat_alloc_output_context`

- (AVFormatContext * ctx, AVOutputFormat * oformat, const char * format_name, const char * file_name);
initialize wrapper context
6. AVStream * avformat_new_stream
(AVFormatContext * s, const AVCodec * c);
add a stream of information to the wrapper context
 7. int avio_open (AVIOContext ** s, const char * url, int flags);
open the context for preparing to write data
 8. int avcodec_parameters_from_context
(AVCodecParameters * par,
const AVCodecContext * codec);
copy the encoder information to the corresponding AVStream stream
 9. int avcodec_send_frame (AVCodecContext * avctx, const AVFrame * frame);
encoding
 10. .int avcodec_receive_packet
(AVCodecContext * avctx, AVPacket * apk);
Get the encoded data

Here take pcm encoding as aac format as an example, familiar with ffmpeg related code, encoding

into other formats such as MP3 / AC3 and so on are the same, the only thing to note is that ffmpeg needs to enable these encoding libraries when compiling

aac has two packaging formats, ADIF and ADTS, the more commonly used is ADTS. FFmpeg's aac-encoded data is in the adts format. This data can be directly played into a file.

```
void doEncode(CodecFormat format=CodecFormatAAC, bool saveByFile = false);
```

The specific implementation code is as follows:

```
/**  
 * Determine whether the sampling format is supported by the specified  
 encoder, and return the sampling format if supported;  
 otherwise, return the sampling format with the largest enumeration value  
 supported by the encoder  
 */  
static enum AVSampleFormat select_sample_fmt(const AVCodec  
*codec, enum AVSampleFormat sample_fmt)  
{  
    const enum AVSampleFormat *p = codec->sample_fmts;  
    enum AVSampleFormat rfmt = AV_SAMPLE_FMT_NONE;  
    while (*p != AV_SAMPLE_FMT_NONE) {  
        if (*p == sample_fmt) {  
            return sample_fmt;  
        }  
        if (rfmt == AV_SAMPLE_FMT_NONE) {  
            rfmt = *p;  
        }  
        p++;
```

```
    }

    return rfmt;
}

/***
*
>Returns the sampling rate of the specified encoder as close to 44100 as
possible
*/
static int select_sample_rate(const AVCodec *codec,int
defalt_sample_rate)
{
    const int *p = 0;
    int best_samplerate = 0;
    if (!codec->supported_samplerates) {
        return 44100;
    }

    p = codec->supported_samplerates;
    while (*p) {
        if (*p == defalt_sample_rate) {
            return *p;
        }

        if (!best_samplerate || abs(44100 - *p) < abs(44100 -
best_samplerate)) {
            best_samplerate = *p;
        }

        p++;
    }

    return best_samplerate;
}

/***
>Returns the channel format with the largest number of channels among the
channel formats supported by the encoder

```

```
*  
Channel format and channel number one-to-one correspondence  
*/  
static uint64_t select_channel_layout(const AVCodec *codec,uint64_t  
default_layout)  
{  
    uint64_t best_ch_layout = AV_CH_LAYOUT_STEREO;  
    const uint64_t *p = codec->channel_layouts;  
    if (p == NULL) {  
        return AV_CH_LAYOUT_STEREO;  
    }  
    int best_ch_layouts = 0;  
    while (*p) {  
        int layouts = av_get_channel_layout_nb_channels(*p);  
        if (*p == default_layout) {  
            return *p;  
        }  
        if (layouts > best_ch_layouts) {  
            best_ch_layout = *p;  
            best_ch_layouts = layouts;  
        }  
        p++;  
    }  
    return best_ch_layout;  
}  
  
AudioEncode::AudioEncode()  
{  
    pFormatCtx = NULL;  
    pCodecCtx = NULL;  
    pSwrCtx = NULL;  
}  
  
AudioEncode::~AudioEncode()  
{
```

```

}

void AudioEncode::privateLeaseResources()
{
    if (pFormatCtx) {
        avformat_close_input(&pFormatCtx);
        pFormatCtx = NULL;
    }
    if (pCodecCtx) {
        avcodec_free_context(&pCodecCtx);
        pCodecCtx = NULL;
    }
    if (pSwrCtx) {
        swr_free(&pSwrCtx);
        pSwrCtx = NULL;
    }
}

void
AudioEncode::doEncode1(AVFormatContext*fmtCtx,AVCodecContext
*cCtx,AVPacket *packet,AVFrame *srcFrame,FILE *file)
{
    if (fmtCtx == NULL || cCtx == NULL || packet == NULL) {
        return;
    }

    int ret = 0;
    // Start encoding; for audio encoding, there is no need to set the value of
    pts (but a warning will appear);
    //if frame is NULL, it means that all the remaining data in the
    encoding buffer has been encoded
    ret = avcodec_send_frame(cCtx, srcFrame);
    while (ret >= 0) {
        ret = avcodec_receive_packet(cCtx, packet);
        if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF) { // EAGAIN may be that the encoder needs to buffer part of the data first,
        which is not a real encoding error
            LOGD("encode error %d",ret);
        }
    }
}

```

```

        return;
    } else if (ret < 0) { // Produced a real coding error
        return;
    }
    LOGD("packet size %d dts:%d pts:%d duration:%d",packet-
>size,packet->dts,packet->pts,packet->duration);
    av_write_frame(fmtCtx, packet);
    if (file) {
        fwrite(packet->data, packet->size, 1, file);
    }
    // Every time encoding avcodec_receive_packet will reallocate
    memory for the packet, so it must be actively released after it is used here
    av_packet_unref(packet);
}
/***
 * 1. A coding format can have multiple packaging formats, and a
 packaging format can also have multiple coding formats. To query all
 packaging formats supported by an encoding format, open the ffmpeg
 source code and query in the following way:
 * Taking AAC as an example, enter .audio_codec =
 AV_CODEC_ID_AAC to query all the packaging formats that support
 AAC. The corresponding file extension is .extensions = corresponding
 value
 *
 2. When encapsulating with AVFormatContext, ffmpeg must compile the
 corresponding encapsulation format option. The corresponding
 encapsulation format option name is .name = corresponding value. That is,
 add --enable_muxer = this value at compile time
*/
static string muxer_file_name(CodecFormat format)
{
    string name = "test.aac";
    if (format == CodecFormatMP3) {
        name = "test.mp3";
    } else if (format == CodecFormatAC3) {

```

```
    /** Unable to find a suitable output format for 'test.rm'
     * Reason analysis: Since ac3 muxer is not compiled in, you can
     see #define CONFIG_RM_MUXER 0 from the config.h file generated by
     compilation
     * Solution: Recompile
     */
     name = "test.ac3";
}

return name;
}

/**
 * 1. Open the source code of ffmpeg and pass .id =
AV_CODEC_ID_AAC
* (Take the AAC encoder as an example here) Query the encoder
information
* 2. The registration method of the new version of the encoder (the old
version is added through av_register_xxx and the macro definition). The
new version is automatically generated according to the configuration after
the .configure and a file codec_list.c under the libavcodec folder, this
* A static array codec_list [] defines all codecs that have been compiled
into the file; refer to .configure configuration code
print_enabled_components libavcodec / codec_list.c AVCodec codec_list $_
CODEC_LIST
* 3. The same id may appear in the source code of the ffmpeg library,
corresponding to two encoding libraries, such as AV_CODEC_ID_AAC
corresponding to the two libraries libshine and libmp3lame, then when
looking up the encoder by ID, it means that the index in codec_list [] is
then
* The former encoder library will be found first (note here), it is best to
search by name
* 4. If wrapper_name has a value, it means that the codec is generally an
external library or a system library. NULL means that ffmpeg's own code
contains
*/
static AVCodec* select_codec(CodecFormat format)
{
```

```

AVCodec *codec = NULL;
if (format == CodecFormatAAC) {
    /* You can find the encoder by the following two methods, the
library corresponding to the aac encoder is fdk_aac
*/
//    codec = avcodec_find_encoder(AV_CODEC_ID_AAC);
//    codec = avcodec_find_encoder_by_name("libfdk_aac");
} else if (format == CodecFormatMP3) {
//    codec = avcodec_find_encoder(AV_CODEC_ID_MP3);
//    codec = avcodec_find_encoder_by_name("libmp3lame");
} else if (format == CodecFormatAC3) {
    codec = avcodec_find_encoder(AV_CODEC_ID_AC3);
}

return codec;
}

static int select_bit_rate(CodecFormat format)
{
    // For different encoders, the optimal code rate is different, unit bit / s;
for mp3, 192kbps can get better sound quality.
    int bit_rate = 64000;
    if (format == CodecFormatMP3) {
        bit_rate = 192000;
    } else if (format == CodecFormatAC3) {
        bit_rate = 192000;
    }

    return bit_rate;
}

void AudioEncode::doEncode(CodecFormat format, bool saveByFile)
{
    string curFile(__FILE__);
    unsigned long pos = curFile.find("1-audio_encode_decode");
    if (pos == string::npos) {
        return;
    }
}

```

```

string resourceDir = curFile.substr(0,pos)+"filesources/";

string pcmpath = resourceDir+"test_441_f32le_2.pcm";
string dstpath = muxer_file_name(format);
string dstpath1 = "save_"+dstpath;
uint64_t src_ch_layout=AV_CH_LAYOUT_STEREO,dst_ch_layout =
AV_CH_LAYOUT_STEREO;
int src_sample_rate = 44100,dst_sample_rate=44100;
int src_nb_samples = 1024;
int src_nb_channels =
av_get_channel_layout_nb_channels(src_ch_layout);
enum AVSampleFormat src_sample_fmt =
AV_SAMPLE_FMT_FLT;// The file is stored as a 16-bit integer type
enum AVSampleFormat dst_sample_fmt = AV_SAMPLE_FMT_FLT;
LOGD("srcpath %s dstpath %s \n",pcmpath.c_str(),dstpath.c_str());

int ret = 0;
// 1. Find the encoder
AVCodec *pCodec = select_codec(CodecFormatAAC);
if (pCodec == NULL) {
    LOGD("pCodec is null \n");
    return;
}

// 2. Create an encoder context
pCodecCtx = avcodec_alloc_context3(pCodec);
if (pCodecCtx == NULL) {
    LOGD("pCodecCtx is null \n");
    privateLeaseResources();
    return;
}

// 3. Set the encoding parameters
// The optimal code rate is different for different encoders, the unit is bit
/s;
pCodecCtx->bit_rate = select_bit_rate(format);
// Sampling rate; each encoder may support multiple sampling rates, a
sampling rate is specified by default (encoders may not support the default)

```

```

pCodecCtx->sample_rate =
select_sample_rate(pCodec,dst_sample_rate);
    // Sampling format; the sampling format supported by each encoder is
different, and a sampling format is specified by default (the encoder does
not necessarily support the default)
pCodecCtx->sample_fmt = select_sample_fmt(pCodec,
dst_sample_fmt);
    // Channel format; the channel format supported by each encoder is
different, and a channel format is specified by default (the encoder does not
necessarily support the default)
pCodecCtx->channel_layout =
select_channel_layout(pCodec,dst_ch_layout);
    // The number of channels; the number of channels corresponds to the
format of the channels. I do n't know why they need to be reset here?
pCodecCtx->channels =
av_get_channel_layout_nb_channels(pCodecCtx->channel_layout);
    // Set the time base. For audio encoding, this option is not necessary
// pCodecCtx->time_base = AVRational{1,1024};

/** 4. Open the encoder context
 * Will do some initialization operations on the encoder
 */
ret = avcodec_open2(pCodecCtx, pCodec, NULL);
if (ret < 0) {
    LOGD("avcodec_open2 fail");
    privateLeaseResources();
    return;
}
LOGD("rate %d select fmt %s chl %d frame_size %d",pCodecCtx-
>sample_rate,av_get_sample_fmt_name(pCodecCtx-
>sample_fmt),pCodecCtx->channels,pCodecCtx->frame_size);

FILE *file = fopen(pcmpath.c_str(), "rb");
FILE *file1 = NULL;
if (saveByFile) {
    file1 = fopen(dstpath1.c_str(),"wb+");
}
if (file == NULL) {

```

```

LOGD("fopen fail");
privateLeaseResources();
return;
}

/** Ways of storing data in audio in AVFrame:
 * planner method: the data of each channel is stored in data [0], data
[1] ...
 * packet mode: sequential storage mode, the sampled data of each
channel is stored in data [0], and stored sequentially in the memory, taking
two channels as an example, such as LRLRLRLR .....,,
 * L represents left channel data, R represents right channel data
 * The specific storage method is determined by the format attribute of
AVFrame, refer to the AVSampleFormat enumeration, the planner method
with P, otherwise the packet method
 *
 * How to store audio data in PCM files:
 * Take two-channel as an example, it is generally stored in the way of
LRLR ....., that is, the packet way. Therefore, when reading data from the
pcm file into the AVFrame, pay attention to whether the storage method
corresponds.
 * A summary of the memory size allocated for byte alignment: similar
to the last parameter of the av_samples_alloc_array_and_samples ()
function and av_frame_get_buffer () function
 * 1. The total size of the allocated memory is line_size * number of
channels, where line_size> = nb_samples * bytes per sample (audio);
 * 2. If the align parameter is specified, the size of line_size is an
integer multiple of the align parameter, if it is 1, then line_size =
nb_samples * bytes per sample
 * 3. When align is 0, it is automatically allocated according to the
current CPU architecture bit number, but ultimately it is not necessarily
allocated according to the parameters of 32 or 64
*/
// 5. Allocate memory blocks for storing uncompressed audio data
bool needConvert = false;
AVPacket *packet = av_packet_alloc();
AVFrame *srcFrame = av_frame_alloc();

```

```
AVFrame *dstFrame = av_frame_alloc();
// Number of audio samples per channel
srcFrame->nb_samples = src_nb_samples;
dstFrame->nb_samples = pCodecCtx->frame_size;
// Sampling format
srcFrame->format = src_sample_fmt;
dstFrame->format = pCodecCtx->sample_fmt;
// Channel format
srcFrame->channel_layout = src_ch_layout;
dstFrame->channel_layout = pCodecCtx->channel_layout;
// Sampling Rate
srcFrame->sample_rate = src_sample_rate;
dstFrame->sample_rate = pCodecCtx->sample_rate;
// Through the above three parameters, you can determine the size of an
AVFrame, that is, the size of the audio data;
// Then use this method to allocate the corresponding memory block;
the second parameter represents the number of alignment bits automatically
selected according to the architecture of the CPU, preferably filled in as 0
ret = av_frame_get_buffer(srcFrame, 0);
if (ret < 0) {
    LOGD("av_frame_get_buffer fail %d",ret);
    privateLeaseResources();
    return;
}
// Make the memory block writable, it is best to call this method
ret = av_frame_make_writable(srcFrame);
if (ret < 0) {
    LOGD("av_frame_make_writable fail %d",ret);
    privateLeaseResources();
    return;
}
// Whether format conversion is required
if (pCodecCtx->sample_fmt != srcFrame->format) {
    needConvert = true;
}
if (pCodecCtx->channel_layout != srcFrame->channel_layout) {
    needConvert = true;
```

```

    }

    if (pCodecCtx->sample_rate != srcFrame->sample_rate) {
        needConvert = true;
    }

    /** Format conversion
     * Encountered a problem: crash
     * Reason analysis: the audio data format in AVFrame and the audio
     * data format (such as sampling format) required by AVCodecContext
     * encoding are inconsistent
     * Solution: Format conversion before encoding
     */
}

if (needConvert) {
    if (pSwrCtx == NULL) {
        pSwrCtx = swr_alloc_set_opts(NULL, pCodecCtx-
>channel_layout, pCodecCtx->sample_fmt, pCodecCtx->sample_rate,
                                srcFrame->channel_layout, (enum
AVSampleFormat)srcFrame->format, srcFrame->sample_rate, 0, NULL);
//        pSwrCtx = swr_alloc();
        if (pSwrCtx == NULL) {
            LOGD("swr_alloc_set_opts() fail");
            return;
        }
        av_opt_set_int(pSwrCtx, "in_channel_layout", srcFrame-
>channel_layout, AV_OPT_SEARCH_CHILDREN);
//        av_opt_set_sample_fmt(pSwrCtx, "in_sample_fmt", (enum
AVSampleFormat)srcFrame->format, AV_OPT_SEARCH_CHILDREN);
//        av_opt_set_int(pSwrCtx, "in_sample_rate", srcFrame-
>sample_rate, AV_OPT_SEARCH_CHILDREN);
//        av_opt_set_int(pSwrCtx, "out_channel_layout", pCodecCtx-
>channel_layout, AV_OPT_SEARCH_CHILDREN);
//        av_opt_set_sample_fmt(pSwrCtx, "out_sample_fmt", pCodecCtx-
>sample_fmt, AV_OPT_SEARCH_CHILDREN);
//        av_opt_set_int(pSwrCtx, "out_sample_rate", pCodecCtx-
>sample_rate, AV_OPT_SEARCH_CHILDREN);
//
//        ret = swr_init(pSwrCtx);
//        if (ret < 0) {

```

```

//          LOGD("swr_init fail");
//          return;
//      }
}

ret = av_frame_get_buffer(dstFrame,0);
if (ret < 0) {
    LOGD("av_frame_get_buffer fail %d",ret);
    return;
}
ret = av_frame_make_writable(dstFrame);
if (ret < 0) {
    LOGD("av_frame_make_writable %d",ret);
    return;
}
}

```

/** 6. Create a packaged Muxer context environment in aac format
AVFormatContext context environment. For each wrapper Muxer, it must
include:

```

* Audio stream parameter information, including the package
parameter information expressed by AVStream
*/
// Guess the AVFormatContext context based on the output file suffix.
ret = avformat_alloc_output_context2(&pFormatCtx, NULL, NULL,
dstpath.c_str());
if (ret < 0) {
    LOGD("avformat_alloc_output_context2 fail %d",ret);
    privateLeaseResources();
    return;
}
// Add audio stream parameter information
AVStream *ostream = avformat_new_stream(pFormatCtx, NULL);
if (!ostream) {
    LOGD("avformat_new_stream fail");
    privateLeaseResources();
    return;
}

```

```

//Open the context for preparing to write data
ret = avio_open(&pFormatCtx->pb, dstpath.c_str(),
AVIO_FLAG_READ_WRITE);
if (ret < 0) {
    LOGD("avio_open fail %d",ret);
    privateLeaseResources();
    return;
}
/** Write header information Important: Copy the encoder information
to the corresponding AVStream stream of AVFormatContext
 * Encountered a problem: returned error -22, suggesting Only AAC
streams can be muxed by the ADTS muxer
 * Solution: Copy the encoder information to the AVStream codecpar
parameter in the output stream through the
avcodec_parameters_from_context method
*/
avcodec_parameters_from_context(ostream->codecpar, pCodecCtx);
ret = avformat_write_header(pFormatCtx, NULL);
if (ret < 0) {
    LOGD("avformat_write_header %d",ret);
    privateLeaseResources();
    return;
}
av_dump_format(pFormatCtx, 0, NULL, 1);

/** Encountered a problem: Encoder did not produce proper pts,
making some up.
 * Reason for analysis: The pts parameter of AVFrame has no set
value, just set it.
*/
size_t readsize = 0;
int ptsIndex = 0;
int require_size = av_samples_get_buffer_size(NULL,
src_nb_channels, src_nb_samples, src_sample_fmt, 0);
while ((readsize = fread(srcFrame->data[0], 1, require_size, file)) > 0) {
    if (needConvert) {
        ret = swr_convert_frame(pSwrCtx,dstFrame,srcFrame);

```

```

if (ret < 0) {
    LOGD("swr_convert_frame fail %d",ret);
    continue;
}
dstFrame->pts = ptsIndex;
doEncode1(pFormatCtx, pCodecCtx, packet, dstFrame,file1);
} else {
    srcFrame->pts = ptsIndex;
    doEncode1(pFormatCtx,pCodecCtx,packet,srcFrame,file1);
}
ptsIndex++;

// // First convert the packet format to planner format, each sample is 4
// bytes, so here can be converted to 32 unsigned integers so that you can read
// 4 bytes at a time, the code is more concise
// /** Encountered a problem: the playback sound has deteriorated
// * Solution: It is because fdst [j] [i] = fsr [0] [i * src_nb_channels +
// j]; the previous code is fdst [j] [i] = fsr [0] [i * + j]; causing data to be
// messy Just correct
// */
// uint32_t **fdst = (uint32_t**)aframe->data;
// uint32_t **fsr = (uint32_t**)src_data;
// for (int i=0; i<src_nb_samples ; i++) {
//     for (int j=0; j<src_nb_channels; j++) {
//         if (av_sample_fmt_is_planar((AVSampleFormat)aframe-
// >format)) {
//             fdst[j][i] = fsr[0][i*src_nb_channels+j];
//             LOGD("add[%d] %p",j,fdst[j][i]);
//             printUInt32toHex(fdst[j][i]);
//         } else {
//             fdst[0][i*src_nb_channels+j] = fsr[0][i*src_nb_channels+j];
//         }
//     }
// }
// }

// If there is format conversion, all the contents of the buffer will be
// taken out

```

```

if (needConvert) {
    while (swr_convert_frame(pSwrCtx,dstFrame,NULL) == 0) {
        if (dstFrame->nb_samples > 0) {
            LOGD("清除剩余的 %d",dstFrame->nb_samples);
            dstFrame->pts = ptsIndex;
            doEncode1(pFormatCtx, pCodecCtx, packet, dstFrame,file1);
        } else {
            break;
        }
    }
}

// Incoming NULL will encode all remaining unencoded data
doEncode1(pFormatCtx,pCodecCtx,packet,NULL,file1);

// The ending information must be written, otherwise the file cannot be
played
av_write_trailer(pFormatCtx);
fclose(file);
if (file1) {
    fclose(file1);
}
av_packet_free(&packet);
av_frame_unref(srcFrame);
av_frame_unref(dstFrame);

privateLeaseResources();
}

```

Detailed code:

1. For encoded audio, you must set encoding related parameters before the avcodec_open () function. Here is to set the bit rate, sampling format, sampling rate, channel type, channel number, etc. Each encoder supports different sampling formats, sampling rates,

channel types, optimal bit rates, etc.

2. When the audio format supported by the encoder is different from the audio format to be encoded, format conversion (also (It is generalized resampling) .For details, refer to the following code.

```
if (needConvert) {  
    if (pSwrCtx == NULL) {  
        pSwrCtx = swr_alloc_set_opts(NULL, pCodecCtx->channel_layout, pCodecCtx->sample_fmt, pCodecCtx->sample_rate,  
                                    srcFrame->channe  
.....
```

There will be a part of data in the internal buffer during audio encoding. To get all of this data, only the avcodec_send_frame () function is required to pass NULL.

PICTURE ENCODING

The encoder in this sample realizes the data encoding of YUV420P as JPEG picture. In accordance with the simple principle, the code is basically simplified to the limit.

The program is very simple, you can directly encode the YUV data to JPEG after running the project. This program is very flexible and can be modified into encoders that encode various image formats, such as PNG, GIF, etc. as needed.

```
/**  
*  
* Simplest FFmpeg Picture Encoder  
*  
*/  
  
#include <stdio.h>  
  
#define __STDC_CONSTANT_MACROS  
  
#ifdef _WIN32  
//Windows  
extern "C"  
{  
#include "libavcodec/avcodec.h"  
#include "libavformat/avformat.h"  
};
```

```

#else
//Linux...
#ifndef __cplusplus
extern "C"
{
#endif
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#ifndef __cplusplus
};
#endif
#endif

int main(int argc, char* argv[])
{
    AVFormatContext* pFormatCtx;
    AVOutputFormat* fmt;
    AVStream* video_st;
    AVCodecContext* pCodecCtx;
    AVCodec* pCodec;

    uint8_t* picture_buf;
    AVFrame* picture;
    AVPacket pkt;
    int y_size;
    int got_picture=0;
    int size;

    int ret=0;

    FILE *in_file = NULL;                      //YUV source
    int in_w=480,in_h=272;                     //YUV's width and height
    const char* out_file = "cuc_view_encode.jpg"; //Output file

    in_file = fopen("cuc_view_480x272.yuv", "rb");
    av_register_all();

    //Method 1

```

```
pFormatCtx = avformat_alloc_context();
//Guess format
fmt = av_guess_format("mjpeg", NULL, NULL);
pFormatCtx->oformat = fmt;
//Output URL
if (avio_open(&pFormatCtx->pb, out_file,
AVIO_FLAG_READ_WRITE) < 0){
    printf("Couldn't open output file.");
    return -1;
}

//Method 2. More simple
//avformat_alloc_output_context2(&pFormatCtx, NULL, NULL,
out_file);
//fmt = pFormatCtx->oformat;

video_st = avformat_new_stream(pFormatCtx, 0);
if (video_st==NULL){
return -1;
}
pCodecCtx = video_st->codec;
pCodecCtx->codec_id = fmt->video_codec;
pCodecCtx->codec_type = AVMEDIA_TYPE_VIDEO;
pCodecCtx->pix_fmt = AV_PIX_FMT_YUVJ420P;

pCodecCtx->width = in_w;
pCodecCtx->height = in_h;

pCodecCtx->time_base.num = 1;
pCodecCtx->time_base.den = 25;
//Output some information
av_dump_format(pFormatCtx, 0, out_file, 1);

pCodec = avcodec_find_encoder(pCodecCtx->codec_id);
if (!pCodec){
printf("Codec not found.");
return -1;
}
if (avcodec_open2(pCodecCtx, pCodec, NULL) < 0){
```

```

printf("Could not open codec.");
return -1;
}
picture = av_frame_alloc();
size = avpicture_get_size(pCodecCtx->pix_fmt, pCodecCtx->width,
pCodecCtx->height);
picture_buf = (uint8_t *)av_malloc(size);
if (!picture_buf)
{
return -1;
}
avpicture_fill((AVPicture *)picture, picture_buf, pCodecCtx-
>pix_fmt, pCodecCtx->width, pCodecCtx->height);

//Write Header
avformat_write_header(pFormatCtx,NULL);

y_size = pCodecCtx->width * pCodecCtx->height;
av_new_packet(&pkt,y_size*3);
//Read YUV
if (fread(picture_buf, 1, y_size*3/2, in_file) <=0)
{
printf("Could not read input file.");
return -1;
}
picture->data[0] = picture_buf;           // Y
picture->data[1] = picture_buf+ y_size;   // U
picture->data[2] = picture_buf+ y_size*5/4; // V

//Encode
ret = avcodec_encode_video2(pCodecCtx, &pkt,picture,
&got_picture);
if(ret < 0){
printf("Encode Error.\n");
return -1;
}
if (got_picture==1){
pkt.stream_index = video_st->index;
}

```

```
ret = av_write_frame(pFormatCtx, &pkt);
}

av_free_packet(&pkt);
//Write Trailer
av_write_trailer(pFormatCtx);

printf("Encode Successful.\n");

if (video_st){
    avcodec_close(video_st->codec);
    av_free(picture);
    av_free(picture_buf);
}
avio_close(pFormatCtx->pb);
avformat_free_context(pFormatCtx);

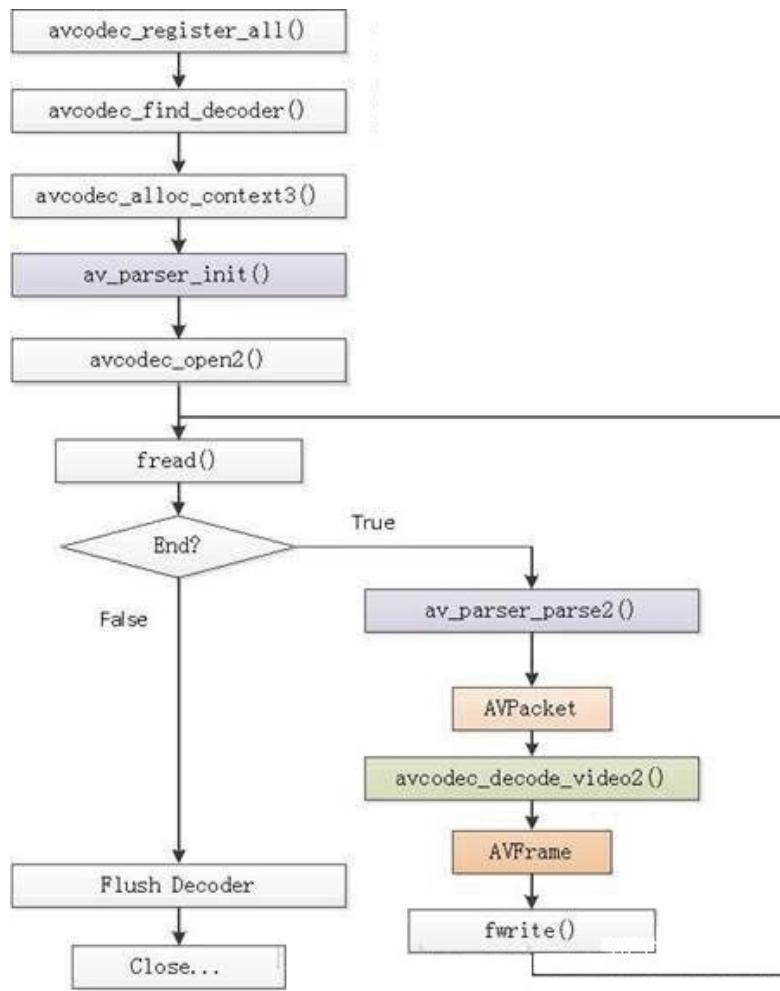
fclose(in_file);

return 0;
}
```

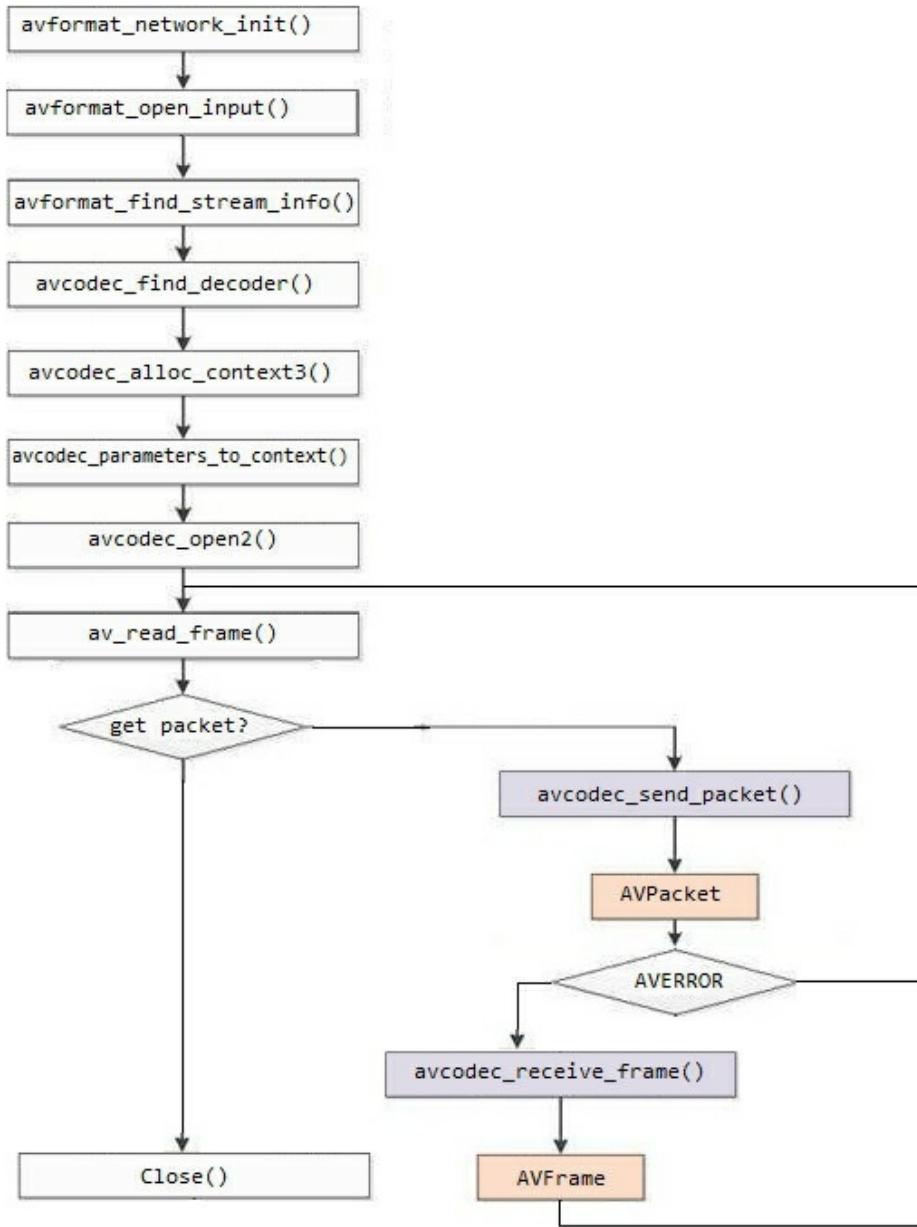
8. DECODING

Video decoding is the process of restoring the compressed video (compression format such as H264) into a YUV video stream through the corresponding decoding algorithm; in the eyes of the computer, first input a 01 string (compressed video) and then perform a large number of floating-point operations , And finally output a longer 01 string (restored uncompressed video). The internal part of the computer that can perform floating-point calculations is the CPU. At present, a number of GPU and GPU-like chips have emerged on the market, such as Nvidia, Hisilicon chips and even Intel's own nuclear display. Decoding using the former is generally called "soft decoding ", and the latter is called "hard decoding". If there is no special designation, FFMPEG is decoded by the CPU, that is, soft decoding.

Decoding flow chart (old version)



Decoding flowchart(New version)



1. CONNECT AND OPEN THE VIDEO STREAM

Connecting and opening the video stream must be the key to subsequent decoding. The API calls corresponding to this step are:

- **Int_avformat_network_init(void)**
this function will initialize and start the underlying TLS Library, which also explains a lot of information on the Internet if you want to open the network stream, This API is necessary .
- **int avformat_open_input(AVFormatContext** ps, const char* filename, AVInputFormat* fmt, AVDictionary ** options)**
The official statement is "open and read the video header information". This function is more complicated. The author has not fully understood every line of his source code, and roughly understands its function as the memory allocation of AVFormatContext. If it is a video file, it will detect its packaging format and load

the video source into the internal buffer; if it is a network streaming video, it will create a working connection video such as a socket to obtain its content and load it into the internal buffer. Finally, read the video header information.

Supplementary note: After this step, you can call the API to `av_dump_format()`print the basic information of the file, such as file length, bit rate, fps, encoding format, etc. The information is as follows:
Input # 0, avi, from '\$ {input_video_file_name}':
Metadata: encoder: Lavf57.83.100 Duration: 00: 10:
00.00, start: 0.000000, bitrate: 4196 kb / s Stream # 0:
0: Video: h264 (High) (H264 / 0x34363248),
yuvj420p (pc, bt709, progressive), 1920x1080, 4194 kb
/ s, 12 fps, 12 tbr, 12 tbn, 24 tbc

2. LOCATE VIDEO STREAM DATA

Whether it is offline or online video files, the relatively correct name should be "multimedia" files. Because these files generally have more than one channel of video stream data, they may also include multiple channels of audio data, video data, and even subtitle data. Therefore, before we do decoding, we need to first find the video stream data we need.

- **int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options)**
We can understand that this function has already done a complete set of decoding processes to obtain information about multimedia streams. A video file may include multiple media simultaneously streaming video files and audio files, etc. This also explains why even the subsequent traversal AVFormatContextof the streamsmembers of the (type AVStream) to do the corresponding decoding.

Basic information of the video stream, such as fps, code rate, and video length, is obtained in the first step of `avformat_open_input` opening the video header file. However, if the video file is h264 / mpeg bare stream data, there may be no header information to obtain it.

3. PREPARE THE DECODER CODEC

Codec is the soul of FFmpeg. As the name implies, decoding must be done by the decoder. The steps to prepare the decoder include: finding a suitable decoder-> copying the decoder (optional)-> opening the decoder.

- **AVCodec* avcodec_find_decoder(enum AVCodecID id)**

There is a corresponding API for searching for decoders AVCodec*
avcodec_find_decoder_by_name(const char* name). This function returns decoder named from the codec library. Generally, during hard decoding, the decoder should be specified by the decoder name (the process of hard decoding will be more complicated, often need to open the underlying library driver of related hardware, etc).

- **AVCodecContext***

```
avcodec_alloc_context3(const AVCodec*  
codec)andint  
avcodec_parameters_to_context(const  
AVCodec* codec, const AVCodecParameters*  
par)
```

avcodec_alloc_context3() created AVCodecContext and avcodec_parameters_to_context() really executed copies of the content. avcodec_parameters_to_context() It is a new API that replaces the old version avcodec_copy_context().

- **avcodec_open2 (AVCodecContext * avctx,
const AVCodec * codec, AVDictionary **
options)**
this function mainly serves the decoder, including assigning related variables to it Memory, check decoder status, etc. The above steps 1-3 are collectively referred to as the "decoding initialization stage". At this point, if the API return value of each step is OK, you can start the real decoding work!

4. DECODE

Core decoder are repeated Get_packet, unpacking work deframed here that the package is one of the important data structures FFmpeg: AVPacket, wherein the frame is equally important data structure: AVFrame.

- **AVPacket**

introduces and analyzes the data structure of many online materials. It is recommended to read the FFmpeg structure: AVPacket analysis. In short, the structure saves the multimedia data before decoding or decompression, including the stream data itself and additional information.

AVPacket is obtained by the function `av_read_frame(AVFormatContext* s, AVPacket* pkt)`. The specific implementation of this function has been improved in the new version to ensure that the complete frame data must be taken out every time. The introduction and analysis of the data structure of AVFrame are also many online materials. In short, the structure saves the data and additional

information of the decompressed frame itself after decoding. AVFrameIn the new version, it is generated by the function int avcodec_send_packet(AVCodecContext* avctx, AVPacket* pkt)and int avcodec_receive_frame(AVCodecContext* avctx, AVFrame* frame), the former actually performs the decoding operation, and the latter takes the decompressed frame data from the cache or decoder memory. What was used in the old version is avcodec_decode_video2() now deprecated. In addition, it should be noted that, generally speaking, it avcodec_send_packet() corresponds to once avcodec_receive_frame(), but there may be a case where it corresponds to multiple times. This relationship refers to one AVPacketcorresponding to one or more AVFrame.

SIMPLE DECODER BASED ON FFMPEG.

The functions of the key functions in the sample are listed as follows:

1. **avcodec_register_all ()**: Register all codecs.
2. **avcodec_find_decoder ()**: Find the decoder.
3. **avcodec_alloc_context3 ()**: allocate memory for AVCodecContext.
4. **avcodec_open2 ()**: Open the decoder.
5. **avcodec_decode_video2 ()**: decode a frame of data.
6. There are two normally "uncommon" functions:
7. **av_parser_init ()**: Initialize AVCodecParserContext.
8. **av_parser_parse2 ()**: Parse to get a Packet.
9. The two data storage structures are listed below:
10. **AVFrame**: store one frame of decoded pixel data
11. **AVPacket**: store one frame (generally)

compressed and encoded data

12. **AVCodecParser:** AVCodecParser is used to parse the input data stream and divide it into compressed encoded data frame by frame. The more vivid statement is to "cut" a long piece of continuous data into pieces of data. His core function is `av_parser_parse2()`. Its definition is shown below.

```
/**  
 * Parse a packet.  
 *  
 * @param s      parser context.  
 * @param avctx   codec context.  
 * @param poutbuf set to pointer to parsed buffer or NULL if not yet  
 * finished.  
 * @param poutbuf_size set to size of parsed buffer or zero if not yet  
 * finished.  
 * @param buf     input buffer.  
 * @param buf_size input length, to signal EOF, this should be 0 (so that  
 * the last frame can be output).  
 * @param pts     input presentation timestamp.  
 * @param dts     input decoding timestamp.  
 * @param pos     input byte position in stream.  
 * @return the number of bytes of the input bitstream used.  
 *  
 * Example:  
 * @code  
 * while(in_len){  
 *     len = av_parser_parse2(myparser, AVCodecContext, &data, &size,  
 *                           in_data, in_len,  
 *                           pts, dts, pos);  
 *     in_data += len;  
 }
```

```

*     in_len -= len;
*
*     if(size)
*         decode_frame(data, size);
* }
* @endcode
*/
int av_parser_parse2(AVCodecParserContext *s,
                     AVCodecContext *avctx,
                     uint8_t **poutbuf, int *poutbuf_size,
                     const uint8_t *buf, int buf_size,
                     int64_t pts, int64_t dts,
                     int64_t pos);

```

Among them, poutbuf points to the compressed encoded data frame output after parsing, and buf points to the input compressed encoded data. If the output data is empty after the function is executed (poutbuf_size is 0), it means that the parsing has not been completed, and you need to call av_parser_parse2() again to parse part of the data to get the parsed data frame. When the output data is not empty after the function is executed, it means that the parsing is complete. You can take out the frame of data in poutbuf for subsequent processing.

```

#include <stdio.h>

#define __STDC_CONSTANT_MACROS

#ifndef _WIN32
#endif

```

```
extern "C"
{
#include "libavcodec/avcodec.h"
};

#ifndef __cplusplus
extern "C"
{
#endif
#ifndef __cplusplus
#endif
#ifndef __cplusplus
#endif

//test different codec
#define TEST_H264 1
#define TEST_HEVC 0

int main(int argc, char* argv[])
{
    AVCodec *pCodec;
    AVCodecContext *pCodecCtx=NULL;
    AVCodecParserContext *pCodecParserCtx=NULL;

    FILE *fp_in;
    FILE *fp_out;
    AVFrame *pFrame;
    const int in_buffer_size=4096;
    uint8_t in_buffer[in_buffer_size +
FF_INPUT_BUFFER_PADDING_SIZE]={0};
    uint8_t *cur_ptr;
    int cur_size;
    AVPacket packet;
    int ret, got_picture;
    int y_size;
```

```

#if TEST_HEVC
    enum AVCodecID codec_id=AV_CODEC_ID_HEVC;
    char filepath_in[]{"bigbuckbunny_480x272.hevc"};
#elif TEST_H264
    AVCodecID codec_id=AV_CODEC_ID_H264;
    char filepath_in[]{"bigbuckbunny_480x272.h264"};
#else
    AVCodecID codec_id=AV_CODEC_ID_MPEG2VIDEO;
    char filepath_in[]{"bigbuckbunny_480x272.m2v"};
#endif

    char filepath_out[]{"bigbuckbunny_480x272.yuv"};
    int first_time=1;

    //av_log_set_level(AV_LOG_DEBUG);
    avcodec_register_all();

pCodec = avcodec_find_decoder(codec_id);
if (!pCodec) {
    printf("Codec not found\n");
    return -1;
}
pCodecCtx = avcodec_alloc_context3(pCodec);
if (!pCodecCtx){
    printf("Could not allocate video codec context\n");
    return -1;
}

pCodecParserCtx=av_parser_init(codec_id);
if (!pCodecParserCtx){
    printf("Could not allocate video parser context\n");
    return -1;
}

//if(pCodec->capabilities&CODEC_CAP_TRUNCATED)
//  pCodecCtx->flags|= CODEC_FLAG_TRUNCATED;

if (avcodec_open2(pCodecCtx, pCodec, NULL) < 0) {
    printf("Could not open codec\n");
}

```

```
    return -1;
}
//Input File
fp_in = fopen(filepath_in, "rb");
if (!fp_in) {
    printf("Could not open input stream\n");
    return -1;
}
//Output File
fp_out = fopen(filepath_out, "wb");
if (!fp_out) {
    printf("Could not open output YUV file\n");
    return -1;
}

pFrame = av_frame_alloc();
av_init_packet(&packet);

while (1) {

    cur_size = fread(in_buffer, 1, in_buffer_size, fp_in);
    if (cur_size == 0)
        break;
    cur_ptr=in_buffer;

    while (cur_size>0){

        int len = av_parser_parse2(
            pCodecParserCtx, pCodecCtx,
            &packet.data, &packet.size,
            cur_ptr , cur_size ,
            AV_NOPTS_VALUE, AV_NOPTS_VALUE,
            AV_NOPTS_VALUE);

        cur_ptr += len;
        cur_size -= len;

        if(packet.size==0)
            continue;
    }
}
```

```

//Some Info from AVCodecParserContext
printf("[Packet]Size:%6d\t",packet.size);
switch(pCodecParserCtx->pict_type){
case AV_PICTURE_TYPE_I: printf("Type:I\t");break;
case AV_PICTURE_TYPE_P: printf("Type:P\t");break;
case AV_PICTURE_TYPE_B: printf("Type:B\t");break;
default: printf("Type:Other\t");break;
}
printf("Number:%4d\n",pCodecParserCtx->output_picture_number);
ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
&packet);
if (ret < 0) {
printf("Decode Error.\n");
return ret;
}
if (got_picture) {
if(first_time){
printf("\nCodec Full Name:%s\n",pCodecCtx->codec->long_name);
printf("width:%d\nheight:%d\n\n",pCodecCtx->width,pCodecCtx-
>height);
first_time=0;
}
//Y, U, V
for(int i=0;i<pFrame->height;i++){
fwrite(pFrame->data[0]+pFrame->linesize[0]*i,1,pFrame-
>width,fp_out);
}
for(int i=0;i<pFrame->height/2;i++){
fwrite(pFrame->data[1]+pFrame->linesize[1]*i,1,pFrame-
>width/2,fp_out);
}
for(int i=0;i<pFrame->height/2;i++){
fwrite(pFrame->data[2]+pFrame->linesize[2]*i,1,pFrame-
>width/2,fp_out);
}
}

printf("Succeed to decode 1 frame!\n");

```

```
        }
    }
}

//Flush Decoder
packet.data = NULL;
packet.size = 0;
while(1){
    ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
&packet);
    if (ret < 0) {
        printf("Decode Error.\n");
        return ret;
    }
    if (!got_picture){
        break;
    }else {
        //Y, U, V
        for(int i=0;i<pFrame->height;i++){
            fwrite(pFrame->data[0]+pFrame->linesize[0]*i,1,pFrame-
>width,fp_out);
        }
        for(int i=0;i<pFrame->height/2;i++){
            fwrite(pFrame->data[1]+pFrame->linesize[1]*i,1,pFrame-
>width/2,fp_out);
        }
        for(int i=0;i<pFrame->height/2;i++){
            fwrite(pFrame->data[2]+pFrame->linesize[2]*i,1,pFrame-
>width/2,fp_out);
        }
    }
    printf("Flush Decoder: Succeed to decode 1 frame!\n");
}
}

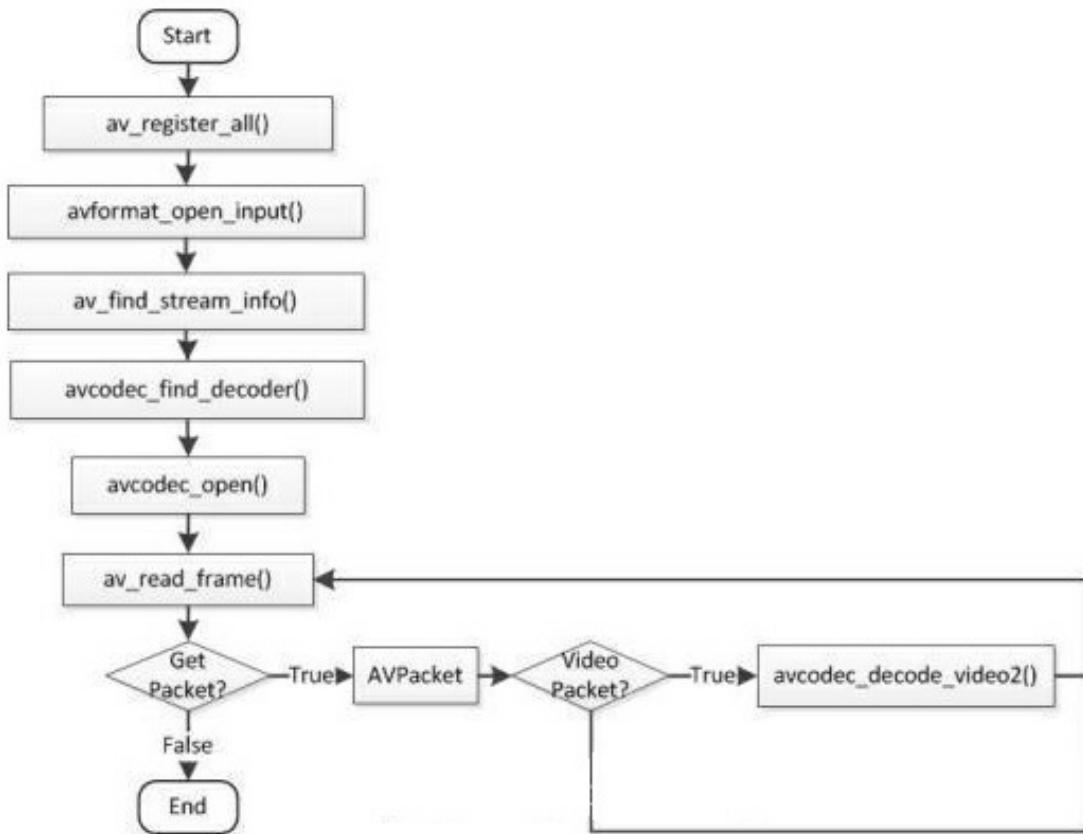
fclose(fp_in);
fclose(fp_out);
av_parser_close(pCodecParserCtx);
```

```
    av_frame_free(&pFrame);
    avcodec_close(pCodecCtx);
    av_free(pCodecCtx);
    return 0;
}
```

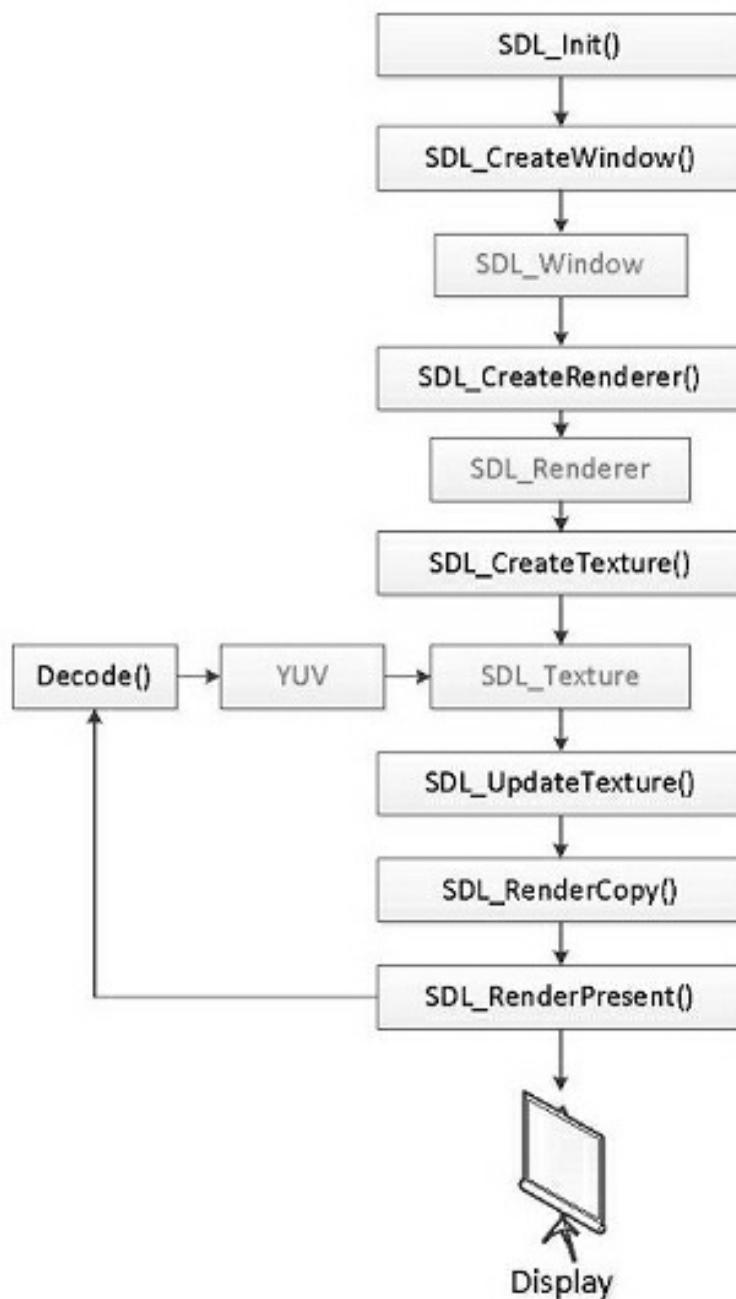
SIMPLE VIDEO PLAYER BASED ON FFMPEG + SDL

Although the player is simple, it almost contains all the necessary APIs to play a video using FFMPEG, and uses SDL to display the decoded video. And support a variety of video inputs such as streaming media, in a simple consideration, there is no audio part, while video playback uses a direct delay of 40ms. The decoding process of the player can be expressed in the form of diagram as follows:

FFmpeg decodes a video process as shown below:



SDL2.0 SHOWS THE FLOW CHART OF YUV



Briefly explain the role of each variable:

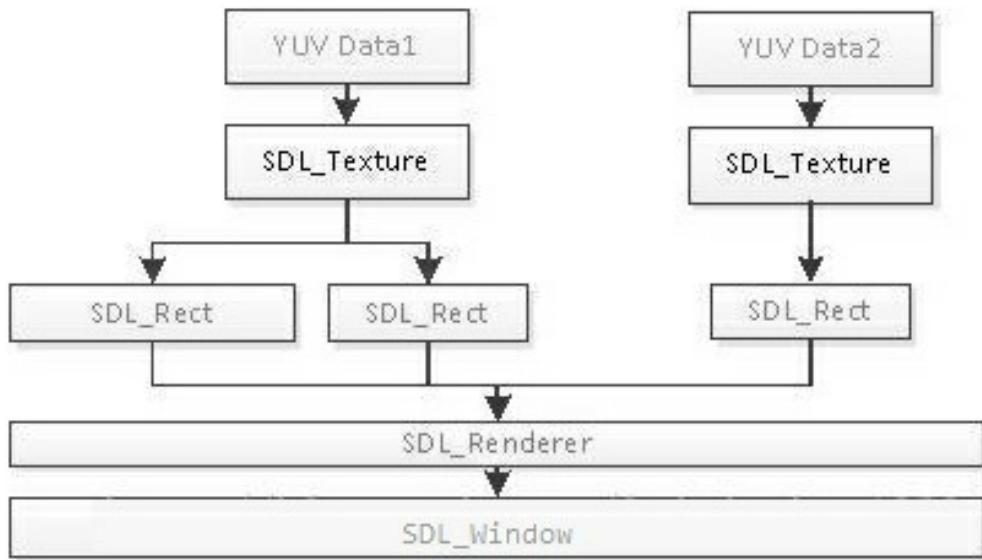
SDL_Window is the window that pops up when using SDL. In SDL 1.x version, only one window can be created. In the SDL2.0 version, multiple windows can be created.

SDL_Texture is used to display YUV data. One SDL_Texture corresponds to one frame of YUV data.

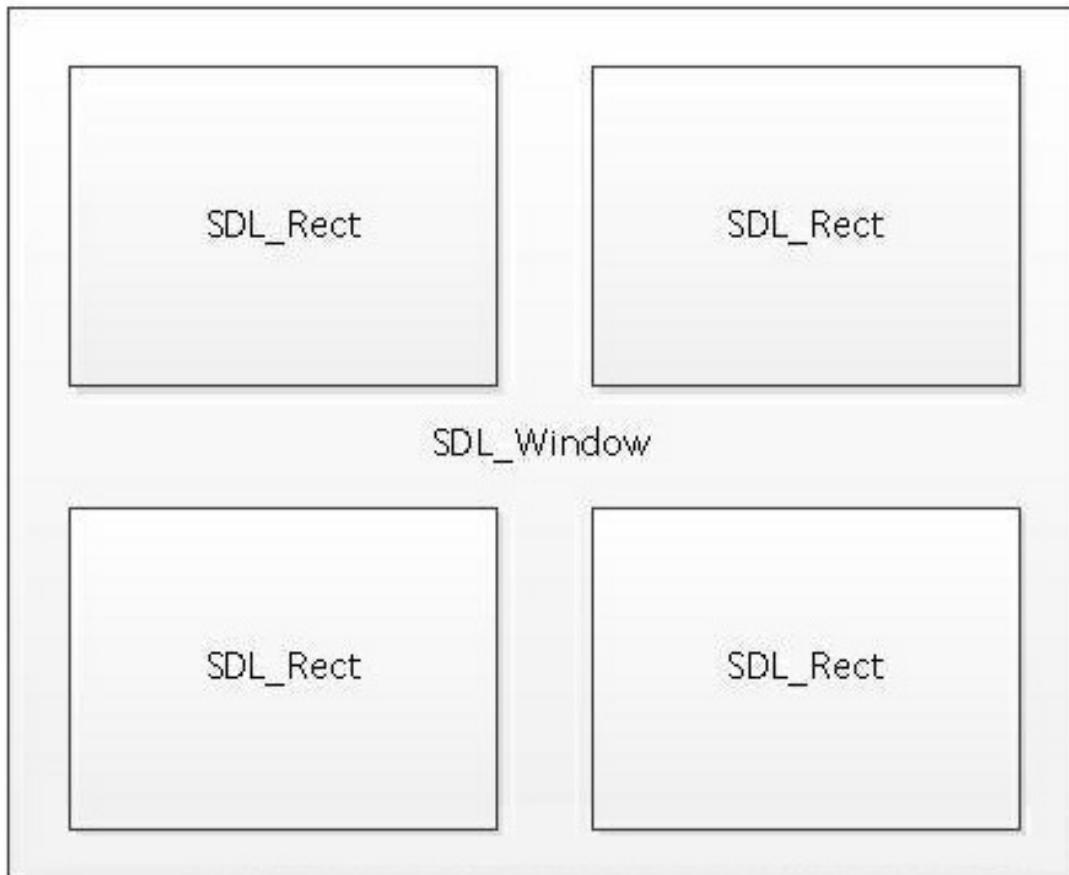
SDL_Renderer is used to render SDL_Texture to SDL_Window.

SDL_Rect is used to determine the location of SDL_Texture display. Note: A SDL_Texture can specify multiple different SDL_Rect, so that the same content can be displayed in different positions of SDL_Window (using the SDL_RenderCopy () function).

Their relationship is shown below:



The following figure gives an example, specifying 4 `SDL_Rect`, you can achieve 4-split screen display.



The most basic version, the beginning of learning.

```
#include <stdio.h>
#define __STDC_CONSTANT_MACROS

#ifndef _WIN32
//Windows
extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include "libavutil/imgutils.h"
#include "SDL2/SDL.h"
};
#else
//Linux...
#ifndef __cplusplus
extern "C"
{
#endif
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
#include <SDL2/SDL.h>
#include <libavutil/imgutils.h>
#ifndef __cplusplus
};
#endif
#endif

//Output YUV420P data as a file
#define OUTPUT_YUV420P 0

int main(int argc, char* argv[])
{
    AVFormatContext *pFormatCtx;
```

```
int          i, videoindex;
AVCodecContext *pCodecCtx;
AVCodec      *pCodec;
AVFrame     *pFrame,*pFrameYUV;
unsigned char *out_buffer;
AVPacket   *packet;
int y_size;
int ret, got_picture;
struct SwsContext *img_convert_ctx;

char filepath[]="bigbuckbunny_480x272.h265";
//SDL-----
int screen_w=0,screen_h=0;
SDL_Window *screen;
SDL_Renderer* sdlRenderer;
SDL_Texture* sdlTexture;
SDL_Rect sdlRect;

FILE *fp_yuv;

av_register_all();
avformat_network_init();
pFormatCtx = avformat_alloc_context();

if(avformat_open_input(&pFormatCtx,filepath,NULL,NULL)!=0){
printf("Couldn't open input stream.\n");
return -1;
}
if(avformat_find_stream_info(pFormatCtx,NULL)<0){
printf("Couldn't find stream information.\n");
return -1;
}
videoindex=-1;
for(i=0; i<pFormatCtx->nb_streams; i++)
if(pFormatCtx->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_VIDEO){
videoindex=i;
break;
```

```

    }

    if(videoindex== -1){
        printf(" Didn't find a video stream.\n");
        return -1;
    }

pCodecCtx=pFormatCtx->streams[videoindex]->codec;
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL){
    printf(" Codec not found.\n");
    return -1;
}
if(avcodec_open2(pCodecCtx, pCodec,NULL)<0){
    printf(" Could not open codec.\n");
    return -1;
}
pFrame=av_frame_alloc();
pFrameYUV=av_frame_alloc();
out_buffer=(unsigned char
*)av_malloc(av_image_get_buffer_size(AV_PIX_FMT_YUV420P,
pCodecCtx->width, pCodecCtx->height,1));
av_image_fill_arrays(pFrameYUV->data, pFrameYUV-
>linesize,out_buffer,
AV_PIX_FMT_YUV420P,pCodecCtx->width, pCodecCtx-
>height,1);
packet=(AVPacket *)av_malloc(sizeof(AVPacket));
//Output Info-----
printf("----- File Information ----- \n");
av_dump_format(pFormatCtx,0,filepath,0);
printf("-----\n");
img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx-
>height, pCodecCtx->pix_fmt,
pCodecCtx->width, pCodecCtx->height, AV_PIX_FMT_YUV420P,
SWS_BICUBIC, NULL, NULL, NULL);

#if OUTPUT_YUV420P
fp_yuv=fopen("output.yuv","wb+");
#endif

```

```

    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO |
SDL_INIT_TIMER)) {
        printf( "Could not initialize SDL - %s\n", SDL_GetError());
        return -1;
    }

    screen_w = pCodecCtx->width;
    screen_h = pCodecCtx->height;
    //SDL 2.0 Support for multiple windows
    screen = SDL_CreateWindow("Simplest ffmpeg player's Window",
SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED,
    screen_w, screen_h,
    SDL_WINDOW_OPENGL);

    if(!screen) {
        printf("SDL: could not create window -
exiting:%s\n",SDL_GetError());
        return -1;
    }

    sdlRenderer = SDL_CreateRenderer(screen, -1, 0);
    //IYUV: Y + U + V (3 planes)
    //YV12: Y + V + U (3 planes)
    sdlTexture = SDL_CreateTexture(sdlRenderer,
SDL_PIXELFORMAT_IYUV,
SDL_TEXTUREACCESS_STREAMING,pCodecCtx->width,pCodecCtx-
>height);

    sdlRect.x=0;
    sdlRect.y=0;
    sdlRect.w=screen_w;
    sdlRect.h=screen_h;

    //SDL End-----
    while(av_read_frame(pFormatCtx, packet)>=0){
        if(packet->stream_index==videoindex){
            ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
packet);

```

```

if(ret < 0){
printf("Decode Error.\n");
return -1;
}
if(got_picture){
sws_scale(img_convert_ctx, (const unsigned char* const*)pFrame-
>data, pFrame->linesize, 0, pCodecCtx->height,
pFrameYUV->data, pFrameYUV->linesize);
#endif OUTPUT_YUV420P
y_size=pCodecCtx->width*pCodecCtx->height;
fwrite(pFrameYUV->data[0],1,y_size,fp_yuv); //Y
fwrite(pFrameYUV->data[1],1,y_size/4,fp_yuv); //U
fwrite(pFrameYUV->data[2],1,y_size/4,fp_yuv); //V
#endif
//SDL-----
#if 0
SDL_UpdateTexture( sdlTexture, NULL, pFrameYUV->data[0],
pFrameYUV->linesize[0] );
#else
SDL_UpdateYUVTexture(sdlTexture, &sdlRect,
pFrameYUV->data[0], pFrameYUV->linesize[0],
pFrameYUV->data[1], pFrameYUV->linesize[1],
pFrameYUV->data[2], pFrameYUV->linesize[2]);
#endif
SDL_RenderClear( sdlRenderer );
SDL_RenderCopy( sdlRenderer, sdlTexture, NULL, &sdlRect);
SDL_RenderPresent( sdlRenderer );
//SDL End-----
//Delay 40ms
SDL_Delay(40);
}
}
av_free_packet(packet);
}
//flush decoder
//FIX: Flush Frames remained in Codec
while (1) {

```

```

    ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
packet);
    if (ret < 0)
        break;
    if (!got_picture)
        break;
    sws_scale(img_convert_ctx, (const unsigned char* const*)pFrame-
>data, pFrame->linesize, 0, pCodecCtx->height,
    pFrameYUV->data, pFrameYUV->linesize);
#ifndef OUTPUT_YUV420P
    int y_size=pCodecCtx->width*pCodecCtx->height;
    fwrite(pFrameYUV->data[0],1,y_size,fp_yuv); //Y
    fwrite(pFrameYUV->data[1],1,y_size/4,fp_yuv); //U
    fwrite(pFrameYUV->data[2],1,y_size/4,fp_yuv); //V
#endif
//SDL-----
    SDL_UpdateTexture( sdlTexture, &sdlRect, pFrameYUV->data[0],
pFrameYUV->linesize[0] );
    SDL_RenderClear( sdlRenderer );
    SDL_RenderCopy( sdlRenderer, sdlTexture, NULL, &sdlRect);
    SDL_RenderPresent( sdlRenderer );
//SDL End-----
//Delay 40ms
    SDL_Delay(40);
}

    sws_freeContext(img_convert_ctx);

#ifndef OUTPUT_YUV420P
    fclose(fp_yuv);
#endif

    SDL_Quit();

    av_frame_free(&pFrameYUV);
    av_frame_free(&pFrame);
    avcodec_close(pCodecCtx);
    avformat_close_input(&pFormatCtx);

```

```
    return 0;
}

//Based on the standard version, SDL Event is introduced.
//The effect is as follows:
//1. The window popped up by SDL can be moved
//2. The screen display is strictly 40ms per frame

/***
 * The simplest FFmpeg-based video player 2 (SDL upgraded version)
 * Simplest FFmpeg Player
 * Version 2 use SDL 2.0 instead of SDL 1.2 in version 1.
 *
 * This program realizes the decoding and display of video files (supports HEVC, H.264, MPEG2, etc.)
 * It is the simplest tutorial on FFmpeg video decoding.
 * You can understand the decoding process of FFmpeg by studying this example.
 * In this version, the SDL message mechanism is used to refresh the video screen.
 * This software is a simplest video player based on FFmpeg.
 * Suitable for beginner of FFmpeg.
 *
 * Remarks:
 * When the standard version is playing video, the screen display uses a 40ms delay. This has two consequences:
 * (1) The window popped up by SDL cannot be moved, and it always shows that it is busy
 * (2) The picture display is not strictly a 40ms frame, because the decoding time has not been considered.
 * SU(SDL Update) In the process of video decoding, the version no longer uses the 40ms delay method, but is created
 * A thread sends a custom message every 40ms to inform the main function to decode and display. After doing so:
 * (1) The window popped up by SDL can be moved
 * (2) Screen display is strictly 40ms a frame
 * Remark:
 * Standard Version use's SDL_Delay() to control video's frame rate, it has
```

```
2
* disadvantages:
* (1)SDL's Screen can't be moved and always "Busy".
* (2)Frame rate can't be accurate because it doesn't consider the time
consumed
* by avcodec_decode_video2()
* SU(SDL Update) Version solved 2 problems above. It create a thread to
send SDL
* Event every 40ms to tell the main loop to decode and show video frames.
*/
#include <stdio.h>

#define __STDC_CONSTANT_MACROS

#ifndef _WIN32
//Windows
extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include "libavutil/imgutils.h"
#include "SDL2/SDL.h"
};
#else
//Linux...
#ifndef __cplusplus
extern "C"
{
#endif
#endif
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
#include <libavutil/imgutils.h>
#include <SDL2/SDL.h>
#ifndef __cplusplus
};

```

```
#endif
#endif

//Refresh Event
#define SFM_REFRESH_EVENT (SDL_USEREVENT + 1)
#define SFM_BREAK_EVENT (SDL_USEREVENT + 2)
int thread_exit=0;
int thread_pause=0;
int sfp_refresh_thread(void *opaque){
    thread_exit=0;
    thread_pause=0;

    while (!thread_exit) {
        if(!thread_pause){
            SDL_Event event;
            event.type = SFM_REFRESH_EVENT;
            SDL_PushEvent(&event);
        }
        SDL_Delay(40);
    }
    thread_exit=0;
    thread_pause=0;
    //Break
    SDL_Event event;
    event.type = SFM_BREAK_EVENT;
    SDL_PushEvent(&event);
    return 0;
}

int main(int argc, char* argv[])
{
    AVFormatContext *pFormatCtx;
    int          i, videoindex;
    AVCodecContext *pCodecCtx;
    AVCodec      *pCodec;
    AVFrame     *pFrame,*pFrameYUV;
    unsigned char *out_buffer;
```

```
AVPacket *packet;
int ret, got_picture;

//-----SDL-----
int screen_w,screen_h;
SDL_Window *screen;
SDL_Renderer* sdlRenderer;
SDL_Texture* sdlTexture;
SDL_Rect sdlRect;
SDL_Thread *video_tid;
SDL_Event event;

struct SwsContext *img_convert_ctx;
char filepath[]="Titanic.ts";

av_register_all();
avformat_network_init();
pFormatCtx = avformat_alloc_context();

if(avformat_open_input(&pFormatCtx,filepath,NULL,NULL)!=0){
printf("Couldn't open input stream.\n");
return -1;
}
if(avformat_find_stream_info(pFormatCtx,NULL)<0){
printf("Couldn't find stream information.\n");
return -1;
}
videoindex=-1;
for(i=0; i<pFormatCtx->nb_streams; i++)
if(pFormatCtx->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_VIDEO){
videoindex=i;
break;
}
if(videoindex==-1){
printf("Didn't find a video stream.\n");
return -1;
}
```

```

pCodecCtx=pFormatCtx->streams[videoindex]->codec;
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL){
printf("Codec not found.\n");
return -1;
}
if(avcodec_open2(pCodecCtx, pCodec,NULL)<0){
printf("Could not open codec.\n");
return -1;
}
pFrame=av_frame_alloc();
pFrameYUV=av_frame_alloc();

out_buffer=(unsigned char
*)av_malloc(av_image_get_buffer_size(AV_PIX_FMT_YUV420P,
pCodecCtx->width, pCodecCtx->height,1));
av_image_fill_arrays(pFrameYUV->data, pFrameYUV-
>linesize,out_buffer,
AV_PIX_FMT_YUV420P,pCodecCtx->width, pCodecCtx-
>height,1);

//Output Info-----
printf("----- File Information ----- \n");
av_dump_format(pFormatCtx,0,filepath,0);
printf("----- \n");
img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx-
>height, pCodecCtx->pix_fmt,
pCodecCtx->width, pCodecCtx->height, AV_PIX_FMT_YUV420P,
SWS_BICUBIC, NULL, NULL, NULL);

if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO |
SDL_INIT_TIMER)) {
printf( "Could not initialize SDL - %s\n", SDL_GetError());
return -1;
}
//SDL 2.0 Support for multiple windows
screen_w = pCodecCtx->width;
screen_h = pCodecCtx->height;

```

```

screen = SDL_CreateWindow("Simplest ffmpeg player's Window",
SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED,
screen_w, screen_h,SDL_WINDOW_OPENGL);

if(!screen) {
printf("SDL: could not create window -
exiting:%s\n",SDL_GetError());
return -1;
}
sdlRenderer = SDL_CreateRenderer(screen, -1, 0);
//IYUV: Y + U + V (3 planes)
//YYV12: Y + V + U (3 planes)
sdlTexture = SDL_CreateTexture(sdlRenderer,
SDL_PIXELFORMAT_IYUV,
SDL_TEXTUREACCESS_STREAMING,pCodecCtx->width,pCodecCtx-
>height);

sdlRect.x=0;
sdlRect.y=0;
sdlRect.w=screen_w;
sdlRect.h=screen_h;

packet=(AVPacket *)av_malloc(sizeof(AVPacket));
video_tid = SDL_CreateThread(sf_refresh_thread,NULL,NULL);
//-----SDL End-----
//Event Loop
for (;;) {
//Wait
SDL_WaitEvent(&event);
if(event.type==SFM_REFRESH_EVENT){
while(1){
if(av_read_frame(pFormatCtx, packet)<0)
thread_exit=1;

if(packet->stream_index==videoindex)
break;
}
}

```

```
    ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
packet);
    if(ret < 0){
        printf("Decode Error.\n");
        return -1;
    }
    if(got_picture){
        sws_scale(img_convert_ctx, (const unsigned char* const*)pFrame-
>data, pFrame->linesize, 0, pCodecCtx->height, pFrameYUV->data,
pFrameYUV->linesize);
        //SDL-----
        SDL_UpdateTexture( sdlTexture, NULL, pFrameYUV->data[0],
pFrameYUV->linesize[0] );
        SDL_RenderClear( sdlRenderer );
        SDL_RenderCopy( sdlRenderer, sdlTexture, NULL, NULL );
        SDL_RenderPresent( sdlRenderer );
        //SDL End-----
    }
    av_free_packet(packet);
}else if(event.type==SDL_KEYDOWN){
//Pause
if(event.key.keysym.sym==SDLK_SPACE)
thread_pause=!thread_pause;
}else if(event.type==SDL_QUIT){
thread_exit=1;
}else if(event.type==SFM_BREAK_EVENT){
break;
}

}
sws_freeContext(img_convert_ctx);
SDL_Quit();
av_frame_free(&pFrameYUV);
av_frame_free(&pFrame);
avcodec_close(pCodecCtx);
avformat_close_input(&pFormatCtx);
return 0;
```

}

SIMPLE AUDIO PLAYER BASED ON FFMPEG + SDL

```
/**  
 * Simplest FFmpeg Audio Player  
 *  
 * This software decode and play audio streams.  
 * Suitable for beginner of FFmpeg.  
 *  
 * This version use SDL 2.0 instead of SDL 1.2 in version 1  
 * Note:The good news for audio is that, with one exception,  
 * it's entirely backwards compatible with 1.2.  
 * That one really important exception: The audio callback  
 * does NOT start with a fully initialized buffer anymore.  
 * You must fully write to the buffer in all cases. In this  
 * example it is SDL_memset(stream, 0, len);  
 *  
 * Version 2.0  
 */  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
#define __STDC_CONSTANT_MACROS  
#ifdef _WIN32  
//Windows  
extern "C"  
{  
#include "libavcodec/avcodec.h"  
#include "libavformat/avformat.h"  
#include "libswresample/swresample.h"
```

```
#include "SDL2/SDL.h"
};

#ifndef __cplusplus
extern "C"
{
#endif

#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libsresample/swresample.h>
#include <SDL2/SDL.h>
#ifndef __cplusplus
};
#endif
#endif

#define MAX_AUDIO_FRAME_SIZE 192000 // 1 second of 48khz 32bit
audio
//Output PCM
#define OUTPUT_PCM 1
//Use SDL
#define USE SDL 1

//Buffer:
//|-----|-----|
//chunk-----pos---len----|
static Uint8 *audio_chunk;
static Uint32 audio_len;
static Uint8 *audio_pos;

/* The audio function callback takes the following parameters:
 * stream: A pointer to the audio buffer to be filled
 * len: The length (in bytes) of the audio buffer
 */
void fill_audio(void *udata, Uint8 *stream, int len){
    //SDL 2.0
    SDL_memset(stream, 0, len);
```

```
if(audio_len==0)
return;

len=(len>audio_len?
audio_len:len);/* Mix as much data as possible */
    SDL_MixAudio(stream,audio_pos,len,SDL_MIX_MAXVOLUME);
    audio_pos += len;
    audio_len -= len;
}

//-----
int main(int argc, char* argv[])
{
    AVFormatContext *pFormatCtx;
    Int i, audioStream;
    AVCodecContext *pCodecCtx;
    AVCodec      *pCodec;
    AVPacket     *packet;
    uint8_t      *out_buffer;
    AVFrame      *pFrame;
    SDL_AudioSpec wanted_spec;
    int ret;
    uint32_t len = 0;
    int got_picture;
    int index = 0;
    int64_t in_channel_layout;
    struct SwrContext *au_convert_ctx;
    FILE *pFile=NULL;
    char url[]="xiaoqingge.mp3";

    av_register_all();
    avformat_network_init();
    pFormatCtx = avformat_alloc_context();
    //Open
    if(avformat_open_input(&pFormatCtx,url,NULL,NULL)!=0){
        printf("Couldn't open input stream.\n");
        return -1;
    }
    // Retrieve stream information
```

```
if(avformat_find_stream_info(pFormatCtx,NULL)<0){
printf("Couldn't find stream information.\n");
return -1;
}
// Dump valid information onto standard error
av_dump_format(pFormatCtx, 0, url, false);

// Find the first audio stream
audioStream=-1;
for(i=0; i < pFormatCtx->nb_streams; i++)
if(pFormatCtx->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_AUDIO){
audioStream=i;
break;
}

if(audioStream==-1){
printf("Didn't find a audio stream.\n");
return -1;
}

// Get a pointer to the codec context for the audio stream
pCodecCtx=pFormatCtx->streams[audioStream]->codec;

// Find the decoder for the audio stream
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL){
printf("Codec not found.\n");
return -1;
}
// Open codec
if(avcodec_open2(pCodecCtx, pCodec,NULL)<0){
printf("Could not open codec.\n");
return -1;
}
#endif OUTPUT_PCM
pFile=fopen("output.pcm", "wb");
#endif
```

```

packet=(AVPacket *)av_malloc(sizeof(AVPacket));
av_init_packet(packet);
//Out Audio Param
uint64_t out_channel_layout=AV_CH_LAYOUT_STEREO;
//nb_samples: AAC-1024 MP3-1152
int out_nb_samples=pCodecCtx->frame_size;
AVSampleFormat out_sample_fmt=AV_SAMPLE_FMT_S16;
int out_sample_rate=44100;
int
out_channels=av_get_channel_layout_nb_channels(out_channel_layout);
//Out Buffer Size
int out_buffer_size=av_samples_get_buffer_size(NULL,out_channels
,out_nb_samples,out_sample_fmt, 1);

out_buffer=(uint8_t *)av_malloc(MAX_AUDIO_FRAME_SIZE*2);
pFrame=av_frame_alloc();
//SDL-----
#if USE SDL
//Init
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO |
SDL_INIT_TIMER)) {
printf( "Could not initialize SDL - %s\n", SDL_GetError());
return -1;
}
//SDL_AudioSpec
wanted_spec.freq = out_sample_rate;
wanted_spec.format = AUDIO_S16SYS;
wanted_spec.channels = out_channels;
wanted_spec.silence = 0;
wanted_spec.samples = out_nb_samples;
wanted_spec.callback = fill_audio;
wanted_spec.userdata = pCodecCtx;

if (SDL_OpenAudio(&wanted_spec, NULL)<0){
printf("can't open audio.\n");
return -1;
}
#endif

```

```

//FIX:Some Codec's Context Information is missing
in_channel_layout=av_get_default_channel_layout(pCodecCtx->channels);
//Swr
au_convert_ctx = swr_alloc();
au_convert_ctx=swr_alloc_set_opts(au_convert_ctx,out_channel_layout,
out_sample_fmt, out_sample_rate, in_channel_layout,pCodecCtx->sample_fmt, pCodecCtx->sample_rate,0, NULL);
swr_init(au_convert_ctx);

//Play
SDL_PauseAudio(0);

while(av_read_frame(pFormatCtx, packet)>=0){
if(packet->stream_index==audioStream){
ret = avcodec_decode_audio4( pCodecCtx, pFrame,&got_picture,
packet);
if ( ret < 0 ) {
printf("Error in decoding audio frame.\n");
return -1;
}
if ( got_picture > 0 ){
swr_convert(au_convert_ctx,&out_buffer,
MAX_AUDIO_FRAME_SIZE,(const uint8_t **)(pFrame->data , pFrame->nb_samples);
#if 1
printf("index:%5d\t pts:%lld\t packet size:%d\n",index,packet->pts,packet->size);
#endif
#if OUTPUT_PCM
//Write PCM
fwrite(out_buffer, 1, out_buffer_size, pFile);
#endif
index++;
}
#endif
#if USE SDL
while(audio_len>0)//Wait until finish

```

```
    SDL_Delay(1);
    //Set audio buffer (PCM data)
    audio_chunk = (Uint8 *) out_buffer;
    //Audio buffer length
    audio_len =out_buffer_size;
    audio_pos = audio_chunk;

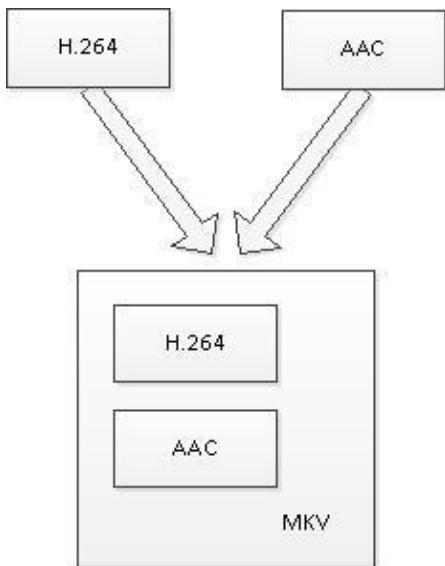
#endif
}
av_free_packet(packet);
}

swr_free(&au_convert_ctx);

#if USE	SDL
    SDL_CloseAudio();//Close SDL
    SDL_Quit();
#endif
#if OUTPUT_PCM
    fclose(pFile);
#endif
av_free(out_buffer);
avcodec_close(pCodecCtx);
avformat_close_input(&pFormatCtx);
return 0;
}
```

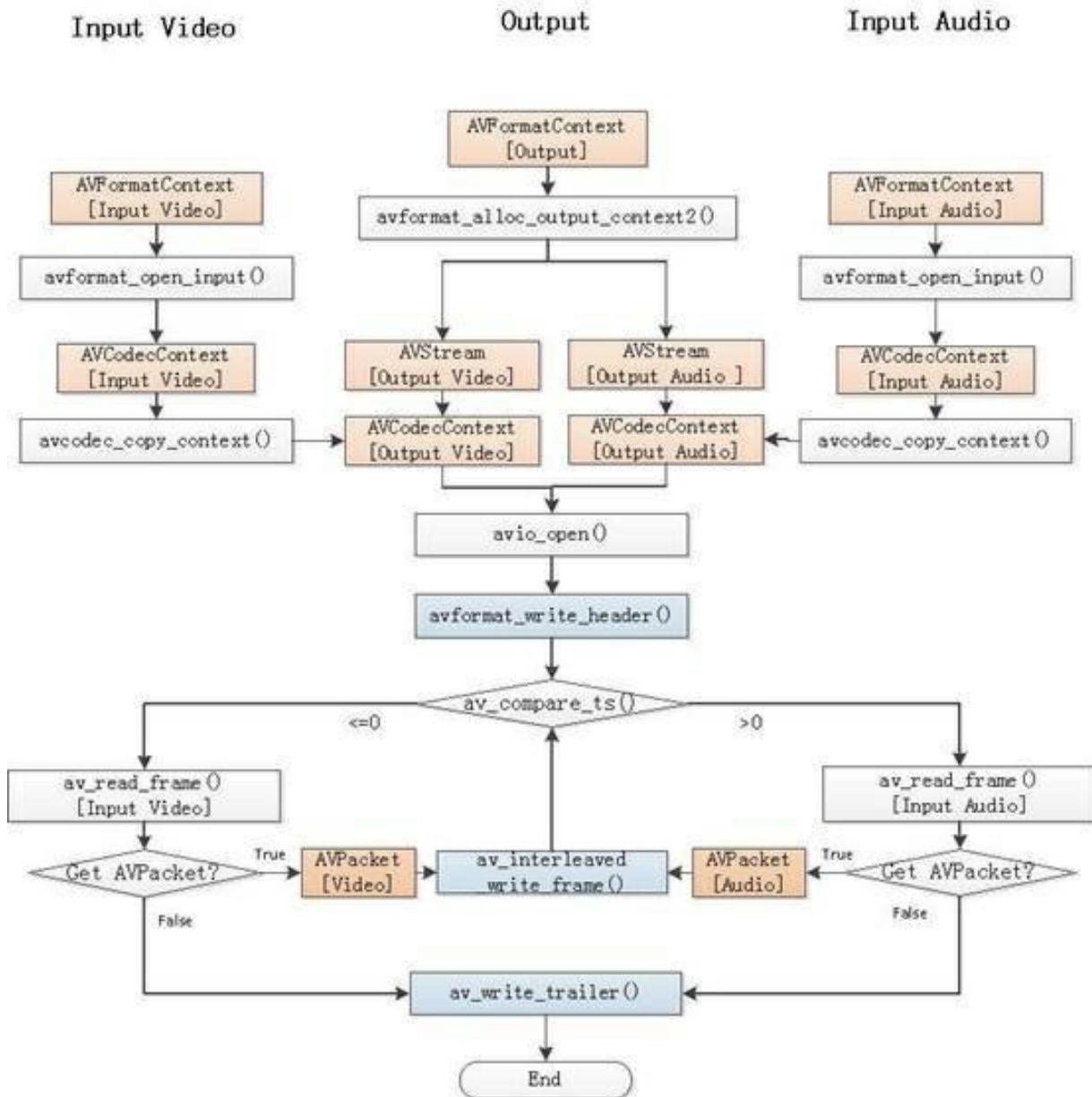
9. MULTIPLEXING

Multiplexing (a.k.a muxing) is the process used in telecommunications, electronics, and video to combine multiple signals into one to be transferred over a shared medium. Multiplexing is used to take one or more video, audio, caption, and/or metadata streams and combine them into a single container such as a TS segment or MP4.



The flow of the program is shown in the figure below. It can be seen from the flowchart that a total of 3 AVFormatContexts are initialized, 2 of which are used for input and 1 for output. After the three

AVFormatContexts are initialized, the input video / audio parameters can be copied to the AVCodecContext structure of the output video / audio through the `avcodec_copy_context()` function. Then call the `av_read_frame()` of the video input stream and the audio input stream respectively, take the AVPacket of the video from the video input stream, take the AVPacket of the audio from the audio input stream, and write the taken AVPacket to the output file respectively. In the meantime, a less common function `av_compare_ts()` is used, which is used to compare timestamps. This function can decide whether to write video or audio.



The input video is not necessarily a H.264 bare stream file, and the audio is not necessarily a pure audio file. You can select two encapsulated video and audio files as input. The program will "pick" the video

stream from the video input file, "pick" the audio stream from the audio input file, and then multiplex the "selected" video and audio streams.

PS1: For H.264 in some packaging formats (such as MP4 / FLV / MKV, etc.), a bitstream filter named "h264_mp4toannexb" is required.

PS2: For AAC in some packaging formats (such as MP4 / FLV / MKV, etc.), a bitstream filter named "aac_adtstoasc" is required.

Briefly introduce the meaning of each important function in the process:

1. `avformat_open_input ()`: Open the input file.
2. `avcodec_copy_context ()`: Assign the parameters of AVCodecContext.
3. `avformat_alloc_output_context2 ()`: Initialize the output file.
4. `avio_open ()`: Open the output file.
5. `avformat_write_header ()`: Write the file header.
6. `av_compare_ts ()`: Compare timestamps and decide whether to write video or audio. This function is relatively rare.
7. `av_read_frame ()`: read an AVPacket from the input file.

8. `av_interleaved_write_frame ()`: Write an AVPacket to the output file.
9. `av_write_trailer ()`: Write to the end of the file.

```
#include <stdio.h>
#define __STDC_CONSTANT_MACROS

#ifndef _WIN32
//Windows
extern "C"
{
#include "libavformat/avformat.h"
};

#ifndef __cplusplus
extern "C"
{
#endif
#endif
#ifndef __cplusplus
#include <libavformat/avformat.h>
#ifndef __cplusplus
#endif
#endif
#endif

/*
FIX: H.264 in some container format (FLV, MP4, MKV etc.) need
"h264_mp4toannexb" bitstream filter (BSF)
 *Add SPS,PPS in front of IDR frame
 *Add start code ("0,0,0,1") in front of NALU
H.264 in some container (MPEG2TS) don't need this BSF.
*/
//'1': Use H.264 Bitstream Filter
#define USE_H264BSF 0

/*
FIX:AAC in some container format (FLV, MP4, MKV etc.) need
```

```
"aac_adtstoasc" bitstream filter (BSF)
*/
//1': Use AAC Bitstream Filter
#define USE_AACBSF 0

int main(int argc, char* argv[])
{
    AVOutputFormat *ofmt = NULL;
    //Input AVFormatContext and Output AVFormatContext
    AVFormatContext *ifmt_ctx_v = NULL, *ifmt_ctx_a =
NULL,*ofmt_ctx = NULL;
    AVPacket pkt;
    int ret, i;
    int videoindex_v=-1,videoindex_out=-1;
    int audioindex_a=-1,audioindex_out=-1;
    int frame_index=0;
    int64_t cur_pts_v=0,cur_pts_a=0;

    //const char *in_filename_v = "cuc_ieschool.ts";//Input file URL
    const char *in_filename_v = "cuc_ieschool.h264";
    //const char *in_filename_a = "cuc_ieschool.mp3";
    //const char *in_filename_a = "gowest.m4a";
    //const char *in_filename_a = "gowest.aac";
    const char *in_filename_a = "huoyuanjia.mp3";

    const char *out_filename = "cuc_ieschool.mp4";//Output file URL
    av_register_all();
    //Input
    if ((ret = avformat_open_input(&ifmt_ctx_v, in_filename_v, 0, 0)) <
0) {
        printf( "Could not open input file.");
        goto end;
    }
    if ((ret = avformat_find_stream_info(ifmt_ctx_v, 0)) < 0) {
        printf( "Failed to retrieve input stream information");
        goto end;
    }
```

```

    if ((ret = avformat_open_input(&ifmt_ctx_a, in_filename_a, 0, 0)) <
0) {
    printf( "Could not open input file.");
    goto end;
}
if ((ret = avformat_find_stream_info(ifmt_ctx_a, 0)) < 0) {
    printf( "Failed to retrieve input stream information");
    goto end;
}
printf("=====Input Information=====\\n");
av_dump_format(ifmt_ctx_v, 0, in_filename_v, 0);
av_dump_format(ifmt_ctx_a, 0, in_filename_a, 0);
printf("=====\\n");
//Output
avformat_alloc_output_context2(&ofmt_ctx, NULL, NULL,
out_filename);
if (!ofmt_ctx) {
    printf( "Could not create output context\\n");
    ret = AVERROR_UNKNOWN;
    goto end;
}
ofmt = ofmt_ctx->oformat;

for (i = 0; i < ifmt_ctx_v->nb_streams; i++) {
//Create output AVStream according to input AVStream
    if(ifmt_ctx_v->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_VIDEO){
        AVStream *in_stream = ifmt_ctx_v->streams[i];
        AVStream *out_stream = avformat_new_stream(ofmt_ctx,
in_stream->codec->codec);
        videoindex_v=i;
        if (!out_stream) {
            printf( "Failed allocating output stream\\n");
            ret = AVERROR_UNKNOWN;
            goto end;
        }
        videoindex_out=out_stream->index;
}

```

```
//Copy the settings of AVCodecContext
if (avcodec_copy_context(out_stream->codec, in_stream->codec) < 0) {
    printf( "Failed to copy context from input to output stream codec
context\n");
    goto end;
}
out_stream->codec->codec_tag = 0;
if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
out_stream->codec->flags |= CODEC_FLAG_GLOBAL_HEADER;
break;
}
}

for (i = 0; i < ifmt_ctx_a->nb_streams; i++) {
//Create output AVStream according to input AVStream
if(ifmt_ctx_a->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_AUDIO){
    AVStream *in_stream = ifmt_ctx_a->streams[i];
    AVStream *out_stream = avformat_new_stream(ofmt_ctx,
in_stream->codec->codec);
    audioindex_a=i;
    if (!out_stream) {
        printf( "Failed allocating output stream\n");
        ret = AVERRORE_UNKOWN;
        goto end;
    }
    audioindex_out=out_stream->index;
    //Copy the settings of AVCodecContext
    if (avcodec_copy_context(out_stream->codec, in_stream->codec) <
0) {
        printf( "Failed to copy context from input to output stream codec
context\n");
        goto end;
    }
    out_stream->codec->codec_tag = 0;
    if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
out_stream->codec->flags |= CODEC_FLAG_GLOBAL_HEADER;
```

```

break;
}
}

printf("=====Output Information=====\\n");
av_dump_format(ofmt_ctx, 0, out_filename, 1);
printf("=====\\n");
//Open output file
if (!(ofmt->flags & AVFMT_NOFILE)) {
    if (avio_open(&ofmt_ctx->pb, out_filename, AVIO_FLAG_WRITE)
< 0) {
        printf( "Could not open output file '%s'", out_filename);
        goto end;
    }
}
//Write file header
if (avformat_write_header(ofmt_ctx, NULL) < 0) {
    printf( "Error occurred when opening output file\\n");
    goto end;
}
//FIX
#if USE_H264BSF
    AVBitStreamFilterContext* h264bsfc =
av_bitstream_filter_init("h264_mp4toannexb");
#endif
#if USE_AACBSF
    AVBitStreamFilterContext* aacbsfc =
av_bitstream_filter_init("aac_adtstoasc");
#endif

while (1) {
AVFormatContext *ifmt_ctx;
int stream_index=0;
AVStream *in_stream, *out_stream;

//Get an AVPacket
if(av_compare_ts(cur_pts_v,ifmt_ctx_v->streams[videoindex_v]-
>time_base,cur_pts_a,ifmt_ctx_a->streams[audioindex_a]->time_base) <=

```

```

0){
    ifmt_ctx=ifmt_ctx_v;
    stream_index=videoindex_out;

    if(av_read_frame(ifmt_ctx, &pkt) >= 0){
        do{
            in_stream = ifmt_ctx->streams[pkt.stream_index];
            out_stream = ofmt_ctx->streams[stream_index];

            if(pkt.stream_index==videoindex_v){
                //FIX£°No PTS (Example: Raw H.264)
                //Simple Write PTS
                if(pkt.pts==AV_NOPTS_VALUE){
                    //Write PTS
                    AVRational time_base1=in_stream->time_base;
                    //Duration between 2 frames (us)
                    int64_t calc_duration=(double)AV_TIME_BASE/av_q2d(in_stream->r_frame_rate);
                    //Parameters    pkt.pts=(double)
                    (frame_index*calc_duration)/(double)
                    (av_q2d(time_base1)*AV_TIME_BASE);
                    pkt.dts=pkt.pts;          pkt.duration=
                    (double)calc_duration/(double)(av_q2d(time_base1)*AV_TIME_BASE);
                    frame_index++;
                }
                cur_pts_v=pkt.pts;
                break;
            }
        }while(av_read_frame(ifmt_ctx, &pkt) >= 0);
    }else{
        break;
    }
}else{
    ifmt_ctx=ifmt_ctx_a;
    stream_index=audioindex_out;
    if(av_read_frame(ifmt_ctx, &pkt) >= 0){
        do{
            in_stream = ifmt_ctx->streams[pkt.stream_index];

```

```

out_stream = ofmt_ctx->streams[stream_index];
if(pkt.stream_index==audioindex_a){

    //Simple Write PTS
    if(pkt.pts==AV_NOPTS_VALUE){
        //Write PTS
        AVRational time_base1=in_stream->time_base;
        //Duration between 2 frames (us)
        int64_t calc_duration=(double)AV_TIME_BASE/av_q2d(in_stream-
>r_frame_rate);
            //Parameters          pkt.pts=(double)
        (frame_index*calc_duration)/(double)
        (av_q2d(time_base1)*AV_TIME_BASE);
            pkt.dts=pkt.pts;
        pkt.duration=(double)calc_duration/(double)
        (av_q2d(time_base1)*AV_TIME_BASE);
            frame_index++;
        }
        cur_pts_a=pkt.pts;
        break;
    }
    }while(av_read_frame(ifmt_ctx, &pkt) >= 0);
}else{
break;
}
}

//FIX:Bitstream Filter
#if USE_H264BSF
    av_bitstream_filter_filter(h264bsfc, in_stream->codec, NULL,
&pkt.data, &pkt.size, pkt.data, pkt.size, 0);
#endif
#if USE_AACBSF
    av_bitstream_filter_filter(aacbsfc, out_stream->codec, NULL,
&pkt.data, &pkt.size, pkt.data, pkt.size, 0);
#endif
//Convert PTS/DTS
pkt.pts = av_rescale_q_rnd(pkt.pts, in_stream->time_base,

```

```

out_stream->time_base, (AVRounding)
(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.dts = av_rescale_q_rnd(pkt.dts, in_stream->time_base,
out_stream->time_base, (AVRounding)
(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.duration = av_rescale_q(pkt.duration, in_stream->time_base,
out_stream->time_base);
    pkt.pos = -1;
    pkt.stream_index=stream_index;

    printf("Write 1 Packet. size:%5d\tpts:%lld\n",pkt.size,pkt.pts);
//Write
if (av_interleaved_write_frame(ofmt_ctx, &pkt) < 0) {
printf( "Error muxing packet\n");
break;
}
av_free_packet(&pkt);
}
//Write file trailer
av_write_trailer(ofmt_ctx);

#if USE_H264BSF
    av_bitstream_filter_close(h264bsfc);
#endif
#if USE_AACBSF
    av_bitstream_filter_close(aacbsfc);
#endif

end:
avformat_close_input(&ifmt_ctx_v);
avformat_close_input(&ifmt_ctx_a);
/* close output */
if (ofmt_ctx && !(ofmt->flags & AVFMT_NOFILE))
avio_close(ofmt_ctx->pb);
avformat_free_context(ofmt_ctx);
if (ret < 0 && ret != AVERROR_EOF) {
printf( "Error occurred.\n");
return -1;
}

```

```
    return 0;  
}
```

10. DEMULTIPLEXING

Demultiplexing is the process of splitting a digital media file into streams with video, audio and subtitles. The main tasks of demultiplexing are:

1. Determine whether the stream format is supported.
2. Extract the header information of the stream, such as the length and width of the video and the number of audio channel samples.
3. Read the compressed data stream for decoding by the decoder.

These 3 steps are the 3 abstract functions we use:

`avformat_open_input-> av_find_stream_info->
av_read_frame.`

Let's take a look at the core structure of the demultiplexer in ffmpeg (taking FLAC audio as an example):

```
AVInputFormat ff_flac_demuxer = {  
    .name = "flac",  
    .long_name = NULL_IF_CONFIG_SMALL ("raw FLAC"),  
    .read_probe = flac_probe,  
    .read_header = flac_read_header,  
    .read_packet = ff_raw_read_partial_packet,  
    .flags = AVFMT_GENERIC_INDEX,
```

```
.extensions = "flac",
.raw_codec_id = AV_CODEC_ID_FLAC,
};
```

After we execute `av_register_all`, this flac's `AVInputFormat` structure pointer will be registered in a global object table. The work of `avformat_open_input` is very simple. It reads the byte stream header from the source through `avio`, and then runs this global object table, one by one to execute the `read_probe` function, if any `read_probe` recognizes it.

We look at the `read_probe` implementation of flac:

```
static int flac_probe (AVProbeData * p)
{
    if (p->buf_size <4 || memcmp (p->buf, "fLaC", 4))
        return 0;
    return AVPROBE_SCORE_EXTENSION;
}
```

This probe just judges whether the header is the fLaC character, it is ok, I recognize it, no, let `avformat_open_input` continue to match the file extension (if any). If all probe functions are not recognized, open fails. If there is a probe recognized, in theory, `read_header` should be executed in `av_find_stream_info`, but ffmpeg still put it in `avformat_open_input`, in fact, the effect is the same,

let's take a look at the read_header of flac:

```
static int flac_read_header (AVFormatContext * s)
{
    int ret, metadata_last = 0, metadata_type, metadata_size,
found_streaminfo = 0;
    uint8_t header [4];
    uint8_t * buffer = NULL;
    AVStream * st = avformat_new_stream (s, NULL);
    if (! st)
        return AVERror (ENOMEM);
    st-> codec-> codec_type = AVMEDIA_TYPE_AUDIO;
    st-> codec-> codec_id = AV_CODEC_ID_FLAC;
    st-> need_parsing = AVSTREAM_PARSE_FULL_RAW;
/* the parameters will be extracted from the compressed bitstream */

/* if fLaC marker is not found, assume there is no header */
if (avio_rl32 (s-> pb) != MKTAG ('f', 'L', 'a', 'C')) {
    avio_seek (s-> pb, -4, SEEK_CUR);
    return 0;
}

/* process metadata blocks */
while (! url_feof (s-> pb) && ! metadata_last) {
    avio_read (s-> pb, header, 4);
    avpriv_flac_parse_block_header (header, & metadata_last, &
metadata_type,
                                    & metadata_size);
    switch (metadata_type) {
/* allocate and read metadata block for supported types */
    case FLAC_METADATA_TYPE_STREAMINFO:
    case FLAC_METADATA_TYPE_CUESHEET:
    case FLAC_METADATA_TYPE_PICTURE:
    case FLAC_METADATA_TYPE_VORBIS_COMMENT:
        buffer = av_mallocz (metadata_size +
FF_INPUT_BUFFER_PADDING_SIZE);
        if (! buffer) {

```

```

        return AVERROR (ENOMEM);
    }
    if (avio_read (s-> pb, buffer, metadata_size)! = metadata_size) {
        RETURN_ERROR (AVERROR (EIO));
    }
    break;
/* skip metadata block for unsupported types */
default :
    ret = avio_skip (s-> pb, metadata_size);
    if (ret <0)
        return ret;
}

if (metadata_type ==
FLAC_METADATA_TYPE_STREAMINFO) {
    FLACStreaminfo si;
    /* STREAMINFO can only occur once */
    if (found_streaminfo) {
        RETURN_ERROR (AVERROR_INVALIDDATA);
    }
    if (metadata_size! = FLAC_STREAMINFO_SIZE) {
        RETURN_ERROR (AVERROR_INVALIDDATA);
    }
    found_streaminfo = 1;
    st-> codec-> extradata = buffer;
    st-> codec-> extradata_size = metadata_size;
    buffer = NULL;

    /* get codec params from STREAMINFO header */
    avpriv_flac_parse_streaminfo (st-> codec, & si, st-> codec->
extradata);

    /* set time base and duration */
    if (si.samplerate> 0) {
        avpriv_set_pts_info (st, 64, 1, si.samplerate);
        if (si.samples> 0)
            st-> duration = si.samples;
    }
}

```

```

    }

else if (metadata_type == FLAC_METADATA_TYPE_CUESHEET) {
    uint8_t isrc [13];
    uint64_t start;
    const uint8_t * offset;
    int i, chapters, track, ti;
    if (metadata_size <431)
        RETURN_ERROR (AVERROR_INVALIDDATA);
    offset = buffer + 395;
    chapters = bytestream_get_byte (& offset)-1;
    if (chapters <= 0)
        RETURN_ERROR (AVERROR_INVALIDDATA);

    for (i = 0; i <chapters; i++)
    {
        if (offset + 36-buffer> metadata_size)
            RETURN_ERROR (AVERROR_INVALIDDATA);
        start = bytestream_get_be64 (& offset);
        track = bytestream_get_byte (& offset);
        bytestream_get_buffer (& offset, isrc, 12);
        isrc [12] = 0;
        offset += 14;
        ti = bytestream_get_byte (& offset);
        if (ti <= 0) RETURN_ERROR
(AVERROR_INVALIDDATA);
        offset += ti * 12;
        avpriv_new_chapter (s, track, st-> time_base, start,
AV_NOPTS_VALUE, isrc);
    }
    av_freep (& buffer);
}

else if (metadata_type == FLAC_METADATA_TYPE_PICTURE) {
    ret = ff_flac_parse_picture (s, buffer, metadata_size);
    av_freep (& buffer);
    if (ret <0) {

```

```

        av_log (s, AV_LOG_ERROR, "Error parsing attached
picture. \n" );
        return ret;
    }
} else {
    /* STREAMINFO must be the first block */
    if (! found_streaminfo) {
        RETURN_ERROR (AVERROR_INVALIDDATA);
    }
    /* process supported blocks other than STREAMINFO */
    if (metadata_type ==
FLAC_METADATA_TYPE_VORBIS_COMMENT) {
        if (ff_vorbis_comment (s, & s-> metadata, buffer,
metadata_size)) {
            av_log (s, AV_LOG_WARNING, "error parsing VorbisComment
metadata \n" );
            }
        }
        av_freep (& buffer);
    }
}
return 0;
fail:
av_free (buffer);
return ret;
}

```

First, we see that `avformat_new_stream` has a stream, because flac audio has only one stream. Then we can see that it sets the type and id of the codec, just like we set `MajorType` and `SubType` in DShow. We see the following line of code:

```
st-> need_parsing =
```

`AVSTREAM_PARSE_FULL_RAW;`

This is the flag that indicates the parser (packet parser), described later. Then come into the process of traversing the metadata block of flac. Because we are not analyzing the flac format, I will not say how the metadata block of flac is. We only say what the structure of AVFormatContext becomes after `read_header`. The flow information of the flac file is stored in the

`FLAC_METADATA_TYPE_STREAMINFO` block.

We see that when running the

`FLAC_METADATA_TYPE_STREAMINFO` block:

```
if (metadata_type == FLAC_METADATA_TYPE_STREAMINFO) {
    FLACStreaminfo si;
    /* STREAMINFO can only occur once */
    if (found_streaminfo) {
        RETURN_ERROR(AVERROR_INVALIDDATA);
    }
    if (metadata_size != FLAC_STREAMINFO_SIZE) {
        RETURN_ERROR(AVERROR_INVALIDDATA);
    }
    found_streaminfo = 1;
    st-> codec-> extradata = buffer;
    st-> codec-> extradata_size = metadata_size;
    buffer = NULL;
    /* get codec params from STREAMINFO header */
    avpriv_flac_parse_streaminfo(st-> codec, & si, st-> codec->
extradata);

    /* set time base and duration */
    if (si.samplerate > 0) {
```

```

    avpriv_set_pts_info (st, 64, 1, si.samplerate);
    if (si.samples > 0)
        st-> duration = si.samples;
}

```

First, codec->extradata has data (just like the H264 decoder requires SPS \ PPS \ NAL header).

Then in the code, we can see that the total length of the stream is available, and the sampling rate of the stream is available. Let us look at the `avpriv_flac_parse_streaminfo` function:

```

void avpriv_flac_parse_streaminfo (AVCodecContext * avctx, struct
FLACStreaminfo * s, const uint8_t * buffer)
{
    GetBitContext gb;
    init_get_bits (& gb, buffer, FLAC_STREAMINFO_SIZE * 8);

    skip_bits (& gb, 16); /* skip min blocksize */
    s-> max_blocksize = get_bits (& gb, 16);
    if (s-> max_blocksize < FLAC_MIN_BLOCKSIZE) {
        av_log (avctx, AV_LOG_WARNING, "invalid max
blocksize:% d \ n",
                s-> max_blocksize);
        s-> max_blocksize = 16;
    }

    skip_bits (& gb, 24); /* skip min frame size */
    s-> max_framesize = get_bits_long (& gb, 24);

    s-> samplerate = get_bits_long (& gb, 20);
    s-> channels = get_bits (& gb, 3) + 1;
    s-> bps = get_bits (& gb, 5) + 1;

    avctx-> channels = s-> channels;
    avctx-> sample_rate = s-> samplerate;
}

```

```
    avctx->bits_per_raw_sample = s->bps;
    ff_flac_set_channel_layout(avctx);

    s->samples = get_bits64(&gb, 36);

    skip_bits_long(&gb, 64); /* md5 sum */
    skip_bits_long(&gb, 64); /* md5 sum */
}
```

It can be seen that everything needed to decode an audio is available, and the channels, sample rate, rate, channel layout, and total number of samples have been saved.

All of this information is set in AVCodecContext. From this, we can conclude that `read_header` is a function to extract the private information needed by the decoder. After `read_header` is completed, basically `avformat_open_input` will return. At this time, although `AVCodecContext` has some information, `AVFormatContext` is quite empty (at this time it already has the total number of streams), we need to fill it, it should be time to call `av_find_stream_info`.

`av_find_stream_info` mainly does several things:

1. Fill in `AVFormatContext` information, such as total length, timebase, pts, dts and some internal information.
2. Some information needed to initialize the decoder.
3. If necessary, initialize the parser.

4. If necessary, try to decode some data to obtain media information.

Here we ignore the first two points, because they have little to do with decapsulation, and the third point is related to decapsulation.

Let us first assume that our `av_find_stream_info` succeeded. At this point we can `av_read_frame`.

We know that the AVFrame read out by `av_read_frame` is a frame of video (in IPB) if it is video, and a few frames if it is audio. But `av_read_frame` will not have broken frames, that is, there will be no 0.5 frames. How exactly does this work? Let's see.

First we return to the above:

```
AVInputFormat ff_flac_demuxer = {
    .name = "flac",
    .long_name = NULL_IF_CONFIG_SMALL( "raw FLAC" ),
    .read_probe = flac_probe,
    .read_header = flac_read_header,
    .read_packet = ff_raw_read_partial_packet,
    .flags = AVFMT_GENERIC_INDEX,
    .extensions = "flac",
    .raw_codec_id = AV_CODEC_ID_FLAC,
};
```

We read `read_probe` for judgment and `read_header` for parsing the header. There is another key thing that we

haven't read, that is `read_packet`. We can see that the `read_packet` here in `flac` points to a common function `ff_raw_read_partial_packet`, let's take a look at this function:

```
#define RAW_PACKET_SIZE 1024
int ff_raw_read_partial_packet (AVFormatContext * s, AVPacket * pkt)
{
    int ret, size;
    size = RAW_PACKET_SIZE;
    if (av_new_packet (pkt, size) <0)
        return AVERRORE (ENOMEM);

    pkt-> pos = avio_tell (s-> pb);
    pkt-> stream_index = 0;
    ret = ffio_read_partial (s-> pb, pkt-> data, size);
    if (ret <0) {
        av_free_packet (pkt);
        return ret;
    }
    av_shrink_packet (pkt, ret);
    return ret;
}
```

This function is really simple, and the work it pays attention to is:

1. Create a new 1024 byte `AVPacket`.
2. Read 1024 bytes of data from IO (if there is no 1024 bytes from the current pointer to the end of the file, the difference is read).
3. Write the actually read data back to the packet size

(av_shrink_packet).

We can see that this thing reads 1024 bytes every time, but even if it is an audio file, it is impossible to say that the offset of all frames in the file is aligned to 1024 bytes, and it is impossible to say that a frame size must be It is 1024 bytes, and for video files, the read 1024 bytes of data will be interspersed with video \ audio frames, so this AVPacket will definitely not be such a beautiful AVPakcet that we call av_read_frame and must be "trimmed" Yes, where is this trimming module? That is the parser.

Let's look at the parc structure of flac:

```
AVCodecParser ff_flac_parser = {  
    .codec_ids = {AV_CODEC_ID_FLAC},  
    .priv_data_size = sizeof(FLACParseContext),  
    .parser_init = flac_parse_init,  
    .parser_parse = flac_parse,  
    .parser_close = flac_parse_close,  
};
```

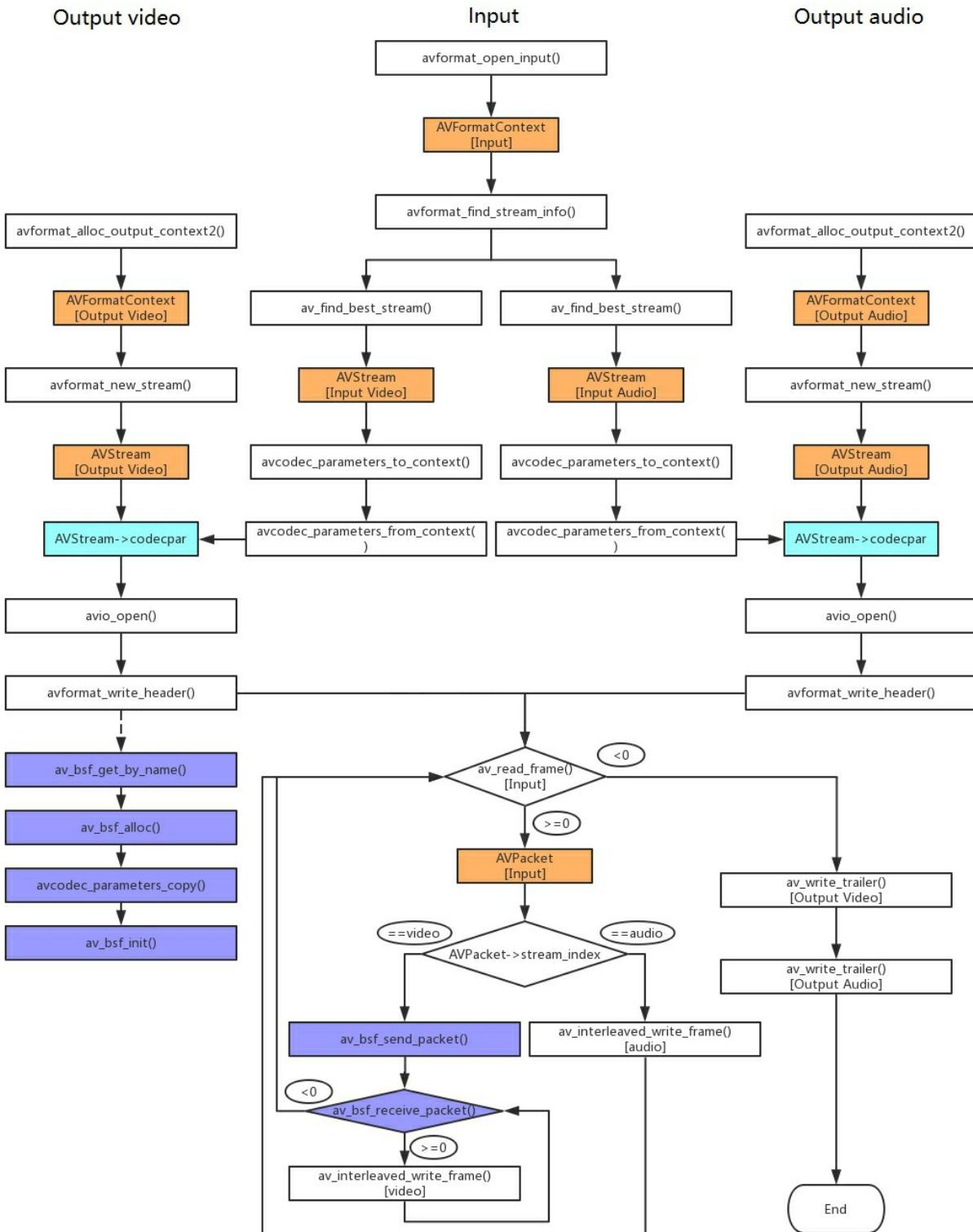
We can see that both demuxer and parser are associated with the same codec id, ie AV_CODEC_ID_FLAC, indicating that they are connected together. The parser structure is initialized by av_find_stream_info, and av_find_stream_info will execute av_parser_init to initialize the parser for each

stream through codec_id. The search method of av_parser_init is also the same as searching for demuxer. The parser has been registered in the global table, and the stopwatch can be matched.

DEMULTIPLEXING TO H264 AND MP3

Based on ffmpeg audio and video demultiplexer, the packaged mp4 file is decomposed into h264 naked stream and mp3.

Flow chart:



Note: In the flowchart, the white box is the specific ffmpeg api, the orange box is the structure that will be

generated after calling the api, and the purple box is the bit stream filter that h264 needs to use. See the white box for the specific process.

A brief function at each function:

1. `avformat_open_input ()`: open stream input
 2. `avformat_find_stream_info ()`: Find information flow, filling the `AVFormatContext`
 3. `av_find_best_stream ()`: input stream to find the desired flow and return flow number of
 4. `avformat_alloc_output_context2 ()`: Initialization Output Stream `AVFormatContext`
 5. `avformat_new_stream ()`: create a stream
 6. `avcodec_parameters_to_context ()`: copy the input stream `AVStream-> codecpar` structure to `AVCodecContext`
 7. `avcodec_parameters_from_context ()`: copy `AVCodecContext` structure to the output stream `AVStream-> codecpar` structure
 8. `avio_open ()`: open Output file
 9. `avformat_write_header ()`: write file header
 10. `av_interleaved_write_frame ()`: write `AVPacket` packets to the stream
 11. `av_write_trailer ()`: write file tail
-

```
#include <stdio.h>
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>

typedef struct {
    AVFormatContext *fmt_ctx;
    AVStream *stream;
} MyFmtContext;

static int bit_stream_filter_deal(AVBSFContext *bsf_ctx, AVPacket *pkt,
AVFormatContext *fmt_ctx)
{
    int ret = 0;

    if ((ret = av_bsf_send_packet(bsf_ctx, pkt)) < 0)
    {
        fprintf(stderr, "av_bsf_send_packet failed.\n");
        return -1;
    }

    while ((ret = av_bsf_receive_packet(bsf_ctx, pkt)) >= 0)
    {
        if (av_interleaved_write_frame(fmt_ctx, pkt) < 0)
        {
            fprintf(stderr, "av_interleaved_write_frame for video failed.\n");
            return -1;
        }
    }

    if (ret != AVERROR(EAGAIN) && ret != AVERROR_EOF)
    {
        fprintf(stderr, "av_bsf_receive_packet failed.\n");
        return -1;
    }
    return 0;
}

static int bit_stream_filter_init(const char *bit_stream_filter_name,
```

```
AVBSFContext **bsf_ctx, AVStream *stream)
{
    int ret = 0;
    const AVBitStreamFilter *bs_filter = NULL;

    if ((bs_filter = av_bsf_get_by_name(bit_stream_filter_name)) ==
NULL)
    {
        fprintf(stderr, "av_bsf_get_by_name failed.\n");
        return -1;
    }

    if ((ret = av_bsf_alloc(bs_filter, bsf_ctx)) < 0)
    {
        fprintf(stderr, "av_bsf_alloc failed.\n");
        return -1;
    }

    if ((ret = avcodec_parameters_copy((*bsf_ctx)->par_in, stream-
>codecpar)) < 0)
    {
        fprintf(stderr, "avcodec_parameters_copy failed.\n");
        return -1;
    }
    (*bsf_ctx)->time_base_in = stream->time_base;

    if ((ret = av_bsf_init(*bsf_ctx)) < 0)
    {
        fprintf(stderr, "av_bsf_init failed.\n");
        return -1;
    }

    return 0;
}

static int create_output_fmt_ctx(MyFmtContext *mfmt_ctx, AVStream
*stream_in, const char *filename)
{
    int ret = 0;
```

```
AVStream *out_stream = NULL;
AVCodecContext * cdc_ctx = NULL;

    if ((ret = avformat_alloc_output_context2(&mfmt_ctx->fmt_ctx,
NULL, NULL, filename)) < 0)
{
    fprintf(stderr, "avformat_alloc_output_context2 for %s failed.\n",
filename);
    goto ret1;
}

    if ((out_stream = avformat_new_stream(mfmt_ctx->fmt_ctx,
NULL)) == NULL)
{
    fprintf(stderr, "avformat_new_stream for %s \n", filename);
    goto ret2;
}

    if ((cdc_ctx = avcodec_alloc_context3(NULL)) == NULL)
{
    fprintf(stderr, "ready copy failed.\n");
    goto ret2;
}

    if ((ret = avcodec_parameters_to_context(cdc_ctx, stream_in-
>codecpar)) < 0)
{
    fprintf(stderr, "avcodec_parameters_to_context for %s failed.\n",
filename);
    goto ret3;
}
    cdc_ctx->codec_tag = 0;
    if (mfmt_ctx->fmt_ctx->oformat->flags &
AVFMT_GLOBALHEADER)
        cdc_ctx->flags |= AV_CODEC_FLAG_GLOBAL_HEADER;
    if ((ret = avcodec_parameters_from_context(out_stream->codecpar,
cdc_ctx)) < 0)
{
```

```

        fprintf(stderr, "avcodec_parameters_from_context for %s failed.\n",
filename);
    goto ret3;
}

mfmt_ctx->stream = out_stream;

avcodec_free_context(&cdc_ctx);
return 0;
ret3:
avcodec_free_context(&cdc_ctx);
ret2:
avformat_free_context(mfmt_ctx->fmt_ctx);
ret1:
return -1;
}

void demuxer_simple(const char *input_file, const char *output_video,
const char *output_audio)
{
    int ret;
    AVFormatContext *fmt_ctx = NULL;
    MyFmtContext fmt_ctx_v = {0};
    MyFmtContext fmt_ctx_a = {0};
    AVPacket *pkt = NULL;
    int video_stream_index = -1, audio_stream_index = -1;
    AVStream *input_stream_v = NULL, *input_stream_a = NULL;
    AVBSFContext *bsf_ctx = NULL;

    /*Open the input file (.mp4) and find the video stream and audio
stream*/
    if ((ret = avformat_open_input(&fmt_ctx, input_file, NULL, NULL)) < 0)
    {
        fprintf(stderr, "avformat_open_input failed.\n");
        goto ret1;
    }

    if ((ret = avformat_find_stream_info(fmt_ctx, NULL)) < 0)

```

```
{  
    fprintf(stderr, "avformat_find_stream_info failed.\n");  
    goto ret2;  
}  
  
if ((ret = av_find_best_stream(fmt_ctx, AVMEDIA_TYPE_VIDEO, -1,  
-1, NULL, 0)) < 0)  
{  
    fprintf(stderr, "avformat_find_best_stream for video failed.\n");  
    goto ret2;  
}  
video_stream_index = ret;  
input_stream_v = fmt_ctx->streams[ret];  
if ((ret = av_find_best_stream(fmt_ctx, AVMEDIA_TYPE_AUDIO, -1,  
-1, NULL, 0)) < 0)  
{  
    fprintf(stderr, "avformat_find_best_stream for audio failed.\n");  
    goto ret2;  
}  
audio_stream_index = ret;  
input_stream_a = fmt_ctx->streams[ret];  
  
/*Create output file video (.h264)*/  
if ((ret = create_output_fmt_ctx(&fmt_ctx_v, input_stream_v,  
output_video)) < 0)  
{  
    fprintf(stderr, "create_output_fmt_ctx for video failed.\n");  
    goto ret2;  
}  
  
/*Create output file audio (.aac / .mp3)*/  
if ((ret = create_output_fmt_ctx(&fmt_ctx_a, input_stream_a,  
output_audio)) < 0)  
{  
    fprintf(stderr, "create_output_fmt_ctx for audio failed.\n");  
    goto ret2;  
}
```

```
if ((pkt = av_packet_alloc()) == NULL)
{
    fprintf(stderr, "av_packet_alloc failed.\n");
    goto ret2;
}

/*Initialize bit stream filter, h264 required*/
if ((ret = bit_stream_filter_init("h264_mp4toannexb", &bsf_ctx,
input_stream_v)) < 0)
{
    fprintf(stderr, "bit_stream_filter_init failed.\n");
    goto ret3;
}

if (!(fmt_ctx_v.fmt_ctx->oformat->flags & AVFMT_NOFILE))
{
    if (avio_open(&fmt_ctx_v.fmt_ctx->pb, output_video,
AVIO_FLAG_WRITE) < 0)
    {
        fprintf(stderr, "avio_open video failed.\n");
        goto ret3;
    }
}

if (!(fmt_ctx_a.fmt_ctx->oformat->flags & AVFMT_NOFILE))
{
    if (avio_open(&fmt_ctx_a.fmt_ctx->pb, output_audio,
AVIO_FLAG_WRITE) < 0)
    {
        fprintf(stderr, "avio_open audio failed.\n");
        goto ret4;
    }
}

/*Write file header*/
if (avformat_write_header(fmt_ctx_v.fmt_ctx, NULL) < 0)
{
    fprintf(stderr, "avformat_write_header for video failed.\n");
```

```
    goto ret5;
}

if (avformat_write_header(fmt_ctx_a.fmt_ctx, NULL) < 0)
{
fprintf(stderr, "avformat_write_header for audio failed.\n");
goto ret5;
}

/*demuxer*/
while (av_read_frame(fmt_ctx, pkt) >= 0)
{
if (pkt->size > 0)
{
if (pkt->stream_index == video_stream_index)
{
pkt->stream_index = fmt_ctx_v.stream->index;

if (bit_stream_filter_deal(bsf_ctx, pkt, fmt_ctx_v.fmt_ctx) < 0)
goto ret5;
}
else if (pkt->stream_index == audio_stream_index)
{
pkt->stream_index = fmt_ctx_a.stream->index;

if ((ret = av_interleaved_write_frame(fmt_ctx_a.fmt_ctx, pkt)) < 0)
{
fprintf(stderr, "av_interleaved_write_frame for audio failed.\n");
fprintf(stderr, "av_interleaved_write_frame for audio failed:%s.\n",
av_err2str(ret));
goto ret5;
}
}
av_packet_unref(pkt);
}
}

av_write_trailer(fmt_ctx_v.fmt_ctx);
av_write_trailer(fmt_ctx_a.fmt_ctx);
```

```

av_bsfree(&bsf_ctx);
av_packet_free(&pkt);
if (!(fmt_ctx_a.fmt_ctx->oformat->flags & AVFMT_NOFILE))
    avio_close(fmt_ctx_a.fmt_ctx->pb);
if (!(fmt_ctx_v.fmt_ctx->oformat->flags & AVFMT_NOFILE))
    avio_close(fmt_ctx_v.fmt_ctx->pb);
    avformat_free_context(fmt_ctx_v.fmt_ctx);
    avformat_free_context(fmt_ctx_a.fmt_ctx);
    avformat_close_input(&fmt_ctx);
    return;
ret5:
if (!(fmt_ctx_a.fmt_ctx->oformat->flags & AVFMT_NOFILE))
    avio_close(fmt_ctx_a.fmt_ctx->pb);
ret4:
if (!(fmt_ctx_v.fmt_ctx->oformat->flags & AVFMT_NOFILE))
    avio_close(fmt_ctx_v.fmt_ctx->pb);
ret3:
av_packet_free(&pkt);
ret2:
avformat_close_input(&fmt_ctx);
ret1:
return;
}

int main(int argc, const char *argv[])
{
    if (argc < 4)
    {
        fprintf(stderr, "Usage: <input file> <output file video> <output file
audio>\n");
        return -1;
    }

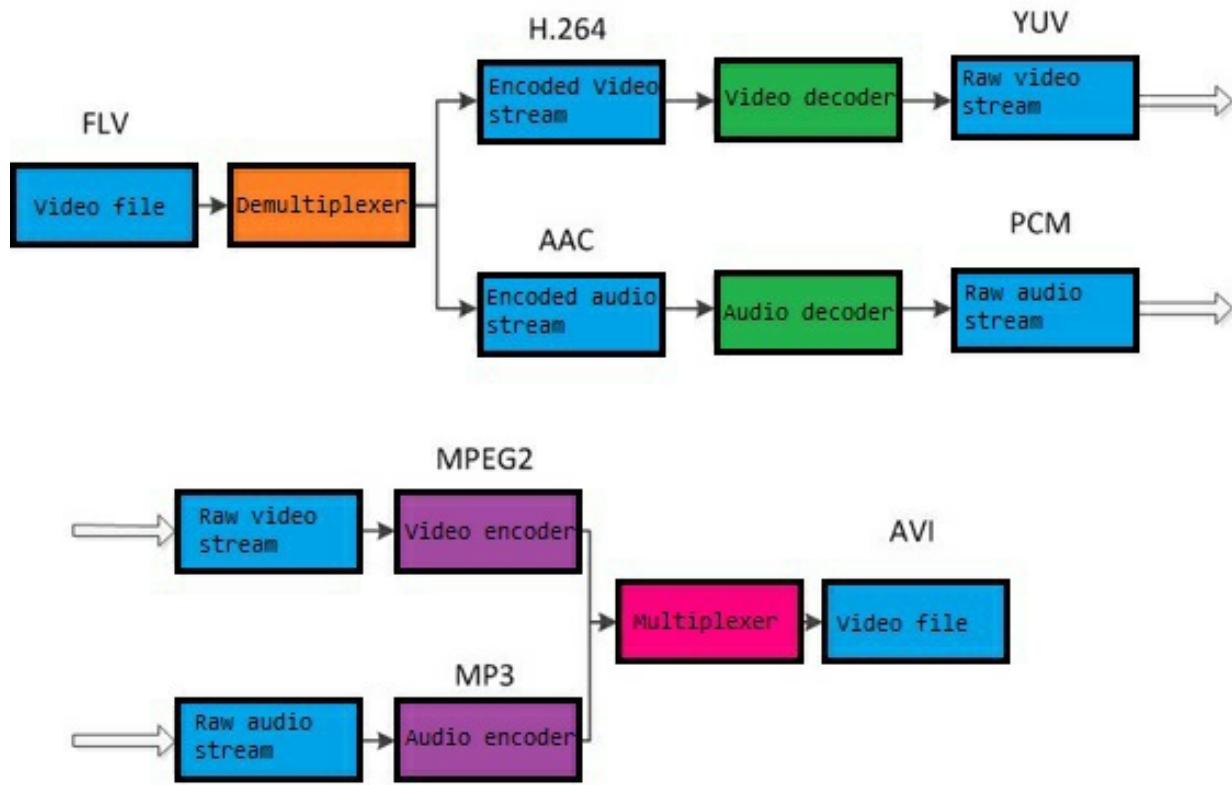
    demuxer_simple(argv[1], argv[2], argv[3]);
    return 0;
}

```

After reading a frame from the input stream, you need to change the stream number to the number of the output stream. For example: generally in mp4 files, the video stream is number 0, the audio stream is number 1, and h264 and mp3 have only one stream, the number is 0. The h264 video stream needs to use the bit stream filter named "h264_mp4toannexb", otherwise, the decomposed video cannot be played.

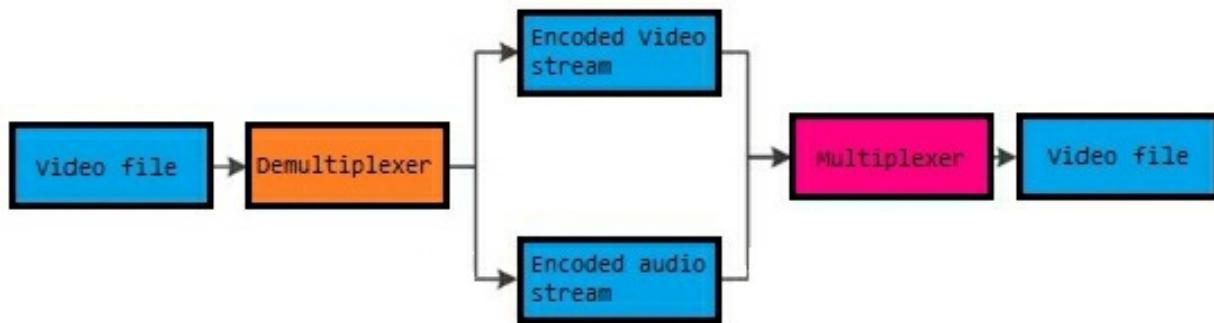
11. REMULTIPLEXING

This sample introduces a FFMPEG-based packaging format converter. The so-called package format conversion is to convert between AVI, FLV, MKV, MP4 (corresponding to .avi, .flv, .mkv, .mp4 files). It should be noted that this program does not perform video and audio encoding and decoding work. Instead, the video and audio compressed code stream is directly obtained from one package format file and then packaged into another package format file. The working principle of the traditional transcoding program is shown below:



The above picture shows an example: FLV (video: H.264, audio: AAC) transcoding to AVI (video: MPEG2, audio MP3). It can be seen that the process of video transcoding is generally equivalent to re-recording video and audio.

The working principle of this program is shown below:



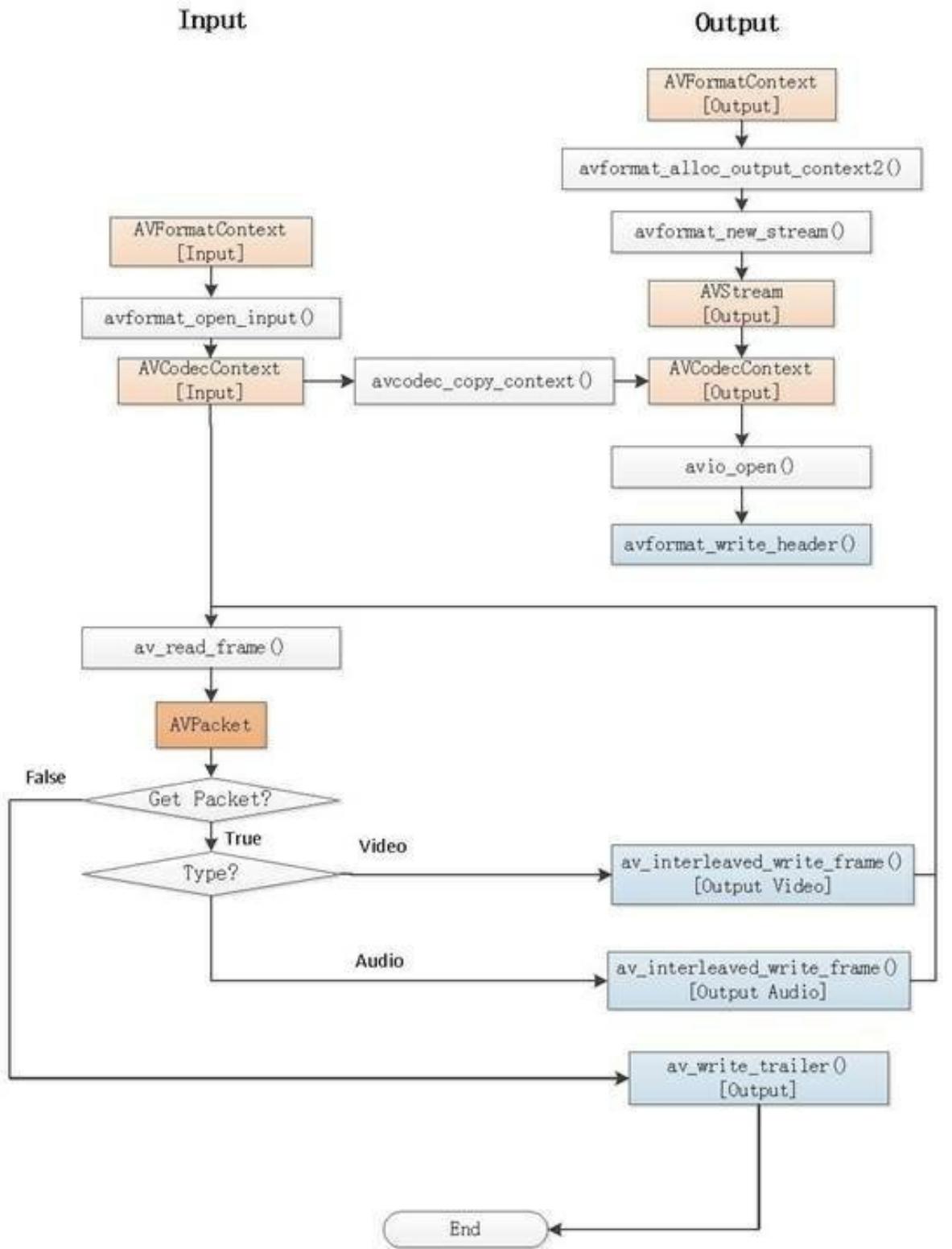
This program does not perform video and audio coding and decoding. Therefore, compared with ordinary transcoding software, this program has the following two characteristics:

Extremely fast processing speed. The video and audio encoding and decoding algorithms are very complex and occupy most of the time for transcoding. Because there is no need to encode and decode video and audio, a lot of time is saved.

Video and audio quality is lossless. Because there is no need to encode and decode video and audio, there will be no video or audio compression damage.

The flowchart of Remuxer based on FFmpeg is attached below. The key data structure is marked in light red in the figure, and the function of outputting video data is marked in light blue. It can be seen that a program contains the processing of two

files: reading the input file (located on the left) and writing the output file (located on the right). In the middle, an `avcodec_copy_context()` is used to copy the input `AVCodecContext` to the output `AVCodecContext`.



Briefly introduce the significance of key functions in

the process:

Input file operation:

1. `avformat_open_input()`: Open the input file and initialize the AVFormatContext of the input video stream.
2. `av_read_frame()`: Read an AVPacket from the input file.
3. Output file operation:
4. `avformat_alloc_output_context2()`: Initialize the AVFormatContext of the output video stream.
5. `avformat_new_stream()`: Create AVStream of output stream.
6. `avcodec_copy_context()`: copy the value t of the AVCodecContex of the input video stream to the AVCodecContext of the output video.
7. `avio_open()`: Open the output file.
8. `avformat_write_header()`: Write the file header (for some encapsulation formats without a file header, this function is not needed. For example, MPEG2TS).
9. `av_interleaved_write_frame()`: Write AVPacket (storage video compression code stream data) to a file.
10. `av_write_trailer()`: Write the end of the file

(for some encapsulation formats without a file header, this function is not required. For example, MPEG2TS).

```
#include "stdafx.h"
extern "C"
{
#include "libavformat/avformat.h"
};

int _tmain(int argc, _TCHAR* argv[])
{
    AVOutputFormat *ofmt = NULL;
    //Input AVFormatContext and Output AVFormatContext
    AVFormatContext *ifmt_ctx = NULL, *ofmt_ctx = NULL;
    AVPacket pkt;
    const char *in_filename, *out_filename;
    int ret, i;
    if (argc < 3) {
        printf("usage: %s input output\n"
               "Remux a media file with libavformat and libavcodec.\n"
               "The output format is guessed according to the file extension.\n"
               , argv[0]);
        return 1;
    }
    in_filename = argv[1];//Input file URL
    out_filename = argv[2];//Output file URL
    av_register_all();
    //Input
    if ((ret = avformat_open_input(&ifmt_ctx, in_filename, 0, 0)) < 0) {
        printf( "Could not open input file.");
        goto end;
    }
    if ((ret = avformat_find_stream_info(ifmt_ctx, 0)) < 0) {
        printf( "Failed to retrieve input stream information");
        goto end;
    }
```

```

}

av_dump_format(ifmt_ctx, 0, in_filename, 0);
//Output
avformat_alloc_output_context2(&ofmt_ctx, NULL, NULL,
out_filename);
if (!ofmt_ctx) {
    printf( "Could not create output context\n");
    ret = AERROR_UNKNOWN;
    goto end;
}
ofmt = ofmt_ctx->oformat;
for (i = 0; i < ifmt_ctx->nb_streams; i++) {
    //Create output AVStream according to input AVStream
    AVStream *in_stream = ifmt_ctx->streams[i];
    AVStream *out_stream = avformat_new_stream(ofmt_ctx,
in_stream->codec->codec);
    if (!out_stream) {
        printf( "Failed allocating output stream\n");
        ret = AERROR_UNKNOWN;
        goto end;
    }
    //Copy the settings of AVCodecContext
    ret = avcodec_copy_context(out_stream->codec, in_stream-
>codec);
    if (ret < 0) {
        printf( "Failed to copy context from input to output stream codec
context\n");
        goto end;
    }
    out_stream->codec->codec_tag = 0;
    if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
        out_stream->codec->flags |=
CODEC_FLAG_GLOBAL_HEADER;
}

av_dump_format(ofmt_ctx, 0, out_filename, 1);
//Open output file

```

```

if (!(ofmt->flags & AVFMT_NOFILE)) {
    ret = avio_open(&ofmt_ctx->pb, out_filename,
AVIO_FLAG_WRITE);
    if (ret < 0) {
        printf( "Could not open output file '%s'", out_filename);
        goto end;
    }
}
//Write file header
ret = avformat_write_header(ofmt_ctx, NULL);
if (ret < 0) {
    printf( "Error occurred when opening output file\n");
    goto end;
}
int frame_index=0;
while (1) {
    AVStream *in_stream, *out_stream;
    //Get an AVPacket
    ret = av_read_frame(ifmt_ctx, &pkt);
    if (ret < 0)
        break;
    in_stream = ifmt_ctx->streams[pkt.stream_index];
    out_stream = ofmt_ctx->streams[pkt.stream_index];
    /* copy packet */
    //Convert PTS/DTS
    pkt.pts = av_rescale_q_rnd(pkt.pts, in_stream->time_base,
out_stream->time_base, (AVRounding)
(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.dts = av_rescale_q_rnd(pkt.dts, in_stream->time_base,
out_stream->time_base, (AVRounding)
(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.duration = av_rescale_q(pkt.duration, in_stream->time_base,
out_stream->time_base);
    pkt.pos = -1;
    //Write)
    ret = av_interleaved_write_frame(ofmt_ctx, &pkt);
    if (ret < 0) {

```

```
    printf( "Error muxing packet\n");
    break;
}
printf("Write %8d frames to output file\n",frame_index);
av_free_packet(&pkt);
frame_index++;
}
//Write file trailer
av_write_trailer(ofmt_ctx);
end:
avformat_close_input(&ifmt_ctx);
/* close output */
if (ofmt_ctx && !(ofmt->flags & AVFMT_NOFILE))
    avio_close(ofmt_ctx->pb);
avformat_free_context(ofmt_ctx);
if (ret < 0 && ret != AVERROR_EOF) {
    printf( "Error occurred.\n");
    return -1;
}
return 0;
}
```

12. TRANSCODING

The big process can be divided into four blocks: input, output, transcoding, and playback. Among them, transcoding involves more processing links. It can be seen from the figure that the transcoding function accounts for a large proportion in the entire functional diagram. The core functions of transcoding are decoding and encoding, but in an available sample program, encoding and decoding are difficult to separate from input and output. The demultiplexer provides input to the decoder, and the decoder will output the original frame. Various complex filter processing can be performed on the original frame. The frame after the filter processing is generated by the encoder to generate encoded frames. The multiplexer outputs to the output file.

WHOLE PROCESS

DEMULTIPLEXING

Read the encoded frames from the input file, determine the stream type, and send the encoded frames to the video decoder or audio decoder according to the stream type.

DECODING

Decode the video and audio coded frames to generate original frames. Details will be described in Decoding chapter.

FILTER

FFmpeg provides a variety of filters for processing raw frame data. In this example, an empty filter is used for each audio stream / video stream, that is, the buffer filter and the buffersink filter are directly connected in the filter diagram. The purpose is to convert the video stream output pixel format to the pixel format used by the encoder through the video buffersink filter; convert the audio stream output channel layout to the encoder's channel layout through the audio buffetsink filter. Prepare for the next encoding operation. If you do not use this method, you will need to deal with image format conversion and audio resampling to ensure that the frames entering the encoder are in a format supported by the encoder. Of course, the routine is extensible, you can easily insert other functional filters between the buffer filter and the buffersink filter, to achieve rich video and audio processing functions.

ENCODING

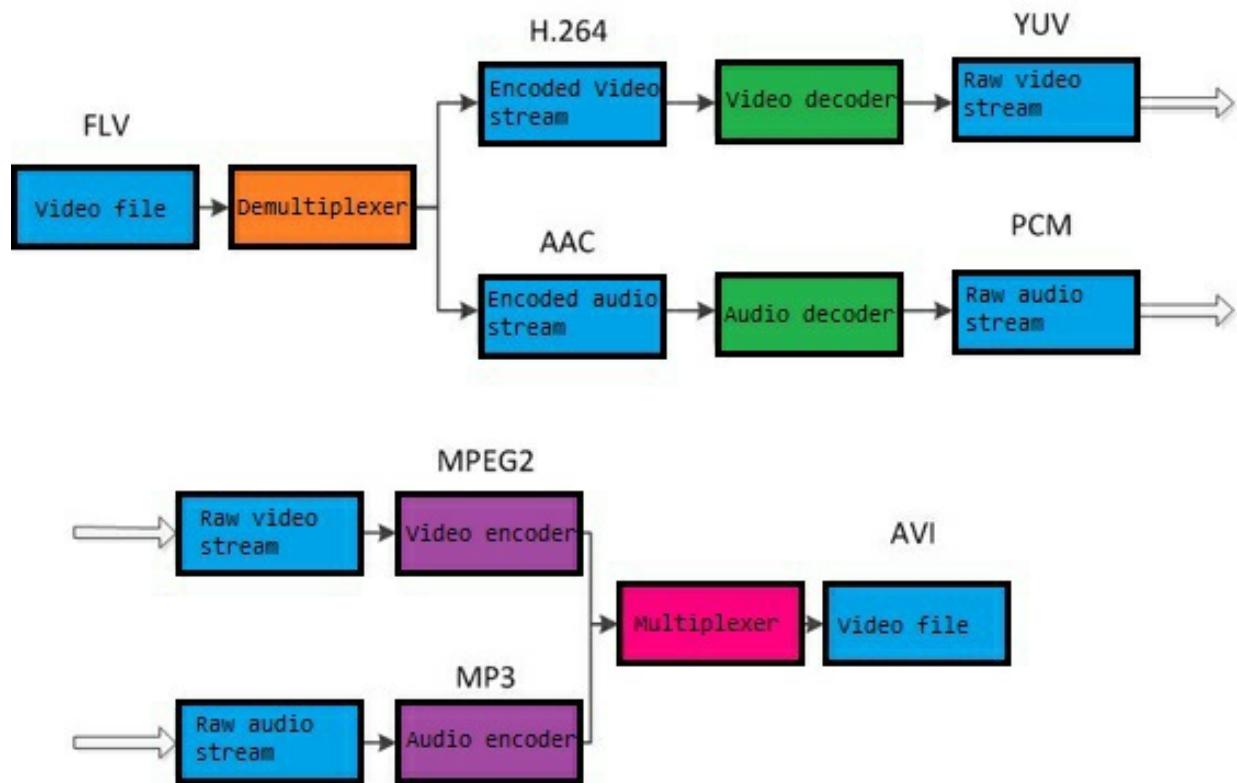
The original video and audio frames are encoded to generate encoded frames. Details will be described later.

REUSE

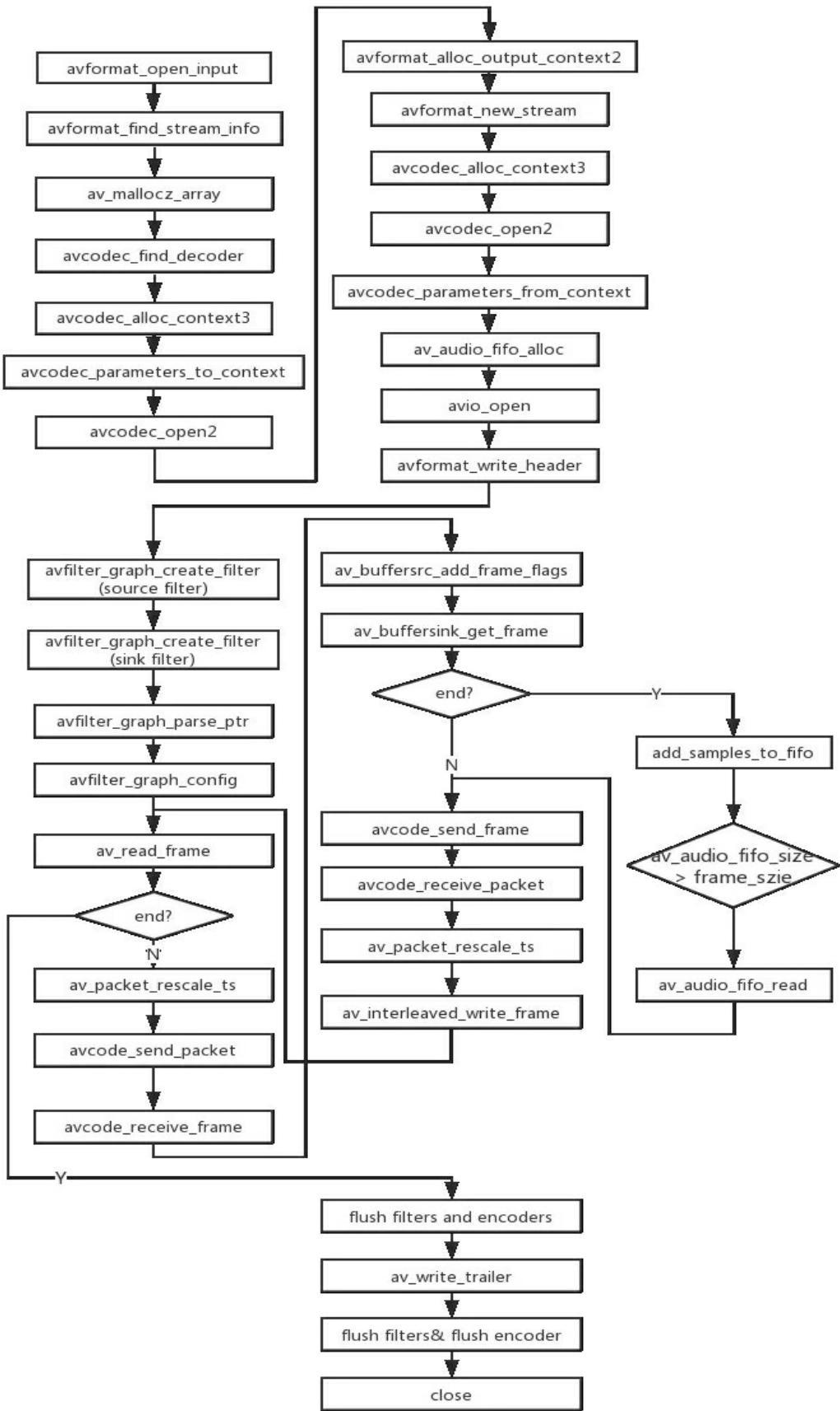
The encoded frames are interleaved into different output types and written to the output file.

```
av_interleaved_write_frame(octx.fmt_ctx, &ipacket);
```

The working principle of the traditional transcoding program is shown below:



An example flowchart is shown below.



TIME STAMP PROCESSING DURING TRANSCODING

In the package format processing routine, it does not matter if you do not understand the time stamp in depth. But in the codec processing routine, the time stamp processing is a very important detail, which must be clarified.

The time base (AVStream.time_base) in the container (file layer) is different from the time base (AVCodecContex.time_base) in the codec context (video layer). Time base conversion is required during decoding and encoding.

The video is played frame by frame, so the original video frame time base is 1 / framerate. Before decoding the video, you need to process the time parameters in the input AVPacket and convert the time base in the input container to 1 / framerate time base; after the video encoding, process the time parameters in the output AVPacket and convert the 1 / framerate time base to the output container Time base.

The audio is played at the sampling point, so the original audio frame time is $1 / \text{sample_rate}$. Before audio decoding, you need to process the time parameters in the input AVPacket and convert the time base in the input container to $1 / \text{sample_rate}$ time base; after the audio encoding, process the time parameters in the output AVPacket and convert the $1 / \text{sample_rate}$ time base to the output container Time base. If the audio FIFO is introduced, the timestamp information of the audio frames read from the FIFO will be lost. You need to use the $1 / \text{sample_rate}$ time base to regenerate pts for each audio frame before sending it to the encoder.

Time base conversion before decoding:

```
av_packet_rescale_ts(ipacket, sctx->i_stream->time_base, sctx->o_codec_ctx->time_base);
```

Encoded time base conversion:

```
av_packet_rescale_ts(&opacket, sctx->o_codec_ctx->time_base, sctx->o_stream->time_base);
```

Sample program from doc/examples/transcoding.c
subdirectory of “<https://ffmpeg.org/>”

```
/**  
* @file  
* API example for demuxing, decoding, filtering, encoding and muxing
```

```
* @example transcoding.c
*/
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavfilter/buffersink.h>
#include <libavfilter/buffersrc.h>
#include <libavutil/opt.h>
#include <libavutil/pixdesc.h>
#include <libavutil/audio_fifo.h>
#include <stdio.h>

static AVFormatContext *ifmt_ctx;
static AVFormatContext *ofmt_ctx;
typedef struct FilteringContext {
    AVFilterContext *buffersink_ctx;
    AVFilterContext *buffersrc_ctx;
    AVFilterGraph *filter_graph;
} FilteringContext;
static FilteringContext *filter_ctx;

typedef struct StreamContext {
    AVCodecContext *dec_ctx;
    AVCodecContext *enc_ctx;
    AVAudioFifo *fifo;
    int64_t pts_audio;
} StreamContext;
static StreamContext *stream_ctx;

static int open_input_file(const char *filename)
{
    int ret;
    unsigned int i;

    ifmt_ctx = NULL;
    if ((ret = avformat_open_input(&ifmt_ctx, filename, NULL, NULL)) < 0)
        av_log(NULL, AV_LOG_ERROR, "Cannot open input file\n");
    return ret;
}
```

```

if ((ret = avformat_find_stream_info(ifmt_ctx, NULL)) < 0) {
    av_log(NULL, AV_LOG_ERROR, "Cannot find stream information\n");
    return ret;
}

stream_ctx = av_mallocz_array(ifmt_ctx->nb_streams, sizeof(*stream_ctx));
if (!stream_ctx)
    return AVERRORE(ENOMEM);

for (i = 0; i < ifmt_ctx->nb_streams; i++) {
    AVStream *stream = ifmt_ctx->streams[i];
    AVCodec *dec = avcodec_find_decoder(stream->codecpar->codec_id);
    AVCodecContext *codec_ctx;
    if (!dec) {
        av_log(NULL, AV_LOG_ERROR, "Failed to find decoder for stream %u\n",
               i);
        return AVERRORE_DECODER_NOT_FOUND;
    }
    codec_ctx = avcodec_alloc_context3(dec);
    if (!codec_ctx) {
        av_log(NULL, AV_LOG_ERROR, "Failed to allocate the decoder context for stream #%"PRIu64"\n", i);
        return AVERRORE(ENOMEM);
    }
    ret = avcodec_parameters_to_context(codec_ctx, stream->codecpar);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Failed to copy decoder parameters to context for stream #%"PRIu64"\n", i);
        return ret;
    }
    /* Reencode video & audio and remux subtitles etc. */
    if (codec_ctx->codec_type == AVMEDIA_TYPE_VIDEO
        || codec_ctx->codec_type == AVMEDIA_TYPE_AUDIO)
        if (codec_ctx->codec_type == AVMEDIA_TYPE_VIDEO)
            codec_ctx->framerate = av_guess_frame_rate(ifmt_ctx, stream,
                                                       i);
    /* Open decoder */
    ret = avcodec_open2(codec_ctx, dec, NULL);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Failed to open decoder for stream %u\n",
               i);
        return AVERRORE(ENOMEM);
    }
}

```

```

        return ret;
    }
}
stream_ctx[i].dec_ctx = codec_ctx;
}

av_dump_format(ifmt_ctx, 0, filename, 0);
return 0;
}

static int open_output_file(const char *filename)
{
AVStream *out_stream;
AVStream *in_stream;
AVCodecContext *dec_ctx, *enc_ctx;
AVCodec *encoder;
int ret;
unsigned int i;

ofmt_ctx = NULL;
avformat_alloc_output_context2(&ofmt_ctx, NULL, NULL, filename);
if (!ofmt_ctx) {
    av_log(NULL, AV_LOG_ERROR, "Could not create output context\n");
    return AERROR_UNKNOWN;
}

for (i = 0; i < ifmt_ctx->nb_streams; i++) {
    out_stream = avformat_new_stream(ofmt_ctx, NULL);
    if (!out_stream) {
        av_log(NULL, AV_LOG_ERROR, "Failed allocating output stream\n");
        return AERROR_UNKNOWN;
    }

    in_stream = ifmt_ctx->streams[i];
    dec_ctx = stream_ctx[i].dec_ctx;

    if (dec_ctx->codec_type == AVMEDIA_TYPE_VIDEO
        || dec_ctx->codec_type == AVMEDIA_TYPE_AUDIO) {

```

```

/* in this example, we choose transcoding to same codec */
encoder = avcodec_find_encoder(dec_ctx->codec_id);
if (!encoder) {
    av_log(NULL, AV_LOG_FATAL, "Necessary encoder n
    return AVERROR_INVALIDDATA;
}
enc_ctx = avcodec_alloc_context3(encoder);
if (!enc_ctx) {
    av_log(NULL, AV_LOG_FATAL, "Failed to allocate the enc
    return AVERROR(ENOMEM);
}

/* In this example, we transcode to same properties (picture size
 * sample rate etc.). These properties can be changed for output
 * streams easily using filters */
if (dec_ctx->codec_type == AVMEDIA_TYPE_VIDEO) {
    enc_ctx->height = dec_ctx->height;
    enc_ctx->width = dec_ctx->width;
    enc_ctx->sample_aspect_ratio = dec_ctx->sample_aspect_ratio;
    /* take first format from list of supported formats */
    if (encoder->pix_fmts)
        enc_ctx->pix_fmt = encoder->pix_fmts[0];
    else
        enc_ctx->pix_fmt = dec_ctx->pix_fmt;
    /* video time_base can be set to whatever is handy and suitable */
    enc_ctx->time_base = av_inv_q(dec_ctx->framerate);
    av_log(NULL, AV_LOG_DEBUG, "enc_ctx->time_base
>time_base.num, enc_ctx->time_base.den);
} else {
    enc_ctx->sample_rate = dec_ctx->sample_rate;
    enc_ctx->channel_layout = dec_ctx->channel_layout;
    enc_ctx->channels = av_get_channel_layout_nb_channels(dec_ctx);
    /* take first format from list of supported formats */
    enc_ctx->sample_fmt = encoder->sample_fmts[0];
    enc_ctx->time_base = (AVRational){1, enc_ctx->sample_rate};
    av_log(NULL, AV_LOG_DEBUG, "enc_ctx->time_base
        "enc_ctx->sample_fmt=%s, dec_ctx->sample_f

```

```

        enc_ctx->time_base.num, enc_ctx->time_base.d
        av_get_sample_fmt_name(enc_ctx->sample_fmt);
        av_get_sample_fmt_name(dec_ctx->sample_fmt);
    }

    if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
        enc_ctx->flags |= AV_CODEC_FLAG_GLOBAL_HEADER;

    /* Third parameter can be used to pass settings to encoder */
    ret = avcodec_open2(enc_ctx, encoder, NULL);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot open video encoder for stream %d\n", i);
        return ret;
    }

    ret = avcodec_parameters_from_context(out_stream->codecpar);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Failed to copy encoder parameters for stream %d\n", i);
        return ret;
    }

    out_stream->time_base = enc_ctx->time_base;
    stream_ctx[i].enc_ctx = enc_ctx;
    stream_ctx[i].fifo = av_audio_fifo_alloc(enc_ctx->sample_fmt,
                                             enc_ctx->frame_size);
    stream_ctx[i].pts_audio = 0;

} else if (dec_ctx->codec_type == AVMEDIA_TYPE_UNKNOWN) {
    av_log(NULL, AV_LOG_FATAL, "Elementary stream #%"PRIu64" is not supported, cannot proceed\n", i);
    return AVERROR_INVALIDDATA;
} else {
    /* if this stream must be remuxed */
    ret = avcodec_parameters_copy(out_stream->codecpar, in_stream->codecpar);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Copying parameters failed for stream %d\n", i);
        return ret;
    }
}

```



```

}

if (dec_ctx->codec_type == AVMEDIA_TYPE_VIDEO) {
    buffersrc = avfilter_get_by_name("buffer");
    buffersink = avfilter_get_by_name("buffersink");
    if (!buffersrc || !buffersink) {
        av_log(NULL, AV_LOG_ERROR, "filtering source or sink element r
            ret = AVERRO
            goto end;
    }
    snprintf(args, sizeof(args),
        "video_size=%dx%d:pix_fmt=%d:time_base=%d/%d:pix
            dec_ctx->width, dec_ctx->height, dec_ctx->pix_fmt,
            dec_ctx->time_base.num, dec_ctx->time_base.den,
            dec_ctx->sample_aspect_ratio.num,
            dec_ctx->sample_aspect_ratio.den);

    ret = avfilter_graph_create_filter(&buffersrc_ctx, buffersrc, "in",
        args, NULL, filter_graph);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot create buffer sour
        goto end;
    }

    ret = avfilter_graph_create_filter(&buffersink_ctx, buffersink, "out",
        NULL, NULL, filter_graph);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot create buffer sink
        goto end;
    }

    ret = av_opt_set_bin(buffersink_ctx, "pix_fmts",
        (uint8_t*)&enc_ctx->pix_fmt, sizeof(enc_ctx->pix_fmt),
        AV_OPT_SEARCH_CHILDREN);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot set output pixel fo
        goto end;
    }
}

```

```

} else if (dec_ctx->codec_type == AVMEDIA_TYPE_AUDIO) {
    buffersrc = avfilter_get_by_name("abuffer");
    buffersink = avfilter_get_by_name("abuffersink");
    if (!buffersrc || !buffersink) {
        av_log(NULL, AV_LOG_ERROR, "filtering source or sink element");
        ret = AVERROR_UNKNOWN;
        goto end;
    }

    if (!dec_ctx->channel_layout)
        dec_ctx->channel_layout =
            av_get_default_channel_layout(dec_ctx->channels);
    snprintf(args,
             sizeof(args),
             "time_base=%d/%d:sample_rate=%d:sample_fmt=%s",
             dec_ctx->time_base.num, dec_ctx->time_base.den, dec_ctx-
                 av_get_sample_fmt_name(dec_ctx->sample_fmt),
                 dec_ctx->channel_layout);
    ret = avfilter_graph_create_filter(&buffersrc_ctx, buffersrc, "in",
                                      args, NULL, filter_graph);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot create audio buffer");
        goto end;
    }

    ret = avfilter_graph_create_filter(&buffersink_ctx, buffersink, "out",
                                      NULL, NULL, filter_graph);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot create audio buffer");
        goto end;
    }

    ret = av_opt_set_bin(buffersink_ctx, "sample_fmts",
                         (uint8_t*)&enc_ctx->sample_fmt, sizeof(enc_ctx->sample_
                           AV_OPT_SEARCH_CHILDREN));
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot set output sample");
        goto end;
    }
}

```

```

    ret = av_opt_set_bin(buffersink_ctx, "channel_layouts",
                         (uint8_t*)&enc_ctx->channel_layout,
                         sizeof(enc_ctx->channel_layout), AV_OPT_SEARCH_CHILDREN);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot set output channel layout");
        goto end;
    }

    ret = av_opt_set_bin(buffersink_ctx, "sample_rates",
                         (uint8_t*)&enc_ctx->sample_rate, sizeof(enc_ctx->sample_rate),
                         AV_OPT_SEARCH_CHILDREN);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot set output sample rate");
        goto end;
    }
} else {
    ret = AERROR_UNKNOWN;
    goto end;
}

/* Endpoints for the filter graph. */
outputs->name      = av_strdup("in");
outputs->filter_ctx = buffersrc_ctx;
outputs->pad_idx    = 0;
outputs->next       = NULL;

inputs->name      = av_strdup("out");
inputs->filter_ctx = buffersink_ctx;
inputs->pad_idx    = 0;
inputs->next       = NULL;

if (!outputs->name || !inputs->name) {
    ret = AERROR(ENOMEM);
    goto end;
}

if ((ret = avfilter_graph_parse_ptr(filter_graph, filter_spec,
                                    &inputs, &outputs, NULL)) < 0)
    goto end;

```



```

        av_log(NULL, AV_LOG_ERROR, "Could not write data to FIFO\n");
        return AVERRORE_EXIT;
    }
    return 0;
}

static int init_filters(void)
{
    const char *filter_spec;
    unsigned int i;
    int ret;
    filter_ctx = av_malloc_array(ifmt_ctx->nb_streams, sizeof(*filter_ctx));
    if (!filter_ctx)
        return AVERRORE(ENOMEM);

    for (i = 0; i < ifmt_ctx->nb_streams; i++) {
        filter_ctx[i].buffersrc_ctx = NULL;
        filter_ctx[i].buffersink_ctx = NULL;
        filter_ctx[i].filter_graph = NULL;
        if (!(ifmt_ctx->streams[i]->codecpar->codec_type == AVMEDIA_TYPE_VIDEO
              || ifmt_ctx->streams[i]->codecpar->codec_type == AVMEDIA_TYPE_AUDIO))
            continue;

        if (ifmt_ctx->streams[i]->codecpar->codec_type == AVMEDIA_TYPE_VIDEO)
            filter_spec = "null"; /* passthrough (dummy) filter for video */
        else
            filter_spec = "anull"; /* passthrough (dummy) filter for audio */
        ret = init_filter(&filter_ctx[i], stream_ctx[i].dec_ctx,
                         stream_ctx[i].enc_ctx, filter_spec);
        if (ret)
            return ret;
    }
    return 0;
}

/**
 * Initialize one input frame for writing to the output file.
 * The frame will be exactly frame_size samples large.

```

```

* @param[out] frame           Frame to be initialized
* @param    output_codec_context Codec context of the output file
* @param    frame_size         Size of the frame
* @return Error code (0 if successful)
*/
static int init_output_frame(AVFrame **frame,
                             AVCodecContext *output_codec_context,
                             int frame_size)
{
    int error;

    /* Create a new frame to store the audio samples. */
    if (!(*frame = av_frame_alloc())) {
        av_log(NULL, AV_LOG_ERROR, "Could not allocate output frame\n");
        return AVERRORE_EXIT;
    }

    /* Set the frame's parameters, especially its size and format.
     * av_frame_get_buffer needs this to allocate memory for the
     * audio samples of the frame.
     * Default channel layouts based on the number of channels
     * are assumed for simplicity.*/
    (*frame)->nb_samples    = frame_size;
    (*frame)->channel_layout = output_codec_context->channel_layout;
    (*frame)->format        = output_codec_context->sample_fmt;
    (*frame)->sample_rate   = output_codec_context->sample_rate;

    /* Allocate the samples of the created frame. This call will make
     * sure that the audio frame can hold as many samples as specified.*/
    if ((error = av_frame_get_buffer(*frame, 0)) < 0) {
        av_log(NULL, AV_LOG_ERROR, "Could not allocate output frame\n");
        av_err2str(error);
        av_frame_free(frame);
        return error;
    }

    return 0;
}

```

```

static int encode_write_frame(AVFrame *filt_frame, unsigned int stream_index)
{
    int ret;
    int got_frame_local;
    AVPacket enc_pkt;

    /* encode filtered frame */
    enc_pkt.data = NULL;
    enc_pkt.size = 0;
    av_init_packet(&enc_pkt);

    if (stream_ctx[stream_index].enc_ctx->codec_type == AVMEDIA_TYPE_AUDIO) {
        if (filt_frame) {
            filt_frame->pts = stream_ctx[stream_index].pts_audio;
            stream_ctx[stream_index].pts_audio += filt_frame->nb_samples;
        }
    }

    ret = avcodec_send_frame(stream_ctx[stream_index].enc_ctx, filt_frame);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Error submitting the frame to the encoder\n");
        av_err2str(ret));
        return ret;
    }

    while (1) {
        ret = avcodec_receive_packet(stream_ctx[stream_index].enc_ctx, &enc_pkt);
        if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF) {
            av_log(NULL, AV_LOG_INFO, "Error of EAGAIN or EOF\n");
            return 0;
        } else if (ret < 0){
            av_log(NULL, AV_LOG_ERROR, "Error during encoding, %s\n");
            return ret;
        } else {
            /* prepare packet for muxing */
            enc_pkt.stream_index = stream_index;
            av_packet_rescale_ts(&enc_pkt,
                stream_ctx[stream_index].dec_ctx->time_base,
                stream_ctx[stream_index].enc_ctx->time_base);
        }
    }
}

```

```

        av_packet_rescale_ts(&enc_pkt,
                                stream_ctx[stream_index].enc_ctx->t
                                ofmt_ctx->streams[stream_index]->ti
AVRational enc_timebase = stream_ctx[stream_index].enc_ct
AVRational ofmt_timebase = ofmt_ctx->streams[stream_index]
av_log(NULL, AV_LOG_DEBUG, "Muxing frame, enc_pkt->
>pts=%ld,"\
                                " enc_ctx->time_base=%d/%d, ofmt_ctx->time_base=
enc_pkt.pts,
                                enc_timebase.num, enc_timebase.den, ofmt_timebase.
/* mux encoded frame */
ret = av_interleaved_write_frame(ofmt_ctx, &enc_pkt);
av_packet_unref(&enc_pkt);
if (ret < 0)
    return ret;
}
}
return ret;
}

static int encode_write_frame_fifo(AVFrame *filt_frame, unsigned int stream_index)
{
    int ret = 0;
    if (stream_ctx[stream_index].enc_ctx->codec_type == AVMEDIA_TYPE_VIDEO)
        const int output_frame_size = stream_ctx[stream_index].enc_ctx->frame_size;

        /* Make sure that there is one frame worth of samples in the FIFO
         * buffer so that the encoder can do its work.
         * Since the decoder's and the encoder's frame size may differ, we
         * need to FIFO buffer to store as many frames worth of input sample
         * that they make up at least one frame worth of output samples.*/
        add_samples_to_fifo(stream_ctx[stream_index].fifo, filt_frame->data
>nb_samples);
        int audio_fifo_size = av_audio_fifo_size(stream_ctx[stream_index].fi
        if (audio_fifo_size < stream_ctx[stream_index].enc_ctx->frame_size)
            /* Decode one frame worth of audio samples, convert it to the
             * output sample format and put it into the FIFO buffer.*/
            return 0;
}

```

```

/* If we have enough samples for the encoder, we encode them.
 * At the end of the file, we pass the remaining samples to
 * the encoder.*/
while (av_audio_fifo_size(stream_ctx[stream_index].fifo) >= stream_
>frame_size) {
    /* Take one frame worth of audio samples from the FIFO buffer
     * encode it and write it to the output file. */

    /* Use the maximum number of possible samples per frame.
     * If there is less than the maximum possible frame size in the
     * buffer use this number. Otherwise, use the maximum possit
     const int frame_size = FFMIN(av_audio_fifo_size(stream_ctx[
stream_ctx[stream_index].enc_ctx->frame_size);

    AVFrame *output_frame;
    /* Initialize temporary storage for one output frame.*/
    if (init_output_frame(&output_frame, stream_ctx[stream_index]) != 0) {
        av_log(NULL, AV_LOG_ERROR, "init_output_frame failed");
        return AVERROR_EXIT;
    }

    /* Read as many samples from the FIFO buffer as required to fill
     * the samples are stored in the frame temporarily.*/
    if (av_audio_fifo_read(stream_ctx[stream_index].fifo, (void **)output_
frame, frame_size) < frame_size) {
        av_log(NULL, AV_LOG_ERROR, "Could not read data from FIFO");
        av_frame_free(&output_frame);
        return AVERROR_EXIT;
    }
    ret = encode_write_frame(output_frame, stream_index);
    av_frame_free(&output_frame);
}

} else {
    ret = encode_write_frame(filt_frame, stream_index);
}
return ret;
}

```

```

static int filter_encode_write_frame(AVFrame *frame, unsigned int stream_index)
{
    int ret;
    AVFrame *filt_frame;

    av_log(NULL, AV_LOG_INFO, "Pushing decoded frame to filters\n");
    /* push the decoded frame into the filtergraph */
    ret = av_buffersrc_add_frame_flags(filter_ctx[stream_index].buffersrc,
                                       frame, 0);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Error while feeding the filtergraph\n");
        return ret;
    }

    /* pull filtered frames from the filtergraph */
    while (1) {
        filt_frame = av_frame_alloc();
        if (!filt_frame) {
            return AVERROR(ENOMEM);
        }
        av_log(NULL, AV_LOG_INFO, "Pulling filtered frame from filters\n");
        ret = av_buffersink_get_frame(filter_ctx[stream_index].buffersink,
                                      filt_frame);
        if (ret < 0) {
            /* if no more frames for output - returns AVERROR(EAGAIN)
             * if flushed and no more frames for output - returns AVERROR_EOF
             * rewrite retcode to 0 to show it as normal procedure completion
             */
            if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
                ret = 0;
            goto cleanup;
        }
        filt_frame->pict_type = AV_PICTURE_TYPE_NONE;
        ret = encode_write_frame_fifo(filt_frame, stream_index);
        if (ret < 0)
            goto cleanup;
    }
cleanup:

```

```
    av_frame_free(&filt_frame);
    return ret;
}

static int flush_encoder(unsigned int stream_index)
{
    int ret;

    if (!(stream_ctx[stream_index].enc_ctx->codec->capabilities &
          AV_CODEC_CAP_DELAY))
        return 0;

    av_log(NULL, AV_LOG_INFO, "Flushing stream #%u encoder\n", stream_index);
    ret = encode_write_frame(NULL, stream_index);
    return ret;
}

int main(int argc, char **argv)
{
    int ret;

    //av_log_set_level(AV_LOG_DEBUG);
    AVPacket packet = { .data = NULL, .size = 0 };
    AVFrame *frame = NULL;
    enum AVMediaType type;
    unsigned int stream_index;
    unsigned int i;
    int got_frame;

    if (argc != 3) {
        av_log(NULL, AV_LOG_ERROR, "Usage: %s <input file> <output file>\n", argv[0]);
        return 1;
    }

    if ((ret = open_input_file(argv[1])) < 0)
        goto end;
    if ((ret = open_output_file(argv[2])) < 0)
        goto end;
    if ((ret = init_filters()) < 0)
```

```

    goto end;

/* read all packets */
while (1) {
    if ((ret = av_read_frame(ifmt_ctx, &packet)) < 0)
        break;
    stream_index = packet.stream_index;
    type = ifmt_ctx->streams[packet.stream_index]->codecpar->codec_ty

    if (filter_ctx[stream_index].filter_graph) {
        av_log(NULL, AV_LOG_DEBUG, "Going to reencode&filter");
        frame = av_frame_alloc();
        if (!frame) {
            ret = AVERRORE(NOMEM);
            break;
        }
        av_packet_rescale_ts(&packet,
                             ifmt_ctx->streams[stream_index]->tir);
        stream_ctx[stream_index].dec_ctx->tir = packet.pts;
        ret = avcodec_send_packet(stream_ctx[stream_index].dec_ctx,
                                  &packet);
        if (ret < 0) {
            av_log(NULL, AV_LOG_ERROR, "Error submitting the
av_err2str(ret));
            goto end;
        }
        while (ret >= 0) {
            ret = avcodec_receive_frame(stream_ctx[stream_index].dec_ctx);
            if (ret == AVERRORE(EAGAIN) || ret == AVERRORE(ECANCELED))
                break;
            } else if (ret < 0) {
                av_log(NULL, AV_LOG_ERROR, "Error during decoding
av_err2str(ret));
                goto end;
            }
        frame->pts = frame->best_effort_timestamp;
        ret = filter_encode_write_frame(frame, stream_index);
        if (ret < 0)
            goto end;
}

```

```

        av_frame_unref(frame);
    }
    av_frame_free(&frame);
} else {
    /* remux this frame without reencoding */
    av_packet_rescale_ts(&packet,
                         ifmt_ctx->streams[stream_index]->tin
                         ofmt_ctx->streams[stream_index]->tin

    ret = av_interleaved_write_frame(ofmt_ctx, &packet);
    if (ret < 0)
        goto end;
}
av_packet_unref(&packet);
}

/* flush filters and encoders */
for (i = 0; i < ifmt_ctx->nb_streams; i++) {
    /* flush filter */
    if (!filter_ctx[i].filter_graph)
        continue;
    ret = filter_encode_write_frame(NULL, i);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Flushing filter failed\n");
        goto end;
    }

    /* flush encoder */
    ret = flush_encoder(i);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Flushing encoder failed\n");
        goto end;
    }
}

av_write_trailer(ofmt_ctx);
end:
av_packet_unref(&packet);

```

```
av_frame_free(&frame);
for (i = 0; i < ifmt_ctx->nb_streams; i++) {
    avcodec_free_context(&stream_ctx[i].dec_ctx);
    av_audio_fifo_free(stream_ctx[i].fifo);
    if (ofmt_ctx && ofmt_ctx->nb_streams > i && ofmt_ctx->streams[i]
        avcodec_free_context(&stream_ctx[i].enc_ctx);
    if (filter_ctx && filter_ctx[i].filter_graph)
        avfilter_graph_free(&filter_ctx[i].filter_graph);
}
av_free(filter_ctx);
av_free(stream_ctx);
avformat_close_input(&ifmt_ctx);
if (ofmt_ctx && !(ofmt_ctx->oformat->flags & AVFMT_NOFILE))
    avio_closep(&ofmt_ctx->pb);
avformat_free_context(ofmt_ctx);

if (ret < 0)
    av_log(NULL, AV_LOG_ERROR, "Error occurred: %s\n", av_err2st
return ret ? 1 : 0;
}
```

13. STREAMING

Streaming media technology is the product of the development of network technology and multimedia technology to a certain stage. The term streaming media can refer to both streaming technology that transmits continuous time-based media over the Internet, or continuous time-based media itself that uses streaming technology. There are currently two main ways to transmit audio, video and other multimedia information on the Internet: download and streaming. Using to download, users need to download the entire media file, before it can be played. Due to the limitation of network bandwidth, downloading often takes a long time, so this processing delay is very large.

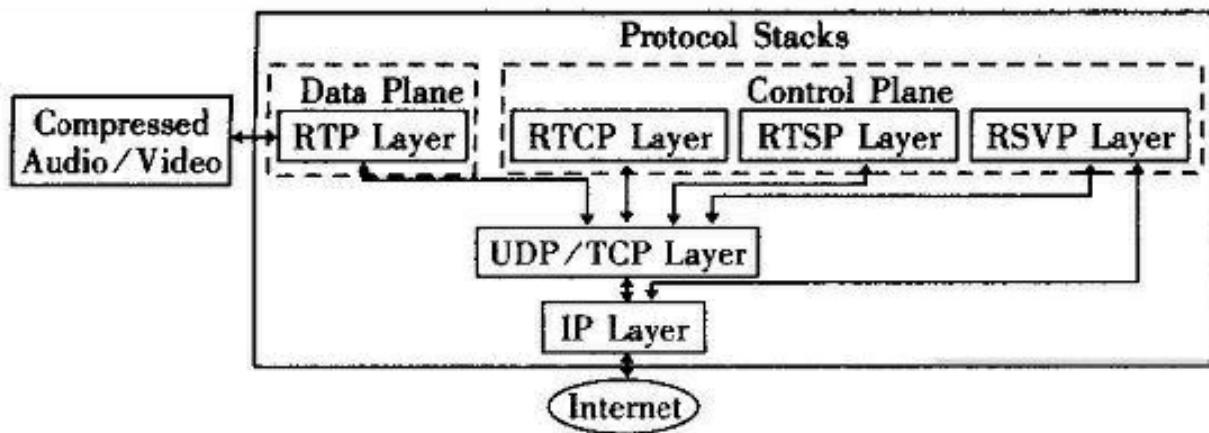
The key technology for streaming media is streaming. Before transmission, the multimedia is preprocessed (reduced quality and efficient compression), and then a buffer system is used to ensure that the data is continuously and correctly transmitted. Using streaming mode, users do not like the way that the use of downloading the entire file to wait until all the next load is completed , but only after

a delay of a few seconds to tens of seconds to start to play it and watch the client. At this point, the rest of the media file will continue to download in the background. Compared with the simple downloading method, this streaming transmission method of downloading and playing multimedia files not only shortens the startup delay greatly, but also greatly reduces the requirement for the system cache capacity. Another benefit of using streaming transmission is to make those who do not know or can not know in advance the size of the media data (such as online broadcast, video conferencing, etc.) possible

A significant feature of multimedia applications is the large amount of data, and many applications have relatively high real-time requirements. Traditional TCP protocol is a connection-oriented protocol, and its retransmission mechanism and congestion control mechanism are not suitable for real-time multimedia transmission. RTP is an application-based transport layer protocol, it does not provide any guarantee of transmission reliability and traffic congestion control mechanism. RTP is on top of UDP (User Datagram Protocol). Although UDP is not as reliable as TCP, and it cannot guarantee the quality of real-time services, RTCP needs to monitor

data transmission and service quality in real time. However, because the transmission delay of UDP is lower than that of TCP, it can cooperate well with audio and video. Therefore, in practical applications, RTP / RTCP / UDP is used for audio / video media, while TCP used for the transmission of data and control signaling. At present, the protocols supporting streaming media transmission mainly include real-time transport protocol RTP (Real-Time Transport Protocol), real-time transport control protocol RTCP (Real-Time Transport Control Protocol) and real-time streaming protocol RTSP (Real-Time Streaming Protocol).

The three protocols are briefly introduced below. The streaming media protocol stack is shown in Figure. First, from OSI's 7-layer protocol layering model and TCP / IP's 4-layer model: OSI's 7-layer model:



1. RTMP

1.1. GENERAL INTRODUCTION

RTMP protocol is an application layer protocol, which depends on the underlying reliable transport layer protocol (usually TCP) to ensure the reliability of information transmission. After the establishment of the link based on the transport layer protocol, the RTMP protocol also requires the client and the server to establish an RTMP Connection link based on the transport layer link by "handshake", and some control information will be transmitted on the Connection link, such as SetChunkSize, SetACKWindowSize. The CreateStream command will create a Stream link, used to transmit specific audio and video data and command information to control the transmission of these information. The RTMP protocol will format the data itself during transmission. This format of message is called RTMP Message. In the actual transmission, in order to better achieve multiplexing, subcontracting and information fairness, the sending end Message will be divided into Chunks with Message ID, each chunk

may be a separate Message, or may be part of Message, the receiving end will be based on the length of the data contained in the chunk, message id and message length chunk Revert to a complete Message, so as to send and receive information.

1.2. HANDSHAKE

To establish a valid RTMP Connection link, you must first " handshake ": the client sends three chunks C0, C1, C2 (in order) to the server, and the server sends S0, S1, S2 (in order) to the client) Three chunks before effective information transmission. The RTMP protocol itself does not stipulate the specific transmission sequence of the six Messages, but the RTMP protocol implementer needs to ensure these points:

- The client must wait to receive S1 before sending C2
- The client must wait for receiving S2 before sending Other information (control information, real audio and video data, etc.)
- The server must wait until receiving C0 to send S1
- The server must wait until receiving C1 before sending S2
- The server must wait until C2 is received before sending other information (control information and real audio and video data).

If a handshake chunk is sent each time, the handshake sequence will be like this: in theory, as long as the

above conditions are met, how to arrange 6 Messages the order is acceptable, but in actual implementation, in order to minimize the number of communications based on the authentication function of the handshake, the general sending order is this. This can be verified by wireshark grabbing the ffmpeg push packet:

client | Server | | --- C0 + C1 --> | | <--- S0 + S1 + S2- | |
--- C2 ---> |

1.3 RTMP CHUNK STREAM

Chunk Stream is a logical abstraction of the stream that transmits RTMP Chunk. Information about RTMP between the client and the server is communicated on this stream. The operation on this stream is also our focus on the RTMP protocol.

1.3.1 MESSAGE

Message here refers to a message that satisfies the protocol format and can be divided into chunks for sending. The fields contained in the message are as follows:

- **Timestamp:** timestamp of the message (but not necessarily the current time, which will be described later), 4 bytes
- **Length (Length):** refers to Message Payload (message load) is the length of audio and video and other information data, 3 bytes
- **TypeId (type Id):** Type Id of the message, 1 byte
- **Message Stream ID (message stream ID):** the unique identifier of each message, when divided into Chunk and Restore Chunk to Message, it is based on this ID to identify whether it is the same message Chunk, 4 bytes, and Little-endian format storage

1.3.2 CHUNKING

RTMP does not use Message as the unit when sending and receiving data, but splits the Message into Chunks for sending, and must send a Chunk before sending the next Chunk. The MessageID in each chunk represents which Message it belongs to, and the receiving end will also assemble the chunk into a Message according to this id.

Why does RTMP split Message into different chunks? By splitting, a large amount of data can be split into smaller "Message", so that you can avoid low-priority messages to continue to send high-priority blocking data, such as in the video transmission process, will include Video frames, audio frames and RTMP control information, if you continue to send audio data or control data, it may cause blockage of the video frame, and then it will cause the most annoying phenomenon when watching video. At the same time, for the Message with a small amount of data, you can compress the information by the field of the Chunk Header, thereby reducing the amount of information transmission. (The specific compression

method will be introduced later)

The default size of Chunk is 128 bytes. During the transmission process, the maximum value of Chunk data can be set through a control message called Set Chunk Size. The sender and the receiver will maintain a Chunk Size, which can be set separately. To change the maximum size of the chunk sent by this party. A larger chunk reduces the time to calculate each chunk and thus reduces the CPU usage, but it will take more time to send, especially in the case of low-bandwidth networks, it is likely to block more important information behind Transmission. A smaller chunk can reduce this blocking problem, but a small chunk will introduce too much extra information (header in the chunk), and a small number of multiple transmissions may also cause intermittent sending, which cannot fully utilize the advantages of high bandwidth. Therefore, it is not suitable for transmission in high bit rate streams. When sending data, you should try different chunk sizes for the data to be sent. You can get the appropriate chunk size through packet capture analysis and other methods. During the transmission process, you can dynamically adjust the chunk size according to the current bandwidth information and the actual information size. Size, so as

to maximize CPU utilization and reduce the probability of information blocking.

1.4. DIFFERENT TYPES OF RTMP MESSAGE

- Command Message (Command Message, Message Type ID = 17 or 20): indicates the command message transmitted between the client box server and performing certain operations on the opposite end, such as connect means connecting to the opposite end, and the opposite end will record if it agrees to connect. The sender sends a message and returns a successful connection message. Publish means to start pushing the stream to the other party. After receiving the command, the receiving end is ready to accept the stream information sent by the opposite end. The more common Command Message will be introduced later. When information is encoded using AMF0, Message Type ID = 20, and Message Type ID = 17 when AMF3 is encoded.
- Data Message (data message, Message Type ID = 15 or 18): transfer some metadata (MetaDataTable, such as video name, resolution, etc.) or some user-defined messages. When the message uses AMF0 encoding, Message Type ID = 18, AMF3 encoding Message

Type ID = 15.

- Shared Object Message (Shared Message, Message Type ID = 16 or 19): Represents a Flash type object, which is composed of a set of key-value pairs, used for multiple clients and multiple instances. When information is encoded using AMF0, Message Type ID = 19, and Message Type ID = 16 when AMF3 is encoded.
- Audio Message (audio information, Message Type ID = 8): audio data.
- Video Message (Video Information, Message Type ID = 9): Video data.
- Aggregate Message (aggregate information, Message Type ID = 22): a collection of multiple RTMP sub-messages
- User Control Message Events (User Control Message, Message Type ID = 4): Tell the other party to execute the user control events contained in this information, such as the Stream Begin event to tell the other party to start the transmission of stream information. Unlike the Protocol Control Message mentioned earlier, this is at the RTMP protocol layer, not at the RTMP chunk stream protocol layer, which is easy to confuse. When this information is sent in the

chunk stream, Message Stream ID = 0, Chunk Stream Id = 2, Message Type Id = 4.

1.5 PUSH RTMP STREAMER SAMPLE

Record a simplest FFmpeg-based streamer (simplest ffmpeg streamer). The function of the pusher is to push the local video data to the streaming media server. The pusher recorded in this sample can push local media files in the format of MOV / AVI / MKV / MP4 / FLV through streaming media protocols (such as RTMP, HTTP, UDP, TCP, RTP, etc.) as live streaming Go out. Due to the wide variety of streaming media protocols, they are not recorded one by one. Here we record how to push local files to RTMP streaming media servers (such as Flash Media Server, Red5, Wowza, etc.) in the form of RTMP live streaming.

On the basis of this flow pusher, it can be modified in various ways to realize various flow pushers. For example:

- * Change the input file to a network stream URL, and a converter can be realized.
- * Change the input file to the form of callback function

(memory read), you can push the video data in the memory.

* Change the input file to the system device (via libavdevice), and at the same time add the encoding function to realize the real-time pusher (live broadcast).
PS: This program does not include the function of video transcoding.

1.5.1 INTRODUCTION

The function of RTMP streamer (Streamer) in the streaming media system can be represented by the following figure. First send the video data to the streaming media server (Server, such as FMS, Red5, Wowza, etc.) in the form of RTMP, and then the client (usually Flash Player) can watch the real-time stream by accessing the streaming media server.

Before running this program, you need to run the RTMP streaming media server and establish the corresponding Application on the streaming media server. After this program runs, you can watch the pushed live stream through the RTMP client (such as Flash Player, FFplay, etc.).

1.5.2 NEED TO PAY ATTENTION TO PACKAGE FORMAT

The package format used by RTMP is FLV. Therefore, when specifying the output streaming media, the encapsulation format must be specified as "flv". Similarly, other streaming media protocols also need to specify their encapsulation format. For example, when using UDP to push streaming media, you can specify its encapsulation format as "mpegts".

1.5.3 DELAY

There is a delay when sending streaming media data. Otherwise, FFmpeg processes data very quickly, and can send all the data out in an instant, which cannot be accepted by the streaming server. Therefore, the data needs to be sent according to the actual frame rate of the video. The pusher recorded in this sample uses the `av_usleep()` function to sleep between video frames to delay transmission. In this way, you can send data according to the frame rate of the video. The reference code is as follows.

```
//...
int64_t start_time=av_gettime();
while (1) {
//...
    //Important:Delay
    if(pkt.stream_index==videoindex){
        AVRational time_base;ifmt_ctx->streams[videoindex]->time_base;
        AVRational time_base_q={1,AV_TIME_BASE};
        int64_t pts_time = av_rescale_q(pkt.dts, time_base, time_base_q);
        int64_t now_time = av_gettime() - start_time;
        if (pts_time > now_time)
            av_usleep(pts_time - now_time);
    }
//...
}
```

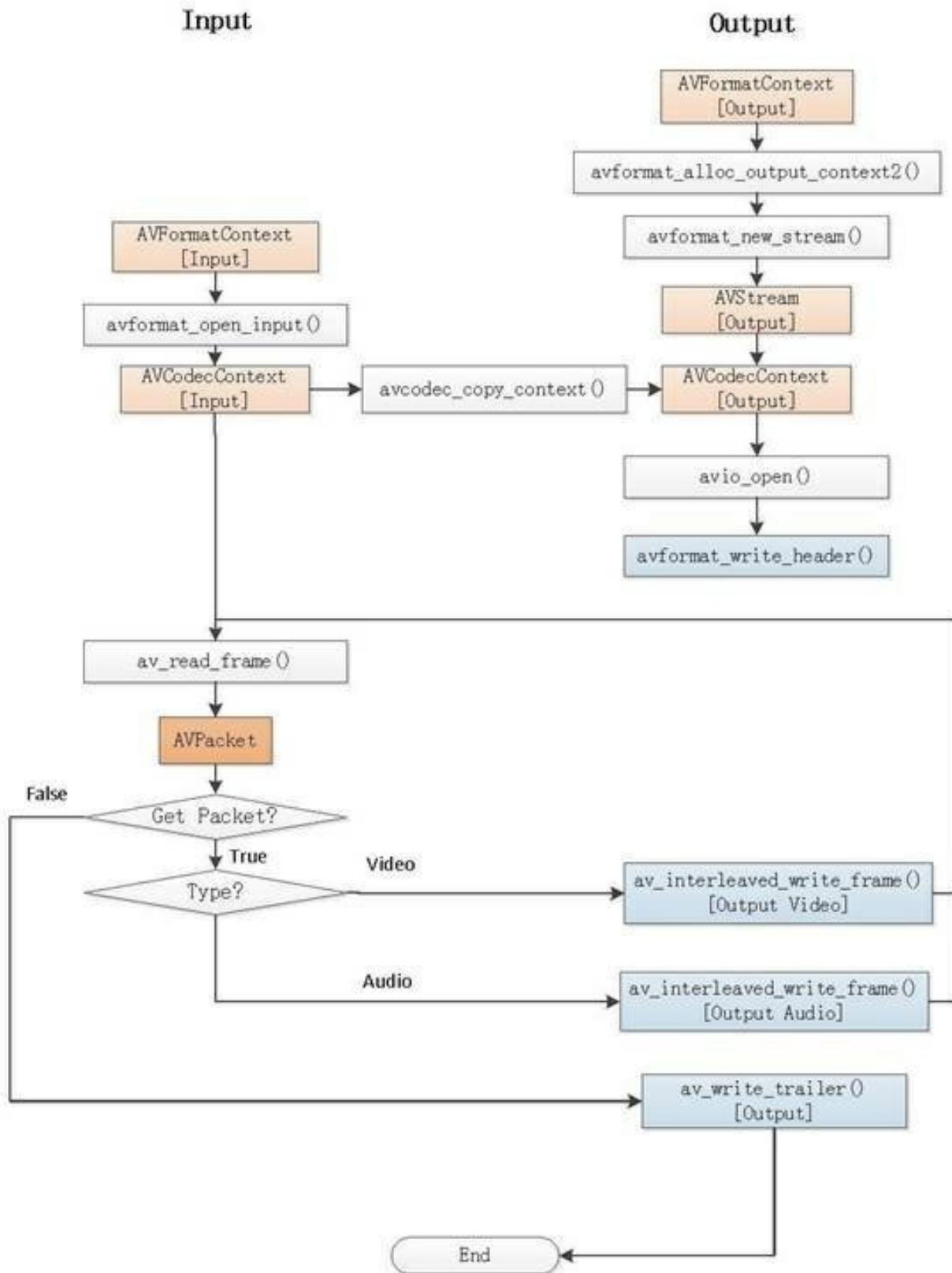
//...

1.5.4 PTS / DTS ISSUE

The bare stream without encapsulation format (such as H.264 bare stream) does not include the parameters of PTS and DTS. When sending such data, you need to calculate and write the PTS, DTS, duration and other parameters of AVPacket yourself.

```
//FIX:No PTS (Example: Raw H.264)
//Simple Write PTS
if(pkt.pts==AV_NOPTS_VALUE){
    //Write PTS
    AVRational time_base1=ifmt_ctx->streams[videoindex]->time_base;
    //Duration between 2 frames (us)
    int64_t calc_duration=(double)AV_TIME_BASE/av_q2d(ifmt_ctx-
>streams[videoindex]->r_frame_rate);
    //Parameters
    pkt.pts=(double)(frame_index*calc_duration)/(double)
(av_q2d(time_base1)*AV_TIME_BASE);
    pkt.dts=pkt.pts;
    pkt.duration=(double)calc_duration/(double)
(av_q2d(time_base1)*AV_TIME_BASE);
}
```

1.5.5 FLOWCHART



1.5.6 Sample Code

```
/**  
*  
* Simplest FFmpeg Streamer (Send RTMP)  
* This example stream local media files to streaming media  
* server (Use RTMP as example).  
* It's the simplest FFmpeg streamer.  
*  
*/  
  
#include <stdio.h>  
  
extern "C"  
{  
#include "libavformat/avformat.h"  
#include "libavutil/mathematics.h"  
#include "libavutil/time.h"  
};  
  
int main(int argc, char* argv[])  
{  
    AVOutputFormat *ofmt = NULL;  
    //AVFormatContext, AVFormatContext  
    //((Input AVFormatContext and Output AVFormatContext)  
    AVFormatContext *ifmt_ctx = NULL, *ofmt_ctx = NULL;  
    AVPacket pkt;  
    const char *in_filename, *out_filename;  
    int ret, i;  
    //in_filename = "cuc_ieschool.mov";  
    //in_filename = "cuc_ieschool.mkv";  
    //in_filename = "cuc_ieschool.ts";  
    //in_filename = "cuc_ieschool.mp4";  
    //in_filename = "cuc_ieschool.h264";  
    in_filename = "cuc_ieschool.flv";//URL (Input file URL)  
    out_filename = "rtmp://localhost/publishlive/livestream";//  
    URL (Output URL) [RTMP]  
    //out_filename = "rtp://233.233.233.233:6666";// URL (Output URL)
```

[UDP]

```
av_register_all();
//Network
avformat_network_init();
//Input)
if ((ret = avformat_open_input(&ifmt_ctx, in_filename, 0, 0)) < 0) {
printf( "Could not open input file.");
goto end;
}
if ((ret = avformat_find_stream_info(ifmt_ctx, 0)) < 0) {
printf( "Failed to retrieve input stream information");
goto end;
}

int videoindex=-1;
for(i=0; i<ifmt_ctx->nb_streams; i++)
if(ifmt_ctx->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_VIDEO){
videoindex=i;
break;
}

av_dump_format(ifmt_ctx, 0, in_filename, 0);
//Output)
avformat_alloc_output_context2(&ofmt_ctx, NULL, "flv",
out_filename); //RTMP
//avformat_alloc_output_context2(&ofmt_ctx, NULL, "mpegts",
out_filename);//UDP

if (!ofmt_ctx) {
printf( "Could not create output context\n");
ret = AVERRORE_UNKOWN;
goto end;
}
ofmt = ofmt_ctx->oformat;
for (i = 0; i < ifmt_ctx->nb_streams; i++) {
//Create output AVStream according to input AVStream)
AVStream *in_stream = ifmt_ctx->streams[i];
```

```
AVStream *out_stream = avformat_new_stream(ofmt_ctx, in_stream->codec->codec);
    if (!out_stream) {
        printf( "Failed allocating output stream\n");
        ret = AVERROR_UNKNOWN;
        goto end;
    }
    //Copy the settings of AVCodecContext)
    ret = avcodec_copy_context(out_stream->codec, in_stream->codec);
    if (ret < 0) {
        printf( "Failed to copy context from input to output stream codec
context\n");
        goto end;
    }
    out_stream->codec->codec_tag = 0;
    if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
        out_stream->codec->flags |= CODEC_FLAG_GLOBAL_HEADER;
    }
    //Dump Format-----
    av_dump_format(ofmt_ctx, 0, out_filename, 1);
    //URL (Open output URL)
    if (!(ofmt->flags & AVFMT_NOFILE)) {
        ret = avio_open(&ofmt_ctx->pb, out_filename,
AVIO_FLAG_WRITE);
        if (ret < 0) {
            printf( "Could not open output URL '%s'", out_filename);
            goto end;
        }
    }
    //((Write file header)
    ret = avformat_write_header(ofmt_ctx, NULL);
    if (ret < 0) {
        printf( "Error occurred when opening output URL\n");
        goto end;
    }

    int frame_index=0;
```

```

int64_t start_time=av_gettime();
while (1) {
    AVStream *in_stream, *out_stream;
    //AVPacket (Get an AVPacket)
    ret = av_read_frame(ifmt_ctx, &pkt);
    if (ret < 0)
        break;
    //FIX: No PTS (Example: Raw H.264)
    //Simple Write PTS
    if(pkt.pts==AV_NOPTS_VALUE){
        //Write PTS
        AVRational time_base1=ifmt_ctx->streams[videoindex]->time_base;
        //Duration between 2 frames (us)
        int64_t calc_duration=
            (double)AV_TIME_BASE/av_q2d(ifmt_ctx->streams[videoindex]-
            >r_frame_rate);                                pkt.pts=(double)
            (frame_index*calc_duration)/(double)
            (av_q2d(time_base1)*AV_TIME_BASE);
            pkt.dts=pkt.pts;pkt.duration=(double)calc_duration/(double)
            (av_q2d(time_base1)*AV_TIME_BASE);
        }
        //Important:Delay
        if(pkt.stream_index==videoindex){
            AVRational time_base=ifmt_ctx->streams[videoindex]->time_base;
            AVRational time_base_q={1,AV_TIME_BASE};
            int64_t pts_time = av_rescale_q(pkt.dts, time_base, time_base_q);
            int64_t now_time = av_gettime() - start_time;
            if (pts_time > now_time)
                av_usleep(pts_time - now_time);
        }
        in_stream = ifmt_ctx->streams[pkt.stream_index];
        out_stream = ofmt_ctx->streams[pkt.stream_index];
        /* copy packet */
        //PTS/DTS (Convert PTS/DTS)
        pkt.pts = av_rescale_q_rnd(pkt.pts, in_stream->time_base,
        out_stream->time_base, (AVRounding))
    }
}

```

```

(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.dts = av_rescale_q_rnd(pkt.dts, in_stream->time_base,
out_stream->time_base, (AVRounding)
(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.duration = av_rescale_q(pkt.duration, in_stream->time_base,
out_stream->time_base);
    pkt.pos = -1;
//Print to Screen
if(pkt.stream_index==videoindex){
printf("Send %8d video frames to output URL\n",frame_index);
frame_index++;
}
//ret = av_write_frame(ofmt_ctx, &pkt);
ret = av_interleaved_write_frame(ofmt_ctx, &pkt);
if (ret < 0) {
printf( "Error muxing packet\n");
break;
}
av_free_packet(&pkt);
}
//(Write file trailer)
av_write_trailer(ofmt_ctx);
end:
avformat_close_input(&ifmt_ctx);
/* close output */
if (ofmt_ctx && !(ofmt->flags & AVFMT_NOFILE))
avio_close(ofmt_ctx->pb);
avformat_free_context(ofmt_ctx);
if (ret < 0 && ret != AVERROREOF) {
printf( "Error occurred.\n");
return -1;
}
return 0;
}

```

1.6 SAVE RTMP STREAMING MEDIA AS A LOCAL FLV FILE

The effect of the stream collector and the streamer is the opposite: the streamer is used to send local files in the form of streaming media, and the stream collector is used to save the streaming media content as local files.

The streamer recorded in this sample can save RTMP streaming media as a local FLV file. Since FFmpeg itself supports many streaming media protocols and encapsulation formats, it also supports other packaging formats and streaming media protocols.

```
/**  
*  
* Simplest FFmpeg Receiver (Receive RTMP)  
* This example saves streaming media data (Use RTMP as example)  
* as a local file.  
* It's the simplest FFmpeg stream receiver.  
*  
*/  
#include <stdio.h>
```

```
#define __STDC_CONSTANT_MACROS
#ifndef _WIN32
//Windows
extern "C"
{
#include "libavformat/avformat.h"
#include "libavutil/mathematics.h"
#include "libavutil/time.h"
};
#else
//Linux...
#endif __cplusplus
extern "C"
{
#endif
#include <libavformat/avformat.h>
#include <libavutil/mathematics.h>
#include <libavutil/time.h>
#endif __cplusplus
};

#endif
#endif

//'1': Use H.264 Bitstream Filter
#define USE_H264BSF 0

int main(int argc, char* argv[])
{
    AVOutputFormat *ofmt = NULL;
    //Input AVFormatContext and Output AVFormatContext
    AVFormatContext *ifmt_ctx = NULL, *ofmt_ctx = NULL;
    AVPacket pkt;
    const char *in_filename, *out_filename;
    int ret, i;
    int videoindex=-1;
    int frame_index=0;
    in_filename = "rtmp://live.hkstv.hk.lxdns.com/live/hks";
    //in_filename = "rtp://233.233.233.233:6666";
```

```
//out_filename = "receive.ts";
//out_filename = "receive.mkv";
out_filename = "receive.flv";

av_register_all();
//Network
avformat_network_init();
//Input
if ((ret = avformat_open_input(&ifmt_ctx, in_filename, 0, 0)) < 0) {
printf( "Could not open input file.");
goto end;
}
if ((ret = avformat_find_stream_info(ifmt_ctx, 0)) < 0) {
printf( "Failed to retrieve input stream information");
goto end;
}

for(i=0; i<ifmt_ctx->nb_streams; i++)
if(ifmt_ctx->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_VIDEO){
videoindex=i;
break;
}

av_dump_format(ifmt_ctx, 0, in_filename, 0);

//Output
avformat_alloc_output_context2(&ofmt_ctx, NULL, NULL,
out_filename); //RTMP

if (!ofmt_ctx) {
printf( "Could not create output context\n");
ret = AVERRORE_UNKOWN;
goto end;
}
ofmt = ofmt_ctx->oformat;
for (i = 0; i < ifmt_ctx->nb_streams; i++) {
//Create output AVStream according to input AVStream
AVStream *in_stream = ifmt_ctx->streams[i];
```

```

AVStream *out_stream = avformat_new_stream(ofmt_ctx,
in_stream->codec->codec);
if (!out_stream) {
printf( "Failed allocating output stream\n");
ret = AVERROR_UNKNOWN;
goto end;
}
//Copy the settings of AVCodecContext
ret = avcodec_copy_context(out_stream->codec, in_stream->codec);
if (ret < 0) {
printf( "Failed to copy context from input to output stream codec
context\n");
goto end;
}
out_stream->codec->codec_tag = 0;
if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
out_stream->codec->flags |= CODEC_FLAG_GLOBAL_HEADER;
}
//Dump Format-----
av_dump_format(ofmt_ctx, 0, out_filename, 1);
//Open output URL
if (!(ofmt->flags & AVFMT_NOFILE)) {
ret = avio_open(&ofmt_ctx->pb, out_filename,
AVIO_FLAG_WRITE);
if (ret < 0) {
printf( "Could not open output URL '%s'", out_filename);
goto end;
}
}
//Write file header
ret = avformat_write_header(ofmt_ctx, NULL);
if (ret < 0) {
printf( "Error occurred when opening output URL\n");
goto end;
}

#if USE_H264BSF

```

```

AVBitStreamFilterContext* h264bsfc =
av_bitstream_filter_init("h264_mp4toannexb");
#endif

while (1) {
    AVStream *in_stream, *out_stream;
    //Get an AVPacket
    ret = av_read_frame(ifmt_ctx, &pkt);
    if (ret < 0)
        break;
    in_stream = ifmt_ctx->streams[pkt.stream_index];
    out_stream = ofmt_ctx->streams[pkt.stream_index];
    /* copy packet */
    //Convert PTS/DTS
    pkt.pts = av_rescale_q_rnd(pkt.pts, in_stream->time_base,
    out_stream->time_base, (AVRounding)
    (AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.dts = av_rescale_q_rnd(pkt.dts, in_stream->time_base,
    out_stream->time_base, (AVRounding)
    (AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
    pkt.duration = av_rescale_q(pkt.duration, in_stream->time_base,
    out_stream->time_base);
    pkt.pos = -1;
    //Print to Screen
    if(pkt.stream_index==videoindex){
        printf("Receive %8d video frames from input URL\n",frame_index);
        frame_index++;
    }

#if USE_H264BSF
    av_bitstream_filter_filter(h264bsfc, in_stream->codec, NULL,
    &pkt.data, &pkt.size, pkt.data, pkt.size, 0);
#endif
    }
    //ret = av_write_frame(ofmt_ctx, &pkt);
    ret = av_interleaved_write_frame(ofmt_ctx, &pkt);

    if (ret < 0) {
        printf( "Error muxing packet\n");
}

```

```
        break;
    }
    av_free_packet(&pkt);
}

#if USE_H264BSF
    av_bitstream_filter_close(h264bsfc);
#endif

//Write file trailer
av_write_trailer(ofmt_ctx);
end:
    avformat_close_input(&ifmt_ctx);
/* close output */
if (ofmt_ctx && !(ofmt->flags & AVFMT_NOFILE))
    avio_close(ofmt_ctx->pb);
    avformat_free_context(ofmt_ctx);
if (ret < 0 && ret != AERROR_EOF) {
    printf( "Error occurred.\n");
    return -1;
}
return 0;
}
```

2. UDP (USER DATA PROTOCOL)

UDP is much simpler than TCP. It does the same as IP and adds the concept of a port so that you can send a message to another recipient with an IP address. It has no sequence or connection, or two-way connection, and no response. You should think that UDP is not reliable, because you know that TCP is a reliable connection scheme, but actually in the same network segment, or in a local area network with a good signal, UDP is actually very reliable. If there is no packet loss and the packets arrive in order (this is almost the normal state of the short local area network), there is no need to retransmit the packets, so all responses and waits of TCP will only waste time and increase network delay. For applications that can tolerate packet loss (real-time audio and video), even if the network is not powerful, UDP is usually a good solution. It is also often used for small messages and notifications. For example, both DHCP and DNS use UDP. It is worth mentioning that Unix Network File

System (NFS) uses UDP in the LAN. You may think that a file system should require a reliable TCP connection, but NFS implementers feel that using UDP can get better performance and establish a special mechanism to ensure reliability. By the way, it is called "User Datagram Protocol" for a reason, because it was designed by a bunch of system administrators. "Datagram" is another name for "Package". "User" has no practical meaning, just like "You". It means that this computer program has nothing to do with the operating system. The reason is that the low-level IP is written by the person who wrote the OS, but UDP provides many of the same functions as datagrams, serving "user" programs (non-OS).

2.1. MULTICASTING (MULTICASTING)

Here we can simplify the discussion about TCP / IP / UDP. By default, we know that IP (UDP and TCP) can send data packets to another device in a network. More precisely, IP sends packets from one IP address to another. The trick to multicasting is to send a data packet to multiple devices at the same time. You can specify a specific IP address as a multicast address and send it to multiple devices at the same time.

The first thing to know about IP multicast is that only UDP has multicast, and there is no such thing as TCP multicast. Why? The focus of multicasting is to efficiently send the same packet to as many different devices as possible, even unknown devices. However, TCP connections may require packet loss and retransmission or delay or reassembly sequence. These operations may consume resources and are not suitable for many application scenarios that use multicast. (At the same time, multicast does not know whether the outgoing packet has arrived, which also leads to the

inability to use TCP).

UDP multicast and broadcast addresses have a range, unicast does not:

It is not necessary to suggest connection before using UDP protocol for information transmission. In other words, the client sends information to the server. The client only needs to give the server's IP address and port number, and then encapsulate the information into a message to be sent and send it out. As for whether the server exists, or whether the message can be received, the client does not care.

Usually the udp programs we discuss are one-to-one unicast programs. This chapter will discuss the many services: broadcast (Broadcast), multicast (multicast). For broadcast, all hosts in the network will receive a copy of the data. For multicast, the message is simply sent to a multicast address, and network knowledge distributes the data to hosts that indicate that they want to receive the data sent to the multicast address. In general, only UDP sockets allow broadcast or multicast.

2.2. UDP BROADCAST

The difference between broadcast UDP and unicast UDP is different IP addresses. Broadcast uses broadcast address 255.255.255.255 to send messages to every host on the same broadcast network. It is worth emphasizing that the local broadcast information will not be forwarded by the router. Of course, this is very easy to understand, because if the router forwards the broadcast information, it will inevitably cause network paralysis. This is why the designers of the IP protocol deliberately did not define an Internet-wide broadcast mechanism.

Broadcast addresses are often used to exchange status information among players on the same local network in online games. Broadcasting is no longer to write a demo program, the reader can change the ip address of the ECHO program to the broadcast address. In fact, as the name implies, the broadcast wants all people in the LAN to speak, but the broadcast still needs to indicate the port number of the receiver, because it is impossible to listen to the broadcast on all ports of the receiver.

2.3. UDP MULTICAST

The same UDP multicast should also indicate the port number of the recipient, and similar to broadcast is the difference between multicast and unicast is the address. The range of multicast addresses in ipv4 is: 224.0.0.0 to 239.255.255.255.

2.4. UDP BROADCAST AND UNICAST

The processing procedures of broadcast and unicast are different. Unicast data is only processed by a specific host that sends and receives data, and broadcast data is processed throughout the local area network.

For example, there are 3 hosts on an Ethernet, and the configuration of the hosts is shown in Table

Host	A	B	C
IP address	192.168.1.150	192.168.1.151	192.168.1.152
MAC address	00: 00: 00: 00: 00: 01	00: 00: 00: 00: 00: 02	00: 00: 00: 00: 00: 03

Unicast process: Host A sends a UDP datagram to Host B. The destination IP is 192.168.1.151, the port is 80, and the destination MAC address is 00: 00: 00: 00: 00: 02. This data passes through the UDP layer and the IP layer and reaches the data link layer. The data is propagated throughout the Ethernet. In this layer, other

hosts determine the destination MAC address. The MAC address of host C is 00: 00: 00: 00: 00: 03, which does not match the destination MAC address 00: 00: 00: 00: 00: 02. The data link layer does not process it and discards the data directly.

The MAC address of host B is 00: 00: 00: 00: 00: 02, which is the same as the destination MAC address 00: 00: 00: 00: 00: 02. This data will go through the IP layer and UDP layer to the application that receives the data program.

Broadcasting process: Host A sends broadcast data to the entire network. The destination IP is 192.168.1.255, the port is 80, and the destination MAC address is FF: FF: FF: FF: FF: FF. This data passes through the UDP layer and the IP layer and reaches the data link layer. The data is propagated throughout the Ethernet. In this layer, other hosts determine the destination MAC address. Since the destination MAC address is FF: FF: FF: FF: FF: FF, Host C and Host B will ignore the comparison of MAC addresses (of course, if the protocol stack does not support broadcasting, the MAC addresses are still compared) and process the received data.

The processing procedures of host B and host C are the same. This data will pass through the IP layer and UDP layer and reach the application that receives the

data.

2.5. PULL RTMP STREAM TO UDP OUTPUT PLAYBACK

Pull the rtmp stream and forward it with udp, as for how to deal with it depends on the software function requirements. It is not difficult to configure it by yourself in the ffmpeg environment. The code is as follows:

```
#include "stdafx.h"
#include "pch.h"
#include <string>
#include <memory>
#include <thread>
#include <iostream>
using namespace std;

AVFormatContext *inputContext = nullptr;
AVFormatContext * outputContext;
int64_t lastReadPacktTime ;

static int interrupt_cb(void *ctx)
{
    int timeout = 10;
    if(av_gettime() - lastReadPacktTime > timeout *1000 *1000)
    {
        return -1;
    }
}
```

```

        }
        return 0;
    }

int OpenInput(string inputUrl)
{
    inputContext = avformat_alloc_context();
    lastReadPacktTime = av_gettime();
    inputContext->interrupt_callback.callback = interrupt_cb;
    int ret = avformat_open_input(&inputContext, inputUrl.c_str(),
nullptr,nullptr);
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "Input file open input failed\n");
        return ret;
    }
    ret = avformat_find_stream_info(inputContext,nullptr);
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "Find input file stream inform
failed\n");
    }
    else
    {
        av_log(NULL, AV_LOG_FATAL, "Open input file %s
success\n",inputUrl.c_str());
    }
    return ret;
}

shared_ptr<AVPacket> ReadPacketFromSource()
{
    shared_ptr<AVPacket> packet(static_cast<AVPacket*>
(av_malloc(sizeof(AVPacket))), [&](AVPacket *p) { av_packet_free(&p);
av_freep(&p);});
    av_init_packet(packet.get());
    lastReadPacktTime = av_gettime();
    int ret = av_read_frame(inputContext, packet.get());
}

```

```

    if(ret >= 0)
    {
        return packet;
    }
    else
    {
        return nullptr;
    }
}

void av_packet_rescale_ts(AVPacket *pkt, AVRational src_tb,
AVRational dst_tb)
{
    if (pkt->pts != AV_NOPTS_VALUE)
        pkt->pts = av_rescale_q(pkt->pts, src_tb, dst_tb);
    if (pkt->dts != AV_NOPTS_VALUE)
        pkt->dts = av_rescale_q(pkt->dts, src_tb, dst_tb);
    if (pkt->duration > 0)
        pkt->duration = av_rescale_q(pkt->duration, src_tb, dst_tb);
}

int WritePacket(shared_ptr<AVPacket> packet)
{
    auto inputStream = inputContext->streams[packet->stream_index];
    auto outputStream = outputContext->streams[packet-
>stream_index];
    av_packet_rescale_ts(packet.get(),inputStream-
>time_base,outputStream->time_base);
    return av_interleaved_write_frame(outputContext, packet.get());
}

int OpenOutput(string outUrl)
{
    int ret = avformat_alloc_output_context2(&outputContext, nullptr,
"mpegts", outUrl.c_str());
    if(ret < 0)
    {
        av_log(NULL, AV_LOG_ERROR, "open output context failed\n");
    }
}

```

```

    goto Error;
}

ret = avio_open2(&outputContext->pb, outUrl.c_str(),
AVIO_FLAG_WRITE,nullptr, nullptr);
if(ret < 0)
{
av_log(NULL, AV_LOG_ERROR, "open avio failed");
goto Error;
}

for(int i = 0; i < inputContext->nb_streams; i++)
{
AVStream * stream = avformat_new_stream(outputContext,
inputContext->streams[i]->codec-
>codec);
ret = avcodec_copy_context(stream->codec, inputContext-
>streams[i]->codec);
if(ret < 0)
{
av_log(NULL, AV_LOG_ERROR, "copy coddec context failed");
goto Error;
}
}

ret = avformat_write_header(outputContext, nullptr);
if(ret < 0)
{
av_log(NULL, AV_LOG_ERROR, "format write header failed");
goto Error;
}

av_log(NULL, AV_LOG_FATAL, " Open output file success
%s\n",outUrl.c_str());
return ret ;
Error:
if(outputContext)
{

```

```
for(int i = 0; i < outputContext->nb_streams; i++)
{
    avcodec_close(outputContext->streams[i]->codec);
}
avformat_close_input(&outputContext);
}
return ret ;
}

void CloseInput()
{
    if(inputContext != nullptr)
    {
        avformat_close_input(&inputContext);
    }
}

void CloseOutput()
{
    if(outputContext != nullptr)
    {
        for(int i = 0 ; i < outputContext->nb_streams; i++)
        {
            AVCodecContext *codecContext = outputContext->streams[i]->codec;
            avcodec_close(codecContext);
        }
        avformat_close_input(&outputContext);
    }
}

void Init()
{
    av_register_all();
    avfilter_register_all();
    avformat_network_init();
    av_log_set_level(AV_LOG_ERROR);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    Init();
    int ret = OpenInput("rtmp://58.200.131.2:1935/livetv/hunantv");
    if(ret >= 0)
    {
        ret = OpenOutput("udp://127.0.0.1:6789");
    }

    if(ret <0)
        goto Error;

    while(true)
    {
        auto packet = ReadPacketFromSource();
        if(packet)
        {
            ret = WritePacket(packet);
            if(ret >= 0)
            {
                cout<<"WritePacket Success!"<<endl;
            }
            else
            {
                cout<<"WritePacket failed!"<<endl;
            }
        }
        else
        {
            break;
        }
    }
}
```

Error:

```
CloseInput();
CloseOutput();
while(true)
{
    this_thread::sleep_for(chrono::seconds(100));
```

```
    }
    return 0;
}
```

It should be noted that because the network conditions will be affected when pulling, the delay setting needs to be longer, I set it to 10 seconds, as shown below

```
static int interrupt_cb(void *ctx)
{
    int timeout = 10;
    if(av_gettime() - lastReadPacktTime > timeout *1000 *1000)
    {
        return -1;
    }
    return 0;
}
```

2.6. UDP RECEIVING TS STREAM

```
#include "utils.h"
#include <pthread.h>
#include <libavcodec / avcodec.h>
#include <libavformat / avformat.h>
UdpQueue recvqueue;
UdpParam udpParam;

// Register the callback function of av_read_frame, here is the simplest
processing, error handling should be added in the actual application,
waiting for timeout ...
int read_data (void * opaque, uint8_t * buf, int buf_size) {
int size = buf_size;
int ret;
// printf ("read data% d \n", buf_size);
do {
ret = get_queue (& recvqueue, buf, buf_size);
} while (ret);
// printf ("read data Ok% d \n", buf_size);
return size;
}
#define BUF_SIZE 4096 * 500
int main (int argc, char ** argv)
{
init_queue (& recvqueue, 1024 * 500);
udpParam.argv = argv;
udpParam.queue = & recvqueue;
uint8_t * buf = av_mallocz (sizeof (uint8_t) * BUF_SIZE);
// UDP receiving thread
```

```

pthread_t udp_recv_thread;
pthread_create (& udp_recv_thread, NULL, udp_ts_recv, & udpParam);
pthread_detach (udp_recv_thread);
av_register_all ();
AVCodec * pVideoCodec, * pAudioCodec;
AVCodecContext * pVideoCodecCtx = NULL;
AVCodecContext * pAudioCodecCtx = NULL;
AVIOContext * pb = NULL;
AVInputFormat * piFmt = NULL;
AVFormatContext * pFmt = NULL;

// step1: apply for an AVIOContext
pb = avio_alloc_context (buf, BUF_SIZE, 0, NULL, read_data, NULL,
NULL);

if (! pb) {
fprintf (stderr, "avio alloc failed! \n");
return -1;
}

// step2: Probe stream format
if (av_probe_input_buffer (pb, & piFmt, "", NULL, 0, 0) <0) {
fprintf (stderr, "probe failed! \n");
return -1;
} else {
fprintf (stdout, "probe success! \n");
fprintf (stdout, "format:% s [% s] \n", piFmt-> name, piFmt-> long_name);
}

pFmt = avformat_alloc_context ();
pFmt-> pb = pb; // step3: this step is critical
// step4: open stream
if (avformat_open_input (& pFmt, "", piFmt, NULL) <0) {
fprintf (stderr, "avformat open failed. \n ");
return -1;
} else {
fprintf (stdout, " open stream success! \n ");
}

// The following is consistent with the file processing

```

```
if (av_find_stream_info (pFmt) <0) {
    fprintf (stderr, "could not fine stream. \n");
    return -1;
}
av_dump_format (pFmt, 0, "", 0);
int videoindex = -1;
int audioindex = -1;
for (int i = 0; i <pFmt-> nb_streams; i++) {
    if ((pFmt-> streams [i]-> codec-> codec_type ==
        AVMEDIA_TYPE_VIDEO) &&
        (videoindex <0)) {
        videoindex = i;
    }
    if ((pFmt-> streams [i]-> codec-> codec_type ==
        AVMEDIA_TYPE_AUDIO) &&
        (audioindex <0)) {
        audioindex = i;
    }
}
if (videoindex <0 || audioindex <0) {
    fprintf (stderr, "videoindex=%d, audioindex=%d \n", videoindex,
            audioindex);
    return -1;
}
AVStream * pVst, * pAst;
pVst = pFmt-> streams [videoindex];
pAst = pFmt-> streams [audioindex];
pVideoCodecCtx = pVst-> codec;
pAudioCodecCtx = pAst-> codec;
pVideoCodec = avcodec_find_decoder (pVideoCodecCtx-> codec_id);
if (! pVideoCodec) {
    fprintf (stderr, "could not find video decoder! \n");
    return -1;
}
if (avcodec_open (pVideoCodecCtx, pVideoCodec) <0) {
    fprintf (stderr, "could not open video codec! \n");
    return -1;
```

```
}

pAudioCodec = avcodec_find_decoder (pAudioCodecCtx-> codec_id);
if (! pAudioCodec) {
    fprintf (stderr, "could not find audio decoder! \n");
    return -1;
}
if (avcodec_open (pAudioCodecCtx, pAudioCodec) <0) {
    fprintf (stderr, "could not open audio codec! \n");
    return -1;
}
int got_picture;
uint8_t samples [AVCODEC_MAX_AUDIO_FRAME_SIZE * 3/2];
AVFrame * pframe = avcodec_alloc_frame ();
AVPacket pkt;
av_init_packet (& pkt);
while (1) {
    if (av_read_frame (pFmt, & pkt)>= 0) {
        if (pkt.stream_index == videoindex) {
            fprintf (stdout, "pkt.size=% d, pkt pts=% lld, pkt.data = 0x% x.", pkt.size,
                    pkt.pts, (unsigned int ) pkt.data);
            avcodec_decode_video2 (pVideoCodecCtx, pframe, & got_picture, & pkt);
            if (got_picture) {
                fprintf (stdout, "decode one video frame! \n");
            }
        } else if (pkt.stream_index == audioindex) {
            int frame_size = AVCODEC_MAX_AUDIO_FRAME_SIZE * 3/2;
            if (avcodec_decode_audio3 (pAudioCodecCtx, (int16_t *) samples, &
            frame_size, & pkt)>= 0) {
                fprintf (stdout, "decode one audio frame! \n");
            }
        }
        (v_free_pack_free );
    }
}
av_free (buf);
av_free (pframe);
free_queue (& recvqueue);
```

```
return 0;  
}
```

3. RTP (REAL-TIME TRANSPORT PROTOCOL)

RTP is a transmission protocol for multimedia data streams on the Internet. It is published by the IETF (Internet Engineering Task Force) as RFC1889. RTP is defined as working under one-to-one or one-to-many transmission. Its purpose is to provide time information and achieve stream synchronization. The typical application of RTP is built on UDP, but it can also work on other protocols such as TCP or ATM. RTP itself only guarantees the transmission of real-time data, and cannot provide a reliable transmission mechanism for sequentially transmitting data packets, nor does it provide flow control or congestion control. It relies on RTCP to provide these services.

3.1. RTP WORKING MECHANISM

A sharp problem that threatens multimedia data transmission is the unpredictable data arrival time. But streaming media transmission requires timely arrival of data for playback and playback. The rtp protocol provides time tags, serial numbers, and other structures for controlling the timely release of data. "Time label" is the most important information in the concept of streaming. The sender conceals the time stamp in the data packet according to the instant sampling. After the receiving end receives the packet, it is restored to the original timely data rate in accordance with the correct time stamp. The timing properties of different media formats are different. But RTP itself is not responsible for synchronization, RTP is only a transport layer protocol, in order to simplify the transport layer processing and improve the efficiency of this layer. Move some transport layer protocol functions (such as flow control) to the application layer to complete. The synchronization is done by the

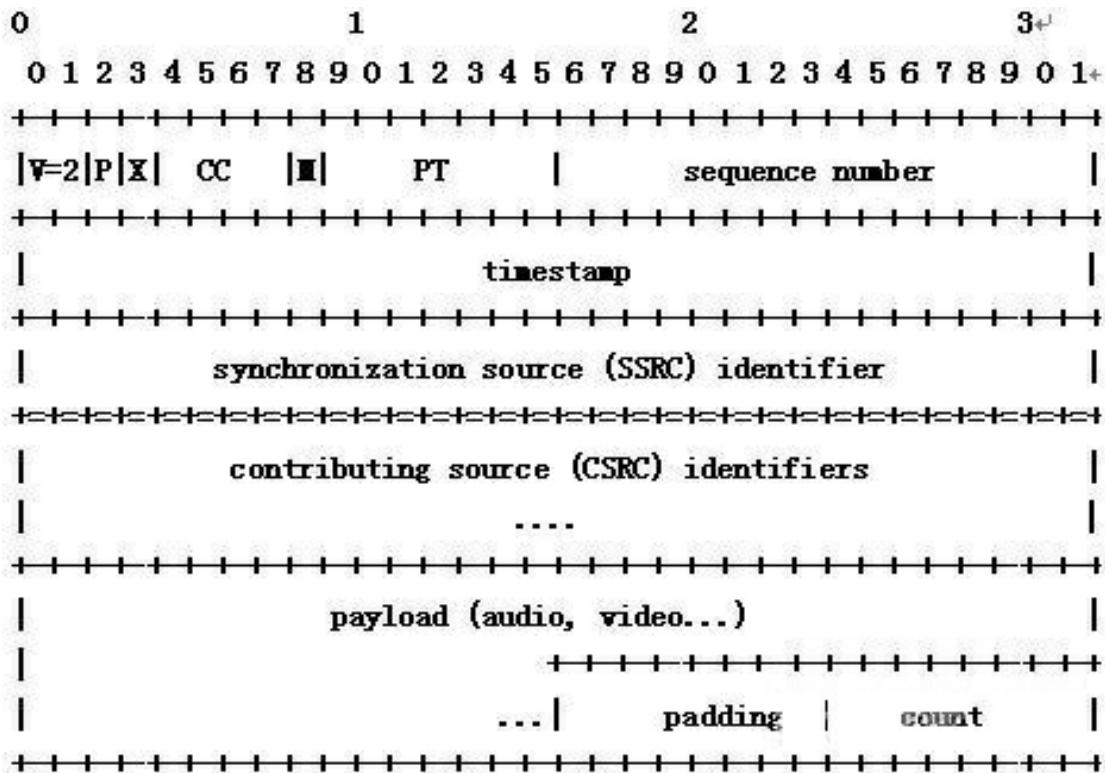
application layer protocol. It does not have the complete function of the transport layer protocol, does not provide any mechanism to ensure the real-time transmission of data, does not support resource reservation, nor guarantee the quality of service. RTP messages do not even include descriptions of length and message boundaries. At the same time, the data packets and control packets of the rtp protocol use adjacent different ports, which greatly improves the flexibility of the protocol and the simplicity of processing.

Both rtp protocol and udp work together to complete the transport layer protocol function. The udp protocol only transmits data packets, regardless of the time sequence of data packet transmission. The protocol data unit of rtp is carried by udp packet. When carrying rtp data packets, sometimes a frame of data is divided into several packets with the same time stamp, you can know that the time stamp is not necessary. The udp multiplexing let rtp protocol supports the use of explicit multi-point delivery to meet the needs of multimedia sessions.

Although the RTP protocol is a transport layer protocol, it is not implemented as a separate layer in the osi architecture. The RTP protocol usually provides services according to a specific application. RTP only

provides a protocol framework, and developers can fully extend the protocol according to the specific requirements of the application.

Protocol message structure:



4. RTCP (RTP CONTROL PROTOCOL)

Real-time Transport Control Protocol (Real-time Transport Control Protocol or RTP Control Protocol or RTCP) is a sister protocol of Real-Time Transport Control (RTP). RTCP provides out-of-band control for RTP media streams. RTCP itself does not transmit data, but works with RTP to package and send multimedia data. RTCP periodically transmits control data between participants of streaming multimedia sessions. The main function of RTCP is to provide feedback for the quality of service provided by RTP.

RTCP collects statistics on related media connections, such as: bytes transferred, packets transferred, packets lost, jitter, unidirectional and bidirectional network delays, etc. Network applications can use the information provided by RTCP to try to improve the quality of service, such as restricting information flow or switching to a codec with less compression. RTCP itself does not provide data encryption or identity authentication. SRTCP can be

used for such purposes.

It can be seen that RTCP + UDP is equivalent to TCP. RTP is just a data format. In other words, providing a transmission control (RTCP) on the basis of UDP is an improvement to UDP. So there can be several combinations: RTP + TCP, RTP + RTCP + UDP

4.1. RTCP WORKING MECHANISM

When the application starts an rtp session, it will use two ports: one for rtp and one for rtcp. RTP itself cannot provide a reliable transmission mechanism for sequentially transmitting data packets, nor does it provide flow control or congestion control. It relies on rtcp to provide these services. Some rtcp packets are periodically issued between rtp sessions for functions such as monitoring service quality and exchanging session user information. The rtcp packet contains statistics such as the number of sent data packets and the number of lost data packets. Therefore, the server can use this information to dynamically change the transmission rate and even change the payload type. rtp and rtcp are used together, they can optimize the transmission efficiency with effective feedback and minimum overhead, so they are particularly suitable for transmitting real-time data on the Internet. According to the data transmission feedback information between users, a flow control strategy can be formulated, and

the interaction of session user information can be formulated a session control strategy.

The rtcp protocol processor defines five types of messages as needed-

1. rr: receiver report
2. sr: sender report
3. sdes: source description items.
4. bye: indicates end of participation.
5. app: application specific functions

They complete receiving, analyzing, generating and the function of sending control messages.

5. RTSP (REALTIME STREAMING PTOCOL)

Real-time streaming protocol RTSP establishes and controls one or several time-synchronous continuous streaming media. Although it is possible for continuous media streams to intersect with control streams, usually it does not send continuous streams itself, such as audio and video. In other words, RTSP acts as a network server for remote control of the multimedia server. RTSP provides an extensible framework that enables the controlled, on-demand transmission of real-time data such as audio and video. Data sources include live data and data stored in clips. The purpose of the protocol is to control multiple data transmission sessions, provide a way to select transmission channels such as UDP, multicast UDP, and TCP, and provide a method for selecting an RTP-based transmission mechanism.

Before requesting the video service from the video server, the client first obtains the presentation description (Presentation description) file of the

requested video service from the web server through HTTP protocol, and uses the information provided by the file to locate the video service address (including the video server address and port number) And video service encoding methods and other information. The client then requests video services from the video server based on the above information. After the video service is initialized, the video server establishes a new video service stream for the client. The client and server run the real-time stream control protocol RTSP to exchange various VCR control signals for the stream, such as PLAY and stop (the PAUSE), fast forward, rewind and the like. When the service is completed, the client submits a TEARDOWN request.
The second part: TS (transport Stream)

TS is an encapsulation format of streaming media proposed in MPEG2, suitable for streaming media playback.

Two media playback formats are supported in MPEG2: PS stream (program stream) for local playback, TS stream (transport Stream) for network playback, TS stream syntax: Transport Stream packets shall be 188 bytes long visible, TS stream The length of the packet is fixed at 188bytes.

Receiving frames with rtsp sample code:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <sstream>

extern "C" {

#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavformat/avio.h>
#include <libswscale/swscale.h>
}

int main(int argc, char** argv) {

    // Open the initial context variables that are needed
    SwsContext *img_convert_ctx;
    AVFormatContext* format_ctx = avformat_alloc_context();
    AVCodecContext* codec_ctx = NULL;
    int video_stream_index;

    // Register everything
    av_register_all();
    avformat_network_init();

    //open RTSP
    if (avformat_open_input(&format_ctx,
                           "rtsp://134.169.178.187:8554/h264.3gp",
                           NULL, NULL) != 0) {
        return EXIT_FAILURE;
    }

    if (avformat_find_stream_info(format_ctx, NULL) < 0) {
        return EXIT_FAILURE;
    }

    //search video stream
    for (int i = 0; i < format_ctx->nb_streams; i++) {
        if (format_ctx->streams[i]->codec->codec_type ==
```

```
AVMEDIA_TYPE_VIDEO)
    video_stream_index = i;
}

AVPacket packet;
av_init_packet(&packet);

//open output file
AVFormatContext* output_ctx = avformat_alloc_context();

AVStream* stream = NULL;
int cnt = 0;

//start reading packets from stream and write them to file
av_read_play(format_ctx); //play RTSP

// Get the codec
AVCodec *codec = NULL;
codec = avcodec_find_decoder(AV_CODEC_ID_H264);
if (!codec) {
    exit(1);
}

// Add this to allocate the context by codec
codec_ctx = avcodec_alloc_context3(codec);

avcodec_get_context_defaults3(codec_ctx, codec);
avcodec_copy_context(codec_ctx, format_ctx-
>streams[video_stream_index]->codec);
std::ofstream output_file;

if (avcodec_open2(codec_ctx, codec, NULL) < 0)
    exit(1);

img_convert_ctx = sws_getContext(codec_ctx->width, codec_ctx-
>height,
        codec_ctx->pix_fmt, codec_ctx->width, codec_ctx->height,
AV_PIX_FMT_RGB24,
        SWS_BICUBIC, NULL, NULL);
```

```

int size = avpicture_get_size(AV_PIX_FMT_YUV420P, codec_ctx->width,
                               codec_ctx->height);
uint8_t* picture_buffer = (uint8_t*) (av_malloc(size));
AVFrame* picture = av_frame_alloc();
AVFrame* picture_rgb = av_frame_alloc();
int size2 = avpicture_get_size(AV_PIX_FMT_RGB24, codec_ctx->width,
                               codec_ctx->height);
uint8_t* picture_buffer_2 = (uint8_t*) (av_malloc(size2));
avpicture_fill((AVPicture *) picture, picture_buffer,
AV_PIX_FMT_YUV420P,
               codec_ctx->width, codec_ctx->height);
avpicture_fill((AVPicture *) picture_rgb, picture_buffer_2,
AV_PIX_FMT_RGB24,
               codec_ctx->width, codec_ctx->height);

while (av_read_frame(format_ctx, &packet) >= 0 && cnt < 1000) {
//read ~ 1000 frames

    std::cout << "1 Frame: " << cnt << std::endl;
    if (packet.stream_index == video_stream_index) { //packet is
video
        std::cout << "2 Is Video" << std::endl;
        if (stream == NULL) { //create stream in file
            std::cout << "3 create stream" << std::endl;
            stream = avformat_new_stream(output_ctx,
format_ctx->streams[video_stream_index]->codec->codec);
            avcodec_copy_context(stream->codec,
format_ctx->streams[video_stream_index]->codec);
            stream->sample_aspect_ratio =
format_ctx->streams[video_stream_index]->codec->sample_aspect_ratio;
        }
        int check = 0;
        packet.stream_index = stream->id;
        std::cout << "4 decoding" << std::endl;
    }
}

```

```
    int result = avcodec_decode_video2(codec_ctx, picture, &check,
&packet);
    std::cout << "Bytes decoded " << result << " check " << check
        << std::endl;
    if (cnt > 100) //cnt < 0)
    {
        sws_scale(img_convert_ctx, picture->data, picture->linesize,
0,
            codec_ctx->height, picture_rgb->data, picture_rgb-
>linesize);
        std::stringstream file_name;
        file_name << "test" << cnt << ".ppm";
        output_file.open(file_name.str().c_str());
        output_file << "P3 " << codec_ctx->width << " " <<
codec_ctx->height
            << " 255\n";
        for (int y = 0; y < codec_ctx->height; y++) {
            for (int x = 0; x < codec_ctx->width * 3; x++)
                output_file
                    << (int) (picture_rgb->data[0]
                        + y * picture_rgb->linesize[0])[x] << " ";
        }
        output_file.close();
    }
    cnt++;
}
av_free_packet(&packet);
av_init_packet(&packet);
}
av_free(picture);
av_free(picture_rgb);
av_free(picture_buffer);
av_free(picture_buffer_2);

av_read_pause(format_ctx);
avio_close(output_ctx->pb);
avformat_free_context(output_ctx);
```

```
    return (EXIT_SUCCESS);  
}
```

ANALYSIS OF MPEG2 TRANSPORT STREAM (MPEG2 TS)

When it comes to specific audio or video formats, the first is the theory. That is what the so-called professors of domestic mixed qualifications do. For us, it is not suitable. Let us understand these obscure theories in our own way. In fact, MPEG2 is a family of protocols. At least the following parts have become the ISO standard:

- ISO / IEC – 13818-1: system part;
- ISO / IEC – 13818-2: video encoding format;
- ISO / IEC – 13818-3: audio Coding format;
- ISO / IEC – 13818-4: conformance test;
- ISO / IEC – 13818-5: software part;
- ISO / IEC – 13818-6: digital storage media command and control;
- ISO / IEC – 13818-7: advanced Audio coding;
- ISO / IEC – 13818-8: Real-time interface for system decoding;

The first part (system part) is very important

and constitutes the basis for MPEG2-based applications. It's a roundabout, right, let me briefly explain: for example, DVD is actually based on the PS stream defined by the system part, plus copyright management, which he composed technology. The protagonist of our story is another stream format, TS stream. Its biggest application at this stage is in the transmission and storage of digital TV programs. Therefore, you can understand that TS is actually a transmission protocol. The load of the international transmission is not significant, only the audio, video or other data is transmitted in the TS. Let me talk about why

there appear the two formats, PS store does not apply to the following environmental losses, while TS is applicable to possible loss or mistakenly various physical network environments, such as your point of view in public television, It is likely to be the application of TS-based DVB-T. Let's look at some concepts in the MPEG2 protocol and do our homework for understanding the code:

ES (Elementary Stream):

An elementary stream (ES) is defined by MPEG communication protocol is usually the output of an audio or video encoder ” Well, it 's very simple, it is

the group data compiled by the encoder, which may be audio, video, or other data. Speaking of which, in fact, the encoder's process to think about, is nothing more than the implementation of: sampling, quantization, encoding these three steps in the encoding of it (some devices may contain the previous sampling and quantization) basic theory of video encoding, or please reference other data

PES (Packetized Elementary Stream):

that defines carrying of elementary streams (usually the output of an audio or video encoder) in packets within MPEG program streams and MPEG transport streams.

1 TS (Transport Stream):

2 PS (Program Stream):

These two have already been mentioned above, we will analyze TS in detail later, we are not interested in PS format. We only enter the topic before entering the topic. As mentioned before, TS is a transmission protocol, so, corresponding to FFmpeg, it can be considered as a kind of encapsulation format.

Therefore, the corresponding code should be found in libavformat first, it is easy to find, that is mpegs.

[libavformat / utils.c]

```

int av_open_input_file(AVFormatContext **ic_ptr, const char *filename,
                      AVInputFormat *fmt,
                      int buf_size,
                      AVFormatParameters *ap)
{
    int err, probe_size;
    AVProbeData probe_data, *pd = &probe_data;
    ByteIOContext *pb = NULL;
    pd->filename = "";
    if (filename)
        pd->filename = filename;
    pd->buf = NULL;
    pd->buf_size = 0;
    //#####
    //【1】This code is actually for the detection of the Container Format that d
    // The container format of the AVFMT_NOFILE tag is handled separately
Demuxers using this tag
    // Only image2_demuxer, rtsp_demuxer, so we can ignore this part when
    //#####
    if (!fmt) {
        /* guess format if no file can be opened */
        fmt = av_probe_input_format(pd, 0);
    }
    /* Do not open file if the format does not need it. XXX: specific
       hack needed to handle RTSP/TCP */
    if (!fmt || !(fmt->flags & AVFMT_NOFILE)) {
        /* if no file needed do not try to open one */
        //#####
        //【2】This function seems easy to understand, nothing more than a buf
we have this,
        //May wish to track down to see others' implementation of buffered IC
        //#####
        if ((err=url_fopen(&pb, filename, URL_RDONLY)) < 0) {
            goto fail;
        }
        if (buf_size > 0) {
            url_setbufsize(pb, buf_size);

```

```

    }
    for(probe_size= PROBE_BUF_MIN; probe_size<=PROBE_BUF_MAX;
        probe_size<<=1){
        int score= probe_size < PROBE_BUF_MAX ? AVPROBE_SCORE_MAX : 0;
        /* read probe data */
        pd->buf= av_realloc(pd->buf, probe_size + AVPROBE_PADDING_SIZE);
        //#####
        //【3】The place where the file is actually read into the buffer of
FILE protocol
        // File_read (), read the content into the buf of pd, the specific code
interested
        //#####
        pd->buf_size = get_buffer(pb, pd->buf, probe_size);
        memset(pd->buf+pd->buf_size, 0, AVPROBE_PADDING_SIZE);
        if (url_fseek(pb, 0, SEEK_SET) < 0) {
            url_fclose(pb);
            if (url_fopen(&pb, filename, URL_RDONLY) < 0) {
                pb = NULL;
                err = AVERROR(EIO);
                goto fail;
            }
        }
        //#####
        //【4】At this point, the pd already has the original file that needs
to find the corresponding container
        // Tag comparison to determine exactly why the container format
        //#####
        /* guess file format */
        fmt = av_probe_input_format2(pd, 1, &score);
    }
    av_freep(&pd->buf);
}
/* if still no format found, error */
if (!fmt) {
    err = AVERROR_NOFMT;
    goto fail;
}

```

```

/* check filename in case an image number is expected */
if (fmt->flags & AVFMT_NEEDNUMBER) {
    if (!av_filename_number_test(filename)) {
        err = AVERROR_NUMEXPECTED;
        goto fail;
    }
}
err = av_open_input_stream(ic_ptr, pb, filename, fmt, ap);
if (err)
    goto fail;
return 0;

fail:
av_freep(&pd->buf);
if (pb)
    url_fclose(pb);
*ic_ptr = NULL;
return err;
}

```

Implementation of encapsulation with buffered IO [liavformat / aviobuf.c]

```

int url_fopen(ByteIOContext **s, const char *filename, int flags)
{
    URLContext *h;
    int err;
    err = url_open(&h, filename, flags);
    if (err < 0)
        return err;
    err = url_fdopen(s, h);
    if (err < 0) {
        url_close(h);
        return err;
    }
    return 0;
}

```

It can be seen that the following function first looks for the protocol format supported by FFmpeg. If the file name does not match, the default is the FILE protocol format. Obviously, the protocol judgment here is interpreted by URL, so basically all The IO interface functions are all in the form of url_xxx. You can also see here, the protocols supported by FFmpeg are:

```
/* protocols */
REGISTER_PROTOCOL (FILE, file);
REGISTER_PROTOCOL (HTTP, http);
REGISTER_PROTOCOL (PIPE, pipe);
REGISTER_PROTOCOL (RTP, rtp);
REGISTER_PROTOCOL (TCP, tcp );
REGISTER_PROTOCOL (UDP, udp);
```

and in most cases, if you do not specify a format like file://xxx, http://xxx , it is handled by the FILE protocol .

[liavformat / avio.c]

```
int url_open(URLContext **puc, const char *filename, int flags)
{
    URLProtocol *up;
    const char *p;
    char proto_str[128], *q;
    p = filename;
    q = proto_str;
    while (*p != '\0' && *p != ':') {
```

```

/* protocols can only contain alphabetic chars */
if (!isalpha(*p))
    goto file_proto;
if ((q - proto_str) < sizeof(proto_str) - 1)
    *q++ = *p;
p++;
}
/* if the protocol has length 1, we consider it is a dos drive */
if (*p == '\0' || (q - proto_str) <= 1) {
file_proto:
    strcpy(proto_str, "file");
} else {
    *q = '\0';
}
up = first_protocol;
while (up != NULL) {
    if (!strcmp(proto_str, up->name))
        //#####
        //Obviously, we already know up, filename, flags at this time
        //#####
        return url_open_protocol (puc, up, filename, flags);
    up = up->next;
}
*puc = NULL;
return AVERROR(ENOENT);
}

```

[libavformat / avio.c]

```

int url_open_protocol (URLContext **puc, struct URLProtocol *up,
                      const char *filename, int flags)
{
    URLContext *uc;
    int err;
    //#####

```

```

// 【a】? Why allocate space like this
//#####
uc = av_malloc(sizeof(URLContext) + strlen(filename) + 1);
if (!uc) {
    err = AERROR(ENOMEM);
    goto fail;
}
#endif LIBAVFORMAT_VERSION_MAJOR >= 53
uc->av_class = &urlcontext_class;
#endif

//#####
// 【b】? Why is this intention
//#####

uc->filename = (char *) &uc[1];
strcpy(uc->filename, filename);
uc->prot = up;
uc->flags = flags;
uc->is_streamed = 0; /* default = not streamed */
uc->max_packet_size = 0; /* default: stream file */
err = up->url_open(uc, filename, flags);
if (err < 0) {
    av_free(uc);
    *puc = NULL;
    return err;
}
//We must be carefull here as url_seek() could be slow, for example for
//http
if((flags & (URL_WRONLY | URL_RDWR)) || !strcmp(up->name, "file"))
    if(!uc->is_streamed && url_seek(uc, 0, SEEK_SET) < 0)
        uc->is_streamed= 1;
    *puc = uc;
    return 0;
fail:
    *puc = NULL;
    return err;
}

```

The above function is not difficult to understand, but there are some places worth pondering, such as the question mark above, do you understand why Coding is so? Obviously, at this time, up-> url_open () actually calls file_open () [libavformat / file.c]. After reading this function, whether the above memory allocation is suddenly realized. The above only analyzed url_open () I haven't analyzed url_fopen (s, h); this part of the code is also left to you with a good heart. Well, in order to track this process, it has gone a bit far, but it is not completely useless.

The detection process of MPEG TS format

[libavformat / mpegs.c]

```
static int mpegs_probe(AVProbeData *p)
{
#ifndef 1
    const int size= p->buf_size;
    int score, fec_score, dvhs_score;
#define CHECK_COUNT 10
    if (size < (TS_FEC_PACKET_SIZE * CHECK_COUNT))
        return -1;
    score = analyze(p->buf, TS_PACKET_SIZE * CHECK_COUNT, TS_PACKET_SIZE, NULL);
    dvhs_score = analyze(p->buf, TS_DVHS_PACKET_SIZE *CHECK_COUNT, TS_DVHS_PACKET_SIZE, NULL);
    fec_score= analyze(p->buf, TS_FEC_PACKET_SIZE*CHECK_COUNT, TS_FEC_PACKET_SIZE, NULL);
    // av_log(NULL, AV_LOG_DEBUG, "score: %d, dvhs_score: %d, fec_score: %d", score, dvhs_score, fec_score);
}
```

```

dvhs_score, fec_score);
// we need a clear definition for the returned score otherwise things will become
// or later
    if(score > fec_score && score > dvhs_score && score > 6)
        return AVPROBE_SCORE_MAX + score - CHECK_COUNT;
    else if(dvhs_score > score && dvhs_score > fec_score && dvhs_score >
            return AVPROBE_SCORE_MAX + dvhs_score - CHECK_COUNT;
    else if(fec_score > 6)
        return AVPROBE_SCORE_MAX + fec_score - CHECK_COUNT;
    else
        return -1;
#else
/* only use the extension for safer guess */
    if (match_ext(p->filename, "ts"))
        return AVPROBE_SCORE_MAX;
    else
        return 0;
#endif
}

static int analyze(const uint8_t *buf, int size, int *packet_size, int *index)
{
    int stat[packet_size];
    int i;
    int x=0;
    int best_score=0;
    memset(stat, 0, packet_size*sizeof(int));

    //#####
    //Since the specific format for searching is at least 3 Bytes, at least the last
    //be searched
    //#####
    for(x=i=0; i<size-3; i++){
        //#####
        //See the protocol description below
        //#####
        if(buf[i] == 0x47 && !(buf[i+1] & 0x80) && (buf[i+3] & 0x30)){
            stat[x]++;
        }
    }
}

```

```

        if(stat[x] > best_score){
            best_score= stat[x];
            if(index)
                *index= x;
        }
    }
    x++;
    if(x == packet_size)
        x= 0;
}
return best_score;
}

```

This function is simply to find the number of packets with a length of packet_size in a buf of size, obviously, the larger the returned value, the more likely it is the corresponding format (188/192/204), of which this special set format, in fact prescribed format protocol:

<i>//Syntax</i>	<i>No. of bits</i>	<i>Mnemonic</i>
transport_packet(){		
sync_byte	8	bslbf
transport_error_indicator	1	bslbf
payload_unit_start_indicator	1	bslbf
transport_priority	1	bslbf
PID	13	uimsbf
transport_scrambling_control	2	bslbf
adaptation_field_control	2	bslbf
continuity_counter	4	uimsbf
if(adaptation_field_control=='10' adaptation_field_control=='11') {		
adaptation_field()		
}		
if(adaptation_field_control=='01' adaptation_field_control=='11') {		

```

        for (i=0;i<N;i++){
            data_byte          8      bslbf
        }
    }
}

```

The sync_byte is fixed to 0x47, which is the above: buf [i] == 0x47. Because the TS Packet with transport_error_indicator is 1 actually has an error, it means that the data carried is meaningless. Such a Packet is obviously meaningless, so: !(Buf [i +1] & 0x80)

For adaptation_field_control, if the value is 0x00, it means that it is reserved for the future, and it is not used now, so:

buf [i + 3] & 0x30

This is the MPEG TS detection process, it is very simple behind We analyze how to get the stream from the mpegs file. 5. Gradually getting better, the previous foundation should be close enough, a bit like the feeling of peeling the onion by hand, let's take a look at the corresponding parsing process for MPEG TS.

```

static int mpegs_read_header(AVFormatContext *s,
                             AVFormatParameters *ap)
{
    MpegTSContext *ts = s->priv_data;
    ByteIOContext *pb = s->pb;
}

```

```

uint8_t buf[1024];
int len;
int64_t pos;

.....
/* read the first 1024 bytes to get packet size */
#####
//【1】With the previous experience of analyzing buffered IO, the following
a problem :)
#####
pos = url_ftell(pb);
len = get_buffer(pb, buf, sizeof(buf));
if (len != sizeof(buf))
    goto fail;
#####
//【2】When I detected the file format before, I already knew the size of the
// Some are redundant. It is estimated that because of the decoding framework,
recently detected packet size cannot be brought in from the front.
// It can be seen that although the framework is good, it will also bring some
less adverse effects
#####
ts->raw_packet_size = get_packet_size(buf, sizeof(buf));
if (ts->raw_packet_size <= 0)
    goto fail;
ts->stream = s;
ts->auto_guess = 0;

if (s->iformat == &mpegts_demuxer) {
    /* normal demux */
    /* first do a scaning to get all the services */
    url_fseek(pb, pos, SEEK_SET);
#####
    【3】
#####
    mpegts_scan_sdt(ts);
#####
    【4】
#####
}

```

```

mpegts_set_service(ts);
#####
[5]
#####
handle_packets(ts, s->probesize);
/* if could not find service, enable auto_guess */
ts->auto_guess = 1;
#ifndef DEBUG_SI
    av_log(ts->stream, AV_LOG_DEBUG, "tuning done\n");
#endif
    s->ctx_flags |= AVFMTCTX_NOHEADER;
} else {
    .....
}
url_fseek(pb, pos, SEEK_SET);
return 0;
fail:
    return -1;
}

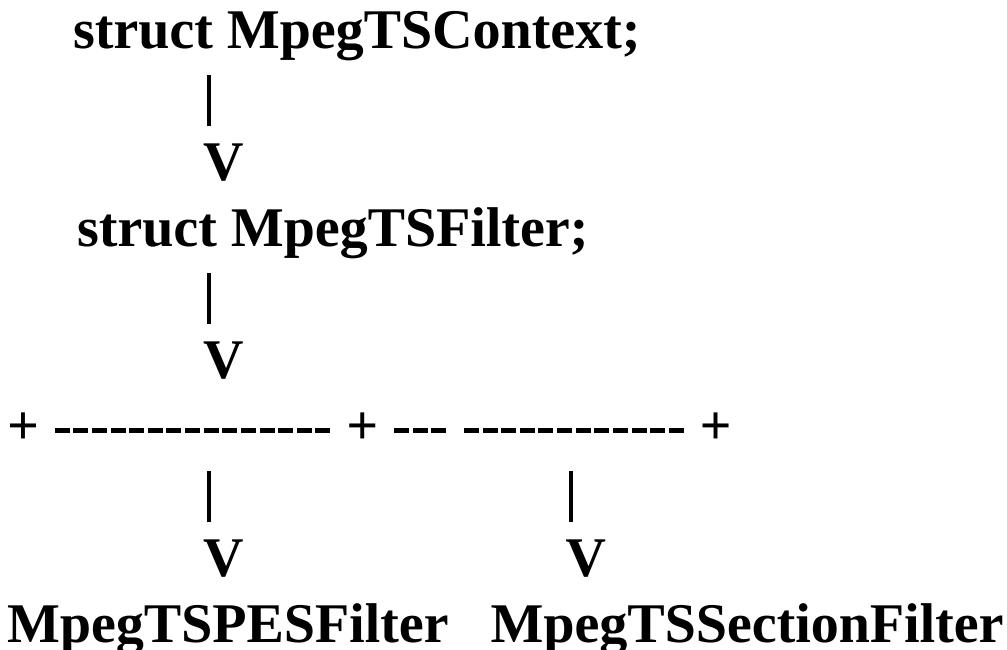
```

Here is a brief talk about MpegTSContext * ts. As you can see from the above, in fact, this is to decode the private data used in different container formats, which can only be used in the corresponding files such as mpegts.c. This increases the modularity of this library , And the biggest benefit of modularity is that the problem is concentrated in a small limited area. If you construct the program yourself, you may wish to refer to its basic idea-such a way the code after you, and after you Life will be much easier. [3] [4] In fact, the same function is called:

`mpegtts_open_section_filter()` Let's see what we intend to do.

```
static MpegTSFilter *mpegtts_open_section_filter(MpegTSContext *ts,
unsigned int pid,
                           SectionCallback *section_cb,
                           void *opaque,
                           int check_crc)
{
    MpegTSFilter *filter;
    MpegTSSectionFilter *sec;
#ifdef DEBUG_SI
    av_log(ts->stream, AV_LOG_DEBUG, "Filter: pid=0x%x\n", pid);
#endif
    if (pid >= NB_PID_MAX || ts->pids[pid])
        return NULL;
    filter = av_mallocz(sizeof(MpegTSFilter));
    if (!filter)
        return NULL;
    ts->pids[pid] = filter;
    filter->type = MPEGTS_SECTION;
    filter->pid = pid;
    filter->last_cc = -1;
    sec = &filter->u.section_filter;
    sec->section_cb = section_cb;
    sec->opaque = opaque;
    sec->section_buf = av_malloc(MAX_SECTION_SIZE);
    sec->check_crc = check_crc;
    if (!sec->section_buf) {
        av_free(filter);
        return NULL;
    }
    return filter;
}
```

To fully understand this part of the code, you actually need to analyze definition of the data structure:



In fact, it is very simple, it is struct MpegTSContext; there are NB_PID_MAX (8192) TS Filter, and each struct MpegTSFilter may be a Filter of PES or a Filter of Section. The syntax structure of TS was given:

//Syntax	No. of bits	Mnemonic
transport_packet(){		
sync_byte	8	bslbf
transport_error_indicator	1	bslbf
payload_unit_start_indicator	1	bslbf
transport_priority	1	bslbf
PID	13	uimsbf
transport_scrambling_control	2	bslbf
adaptation_field_control	2	bslbf
continuity_counter	4	uimsbf

```

    if(adaptation_field_control=='10' || adaptation_field_control=='11'){
        adaptation_field()
    }
    if(adaptation_field_control=='01' || adaptation_field_control=='11')
{
    for (i=0;i<N;i++){
        data_byte      8      bslbf  } }
}

```

Visible [3] [4], that is, two section-type filters are mounted. In fact, in the two loads of TS, section is the metadata of PES. Only when the section is parsed can the PES data be parsed, so first Thefilter in the previous section. Two section filters are mounted, as follows:

PID Section Name Callback

SDT_PID (0x0011)	ServiceDescriptionTable	sdt_cb
PAT_PID (0x0000)	ProgramAssociationTable	
pat_cb		

Since Callback is used, it is naturally used in the back place. Therefore, we continue to look at the code at [5] to see the most important place. In simple terms:

```

handle_packets ()
|
+-> read_packet ()
|

```

+-> **handle_packet ()**

|

+-> **write_section_data ()**

read_packet () is very simple, it is to find sync_byte (0x47), and it seems that handle_packet () is where we really should pay attention. This function is very important. The code for analysis:

```
/* handle one TS packet */
static void handle_packet(MpegTSContext *ts, const uint8_t *packet)
{
    AVFormatContext *s = ts->stream;
    MpegTSFilter *tss;
    int len, pid, cc, cc_ok, afc, is_start;
    const uint8_t *p, *p_end;

    //#####
    //Get the PID of the package
    //#####
    pid = AV_RB16(packet + 1) & 0x1fff;
    if(pid && discard_pid(ts, pid))
        return;
    //#####
    //Whether it is the beginning of PES or Section (payload_unit_start_indic
    //#####

    is_start = packet[1] & 0x40;
    tss = ts->pids[pid];

    //#####
    //ts-> auto_guess is 0 at this time, so the following code is not considered
    //#####
    if (ts->auto_guess && tss == NULL && is_start) {
        add_pes_stream(ts, pid, -1, 0);
        tss = ts->pids[pid];
    }
}
```

```

}

if (!tss)
    return;

//#####
//The code is very clear, although the inspection, but does not use the result
inspection
//#####
/* continuity check (currently not used) */
cc = (packet[3] & 0xf);
cc_ok = (tss->last_cc < 0) || (((tss->last_cc + 1) & 0x0f) == cc));
tss->last_cc = cc;

//#####
//adaptation_field_control
//#####
/* skip adaptation field */
afc = (packet[3] >> 4) & 3;
p = packet + 4;
if (afc == 0) /* reserved value */
    return;
if (afc == 2) /* adaptation field only */
    return;
if (afc == 3) {
    /* skip adapation field */
    p += p[0] + 1;
}

//#####
//p is close to the payload in the TS packet
//#####
/* if past the end of packet, ignore */
p_end = packet + TS_PACKET_SIZE;
if (p >= p_end)
    return;

ts->pos47= url_ftell(ts->stream->pb) % ts->raw_packet_size;

if (tss->type == MPEGTS_SECTION) {

```

```

if (is_start) {
//
#####
// For Section, the first byte of the matching part is the pointer field. If
is 0,
    // It means that the beginning of the section is followed immediately, or
it is the end of a section and
        // At the beginning of another Section, therefore, the process here is ac-
composed of two values is_start
            // (payload_unit_start_indicator) and len (pointer field) to decide
            // #####
/* pointer field present */
len = *p++;
if (p + len > p_end)
    return;
if (len && cc_ok) {
    // #####
    // 1).is_start == 1
    // len > 0
    // The load part is composed of the End part of A Section and the S-
Section.
        // End partial write
        // #####
        /* write remaining section bytes */
        write_section_data(s, tss, p, len, 0);
        /* check whether filter has been closed */
        if (!ts->pids[pid])
            return;
    }
    p += len;
    if (p < p_end) {
        // #####
        // 2).is_start == 1
        // len > 0
        // The load section is composed of the End section of A Section and
of B Section.
            // Start Partial write

```

```

        // or:
        // 3).
        // is_start == 1
        // len == 0
        // The load part is only the Start part of a Section, write it
        // #####
        write_section_data(s, tss, p, p_end - p, 1);
    }
} else if (cc_ok) {
    // #####
    // 4).is_start == 0
    // The load part is only the middle part of the Section, write it
    // #####
    write_section_data(s, tss, p, p_end - p, 0);
} else {
    // #####
    // If it is PES type, call its Callback directly, but obviously, only the Se
    // It is possible to parse PES after parsing is complete
    // #####
    tss->u.pes_filter.pes_cb(tss, p, p_end - p, is_start);
}
}

```

`write_section_data()` function repeatedly collect data buffer in the guide to complete the relevant Section of the restructured process, then registered before calling two `section_cb`:

Later we will analyze hung in the previous two `section_cb`

mpegts.c file analysis

1 Summary

ffmpeg frame parsing code corresponding to MPEG-2 TS stream in mpegs.c file, the file has two demultiplexed instances with: mpegs_demuxer and mpegsraw_demuxer, mpegs_demuxer corresponding real TS stream format type.

2 mpegs_demuxer structure analysis

```
AVInputFormat mpegs_demuxer = { au
    "mpegts", //The name of demux
    NULL_IF_CONFIG_SMALL("MPEG-2 transport stream format"), //
If defined
    sizeof(MpegTSContext),//The size of the private domain of each
demuxer structure
    mpegs_probe,//Check if it is in TS stream format
    mpegs_read_header,//Introduced below
    mpegs_read_packet,//Introduced below
    mpegs_read_close,//Close demuxer
    read_seek,//Introduced below
    mpegs_get_pcr,//Introduced below
    .flags = AVFMT_SHOW_IDS|AVFMT_TS_DISCONT,//Introduced
below
};
```

The main frame by av_register_all function of the configuration register ffmpeg in to by mpegs_probe function detects whether the TS stream format, and then find audio streams and video streams by mpegs_read_header function (note meaning: not find all all audio streams in the function And video stream),

and finally call the mpegs_read_packet function to extract the audio stream and video stream data, and push it into the decoder through the main frame.

3 mpegs_probe function analysis

mpegs_probe av_probe_input_format2 call is to be determined according to the score that format returned can be maximized. mpegs_probe calls the analyze function, we first analyze the analyze function.

```
static int analyze(const uint8_t *buf, int size, int packet_size, int *index)
{
    int stat[TS_MAX_PACKET_SIZE];//Points statistics
    int i;
    int x=0;
    int best_score=0;
    memset(stat, 0, packet_size*sizeof(int));
    for(x=i=0; i<size-3; i++){
        if(buf[i] == 0x47 && !(buf[i+1] & 0x80) && (buf[i+3] & 0x30)){
            stat[x]++;
            if(stat[x] > best_score){
                best_score= stat[x];
                if(index)
                    *index= x;
            }
        }
        x++;
        if(x == packet_size)
            x= 0;
    }
    return best_score;
}
```

The idea of the analyze function:

buf [i] == 0x47 &&! (buf[i + 1] & 0x80) && (buf [i + 3] & 0x30) is the mode
of TS stream synchronization, 0x47 is the symbol of
TS stream synchronization, (Buf [i + 1] & 0x80 is a
transmission error flag. When buf [i + 3] & 0x30 is 0, it
means that it is reserved for future use of ISO / IEC.
There is no such value at present. Remember that this
mode is TS stream synchronization mode "STAT array
variable stored in the TS stream is synchronization
pattern." appears in a location number analyze a
function of scan test data, if the pattern is found, the
stat [i% packet_size] ++ (x varying function of the
amount used is Modulus operation, because when x ==
packet_size, x is reset to zero), after scanning, naturally
the accumulated value of the synchronization position
is the largest. The mpegts_probe function gets the
corresponding score by calling the analyze function,
and the ffmpeg framework will use the number
determines whether it is the corresponding format, and
returns the corresponding AVInputFormat instance. 4
mpegts_read_header function analysis the ellipsis in
the following represents the code related to
mpegtsraw_demuxer, which is not involved for the
time being.

```

static int mpegs_read_header(AVFormatContext *s,
                             AVFormatParameters *ap)
{
    MpegTSContext *ts = s->priv_data;
    ByteIOContext *pb = s->pb;
    uint8_t buf[5*1024];
    int len;
    int64_t pos;
    .....
    //Save the current position of the stream, so that the original position
    can be restored after the detection operation is completed,
    //In this way, there will be no wastage during playback.
    pos = url_ftell(pb);
    //Read segment stream to detect TS packet size
    len = get_buffer(pb, buf, sizeof(buf));
    if (len != sizeof(buf))
        goto fail;
    //The size of the TS stream packet is usually 188 bytes. What I have
    seen so far is 188 bytes.
    //There are three sizes of TS packages:
    //1) 188 bytes under normal circumstances
    //2) 192Bytes DVH-S format
    //3) Based on 188Bytes, plus 16Bytes of FEC (forward error
    correction), which is 204bytes
    ts->raw_packet_size = get_packet_size(buf, sizeof(buf));
    if (ts->raw_packet_size <= 0)
        goto fail;
    ts->stream = s;

    //auto_guess = 1, Then as long as a PES pid is found in the
    handle_packet function
    //Create the PES stream
    //auto_guess = 0, It is ignored.
    //auto_guess The main function is to set it to 1 if there is no service
    information in the TS stream.
    //Then any PID stream will be used as a media stream to establish the
    corresponding PES data structure.

```

```

//The PES pid was found during the mpegts_read_header function, but
//Is not to establish the corresponding stream, just analyze the PSI
information.

//For the relevant code, see the following code of the handle_packet
function:
    //tss = ts->pids[pid];
    //if (ts->auto_guess && tss == NULL && is_start) {
    //  add_pes_stream(ts, pid, -1, 0);
    //  tss = ts->pids[pid];
    //}
    ts->auto_guess = 0;
    if (s->iformat == &mpegts_demuxer) {
        /* normal demux */
        /* first do a scaning to get all the services */
        url_fseek(pb, pos, SEEK_SET);
        //Mount the callback function of the parsed SDT table ts-> pids
variable, // so find the corresponding processing callback function in the
handle_packet function according to the corresponding pid.
        mpegts_open_section_filter(ts, SDT_PID, sdt_cb, ts, 1);
        //Same as above, just hang the callback function for PAT table
analysis
        mpegts_open_section_filter(ts, PAT_PID, pat_cb, ts, 1);
        //Detect segment flow, easy to detect SDT, PAT, PMT table
        handle_packets(ts, s->probesize);
        /* if could not find service, enable auto_guess */

        //Turn on the add pes stream logo, so that the pes is found in the
handle_packet function
        //pid, it will automatically create the stream of the pes.
        ts->auto_guess = 1;
        dprintf(ts->stream, "tuning done\n");
        s->ctx_flags |= AVFMTCTX_NOHEADER;
    } else {
        .....
    }
    //Restore the position before detection.
    url_fseek(pb, pos, SEEK_SET);

```

```

    return 0;
fail:
    return -1;
}
static MpegTSFilter *mpegts_open_section_filter(MpegTSContext *ts,
unsigned int pid,
                                         SectionCallback *section_cb, void
*opaque,
                                         int check_crc)
{
    MpegTSFilter *filter;
    MpegTSSectionFilter *sec;
    dprintf(ts->stream, "Filter: pid=0x%x\n", pid);
    if (pid >= NB_PID_MAX || ts->pids[pid])
        return NULL;
    //Allocate space for the filter, mount the pids of MpegTSContext
    //Is the instance
    filter = av_mallocz(sizeof(MpegTSFilter));
    if (!filter)
        return NULL;
    //Mount the filter instance
    ts->pids[pid] = filter;
    //Set the filter-related parameters, because the analysis of the business
    information table is only a segment,
    //So the filter type is MPEGTS_SECTION
    filter->type = MPEGTS_SECTION;

    //Set pid
    filter->pid = pid;
    filter->last_cc = -1;
    //Set filter callback handler
    sec = &filter->u.section_filter;
    sec->section_cb = section_cb;
    sec->opaque = opaque;
    //Allocate buffers for data processing, it will be called after calling
    handle_packet function
    //write_section_data stores the data of the business information table in

```

the ts package here,

```

//The callback function registered above will be delivered for
processing only after the segment collection is completed.
sec->section_buf = av_malloc(MAX_SECTION_SIZE);
sec->check_crc = check_crc;
if (!sec->section_buf) {
    av_free(filter);
    return NULL;
}
return filter;
}

static int handle_packets(MpegTSContext *ts, int nb_packets)
{
    AVFormatContext *s = ts->stream;
    ByteIOContext *pb = s->pb;
    uint8_t packet[TS_PACKET_SIZE];
    int packet_num, ret;
    //This variable indicates the end of the second handle_packets
processing.
    //When mpegts_read_packet is called, if a PES packet is found, then
    //ts->stop_parse = 1, the current analysis ends.
    ts->stop_parse = 0;
    packet_num = 0;
    for(;;) {
        if (ts->stop_parse>0)
            break;
        packet_num++;
        if (nb_packets != 0 && packet_num >= nb_packets)
            break;
        //Read ts packets, usually 188bytes
        ret = read_packet(pb, packet, ts->raw_packet_size);
        if (ret != 0)
            return ret;
        handle_packet(ts, packet);
    }
    return 0;
}
```

```

}

static int handle_packet(MpegTSContext *ts, const uint8_t *packet)
{
    AVFormatContext *s = ts->stream;
    MpegTSFilter *tss;
    int len, pid, cc, cc_ok, afc, is_start;
    const uint8_t *p, *p_end;
    int64_t pos;

    //Get the PID of the package from the TS package.
    pid = AV_RB16(packet + 1) & 0x1fff;
    if(pid && discard_pid(ts, pid))
        return 0;
    is_start = packet[1] & 0x40;
    tss = ts->pids[pid];
    //ts-> auto_guess is set to 0 in the mpegs_read_header function,
    //In other words, pes stream is not established during the ts detection
    process.
    if (ts->auto_guess && tss == NULL && is_start) {
        add_pes_stream(ts, pid, -1, 0);
        tss = ts->pids[pid];
    }
    //The mpegs_read_header function calls the handle_packet function
    only to process TS streams
    //Business information, because tss is not established for the
    corresponding PES, so tss is empty and returned directly.
    if (!tss)
        return 0;
    /* continuity check (currently not used) */
    cc = (packet[3] & 0xf);
    cc_ok = (tss->last_cc < 0) || (((tss->last_cc + 1) & 0x0f) == cc));
    tss->last_cc = cc;
    /* skip adaptation field */
    afc = (packet[3] >> 4) & 3;
    p = packet + 4;
    if (afc == 0) /* reserved value */
        return 0;
}

```

```

if (afc == 2) /* adaptation field only */
    return 0;
if (afc == 3) {
    /* skip adaption field */
    p += p[0] + 1;
}
/* if past the end of packet, ignore */
p_end = packet + TS_PACKET_SIZE;
if (p >= p_end)
    return 0;
pos = url_ftell(ts->stream->pb);
ts->pos47= pos % ts->raw_packet_size;
if (tss->type == MPEGTS_SECTION) {
    //Indicates that the current TS packet contains new business information
    segments
    if (is_start) {
        //Get the pointer field,
        //The new segment starts at the position indicated by the pointer
        field
        len = *p++;
        if (p + len > p_end)
            return 0;
        if (len && cc_ok) {
            //At this time, the TS load consists of two parts:
            //1)The position indicated by the pointer field starts from the TS
            load;
            //2)From the position indicated by the pointer field, the TS
            packet ends
                //1)The position represents the end of the previous segment.
                //2)The position represents the beginning of the new segment.
                //The following code saves part of the data at the end of the
                last segment, which is
                //1)Location data.
                write_section_data(s, tss, p, len, 0);
                /* check whether filter has been closed */
                if (!ts->pids[pid])

```

```

        return 0;
    }
    p += len;
    //Keep the new segment data, that is, the data of 2) location.
    if (p < p_end) {
        write_section_data(s, tss, p, p_end - p, 1);
    }
} else {
    //Save the data in the middle of the segment.
    if (cc_ok) {
        write_section_data(s, tss, p, p_end - p, 0);
    }
}
} else {
    int ret;
    //Normal PES data processing
    // Note: The position here points actually behind the current
packet.
    if ((ret = tss->u.pes_filter.pes_cb(tss, p, p_end - p, is_start,
                                         pos - ts->raw_packet_size)) < 0) return ret;
}
return 0;
}

static void write_section_data(AVFormatContext *s, MpegTSFilter *tss1,
                               const uint8_t *buf, int buf_size, int is_start)
{
    MpegTSSectionFilter *tss = &tss1->u.section_filter;
    int len;
    //In buf is the beginning of a segment.
    if (is_start) {
        //Copy the content to tss-> section_buf and save
        memcpy(tss->section_buf, buf, buf_size);
        //tss->section_index
        tss->section_index = buf_size;
        //The length of the segment, not yet known, is set to -1
        tss->section_h_size = -1;
        //Whether the end of the paragraph is reached.
    }
}

```

```

tss->end_of_section_reached = 0;
} else {
    //In buf is the data in the middle of the segment.
    if (tss->end_of_section_reached)
        return;
    len = 4096 - tss->section_index;
    if (buf_size < len)
        len = buf_size;
    memcpy(tss->section_buf + tss->section_index, buf, len);
    tss->section_index += len;
}
//If the conditions are met, calculate the length of the segment
if (tss->section_h_size == -1 && tss->section_index >= 3) {
    len = (AV_RB16(tss->section_buf + 1) & 0xffff) + 3;
    if (len > 4096)
        return;
    tss->section_h_size = len;
}
//Determine whether the segment data is collected. If the collection is
completed, call the corresponding callback function to process the
segment.
if (tss->section_h_size != -1 && tss->section_index >= tss-
>section_h_size) {
    tss->end_of_section_reached = 1;
    if (!tss->check_crc ||
        av_crc(av_crc_get_table(AV_CRC_32_IEEE), -1,
               tss->section_buf, tss->section_h_size) == 0)
        tss->section_cb(tss1, tss->section_buf, tss->section_h_size);
}
}

```

14. DEVICES

FFmpeg has a class library that interacts with multimedia devices: Libavdevice. Use this library to read data from multimedia devices on a computer (or other device), or output data to a specified multimedia device. Libavdevice supports the following devices as inputs:

alsa
avfoundation
bktr
dshow
dv1394
fbdev
gdigrab
iec61883
jack
lavfi
libcdio
libdc1394
openal
oss

pulse
qtkit
sndio
video4linux2, v4l2
vfwcap
x11grab
decklink

Libavdevice supports the following devices as outputs:

alsa
caca
decklink
fbdev
opengl
oss
pulse
sdl
sndio
xv

READ CAMERA

First of all, when using libavdevice, you need to include its header file:

```
#include "libavdevice/avdevice.h"
```

Then, you need to register libavdevice in the program:

```
avdevice_register_all();
```

Then you can use the functions of libavdevice.

Reading data using libavdevice is similar to directly opening a video file. Because the device of the system is also considered by FFmpeg as an input format (ie AVInputFormat). Use FFmpeg to open an ordinary video file using the following function:

```
AVFormatContext *pFormatCtx = avformat_alloc_context();
avformat_open_input(&pFormatCtx, "test.h265", NULL, NULL);
```

When using libavdevice, the only difference is that you need to find the device used for input first. Use `av_find_input_format()` here to complete:

```
AVFormatContext *pFormatCtx = avformat_alloc_context();
AVInputFormat *ifmt=av_find_input_format("vfwcap");
avformat_open_input(&pFormatCtx, 0, ifmt,NULL);
```

The above code first specifies the vfw device as the input device, and then specifies the 0th device to open in the URL (which is the camera device on my own computer).

In addition to using the vfw device as the input device on the Windows platform, you can also use DirectShow as the input device:

```
AVFormatContext *pFormatCtx = avformat_alloc_context();
AVInputFormat *ifmt=av_find_input_format("dshow");
avformat_open_input(&pFormatCtx,"video=Integrated
Camera",ifmt,NULL) ;
```

```
/***
*
* Simplest FFmpeg Device (Read Camera)
*
* This software read data from Computer's Camera and play it.
* It's the simplest example about usage of FFmpeg's libavdevice Library.
* It's suitable for the beginner of FFmpeg.
* This software support 2 methods to read camera in Microsoft Windows:
*   1.gdigrab: VfW (Video for Windows) capture input device.
*   The filename passed as input is the capture driver number,
*   ranging from 0 to 9.
*   2.dshow: Use Directshow. Camera's name in author's computer is
```

```
*      "Integrated Camera".
* It use video4linux2 to read Camera in Linux.
* It use avfoundation to read Camera in MacOS.
*
*/
#include <stdio.h>
#define __STDC_CONSTANT_MACROS

#ifndef _WIN32
//Windows
extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include "libavdevice/avdevice.h"
#include "SDL/SDL.h"
};
#else
//Linux...
#ifndef __cplusplus
extern "C"
{
#endif
#endif
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
#include <libavdevice/avdevice.h>
#include <SDL/SDL.h>
#ifndef __cplusplus
};
#endif
#endif
#endif

//Output YUV420P
#define OUTPUT_YUV420P 0
//1' Use Dshow
```

```

//'0' Use VFW
#define USE_DSHOW 0

//Refresh Event
#define SFM_REFRESH_EVENT (SDL_USEREVENT + 1)

#define SFM_BREAK_EVENT (SDL_USEREVENT + 2)

int thread_exit=0;

int sfp_refresh_thread(void *opaque)
{
    thread_exit=0;
    while (!thread_exit) {
        SDL_Event event;
        event.type = SFM_REFRESH_EVENT;
        SDL_PushEvent(&event);
        SDL_Delay(40);
    }
    thread_exit=0;
    //Break
    SDL_Event event;
    event.type = SFM_BREAK_EVENT;
    SDL_PushEvent(&event);

    return 0;
}

//Show Dshow Device
void show_dshow_device(){
    AVFormatContext *pFormatCtx = avformat_alloc_context();
    AVDictionary* options = NULL;
    av_dict_set(&options, "list_devices", "true", 0);
    AVInputFormat *iformat = av_find_input_format("dshow");
    printf("=====Device Info=====\\n");
    avformat_open_input(&pFormatCtx, "video=dummy", iformat, &options);
    printf("=====\\n");
}

```

```

//Show Dshow Device Option
void show_dshow_device_option(){
    AVFormatContext *pFormatCtx = avformat_alloc_context();
    AVDictionary* options = NULL;
    av_dict_set(&options, "list_options", "true", 0);
    AVInputFormat *iformat = av_find_input_format("dshow");
    printf("=====Device Option Info=====\n");
    avformat_open_input(&pFormatCtx, "video=Integrated
Camera", iformat, &options);
    printf("======\n");
}

//Show VFW Device
void show_vfw_device(){
    AVFormatContext *pFormatCtx = avformat_alloc_context();
    AVInputFormat *iformat = av_find_input_format("vfwcap");
    printf("=====VFW Device Info=====\n");
    avformat_open_input(&pFormatCtx, "list", iformat, NULL);
    printf("======\n");
}

//Show AVFoundation Device
void show_avfoundation_device(){
    AVFormatContext *pFormatCtx = avformat_alloc_context();
    AVDictionary* options = NULL;
    av_dict_set(&options, "list_devices", "true", 0);
    AVInputFormat *iformat = av_find_input_format("avfoundation");
    printf("==AVFoundation Device Info===\n");
    avformat_open_input(&pFormatCtx, "", iformat, &options);
    printf("======\n");
}

int main(int argc, char* argv[])
{
    AVFormatContext *pFormatCtx;
    int          i, videoindex;
    AVCodecContext *pCodecCtx;

```

```

AVCodec      *pCodec;
av_register_all();
avformat_network_init();
pFormatCtx = avformat_alloc_context();
//Open File
//char filepath[]="src01_480x272_22.h265";
//avformat_open_input(&pFormatCtx,filepath,NULL,NULL)

//Register Device
avdevice_register_all();

//Windows
#define _WIN32

//Show Dshow Device
show_dshow_device();
//Show Device Options
show_dshow_device_option();
//Show VFW Options
show_vfw_device();

#if USE_DSHOW
    AVInputFormat *ifmt=av_find_input_format("dshow");
    //Set own video device's name
    if(avformat_open_input(&pFormatCtx,"video=Integrated
Camera",ifmt,NULL)!=0){
        printf("Couldn't open input stream.\n");
        return -1;
    }
#else
    AVInputFormat *ifmt=av_find_input_format("vfwcap");
    if(avformat_open_input(&pFormatCtx,"0",ifmt,NULL)!=0){
        printf("Couldn't open input stream.\n");
        return -1;
    }
#endif
#elif defined linux
//Linux

```

```

AVInputFormat *ifmt=av_find_input_format("video4linux2");
if(avformat_open_input(&pFormatCtx,"/dev/video0",ifmt,NULL)!=0)
{
    printf("Couldn't open input stream.\n");
    return -1;
}
#else
show_avfoundation_device();
//Mac
AVInputFormat *ifmt=av_find_input_format("avfoundation");
//Avfoundation
//[video]:[audio]
if(avformat_open_input(&pFormatCtx,"0",ifmt,NULL)!=0){
    printf("Couldn't open input stream.\n");
    return -1;
}
#endif

if(avformat_find_stream_info(pFormatCtx,NULL)<0)
{
    printf("Couldn't find stream information.\n");
    return -1;
}
videoindex=-1;
for(i=0; i<pFormatCtx->nb_streams; i++)
if(pFormatCtx->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_VIDEO)
{
    videoindex=i;
    break;
}
if(videoindex==-1)
{
    printf("Couldn't find a video stream.\n");
    return -1;
}
pCodecCtx=pFormatCtx->streams[videoindex]->codec;

```

```

pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL)
{
printf("Codec not found.\n");
return -1;
}
if(avcodec_open2(pCodecCtx, pCodec,NULL)<0)
{
printf("Could not open codec.\n");
return -1;
}
AVFrame *pFrame,*pFrameYUV;
pFrame=av_frame_alloc();
pFrameYUV=av_frame_alloc();
//unsigned char *out_buffer=(unsigned char
*)av_malloc(avpicture_get_size(AV_PIX_FMT_YUV420P, pCodecCtx-
>width, pCodecCtx->height));
    //avpicture_fill((AVPicture *)pFrameYUV, out_buffer,
AV_PIX_FMT_YUV420P, pCodecCtx->width, pCodecCtx->height);
//SDL-----
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO |
SDL_INIT_TIMER)) {
printf( "Could not initialize SDL - %s\n", SDL_GetError());
return -1;
}
int screen_w=0,screen_h=0;
SDL_Surface *screen;
screen_w = pCodecCtx->width;
screen_h = pCodecCtx->height;
screen = SDL_SetVideoMode(screen_w, screen_h, 0,0);

if(!screen) {
printf("SDL: could not set video mode -
exiting:%s\n",SDL_GetError());
return -1;
}
SDL_Overlay *bmp;

```

```

    bmp = SDL_CreateYUVOverlay(pCodecCtx->width, pCodecCtx-
>height,SDL_YV12_OVERLAY, screen);
    SDL_Rect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = screen_w;
    rect.h = screen_h;
//SDL End-----
    int ret, got_picture;

    AVPacket *packet=(AVPacket *)av_malloc(sizeof(AVPacket));

#ifndef OUTPUT_YUV420P
    FILE *fp_yuv=fopen("output.yuv","wb+");
#endif

    struct SwsContext *img_convert_ctx;
    img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx-
>height, pCodecCtx->pix_fmt, pCodecCtx->width, pCodecCtx->height,
AV_PIX_FMT_YUV420P, SWS_BICUBIC, NULL, NULL, NULL);
//-----
    SDL_Thread *video_tid =
SDL_CreateThread(sf_refresh_thread,NULL);
//
    SDL_WM_SetCaption("Simplest FFmpeg Read Camera",NULL);
//Event Loop
    SDL_Event event;

    for (;;) {
//Wait
    SDL_WaitEvent(&event);
    if(event.type==SFM_REFRESH_EVENT){
//-----
    if(av_read_frame(pFormatCtx, packet)>=0){
        if(packet->stream_index==videoindex){
            ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
packet);
            if(ret < 0){

```

```

printf("Decode Error.\n");
return -1;
}
if(got_picture){
SDL_LockYUVOverlay(bmp);
pFrameYUV->data[0]=bmp->pixels[0];
pFrameYUV->data[1]=bmp->pixels[2];
pFrameYUV->data[2]=bmp->pixels[1];
pFrameYUV->linesize[0]=bmp->pitches[0];
pFrameYUV->linesize[1]=bmp->pitches[2];
pFrameYUV->linesize[2]=bmp->pitches[1];
sws_scale(img_convert_ctx, (const unsigned char* const*)pFrame-
>data, pFrame->linesize, 0, pCodecCtx->height, pFrameYUV->data,
pFrameYUV->linesize);

#if OUTPUT_YUV420P
int y_size=pCodecCtx->width*pCodecCtx->height;
fwrite(pFrameYUV->data[0],1,y_size,fp_yuv); //Y
fwrite(pFrameYUV->data[1],1,y_size/4,fp_yuv); //U
fwrite(pFrameYUV->data[2],1,y_size/4,fp_yuv); //V
#endif

SDL_UnlockYUVOverlay(bmp);
SDL_DisplayYUVOverlay(bmp, &rect);

}
}

av_free_packet(packet);
}else{
//Exit Thread
thread_exit=1;
}
}else if(event.type==SDL_QUIT){
thread_exit=1;
}else if(event.type==SFM_BREAK_EVENT){
break;
}

```

```
}

sws_freeContext(img_convert_ctx);

#if OUTPUT_YUV420P
    fclose(fp_yuv);
#endif

SDL_Quit();

av_free(pFrameYUV);
avcodec_close(pCodecCtx);
avformat_close_input(&pFormatCtx);

return 0;
}
```

SCREEN CAPTURE

Here are two ways to grab screen data using libavdevice on Windows systems: gdigrab and dshow. Introduced separately below.

1. gdigrab

“gdigrab” is a device used by FFmpeg to capture Windows desktop. Very suitable for screen recording. It supports two ways of crawling through different input URLs:

- "desktop": grab the entire desktop. Or grab a specific area on the desktop.
- "title = {window name)": grab a specific window in the screen.
- gdigrab also supports some parameters for setting the position of the screen capture :
- offset_x: the horizontal coordinate of the starting point of the screen capture.
- offset_y: the longitudinal coordinate of the starting point of the screen capture.
- video_size: the size of the screen capture.
- framerate: Frame rate of screen capture.

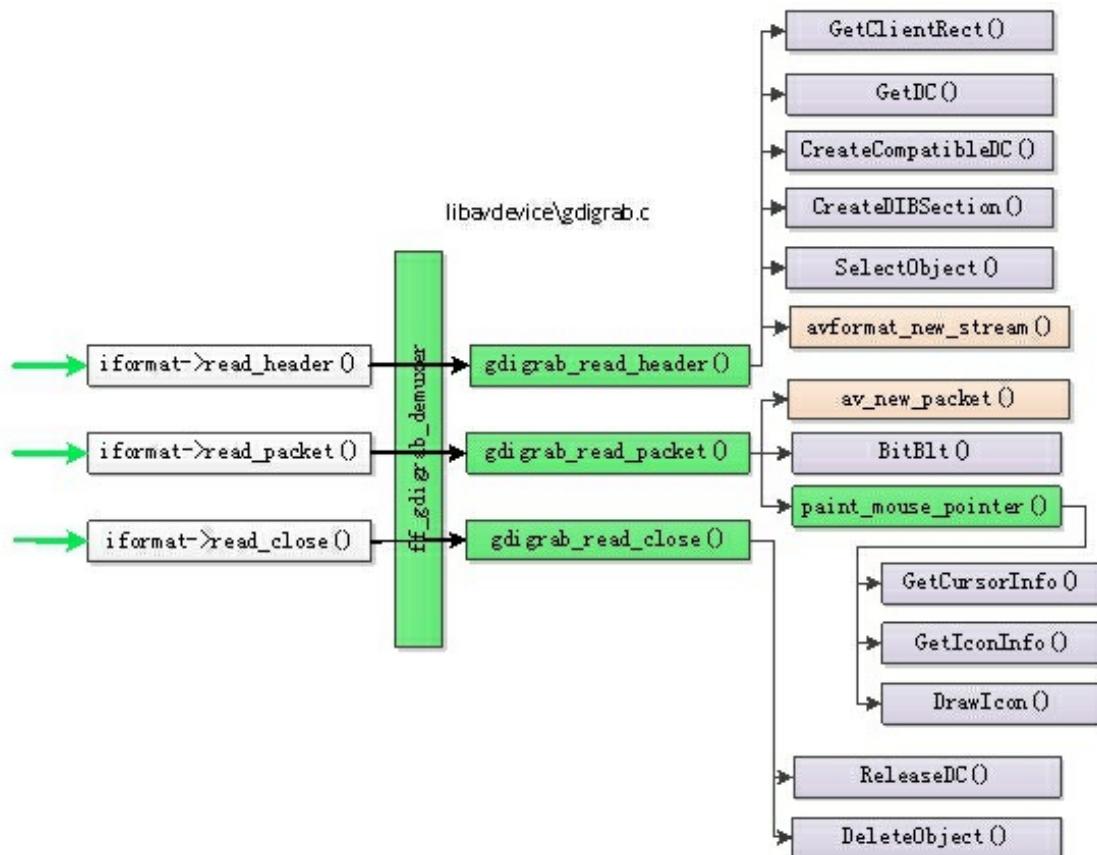
```

//Use gdigrab
AVDictionary* options = NULL;
//Set some options
//grabbing frame rate
//av_dict_set(&options,"framerate","5",0);
//The distance from the left edge of the screen or desktop
//av_dict_set(&options,"offset_x","20",0);
//The distance from the top edge of the screen or desktop
//av_dict_set(&options,"offset_y","40",0);
//Video frame size. The default is to capture the full screen
//av_dict_set(&options,"video_size","640x480",0);
AVInputFormat *ifmt=av_find_input_format("gdigrab");
if(avformat_open_input(&pFormatCtx,"desktop",ifmt,&options)!=0){
printf("Couldn't open input stream.\n");
return -1;
}

```

GDIGrab is used for screen recording (screen capture) under Windows.

The source code of gdigrab is located at libavdevice \ gdigrab.c. The call relationship diagram of key functions is shown in the figure below. The functions with green background in the figure represent the functions declared in the source code, and the functions with purple background represent the API functions of Win32.



2. THE DSHOW

Screen capture software needs to be installed when using dshow screen capture: screen-capture-recorder software address:

<http://sourceforge.net/projects/screencapturer/>

After the download software is installed, you can specify the input device of dshow as "screen-capture" - recorder ". The reference code is as follows:

```
AVInputFormat *ifmt=av_find_input_format("dshow");
if(avformat_open_input(&pFormatCtx,"video=screen-capture-
recorder",ifmt,NULL)!=0){
    printf("Couldn't open input stream.\n");
    return -1;
}
```

Screen capture sample:

```
/**
 * Simplest FFmpeg Device (Screen Capture)
 *
 * This software capture screen of computer. It's the simplest example
 * about usage of FFmpeg's libavdevice Library.
 * It's suitable for the beginner of FFmpeg.
 * This software support 2 methods to capture screen in Microsoft
 Windows:
 * 1.gdigrab: Win32 GDI-based screen capture device.
```

```
* Input URL in avformat_open_input() is "desktop".
* 2.dshow: Use Directshow. Need to install screen-capture-recorder.
* It use x11grab to capture screen in Linux.
* It use avfoundation to capture screen in MacOS.
*/
#include <stdio.h>
#define __STDC_CONSTANT_MACROS
#ifdef _WIN32
//Windows
extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libswscale/swscale.h"
#include "libavdevice/avdevice.h"
#include "SDL/SDL.h"
};
#else
//Linux...
#ifdef __cplusplus
extern "C"
{
#endif
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
#include <libavdevice/avdevice.h>
#include <SDL/SDL.h>
#ifdef __cplusplus
};
#endif
#endif
//Output YUV420P
#define OUTPUT_YUV420P 0
//'1' Use Dshow
//'0' Use GDIgrab
```

```

#define USE_DSHOW 0

//Refresh Event
#define SFM_REFRESH_EVENT (SDL_USEREVENT + 1)

#define SFM_BREAK_EVENT (SDL_USEREVENT + 2)

int thread_exit=0;

int sfp_refresh_thread(void *opaque)
{
    thread_exit=0;
    while (!thread_exit) {
        SDL_Event event;
        event.type = SFM_REFRESH_EVENT;
        SDL_PushEvent(&event);
        SDL_Delay(40);
    }
    thread_exit=0;
    //Break
    SDL_Event event;
    event.type = SFM_BREAK_EVENT;
    SDL_PushEvent(&event);

    return 0;
}

//Show Dshow Device
void show_dshow_device(){
    AVFormatContext *pFormatCtx = avformat_alloc_context();
    AVDictionary* options = NULL;
    av_dict_set(&options,"list_devices","true",0);
    AVInputFormat *iformat = av_find_input_format("dshow");
    printf("=====Device Info=====\\n");
    avformat_open_input(&pFormatCtx,"video=dummy",iformat,&options);
    printf("=====\\n");
}

//Show AVFoundation Device

```

```

void show_avfoundation_device(){
    AVFormatContext *pFormatCtx = avformat_alloc_context();
    AVDictionary* options = NULL;
    av_dict_set(&options, "list_devices", "true", 0);
    AVInputFormat *iformat = av_find_input_format("avfoundation");
    printf("==AVFoundation Device Info===\n");
    avformat_open_input(&pFormatCtx, "", iformat, &options);
    printf("======\n");
}

int main(int argc, char* argv[])
{
    AVFormatContext *pFormatCtx;
    int i, videoindex;
    AVCodecContext *pCodecCtx;
    AVCodec *pCodec;
    av_register_all();
    avformat_network_init();
    pFormatCtx = avformat_alloc_context();
    //Open File
    //char filepath[]="src01_480x272_22.h265";
    //avformat_open_input(&pFormatCtx,filepath,NULL,NULL)

    //Register Device
    avdevice_register_all();
    //Windows
#ifndef _WIN32
#if USE_DSHOW
    //Use dshow
    //
    //Need to Install screen-capture-recorder
    //screen-capture-recorder
    //Website: http://sourceforge.net/projects/screencapturer/
    //
    AVInputFormat *ifmt=av_find_input_format("dshow");

```

```

    if(avformat_open_input(&pFormatCtx, "video=screen-capture-
recorder", ifmt, NULL)!=0){
        printf("Couldn't open input stream.\n");
        return -1;
    }
#else
//Use gdigrab
AVDictionary* options = NULL;
//Set some options
//grabbing frame rate
//av_dict_set(&options,"framerate","5",0);
//The distance from the left edge of the screen or desktop
//av_dict_set(&options,"offset_x","20",0);
//The distance from the top edge of the screen or desktop
//av_dict_set(&options,"offset_y","40",0);
//Video frame size. The default is to capture the full screen
//av_dict_set(&options,"video_size","640x480",0);
AVInputFormat *ifmt=av_find_input_format("gdigrab");
if(avformat_open_input(&pFormatCtx, "desktop", ifmt, &options)!=0)
{
    printf("Couldn't open input stream.\n");
    return -1;
}

#endif
#elif defined linux
//Linux
AVDictionary* options = NULL;
//Set some options
//grabbing frame rate
//av_dict_set(&options,"framerate","5",0);
//Make the grabbed area follow the mouse
//av_dict_set(&options,"follow_mouse","centered",0);
//Video frame size. The default is to capture the full screen
//av_dict_set(&options,"video_size","640x480",0);
AVInputFormat *ifmt=av_find_input_format("x11grab");
//Grab at position 10,20

```

```

    if(avformat_open_input(&pFormatCtx,":0.0+10,20",ifmt,&options)!=0)
{
    printf("Couldn't open input stream.\n");
    return -1;
}
#endif

show_avfoundation_device();
//Mac
AVInputFormat *ifmt=av_find_input_format("avfoundation");
//Avfoundation
//[video]:[audio]
if(avformat_open_input(&pFormatCtx,"1",ifmt,NULL)!=0){
    printf("Couldn't open input stream.\n");
    return -1;
}
#endif

if(avformat_find_stream_info(pFormatCtx,NULL)<0)
{
    printf("Couldn't find stream information.\n");
    return -1;
}
videoindex=-1;
for(i=0; i<pFormatCtx->nb_streams; i++)
if(pFormatCtx->streams[i]->codec-
>codec_type==AVMEDIA_TYPE_VIDEO)
{
    videoindex=i;
    break;
}
if(videoindex==-1)
{
    printf("Didn't find a video stream.\n");
    return -1;
}
pCodecCtx=pFormatCtx->streams[videoindex]->codec;
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);

```

```

if(pCodec==NULL)
{
printf("Codec not found.\n");
return -1;
}
if(avcodec_open2(pCodecCtx, pCodec,NULL)<0)
{
printf("Could not open codec.\n");
return -1;
}
AVFrame *pFrame,*pFrameYUV;
pFrame=av_frame_alloc();
pFrameYUV=av_frame_alloc();
//unsigned char *out_buffer=(unsigned char
*)av_malloc(avpicture_get_size(AV_PIX_FMT_YUV420P, pCodecCtx-
>width, pCodecCtx->height));
//avpicture_fill((AVPicture *)pFrameYUV, out_buffer,
AV_PIX_FMT_YUV420P, pCodecCtx->width, pCodecCtx->height);
//SDL-----
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO |
SDL_INIT_TIMER)) {
printf( "Could not initialize SDL - %s\n", SDL_GetError());
return -1;
}
int screen_w=640,screen_h=360;
const SDL_VideoInfo *vi = SDL_GetVideoInfo();
//Half of the Desktop's width and height.
screen_w = vi->current_w/2;
screen_h = vi->current_h/2;
SDL_Surface *screen;
screen = SDL_SetVideoMode(screen_w, screen_h, 0,0);

if(!screen) {
printf("SDL: could not set video mode -
exiting:%s\n",SDL_GetError());
return -1;
}

```

```

SDL_Overlay *bmp;
bmp = SDL_CreateYUVOverlay(pCodecCtx->width, pCodecCtx-
>height,SDL_YV12_OVERLAY, screen);
SDL_Rect rect;
rect.x = 0;
rect.y = 0;
rect.w = screen_w;
rect.h = screen_h;
//SDL End-----
int ret, got_picture;

AVPacket *packet=(AVPacket *)av_malloc(sizeof(AVPacket));

#if OUTPUT_YUV420P
FILE *fp_yuv=fopen("output.yuv","wb+");
#endif

struct SwsContext *img_convert_ctx;
img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx-
>height, pCodecCtx->pix_fmt, pCodecCtx->width, pCodecCtx->height,
AV_PIX_FMT_YUV420P, SWS_BICUBIC, NULL, NULL, NULL);
//-----
SDL_Thread *video_tid =
SDL_CreateThread(sfp_refresh_thread,NULL);
//
SDL_WM_SetCaption("Simplest FFmpeg Grab Desktop",NULL);
//Event Loop
SDL_Event event;

for (;;) {
//Wait
SDL_WaitEvent(&event);
if(event.type==SFM_REFRESH_EVENT){
//-----
if(av_read_frame(pFormatCtx, packet)>=0){
if(packet->stream_index==videoindex){
ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
packet);
}
}
}
}

```

```

if(ret < 0){
printf("Decode Error.\n");
return -1;
}
if(got_picture){
SDL_LockYUVOverlay(bmp);
pFrameYUV->data[0]=bmp->pixels[0];
pFrameYUV->data[1]=bmp->pixels[2];
pFrameYUV->data[2]=bmp->pixels[1];
pFrameYUV->linesize[0]=bmp->pitches[0];
pFrameYUV->linesize[1]=bmp->pitches[2];
pFrameYUV->linesize[2]=bmp->pitches[1];
sws_scale(img_convert_ctx, (const unsigned char* const*)pFrame-
>data, pFrame->linesize, 0, pCodecCtx->height, pFrameYUV->data,
pFrameYUV->linesize);

#if OUTPUT_YUV420P
int y_size=pCodecCtx->width*pCodecCtx->height;
fwrite(pFrameYUV->data[0],1,y_size,fp_yuv); //Y
fwrite(pFrameYUV->data[1],1,y_size/4,fp_yuv); //U
fwrite(pFrameYUV->data[2],1,y_size/4,fp_yuv); //V
#endif

SDL_UnlockYUVOverlay(bmp);
SDL_DisplayYUVOverlay(bmp, &rect);

}
}

av_free_packet(packet);
}else{
//Exit Thread
thread_exit=1;
}
}else if(event.type==SDL_QUIT){
thread_exit=1;
}else if(event.type==SFM_BREAK_EVENT){
break;
}

```

```
}

    sws_freeContext(img_convert_ctx);

#if OUTPUT_YUV420P
    fclose(fp_yuv);
#endif

    SDL_Quit();

//av_free(out_buffer);
av_free(pFrameYUV);
avcodec_close(pCodecCtx);
avformat_close_input(&pFormatCtx);

return 0;
}
```

REFERENCES

<https://blog.csdn.net>

https://blog.csdn.net/li_wen01/

<https://trac.ffmpeg.org/wiki>

<http://www.ffmpeg.org/documentation.html>

<http://www.ffmpeg.org>

<http://chunlin.li/tech/doku.php/tech:multimedia:ffmpeg>

<http://bbs.chinaffmpeg.com/>

<https://blog.csdn.net/leixiaohua1020/>

<https://blog.csdn.net/crazyman2010/>

http://blog.163.com/liukang_0404@126/blog/

<https://zhuanlan.zhihu.com/>

<https://www.cnblogs.com/ranson7zop/>

<http://trace.eas.asu.edu/yuv/>

<https://en.wikipedia.org/wiki>

<https://write.corbpie.com/adding-audio-to-a-video-file-with-ffmpeg/>

<https://slhck.info/ffmpeg-encoding-course/#/13>

<http://www.zhiboyun.com/zh/document/newbie/concep>

<https://blog.csdn.net/goldfish3/article>

ABOUT THE AUTHOR

Akın Yaprakgül is a computer engineer and his work includes programming, Broadcast and Tv media softwares like Ingest, Character generator, Channel in a box software. He graduated from Karadeniz Technical University, Computer Engineering department. He worked for 2 years on image processing in Ankara in 2010. Then in 2013, he stepped into the broadcast industry in Istanbul and has been working in the broadcast industry until today. He is married and has one child.