FFMPEG/FFPLAY源 码 剖 析

作 者：杨 书 良

前 言

古有"民以食为天"，今有"民以玩为天"，当今各种各样的电子产品的影音娱乐功能越来越强悍，或者说影音娱乐功能推着各种各样的电子产品大踏步前进，于是很多有心人开始研究学习当今世界上开源多媒体领域的老祖宗和超级王者 ffmpeg。Linux 平台上就不用说了，ffmpeg 一统天下，虽然也有一些其他的类似产品问世，但都是使用 ffmpeg 的内核，外挂包装层或外挂界面层的系统架构，无一例外。Windows 平台上虽然有其他的类似产品，但是使用 ffmpeg 内核的知名产品也不少，还有一些不幸上了 ffmpeg 的耻辱名单。

站在巨人的肩膀上，自然看得更高看得更远，但是 ffmpeg 是一个非常庞大的系统，几乎实现了当今世界上所有的多媒体编码解码，通吃所有的影音娱乐媒体文件，自然而然地，fmpeg 定义的结构繁多，并且定义的结构多半都是超级复杂，想要透彻理解并站上 ffmpeg 的肩膀非常的不容易。

虽然在网上可以找到一些开发参考资料，但是多半仅停留在源代码的编译步骤，API 的使用介绍，简单的移植方法，还有一些原理性的介绍。当然有总比没有强，但是看着那些资料，总是一种隔靴搔痒的感觉，虽然也可以编译成功，也可以播放一些影音媒体文件，或者还可以做一个调用 ffmpeg dll 文件或者 exe 文件的客户端，做来做去最多也就仅仅只做一个简单的外包装。仅停留在外包装层面上，对 ffmpeg 本身的实现代码并没有什么深入理解。

本书深入 ffmpeg 的内核，在源代码的实现水平上来理解它，是目前国内第一本详细讲解 ffmpeg 源代码方面的书，填补这方面的空白。由于 ffmpeg 的体系结构过于庞大，并且很多视音频编码解码算法都归各商业公司私有，从公共的途径很难获得相关资料，因此本书精挑细选只保留一个视频算法和一个音频算法，其他的视音频算法都删掉，并且只关注解码而删掉编码相关的代码。这样瘦身处理后， ffmpeg 是相当的苗条，除了视音频编解码算法的具体实现外，其他的都保留，仍不失其完整性。这样不仅可以学习理解 ffmpeg 的精华，还大大降低了学习的门槛，可以节省相关工程师 6 个月到 3 个月的研习时间。

附带的源代码在 VS2005 和 VC6 上编译通过，但注意不要开启 VC6 的优化功能或者打 VC6 的补丁。

附带的测试文件因为原始媒体文件分辨率为 321x321 太怪怪，是作者经过比较复杂的流程转压缩转压缩 为 320x320 分辨率(对此媒体文件，常规的转压缩方法失真太大)。

本书为准备研究学习 ffmpeg 的读者而写，需要读者了解一些播放器相关的概念，了解 C 语言，一些阅读源码的功底。本书不严格区分 ffmpeg 和 ffplay，ffplay 有时特指瘦身后的 ffmpeg，有时候混用。

由于作者才学疏浅，ffmpeg 超级庞大复杂，理解难免有误，还请各位网友热心指正。

杨书良 20110110 上海

版权说明

第一章 概述

1.1 ffplay 文件概览

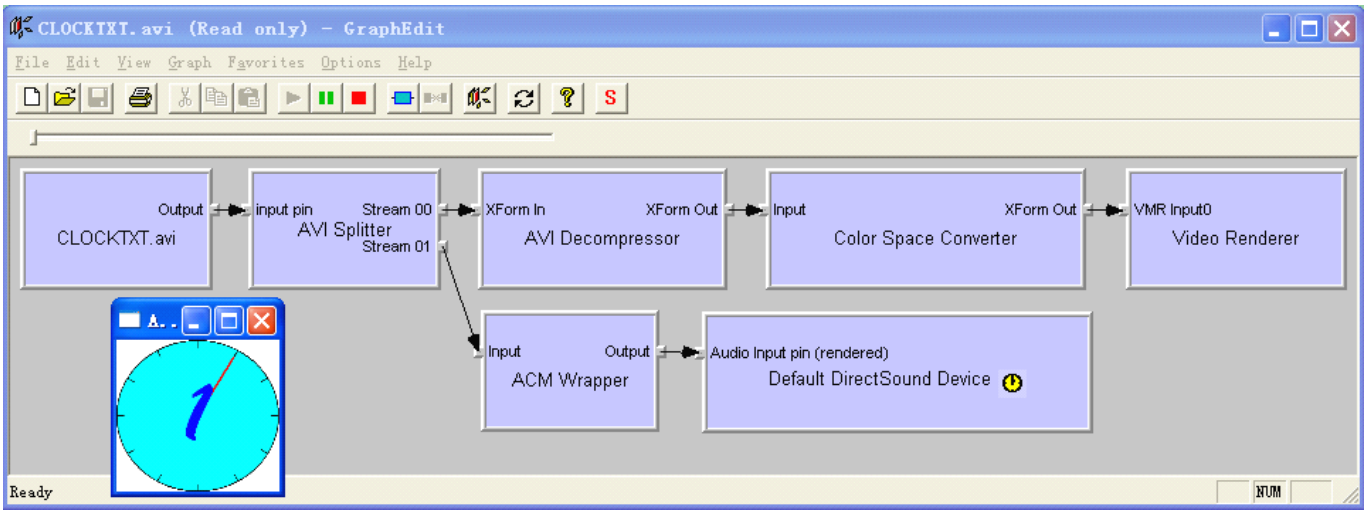
首先我们来暴增一下研习信心。ffmpeg 经作者瘦身处理后的，但保持了原有的架构和完整性，经统计仅包括 26 个.h 和.c 文件，约 5600 行代码，总大小约 176k，相对于原来约十兆的源码包，那可是不一般的小巧，研习的信心是不是超级暴增？

我们再来扫描一下所有目录和所有文件，目录结构和原来的一样，只是每个目录下的文件删减了很多，如下图所示，右边是对每个文件的简单描述。由此，对 ffmpeg 有一个整体的感官，其实 ffmpeg 也可以很小巧，走捷径研习也可以很简单，并不是想象中的那么难。

1.2 播放器一般原理

再接再厉，我们开始学习解码器通用的一些原理性知识，用 Windows DirectShow 的工具 GraphEdit 打开附带的测试文件，可以直观的看到播放这个媒体文件的基本模块，七个模块按广度顺序：读文件模块，解复用模块，视频解码模块，音频解码音频，颜色空间转换模块，视频显示模块，音频播放模块。

按照 DirectShow 的术语，一个模块叫做一个 filter(过滤器)，模块的输入输出口叫做 pin(管脚)，有 input pin 和 output pin 两种；第一个 filter 叫做 Source filter，每种媒体最后一个 filter 叫做 Sink filter，象下图所示连成串的所有 filter 组成一个 Graph。媒体文件的数据就像流水一样在 Graph 中流动，各个相关的 filter 各负其责，最后我们就看到了视频，也听到了声音。



DirectShow 中和播放器有关的 filter 粗略的分为五类，分别是 Source filer, Demux flter, Decoder filter, Color Space converter filter，Render filter，各类 filter 的功能与作用简述如下：

Source filter 源过滤器的作用是为下级 demux filter 以包的形式源源不断的提供数据流。在通常情况下，我们有多种方式可以获得数据流，一种是从本地文件中读取，一种是从网上获取，Sourcefilter 另外一个作用就是屏蔽读本地文件和获取网络数据的差别，在下一级的 demux filter 看来，本地文件和网络数据是一样的。

Demux filter 解复用过滤器的作用是识别文件类型，媒体类型，分离出各媒体原始数据流，打上时钟信息后送给下级 decoder filter。为识别出不同的文件类型和媒体类型，常规的做法是读取一部分数据，然后遍历解复用过滤器支持的文件格式和媒体数据格式，做匹配来确定是哪种文件类型，哪种媒体类型；有些媒体类型的原始数据外面还有其他的信息，比如时间，包大小，是否完整包等等。 demux filter 解析数据包后取出原始数据，有些类型的媒体不管是否是完整包都立即送往下级 decoder filter，有些类型的媒体要送完整数据包，此时可能有一些数据包拼接的动作；当然时钟信息的计算也是 demux filter 的工作内容，这个时钟用于各媒体之间的同步。在本

例中，AVI Splitter 是 Demux filter。

Decoder filter 解码过滤器的作用就是解码数据包，并且把同步时钟信息传递下去。对视频媒体而言，通常是解码成 YUV 数据，然后利用显卡硬件直接支持 YUV 格式数据 Overlay 快速显示的特性让显卡极速显示。YUV 格式是一个统称，常见的有 YV12，YUY2，UYVY 等等。有些非常古老的显卡和嵌入式系统不支持 YUV 数据显示，那就要转换成 RGB 格式的数据，每一帧的每一个像素点都要转换，分别计算 RGB 分量，并且因为转换是浮点运算，虽然有定点算法，还是要耗掉相当一部分 CPU，总体上效率底下；对音频媒体而言，通常是解码成 PCM 数据，然后送给声卡直接输出。在本例中，AVI Decompress 和 ACM Warper 是 decoder filter。

Color space converter filter 颜色空间转换过滤器的作用是把视频解码器解码出来的数据转换成当前显示系统支持的颜色格式。通常视频解码器解码出来的是 YUV 数据，PC 系统是直接支持 YUV 格式的，也支持 RGB 格式，有些嵌入式系统只支持 RGB 格式的。在本例中，视频解码器解码出来的是 RGB8 格式的数据，Color space converter filter 把 RGB8 转换成 RGB32 显示。

Render filter 渲染过滤器的作用就是在适当的时间渲染相应的媒体，对视频媒体就是直接显示图像，对音频就是播放声音。视音频同步的策略方法有好几种，其中最简单的一种就是默认视频和音频基准时间相同，这时音频可以不打时钟信息，通过计算音频的采样频率，量化 bit 数，声道数等基本参数就知道音频 PCM 的数据速率，按照这个速率往前播放即可；视频必须要使用同步时钟信息来决定什么时候显示。DirectShow 采用一个有序链表，把接收到的数据包放进有序链表中，启动一个定时器，每次定时器时间到就扫描链表，比较时钟信息，或者显示相应的帧，或者什么也不做，每次接收到新的数据帧，首先判断时钟信息，如果是历史数据帧就丢弃，如果是将来显示数据帧就进有序链表，如果当前时间帧就直接显示。如此这样，保持视频和音频在人体感觉误差范围内相对的动态同步。在本例中 VideoRender 和 Default DirectSound Device 是 Render filter，同时也是 Sink filter。

GraphEdit 应用程序可以看成是一个支撑平台，支撑框架。它容纳各种 filter，在 filter 间的传递一些通讯消息，控制 filter 的运行(启动暂停停止)，维护 filter 运行状态。GraphEdit 就象超级大管家一样，既维护管理看得见的 filter，又维护管理看不见的运行支撑环境。

1.3 ffplay 播放器原理

ffplay 播放器从原理上来讲和 windows directshow 差不多，只是没有使用 driectshow 那些名词术语来表述。从 directshow 的视角来看 ffplay 播放器，简单的划分一下模块和各个模块的文件，有些文件可能在多个模块中都有用到，只能不严格的划分。

Source filter 读文件模块，可以简单的分为 3 层，最底层的是 file，pipe，tcp，udp，http 等这些具体的本地文

件或网络协议(注意 ffplay 把 file 也当协议看待)；中间抽象层用 URLContext 结构来统一表示底层具体的本地文件或网络协议，相关操作也只是简单的中转一下调用底层具体文件或协议的支撑函数；最上层用 ByteIOContext 结构来扩展 URLProtocol 结构成内部有缓冲机制的广泛意义上的文件，并且仅仅由 ByteIOContext 对模块外提供服务。此模块主要有 libavformat 目录下的 file.c,avio.h, avio.c, aviobuf.c 等文件，实现读媒体文件功能。

Demux filter 解复用模块，可以简单的分为两层，底层是 AVIContext，TCPContext，UDPContext 等等这些具体媒体的解复用结构和相关的基础程序，上层是 AVInputFormat 结构和相关的程序。上下层之间由 AVInputFormat 相对应的 AVFormatContext 结构的 priv_data 字段关联 AVIContext 或 TCPContext 或 UDPContext 等等具体的文件格式。AVInputFormat 和具体的音视频编码算法格式由 AVFormatContext 结构的 streams 字段关联媒体格式，streams 相当于 demux filter 的 output pin，解复用模块分离音视频裸数据通过 streams 传递给下级音视频解码器。此模块主要有 libavformat 目录下的 avidec.c，utils_format.c 文件。

Decoder filter 解码模块，也可以简单的分为两层，由 AVCodec 统一表述。上层是 AVCodec 对应的 AVCodecContext 结构和相关的基础程序，底层是 TSContext，MsrleContext 等等这些具体的编解码器内部使用的结构和相关的基础程序。AVCodecContext 结构的 priv_data 字段关联 TSContext，MsrleContext 等等具体解码器上下文。此模块主要有 libavcodec 目录下的 msrle.c，truespeech.c，truespeech_data.h，utils_codec.c 等文件。

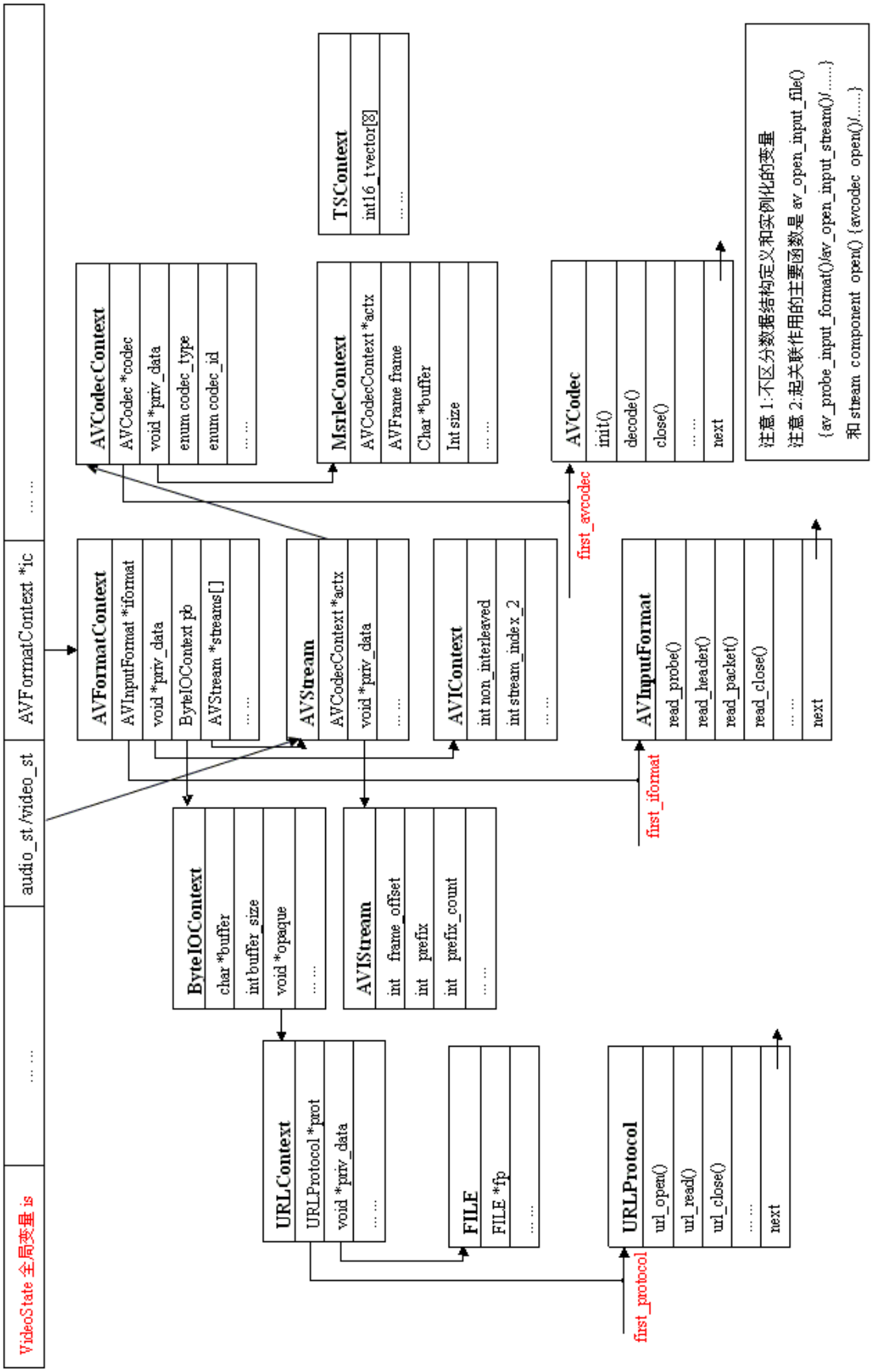
Color space converter 颜色空间转换模块，大多数的 decoder filter 解码出来的数据是 YUV 颜色空间的数据，并不是所有系统都支持的 YUV 颜色空间。如果不匹配，就需要做颜色空间转换，但是目前 PC 上无论是独立显卡还是集成显卡都直接支持 YUV 颜色空间显示。有些视频解码器或视频渲染模块含有颜色空间转换功能，这时就不需要独立的颜色空间转换模块。在本例中，颜色空间转换模块转换 RGB8 格式到 RGB32 格式直接显示。此模块主要有 libavcodec 目录下的 imgconvert.c，imgconvert_template.h 文件。

Render filter 渲染模块，ffplay 中的渲染模块使用 SDL 库，由 SDL 库帮我们显示视频，播放音频。我们只需要会用 SDL 库中的函数就可以了，具体的渲染动作可以不用理；如果确实想看 SDL 库是怎样运作的，那就把 SDL 库的源代码找出来，潜心研读，关于这部分实现的讨论不在本书范围内。

ffplay.c 文件相当于 directShow 里面的 GraphEdit，可以简单理解成一个支撑大平台，支撑大框架。它声明并实例化其他的数据结构/模块，并且把其他数据结构/模块串联起来，最后把其他数据结构/模块关联到 VideoState 数据结构，同时控制程序的启动暂停停止等。

1.4 ffplay 架构概述

纵观瘦身后的 ffplay，我们从它定义的数据结构来理解它的架构，首先看一下主要数据结构之间的关联图。


VideoState 全局变量 is
audio_st /video_st
AVFormatContext *ic
AVFormatContext
AVInputFormat *iformat
void *priv_data
ByteIOContext pb
AVStream *streams[]
AVStream
AVCodecContext *actx
void *priv_data
AVIContext
int non_interleaved
int stream_index_2
AVInputFormat
read_probe()
read_header()
read_packet()
read_close()
next
first_iformat
AVCodecContext
AVCodec *codec
void *priv_data
enum codec_type
enum codec_id
MsrleContext
AVCodecContext *actx
AVFrame frame
Char *buffer
Int size
TSContext
int16_t vector[8]
AVCodec
init()
decode()
close()
next
first_avcodec
注意 1:不区分数据结构定义和实例化的变量
注意 2:起关联作用的主要函数是 av_open_input_file()
{av_probe_input_format()/av_open_input_stream()/.....}
和 stream component open() {avccodec open()/.....}
ByteIOContext
char *buffer
int buffer_size
void *opaque
AVIStream
int frame_offset
int prefix
int prefix_count
URLContext
URLProtocol *prot
void *priv_data
FILE
FILE *fp
URLProtocol
url_open()
url_read()
url_close()
next
first_protocol

ffplay 定义的数据结构很有特色。

有一些是动态与静态的关系，比如， URLProtocol 和 URLContext， AVInputFormat 和 AVFormatContext，AVCodec 和 AVCodecContext。从前面播放器的一般原理我们可知，播放器内部要实现的几大功能是，读文件，识别格式，音视频解码，音视频渲染。其中音视频渲染由 SDL 实现，我们不讨论。ffplay 把其他的每个大功能抽象成一个相当于 C++中 COM 接口的数据结构，着重于功能函数，同时这些功能函数指针在编译的时候就能静态确定。每一个大功能都要支持多种类型的广义数据，ffplay 把这多种类型的广义数据的共同的部分抽象成对应的 Context 结构，这些对应的 context 结构着重于动态性，其核心成员只能在程序运行时动态确定其值。并且 COM 接口类的数据结构在程序运行时有很多很多实例，而相应的 Context 类只有一个实例，这里同时体现了数据结构的划分原则，如果有一对多的关系就要分开定义。

有一些是指针表述的排他性包含关系(因为程序运行时同一类型的多种数据只支持一种，所以就有排他性)。比如，AVCodecContext 用 priv_data 包含 MsrleContext 或 TSContext，AVFormatContext 用 priv_data 包含 AVIContext 或其他类 Context， AVStream 用 priv_data 包含 AVIStream 或其他类 Stream。由前面数据结构的动态与静态关系可知，ffplay 把多种类型的广义数据的共同部分抽象成 context 结构，那么广义数据的各个特征不同部分就抽象成各种具体类型的 context，然后用 priv_data 字段表述的指针排他性的关联起来。由于瘦身后的 ffplay 只支持有限类型，所以 AVFormatContext 只能关联包含 AVIContext，AVStream 只能关联包含 AVIStream。

有一些是扩展包含关系，比如，ByteIOContext 包含 URLContext，就是在应用层把没有缓存的 URLContext 扩展有缓冲区的广义文件 ByteIOContext，改善程序 IO 性能。

有一些是直接包含关系，比如，AVFrame 包含 AVPicture，这两个结构共有的字段，其定义类型、大小、顺序都一模一样，除了更准确的描述各自的意义便于阅读理解维护代码外，还可以方便的把 AVFrame 大结构强制转换 AVPicture 小结构。

我们先来重点分析 AVCodec/AVCodecContext/MsrleContext 这几个数据结构，这几个数据结构定义了编解码器的核心架构，相当于 Directshow 中的各种音视频解码器 decoder。

```
typedef struct AVCodec
{
    const char *name;              // 标示 Codec 的名字, 比如，"msrle" "truespeech" 等。
    enum CodecType type;           // 标示 Codec 的类型，有 Video，Audio，Data 等类型
    enum CodecID id;               // 标示 Codec 的 ID，有 CODEC_ID_MSRLE，CODEC_ID_TRUESPEECH 等
    int priv_data_size;            // 标示具体的 Codec 对应的 Context 的大小，在本例中是 MsrleContext
                                   //     或 TSContext 的大小。
    int(*init)(AVCodecContext*);// 标示 Codec 对外提供的操作
```

```
    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;                   // 标示 Codec 的能力，在瘦身后的 ffplay 中没太大作用，可忽略
    struct AVCodec *next;               // 用于把所有 Codec 串成一个链表，便于遍历
}AVCodec;
```

AVCodec 是类似 COM 接口的数据结构，表示音视频编解码器，着重于功能函数，一种媒体类型对应一个 AVCodec 结构，在程序运行时有多个实例。next 变量用于把所有支持的编解码器连接成链表，便于遍历查找；id 确定了唯一编解码器；priv_data_size 表示具体的 Codec 对应的 Context 结构大小，比如 MsrleContext 或 TSContext，这些具体的结够定义散落于各个.c 文件中，为避免太多的 if else 类语句判断类型再计算大小，这里就直接指明大小，因为这是一个编译时静态确定的字段，所以放在 AVCodec 而不是 AVCodecContext 中。

```
typedef struct AVCodecContext
{
    int bit_rate;
    int frame_number;

    unsigned char *extradata; // Codec 的私有数据，对 Audio 是 WAVEFORMATEX 结构扩展字节。
    int extradata_size;             // 对 Video 是 BITMAPINFOHEADER 后的扩展字节

    int width, height;              // 此逻辑段仅针对视频
    enum PixelFormat pix_fmt;

    int sample_rate;                // 此逻辑段仅针对音频
    int channels;
    int bits_per_sample;
    int block_align;

    struct AVCodec *codec;   // 指向当前 AVCodec 的指针，
    void *priv_data;                // 指向当前具体编解码器 Codec 的上下文 Context。

    enum CodecType codec_type; // see CODEC_TYPE_xxx
    enum CodecID codec_id;       // see CODEC_ID_xxx

    int(*get_buffer)(struct AVCodecContext *c, AVFrame *pic);
    void(*release_buffer)(struct AVCodecContext *c, AVFrame *pic);
    int(*reget_buffer)(struct AVCodecContext *c, AVFrame *pic);

    int internal_buffer_count;
    void *internal_buffer;

    struct AVPaletteControl *palctrl;
```

}AVCodecContext;

AVCodecContext 结构表示程序运行的当前 Codec 使用的上下文，着重于所有 Codec 共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。extradata 和 extradata_size 两个字段表述了相应 Codec 使用的私有数据，对 Codec 全局有效，通常是一些标志信息；codec 字段关联相应的编解码器；priv_data 字段关联各个具体编解码器独有的属性上下文，和 AVCodec 结构中的 priv_data_size 配对使用。

```
typedef struct MsrleContext
{
    AVCodecContext *avctx;
    AVFrame frame;

    unsigned char *buf;
    int size;
} MsrleContext;
```

MsrleContext 结构着重于 RLE 行程长度压缩算法独有的属性值和关联 AVCodecContext 的 avctx 字段。因为 RLE 行程长度算法足够简单，属性值相对较少。

接着来重点分析 AVInputFormat/AVFormatContext/AVIContext 这几个数据结构，这几个数据结构定义了识别文件容器格式的核心架构，相当于 Directshow 中的各种解复用 demuxer。

```
typedef struct AVInputFormat
{
    const char *name;
    int priv_data_size;     // 标示具体的文件容器格式对应的 Context 的大小，在本例中是 AVIContext

    int(*read_probe)(AVProbeData*);
    int(*read_header)(struct AVFormatContext *, AVFormatParameters *ap);
    int(*read_packet)(struct AVFormatContext *, AVPacket *pkt);
    int(*read_close)(struct AVFormatContext*);

    const char *extensions; // 文件扩展名
    struct AVInputFormat *next;
} AVInputFormat;
```

AVInputFormat 是类似 COM 接口的数据结构，表示输入文件容器格式，着重于功能函数，一种文件容器格式对应一个 AVInputFormat 结构，在程序运行时有多个实例。next 变量用于把所有支持的输入文件容器格式连接成链表，便于遍历查找；priv_data_size 标示具体的文件容器格式对应的 Context 的大小，在本例中是 AVIContext，这些具体的结够定义散落于各个 .c 文件中，为避免太多的 if else 类语句判断类型再计算大小，这里就直接指明

大小，因为这是一个编译时静态确定的字段，所以放在 AVInputFormat 而不是 AVFormatContext 中。

```
typedef struct AVFormatContext        // format I/O context
{
    struct AVInputFormat *iformat;
    void *priv_data;                  // 指向具体的文件容器格式的上下文 Context，在本例中是 AVIContext

    ByteIOContext pb;                 // 广泛意义的输入文件
    int nb_streams;

    AVStream *streams[MAX_STREAMS];
} AVFormatContext;
```

AVFormatContext 结构表示程序运行的当前文件容器格式使用的上下文，着重于所有文件容器共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。iformat 字段关联相应的文件容器格式；pb 关联广义的输入文件；streams 关联音视频流；priv_data 字段关联各个具体文件容器独有的属性上下文，和 priv_data_size 配对使用。

```
typedef struct AVIContext
{
    int64_t riff_end;
    int64_t movi_end;
    offset_t movi_list;
    int non_interleaved;
    int stream_index_2; // 为了和 AVPacket 中的 stream_index 相区别，添加后缀标记。
} AVIContext;
```

AVIContext 定义了 AVI 中流的一些属性，其中 stream_index_2 定义了当前应该读取流的索引。

接着我们来重点分析 URLProtocol/URLContext(ByteIOContext)/FILE(Socket)这几个数据结构，这几个数据结构定义了读取文件的核心架构，相当于 Directshow 中的文件源 file source filter。

```
typedef struct URLProtocol
{
    const char *name;     // 便于人性化的识别理解
    int(*url_open)(URLContext *h, const char *filename, int flags);
    int(*url_read)(URLContext *h, unsigned char *buf, int size);
    int(*url_write)(URLContext *h, unsigned char *buf, int size);
    offset_t(*url_seek)(URLContext *h, offset_t pos, int whence);
    int(*url_close)(URLContext *h);
    struct URLProtocol *next;
} URLProtocol;
```

URLProtocol 是类似 COM 接口的数据结构，表示广义的输入文件，着重于功能函数，一种广义的输入文件

对应一个 URLProtocol 结构，比如 file，pipe，tcp 等等，但瘦身后的 ffplay 只支持 file 一种输入文件。next 变量用于把所有支持的广义的输入文件连接成链表，便于遍历查找。

```
typedef struct URLContext
{
    struct URLProtocol *prot;
    int flags;
    int max_packet_size; // if non zero, the stream is packetized with this max packet size
    void *priv_data;     // 文件句柄 fd，网络通信 Scoket 等
    char filename[1];    // specified filename
} URLContext;
```

URLContext 结构表示程序运行的当前广义输入文件使用的上下文，着重于所有广义输入文件共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。prot 字段关联相应的广义输入文件；priv_data 字段关联各个具体广义输入文件的句柄。

```
typedef struct ByteIOContext
{
    unsigned char *buffer;
    int buffer_size;
    unsigned char *buf_ptr,  *buf_end;
    void *opaque;          // 关联 URLContext
    int (*read_buf)(void *opaque, uint8_t *buf, int buf_size);
    int (*write_buf)(void *opaque, uint8_t *buf, int buf_size);
    offset_t(*seek)(void *opaque, offset_t offset, int whence);
    offset_t pos;          // position in the file of the current buffer
    int must_flush;        // true if the next seek should flush
    int eof_reached;       // true if eof reached
    int write_flag;        // true if open for writing
    int max_packet_size;
    int error;             // contains the error code or 0 if no error happened
} ByteIOContext;
```

ByteIOContext 结构扩展 URLProtocol 结构成内部有缓冲机制的广泛意义上的文件，改善广义输入文件的 IO 性能。由其数据结构定义的字段可知，主要是缓冲区相关字段，标记字段，和一个关联字段 opaque 来完成广义文件读写操作。opaque 关联字段用于关联 URLContext 结构，间接关联并扩展 URLProtocol 结构。

接着我们来重点分析 AVStream/AVIStream 这几个数据结构，这几个数据结构定义了解析媒体流的核心属性，主要用于读取媒体流数据，相当于 Directshow 中的解复用 Demux 内部的流解析逻辑。特别注意此结构关联 AVCodecContext 结构，并经此结构能跳转到其他结构。

```
typedef struct AVStream             // 解析文件容器内部使用的逻辑
{
    AVCodecContext *actx;           // codec context, change from AVCodecContext *codec;
    void *priv_data;                // AVIStream

    AVRational time_base;           // 由 av_set_pts_info()函数初始化

    AVIndexEntry *index_entries; // only used if the format does not support seeking natively
    int nb_index_entries;
    int index_entries_allocated_size;

    double frame_last_delay;
} AVStream;
```

AVStream 结构表示当前媒体流的上下文，着重于所有媒体流共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。actx 字段关联当前音视频媒体使用的编解码器；priv_data 字段关联解析各个具体媒体流与文件容器有关的独有的属性；还有一些媒体帧索引和时钟信息。

```
typedef struct AVIStream
{
    int64_t frame_offset; // current frame(video) or byte(audio) counter(used to compute the pts)
    int remaining;
    int packet_size;

    int scale;
    int rate;
    int sample_size;        // size of one sample (or packet) (in the rate/scale sense) in bytes

    int64_t cum_len;      // temporary storage (used during seek)

    int prefix;             // normally 'd'<<8 + 'c' or 'w'<<8 + 'b'
    int prefix_count;
} AVIStream;
```

AVIStream 结构定义了 AVI 文件中媒体流的一些属性，用于解析 AVI 文件。

接着我们来分析 AVPacket/AVPacketList/PacketQueue 这几个数据结构，这几个数据结构定义解复用 demux 模块输出的音视频压缩数据流队列，相当于 Directshow 中 Demux 的 OutputPin，传递数据到解码器。

```
typedef struct AVPacket
{
    int64_t pts; // presentation time stamp in time_base units
```

```
    int64_t dts; // decompression time stamp in time_base units
    int64_t pos; // byte position in stream, -1 if unknown
    uint8_t *data;
    int size;
    int stream_index;
    int flags;
    void(*destruct)(struct AVPacket*);
} AVPacket;
```

AVPacket 代表音视频数据帧，固有的属性是一些标记，时钟信息，和压缩数据首地址，大小等信息。

```
typedef struct AVPacketList
{
    AVPacket pkt;
    struct AVPacketList *next;
} AVPacketList;
```

AVPacketList 把音视频 AVPacket 组成一个小链表。

```
typedef struct PacketQueue
{
    AVPacketList *first_pkt,  *last_pkt;
    int size;
    int abort_request;
    SDL_mutex *mutex;
    SDL_cond *cond;
} PacketQueue;
```

PacketQueue 通过小链表 AVPacketList 把音视频帧 AVPacket 组成一个顺序队列，是数据交换中转站，当然同步互斥控制逻辑是必不可少的。

最后我们来重点分析 VideoState 这个数据结构，这个数据结构把主要的数据结构整合在一起，声明成全局变量，起一个中转的作用，便于在各个子结构之间跳转，相当于一个大背景，大平台的作用。

```
typedef struct VideoState
{
    SDL_Thread *parse_tid;
    SDL_Thread *video_tid;

    int abort_request;

    AVInputFormat *iformat;
    AVFormatContext *ic;    // 关联的主要数据结构是 ByteIOContext 和 AVStream
```

```
    AVStream *audio_st;      // 关联的主要数据结构是 AVCodecContext 和 AVIStream
    AVStream *video_st;

    int audio_stream; // 音频流索引，实际表示 AVFormatContext 结构中 AVStream *streams[]数组中的索引
    int video_stream; // 视频流索引，实际表示 AVFormatContext 结构中 AVStream *streams[]数组中的索引

    PacketQueue audioq;   // 音频数据包队列，注意一包音频数据可能包含几个音频帧
    PacketQueue videoq;   // 视频数据包队列，注意瘦身后的 ffplay 一包视频数据是完整的一帧

    VideoPicture pictq[VIDEO_PICTURE_QUEUE_SIZE]; // 输出视频队列，瘦身后的 ffplay 只有一项
    double frame_last_delay;

    uint8_t audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE *3) / 2]; // 输出的音频缓存
    unsigned int audio_buf_size;
    int audio_buf_index;
    AVPacket audio_pkt;     // 音频包属性，只一个指针指向原始音频包数据，非直接包含音频数据包数据
    uint8_t *audio_pkt_data;
    int audio_pkt_size;

    SDL_mutex *video_decoder_mutex; // 视频线程同步互斥变量
    SDL_mutex *audio_decoder_mutex; // 音频线程同步互斥变量

    char filename[240];   // 媒体文件名
} VideoState;
```

音视频数据流简单流程，由 ByteIOContext(URLContext/URLProtocol)表示的广义输入文件，在 AVStream(AVIStreamt)提供的特定文件容器流信息的指引下，用 AVInputFormat(AVFormatContext /AVInputFormat)接口的 read_packet()函数读取完整的一帧数据，分别放到音频或视频 PacketQueue(AVPacketList/AVPacket)队列中，这部分功能由 decode_thread 线程完成。对于视频数据，video_thread 线程不停的从视频 PacketQueue 队列中取出视频帧，调用 AVCodec(AVCodecContext /MsrleContext)接口的 decode()函数解码视频帧，在适当延时后做颜色空间转化并调用 SDL 库显示出来。对于音频数据，SDL 播放库播放完缓冲区的 PCM 数据后，调用 sdl_audio_callback()函数解码音频数据，并把解码后的 PCM 数据填充到 SDL 音频缓存播放。当下次播放完后，再调用 sdl_audio_callback()函数解码填充，如此循环不已。对于 SDL 库的实现，这里不做讨论。

特别注意 decode_thread 线程的共分为三大逻辑功能。

(1): 首先调用 av_open_input_file()直接识别文件格式和间接识别媒体格式(媒体格式是通过 av_open_input_file 调用 av_open_input_stream 再调用 ic->iformat->read_header(ic, ap)来识别的，所以是间接识别媒体格式)。

(2)：接着调用 stream_component_open()查找打开编解码器 codec 并启动音频和视频解码线程。

(3)：再接着解析文件，分离音视频数据包，排序进入队列。

1.5 ffplay 主要改动

在 ffmpeg 的瘦身过程中，精跳细选留下 MS RLE 视频压缩算法和 True Speech 语音压缩算法，主要是因为这两个算法足够的简单，并且测试文件 CLOCKTXT.avi 就在手边，AVI 容器的解复用也很简单，不好的就是视频分辨率是 321x321 太怪怪，简单转换为 YUV 显示不太正确，所以把它转压为 320x320 格式，无损压缩的转压就是麻烦多多，常规的转压缩工具都不好用，浪费了比较长的时间。

超大量的删除不用支撑的文件格式和媒体编解码算法相关代码，删掉不支持的平台相关代码，删掉了一些不用支撑的功能，同时写死某些参数又精简了很多函数的实现，有些地方牺牲了一些运算进度来简化，有些函数做了合并，为实现最小化把没有实际使用的几乎所有变量和函数也删掉了。

ffplay 有几个函数名字值得商榷，不太符合上下文，为便于理解，做了修改。

url_fread()函数由 get_buffer()改名而来，如果把 ByteIOContext *s 看作是广义上的文件句柄，已有的 url_fopen()/url_fclose()/url_feof()/url_fsize()/url_ftell()/url_fskip()/url_fseek()函数名字非常准确，但是从文件操作的完备性来讲，没有读文件的函数(解码器不写文件所以不需要考虑 url_fwrite()函数)。ffplay 原始的 get_buffer()函数实际就是广义上的读文件函数，把它改成 url_fread()后文件操作类函数就完备了。

在读文件模块，操作内部缓冲区的几个静态内部函数 url_read_buf()/url_seek_buf()/url_write_buf ()是由相应的 url_read_packet()/url_seek_packet()/url_write_packet()修改而来，首先在功能上它们维护模块内部使用的 buffer，其次 read_packet()类函数在其他地方有使用，在那个地方一次读一个完整数据包(packet)，为区分 buffer 和 packet 的不同就修改了名字。

数据结构的修改，把 AVStream 结构中的 AVCodecContext *codec 改成了 actx 更准确的描述了变量的意义。

同步机制的修改，此瘦身后的 ffplay 采用比较简单的同步策略，视频音频都从零开始起步计时，按照各自的时钟往前走，音频由自身特性决定不用时钟，播放速率由采样率，声道数，量化 bit 数决定；视频按照帧率计算每帧间隔，从第二帧开始忽略掉读文件解码显示等等操作消耗的时间，先 sleep 间隔时间再显示视频图像，为此在 AVStream 结构中添加 double frame_last_delay 变量表示帧间隔时间。

1.6 SDL 显示视频

SDL 是 Simple Direct Layer 的缩写。它是一个出色的多媒体库，适用于 PC 平台，并且已经应用在许多工程中，它是如此的优秀，甚至已移植到某些手机平台上。它的官方网站是 http://www.libsdl.org/，在这个网

站上可以下载 SDL 库的源代码自己编译库，也可以直接下载预编译库。

SDL 显示视频图像函数调用序列如下，忽略掉错误处理：

1):初始化 SDL 库，

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)
```

2):创建显示表面，

```
SDL_Surface *screen = SDL_SetVideoMode(width, height, 0, 0)
```

3):创建Overlay表面，

```
SDL_Overlay *bmp = SDL_CreateYUVOverlay(width, height, SDL_YV12_OVERLAY, screen)
```

4):取得独占权和 Overlay 表面首地址，

```
SDL_LockYUVOverlay(bmp);
```

5):填充视频数据，纹理数据

6):释放独占权，

```
SDL_UnlockYUVOverlay(bmp);
```

7):刷新视频，

```
SDL_DisplayYUVOverlay(bmp, &rect);
```

1.7 SDL 播放音频

SDL 播放音频采用回调函数的方式来保证音频的连续性，在设置音频输出参数，向系统注册回调函数后，每次写入的音频数据播放完，系统自动调用注册的回调函数，通常在此回调函数中继续往系统写入音频数据。

SDL 播放音频函数调用序列，忽略掉错误处理：

1):初始化 SDL_AudioSpec 结构，此结构包括音频参数和回调函数，比如

```
SDL_AudioSpec wanted_spec;
wanted_spec.userdata = is;
wanted_spec.channels = 2;
wanted_spec.callback = sdl_audio_callback;
......
```

2):打开音频设备

```
SDL_OpenAudio(&wanted_spec, &spec);
```

3)：激活音频设备开始工作

```
SDL_PauseAudio(0);
```

4):在音频回调函数中写入音频数据，示意代码如下

```
void sdl_audio_callback(void *opaque, Uint8 *stream, int len)
{
   memcpy(stream, (uint8_t*)audio_buf, len);
}
```

5)：播放完后关闭音频

```
SDL_CloseAudio();
```

1.8 AVI 文件格式简介

AVI 是 Audio Video Interleaved 的缩写，意思是视频数据和音频数据交织存放的一种文件格式，一种容器格式，是 Windows 操作系统上最基本的、也是最常用的一种媒体文件格式。通常情况下，一个 AVI 文件可以包含多个不同类型的媒体流（典型的情况下有一个音频流和一个视频流），不过含有单一音频流或单一视频流的 AVI 文件也是合法的。

AVI 文件以数据块(chunk)为单位来组织数据，每个数据块以一个四字符码 FOURCC（four-character code）来表征数据类型，比如'AVI'，'LIST'，'movi'等等，接着是4个字节的整数只表示当前块的大小，注意此大小不包括四字符码和4字节整数占用的共8字节，接着是此数据块的实际有效数据。注意 chunk 块还可以嵌套。

AVI 文件的基本格式如下所示：

```
RIFF('AVI'
    LIST('hdrl'
        avih(<MainAVIHeader>)
           LIST('strl'
               strh(<AVISTREAMHEADER>)
               strf(<BITMAPINFO 或者 WAVEFORMATEX>)
               ... additional header data
               )
           ...
           )
    LIST('movi'
       { LIST('rec'
             SubChunk...
             )
         | SubChunk } ....
        )
```

```
    [ 'idx1' (AVIOLDINDEX)]
    )
RIFF( 'AVI ' …) 表征了 AVI 文件类型。
LIST( 'hdrl') 表示是一个 header list，其后 MainAVIHeader 定义如下：
   typedef struct _avimainheader {
        FOURCC fcc;                  // 'avih'
        DWORD  cb;                   // 此数据结构的大小，不包括最初的8个字节（fcc 和 cb 两个域）
        DWORD  dwMicroSecPerFrame;   // 视频帧间隔时间（以毫秒为单位）
        DWORD  dwMaxBytesPerSec;     // 最大数据率
        DWORD  dwPaddingGranularity; // 数据填充的粒度
        DWORD  dwFlags;              // AVI 文件标记，特别注意 AVIF_MUSTUSEINDEX 标记，
                                     //   其表明应用程序需要使用 index，而不是物理上的顺序，
                                     //   来定义数据的展现顺序。
        DWORD  dwTotalFrames;        // 总帧数
        DWORD  dwInitialFrames;      // 初始帧数,可简单的忽略为0不处理此字段
        DWORD  dwStreams;            // 媒体流的个数
        DWORD  dwSuggestedBufferSize;// 建议读缓冲区大小，指示解复用程序更合理的分配缓冲区大小
        DWORD  dwWidth;              // 视频图像的宽（以像素为单位）
        DWORD  dwHeight;             // 视频图像的高（以像素为单位）
        DWORD  dwReserved[4];        // 保留
   } AVIMAINHEADER;
LIST('strl') 表示是一个 stream list，其后 AVISTREAMHEADER 定义如下：
    typedef struct _avistreamheader {
        FOURCC fcc;         // strh'
        DWORD  cb;          // 此本数据结构的大小，不包括最初的8个字节（fcc 和 cb 两个域）
        FOURCC fccType;     // 流的类型，常见的有 'auds'（音频流）、'vids'（视频流）
        FOURCC fccHandler; // 指定流的处理者，对于音视频来说就是解码器
        DWORD  dwFlags;     // 标记：是否允许这个流输出？调色板是否变化？
        WORD   wPriority;  // 流的优先级（当有多个相同类型的流时优先级最高的为默认流）
        WORD   wLanguage;
        DWORD  dwInitialFrames; // 初始帧数，可简单的忽略为0不处理此字段
        DWORD  dwScale;          // 缩放因子，和 dwRate 配合使用更准确地表述。
        DWORD  dwRate;           // 比如视频帧率为29.97时，dwScale 可设为1001，dwRate 可设为30000
        DWORD  dwStart;          // 流的开始时间,指定这个流开始的时间。其单位由 dwRate 和
                                 //   dwScale 成员定义(其单位是 dwRate/dwScale)。通常 dwStart 是0,
                                 //   但是它也可以为不与文件同时启动的流定义一个时间延迟。
        DWORD  dwLength;         // 流的长度（单位与 dwScale 和 dwRate 的定义有关）
        DWORD  dwSuggestedBufferSize;//建议读缓冲区大小，指示解复用程序更合理的分配缓存大小
        DWORD  dwQuality;        // 流数据的质量指标（0 ~ 10,000）
        DWORD  dwSampleSize;     // Sample 的大小
        struct {
```

```
            short int left;
            short int top;
            short int right;
            short int bottom;
        }rcFrame;              // 指定这个流（视频流或文字流）在视频主窗口中的显示位置
    }AVISTREAMHEADER;
```

每种媒体有一个 strf 块。对视频此块对于 BITMAPINFO，对音频此块对应 WAVEFORMATEX。

LIST('movi'),表示是一个 Movice 块，保存真正的视音频媒体流数据。

最后是索引块，使用一个四字符码‘idx1’来表征，由数据结构 AVIOLDINDEX 来定义

```
    typedef struct _avioldindex {
        FOURCC  fcc;            // 必须为‘idx1’
        DWORD   cb;             // 本数据结构的大小，不包括最初的8个字节（fcc 和 cb 两个域）
        struct _avioldindex_entry {
            DWORD   dwChunkId;  // 表征本数据块的四字符码
            DWORD   dwFlags;    // 说明本数据块是不是关键帧、是不是‘rec ’列表等信息
            DWORD   dwOffset;   // 本数据块在文件中的偏移量，有相对偏移(相对‘movi’偏移)
                                // 和绝对偏移(相对于 AVI 文件头)两种。
            DWORD   dwSize;     // 本数据块的大小
        } aIndex[];             // 为每个媒体数据块都定义一个索引信息的数组
    } AVIOLDINDEX;
```

还有一种特殊的数据块，用一个四字符码‘JUNK’来表征，它用于内部数据的对齐，播放器应该忽略。

AVI 文件媒体数据组织存储方式。媒体数据的各最小存取单元前面都使用了一个四字符码来表征它的类型，四字节的整数表示长度，同样不包括四字符码和4字节整数占用的共8字节。这个四字符码由2个字节的类型码和2个字节的流编号组成。标准的类型码定义如下：‘db’（非压缩视频帧）、‘dc’（压缩视频帧）、‘pc’（改用新的调色板）、‘wb’（压缩音频）。比如第一个流（Stream 0）是音频，则表征音频数据块的四字符码为‘00wb’；第二个流（Stream 1）是视频，则表征视频数据块的四字符码为‘00db’或‘00dc’。对于视频数据来说，在 AVI 数据序列中间还可以定义一个新的调色板，每个改变的调色板数据块用‘xxpc’来表征，新的调色板使用一个数据结构 AVIPALCHANGE 来定义，通常比较少见。

1.9 MS RLE 压缩算法简介

Run Length Encoding (RLE) 数据压缩算法是无损压缩方法的一种，除去一些表示行结束，文件结束等等特殊字符外，用一对数字(每个数字一个字节)表示一个压缩单元，前一个数字表示这个压缩单元的长度，后一个数字表示这个压缩单元的值，比如：

Compressed data　　Expanded data

03 04	04 04 04
02 78	78 78

Microsoft's RLE 压缩算法还支持一种每像素 4Bits 的压缩算法，仍然是一对数字表示一个压缩单元，前一个数字表示这个压缩单元的长度，后一数字每 4bits 表示一个值，循环填充压缩单元直到填满为止。比如：

Compressed data	Expanded data
03 04	0 4 0
05 06	0 6 0 6 0

RLE 压缩算法可以用于 BMP 静态图片和 AVI 动画中，由于单个字节能表达的值有限，所以 RLE 压缩算法的局限性很大，仅在 windows 3.1 年代有过广泛应用，但 MS Windows 系列操作系统的兼容性做得很好，现在还支撑 RLE 压缩算法。

1.10 True Speech 压缩算法简介

True Speech 语音压缩算法基于线性预测编码的语音编码算法，支撑 8k 采样，16bit 量化，单声道语音数据压缩，采样固定比特率，每帧 240 个采样，分为 4 个子帧，windows 版本大约实现了 15：1 的压缩比。

其他资料请各位网友自行查找。

第二章 libavutil 剖析

2.1 文件列表

文件类型	文件名	大小(bytes)
h	common.h	1515
h	bswap.h	489
h	rational.h	257
h	mathematics.h	153
h	avutil.h	1978

2.2 common.h 文件

2.2.1 功能描述

ffplay 使用的工具类数据类型定义，宏定义和两个简单的内联函数,基本上是自注释的。

2.2.2 文件注释

```
1
2   #ifndef COMMON_H
3   #define COMMON_H
4
5   #include <stdlib.h>
6   #include <stdio.h>
7   #include <string.h>
8   #include <ctype.h>
9
10  #if defined(WIN32) && !defined(__MINGW32__) && !defined(__CYGWIN__)
11  #define CONFIG_WIN32
12  #endif
13
```

内联函数的关键字在 linux gcc 和 windows vc 中的定义是不同的，gcc 是 inline，vc 是__inline。因为代码是从 linux 下移植过来的，在这里做一个宏定义修改相对简单。

```
14  #ifdef CONFIG_WIN32
15  #define inline __inline
16  #endif
17
```

简单的数据类型定义，linux gcc 和 windows vc 编译器有稍许不同，用宏开关 CONFIG_WIN32 来屏蔽 64 位整数类型的差别。

```
18  typedef signed char int8_t;
19  typedef signed short int16_t;
20  typedef signed int int32_t;
21  typedef unsigned char uint8_t;
22  typedef unsigned short uint16_t;
23  typedef unsigned int uint32_t;
24
25  #ifdef CONFIG_WIN32
26  typedef signed __int64 int64_t;
27  typedef unsigned __int64 uint64_t;
28  #else
29  typedef signed long long int64_t;
30  typedef unsigned long long uint64_t;
31  #endif
32
```

64 位整数的定义语法，linux gcc 和 windows vc 编译器有稍许不同，用宏开关 CONFIG_WIN32 来屏蔽 64 位整数定义的差别。

Linux 用 LL/ULL 来表示 64 位整数，VC 用 i64 来表示 64 位整数。

是连接符，把##前后的两个字符串连接成一个字符串。

```
33  #ifdef CONFIG_WIN32
34  #define int64_t_C(c)     (c ## i64)
35  #define uint64_t_C(c)    (c ## i64)
36  #else
37  #define int64_t_C(c)     (c ## LL)
38  #define uint64_t_C(c)    (c ## ULL)
39  #endif
40
```

定义最大的 64 位整数。

```
41  #ifndef INT64_MAX
42  #define INT64_MAX int64_t_C(9223372036854775807)
43  #endif
44
```

大小写敏感的字符串比较函数。在 ffplay 中只关心是否相等，不关心谁大谁小。

```
45  static int strcasecmp(char *s1, const char *s2)
46  {
47      while (toupper((unsigned char) *s1) == toupper((unsigned char) *s2++))
48          if (*s1++ == '\0')
49              return 0;
```

```
50
51      return (toupper((unsigned char) *s1) - toupper((unsigned char) *--s2));
52  }
53
```

限幅函数，这个函数使用简单的比较逻辑来实现，比较语句多，容易中断 CPU 的指令流水线，导致性能低下。如果变量 a 的取值范围比较小，可以用常规的空间换时间的查表方法来优化。

```
54  static inline int clip(int a, int amin, int amax)
55  {
56      if (a < amin)
57          return amin;
58      else if (a > amax)
59          return amax;
60      else
61          return a;
62  }
63
64  #endif
```

2.3 bswap.h 文件

2.3.1 功能描述

short 和 int 整数类型字节顺序交换，通常和 CPU 大端或小端有关。

对 int 型整数，小端 CPU 低地址内存存低位字节，高地址内存存高位字节。

对 int 型整数，大端 CPU 低地址内存存高位字节，高地址内存存低位字节。

常见的 CPU 中，Intel X86 序列及其兼容序列只能是小端，Motorola 68 序列只能是大端，ARM 大端小端都支持，但默认小端。

2.3.2 文件注释

```
1   #ifndef __BSWAP_H__
2   #define __BSWAP_H__
3
```

Int 16 位短整数字节交换，简单的移位再或运算。

```
4   static inline uint16_t bswap_16(uint16_t x)
5   {
6       return (x >> 8) | (x << 8);
7   }
8
```

Int 32 位长整数字节交换，看遍所有的开源代码，这个代码是最简洁的 C 代码，并且和上面 16 位短整数字节交换一脉相承。

```
9   static inline uint32_t bswap_32(uint32_t x)
10  {
11      x = ((x << 8) &0xFF00FF00) | ((x >> 8) &0x00FF00FF);
12      return (x >> 16) | (x << 16);
13  }
14
15  // be2me ... BigEndian to MachineEndian
16  // le2me ... LittleEndian to MachineEndian
17
18  #define be2me_16(x) bswap_16(x)
19  #define be2me_32(x) bswap_32(x)
20  #define le2me_16(x) (x)
21  #define le2me_32(x) (x)
22
23  #endif
```

2.4 rational.h 文件

2.4.1 功能描述

用两整数精确表示分数。常规的可以用一个 float 或 double 型数来表示分数，但不是精确表示，在需要相对比较精确计算的时候，为避免非精确表示带来的计算误差，采用两整数来精确表示。

2.4.2 文件注释

```
1   #ifndef RATIONAL_H
2   #define RATIONAL_H
3
```

用分数最原始的分子和分母的定义来表示，用分子和分母的组合来表示分数。

```
4   typedef struct AVRational
5   {
6       int num; // numerator   // 分子
7       int den; // denominator // 分母
8   } AVRational;
9
```

用 float 或 double 表示分数值，强制类型转换后，简单的除法运算。

```
10  static inline double av_q2d(AVRational a)
11  {
12      return a.num / (double)a.den;
13  }
14
15  #endif
```

2.5 mathematics.h 文件

2.5.1 功能描述

数学上的缩放运算。为避免计算误差，缩放因子用两整数表示做精确的整数运算。为防止计算溢出，强制转换为 int 64 位整数后计算。

此处做了一些简化，运算精度会降低，但普通的人很难感知到计算误差。

2.5.2 文件注释

```
1   #ifndef MATHEMATICS_H
2   #define MATHEMATICS_H
3
```

数学上的缩放运算，此处简化了很多，虽然计算结果有稍许误差，但不影响播放效果。

```
4   static inline int64_t av_rescale(int64_t a, int64_t b, int64_t c)
5   {
6       return a * b / c;
7   }
8
9   #endif
```

2.6 avutil.h 文件

2.6.1 功能描述

ffplay 基础工具库使用的一些常数和宏的定义。

2.6.2 文件注释

```
1   #ifndef AVUTIL_H
2   #define AVUTIL_H
3
4   #ifdef __cplusplus
5   extern "C" {
6   #endif
7
```

代码 8 到 15 行是一些版本信息标示的宏定义，便于各位网友和原始版本比对，更深入地学习 ffmpeg。

```
8   #define AV_STRINGIFY(s)         AV_TOSTRING(s)
9   #define AV_TOSTRING(s) #s
10
11  #define LIBAVUTIL_VERSION_INT   ((49<<16)+(0<<8)+0)
12  #define LIBAVUTIL_VERSION       49.0.0
13  #define LIBAVUTIL_BUILD         LIBAVUTIL_VERSION_INT
14
15  #define LIBAVUTIL_IDENT         "Lavu" AV_STRINGIFY(LIBAVUTIL_VERSION)
16
17  #include "common.h"
18  #include "mathematics.h"
19  #include "rational.h"
20
```

像素格式的宏定义，便于代码编写和维护。把一些常数定义成有意义的宏是一个值得鼓励的好习惯。

```
21  enum PixelFormat
22  {
23      PIX_FMT_NONE= -1,
24      PIX_FMT_YUV420P,    // Planar YUV 4:2:0 (1 Cr & Cb sample per 2x2 Y samples)
25      PIX_FMT_YUV422,     // Packed pixel, Y0 Cb Y1 Cr
26      PIX_FMT_RGB24,      // Packed pixel, 3 bytes per pixel, RGBRGB...
27      PIX_FMT_BGR24,      // Packed pixel, 3 bytes per pixel, BGRBGR...
28      PIX_FMT_YUV422P,    // Planar YUV 4:2:2 (1 Cr & Cb sample per 2x1 Y samples)
29      PIX_FMT_YUV444P,    // Planar YUV 4:4:4 (1 Cr & Cb sample per 1x1 Y samples)
30      PIX_FMT_RGBA32,     // Packed pixel, 4 bytes per pixel, BGRABGRA..., stored in cpu endianness
```

```
31      PIX_FMT_YUV410P,    // Planar YUV 4:1:0 (1 Cr & Cb sample per 4x4 Y samples)
32      PIX_FMT_YUV411P,    // Planar YUV 4:1:1 (1 Cr & Cb sample per 4x1 Y samples)
33      PIX_FMT_RGB565,     // always stored in cpu endianness
34      PIX_FMT_RGB555,     // always stored in cpu endianness, most significant bit to 1
35      PIX_FMT_GRAY8,
36      PIX_FMT_MONOWHITE, // 0 is white
37      PIX_FMT_MONOBLACK, // 0 is black
38      PIX_FMT_PAL8,       // 8 bit with RGBA palette
39      PIX_FMT_YUVJ420P,   // Planar YUV 4:2:0 full scale (jpeg)
40      PIX_FMT_YUVJ422P,   // Planar YUV 4:2:2 full scale (jpeg)
41      PIX_FMT_YUVJ444P,   // Planar YUV 4:4:4 full scale (jpeg)
42      PIX_FMT_XVMC_MPEG2_MC,// XVideo Motion Acceleration via common packet passing(xvmc_render.h)
43      PIX_FMT_XVMC_MPEG2_IDCT,
44      PIX_FMT_UYVY422,    // Packed pixel, Cb Y0 Cr Y1
45      PIX_FMT_UYVY411,    // Packed pixel, Cb Y0 Y1 Cr Y2 Y3
46      PIX_FMT_NB,
47  };
48
49  #ifdef __cplusplus
50  }
51  #endif
52
53  #endif
```

第三章 libavformat 剖析

3.1 文件列表

文件类型	文件名	大小(bytes)
h	avformat.h	5352
c	allformats.c	299
c	cutils.c	606
c	file.c	1504
h	avio.h	3103
c	avio.c	2286
c	aviobuf.c	6887
c	utils_format.c	7662
c	avidec.c	21713

3.2 avformat.h 文件

3.2.1 功能描述

定义识别文件格式和媒体类型库使用的宏、数据结构和函数，通常这些宏、数据结构和函数在此模块内相对全局有效。

3.2.2 文件注释

```
1   #ifndef AVFORMAT_H
2   #define AVFORMAT_H
3
4   #ifdef __cplusplus
5   extern "C"
6   {
7   #endif
8
9   #define LIBAVFORMAT_VERSION_INT ((50<<16)+(4<<8)+0)
10  #define LIBAVFORMAT_VERSION     50.4.0
11  #define LIBAVFORMAT_BUILD       LIBAVFORMAT_VERSION_INT
12
13  #define LIBAVFORMAT_IDENT       "Lavf" AV_STRINGIFY(LIBAVFORMAT_VERSION)
14
15  #include "../libavcodec/avcodec.h"
16  #include "avio.h"
17
```

一些简单的宏定义

```
18  #define AVERROR_UNKNOWN      (-1)    // unknown error
19  #define AVERROR_IO           (-2)    // i/o error
20  #define AVERROR_NUMEXPECTED (-3)    // number syntax expected in filename
21  #define AVERROR_INVALIDDATA (-4)    // invalid data found
22  #define AVERROR_NOMEM        (-5)    // not enough memory
23  #define AVERROR_NOFMT        (-6)    // unknown format
24  #define AVERROR_NOTSUPP      (-7)    // operation not supported
25
26  #define AVSEEK_FLAG_BACKWARD 1       // seek backward
27  #define AVSEEK_FLAG_BYTE     2       // seeking based on position in bytes
28  #define AVSEEK_FLAG_ANY      4       // seek to any frame, even non keyframes
29
30  #define AVFMT_NOFILE         0x0001  // no file should be opened
31
32  #define PKT_FLAG_KEY         0x0001
33
34  #define AVINDEX_KEYFRAME     0x0001
35
36  #define AVPROBE_SCORE_MAX    100
37
38  #define MAX_STREAMS 20
39
```

音视频数据包定义，在瘦身后的 ffplay 中，每一个包是一个完整的数据帧。注意保存音视频数据包的内存是 malloc 出来的，用完后应及时用 free 归还给系统。

```
40  typedef struct AVPacket
41  {
42      int64_t pts; // presentation time stamp in time_base units // 表示时间，对视频是显示时间
43      int64_t dts; // decompression time stamp in time_base units// 解码时间，这个不是很重要
44      int64_t pos; // byte position in stream, -1 if unknown
45      uint8_t *data;    // 实际保存音视频数据缓存的首地址
46      int size;         // 实际保存音视频数据缓存的大小
47      int stream_index; // 当前音视频数据包对应的流索引，在本例中用于区别音频还是视频。
48      int flags;        // 数据包的一些标记，比如是否是关键帧等。
49      void(*destruct)(struct AVPacket*);
50  } AVPacket;
51
```

音视频数据包链表定义，注意每一个 AVPacketList 仅含有一个 AVPacket，和传统的很多很多节点的 list 不同，不要被 list 名字迷惑。

```
52  typedef struct AVPacketList
53  {
54      AVPacket pkt;
55      struct AVPacketList *next; // 用于把各个 AVPacketList 串联起来。
56  } AVPacketList;
57
```

释放掉音视频数据包占用的内存，把首地址置空是一个很好的习惯。

```
58  static inline void av_destruct_packet(AVPacket *pkt)
59  {
60      av_free(pkt->data);
61      pkt->data = NULL;
62      pkt->size = 0;
63  }
64
```

判断一些指针，中转一下，释放掉音视频数据包占用的内存。

```
65  static inline void av_free_packet(AVPacket *pkt)
66  {
67      if (pkt && pkt->destruct)
68          pkt->destruct(pkt);
69  }
70
```

读文件往数据包中填数据，注意程序跑到这里时，文件偏移量已确定，要读数据的大小也确定，但是数据包的缓存没有分配。分配好内存后，要初始化包的一些变量。

```
71  static inline int av_get_packet(ByteIOContext *s, AVPacket *pkt, int size)
72  {
73      int ret;
74      unsigned char *data;
75      if ((unsigned)size > (unsigned)size + FF_INPUT_BUFFER_PADDING_SIZE)
76          return AVERROR_NOMEM;
77
```

分配数据包缓存

```
78      data = av_malloc(size + FF_INPUT_BUFFER_PADDING_SIZE);
79      if (!data)
80          return AVERROR_NOMEM;
81
```

把数据包中 pad 部分清 0，这是一个很好的习惯。 缓存清 0 不管在什么情况下都是好习惯。

```
82      memset(data + size, 0, FF_INPUT_BUFFER_PADDING_SIZE);
83
```

设置 AVPacket 其他的成员变量，能确定的就赋确定值，不能确定的赋初值。

```
84      pkt->pts = AV_NOPTS_VALUE;
85      pkt->dts = AV_NOPTS_VALUE;
86      pkt->pos =  - 1;
87      pkt->flags = 0;
88      pkt->stream_index = 0;
89      pkt->data = data;
90      pkt->size = size;
91      pkt->destruct = av_destruct_packet;
92
93      pkt->pos = url_ftell(s);
94
```

实际读广义文件填充数据包，如果读文件错误时通常是到了末尾，要归还刚刚 malloc 出来的内存。

```
95      ret = url_fread(s, pkt->data, size);
96      if (ret <= 0)
97          av_free_packet(pkt);
98      else
99          pkt->size = ret;
100
101     return ret;
102 }
103
```

为识别文件格式，要读一部分文件头数据来分析匹配 ffplay 支持的文件格式文件特征。于是 AVProbeData 结构就定义了文件名，首地址和大小。此处的读独立于其他文件操作。

```
104 typedef struct AVProbeData
105 {
106     const char *filename;
107     unsigned char *buf;
108     int buf_size;
109 } AVProbeData;
110
```

文件索引结构，flags 和 size 位定义是为了节省内存。

```
111 typedef struct AVIndexEntry
112 {
113     int64_t pos;
```

```
114     int64_t timestamp;
115     int flags: 2;
116     int size: 30;
117 } AVIndexEntry;
118
```

AVStream 抽象的表示一个媒体流，定义了所有媒体一些通用的属性。

```
119 typedef struct AVStream
120 {
121     AVCodecContext *actx;// 关联到解码器//codec context,change from AVCodecContext *codec;
122
123     void *priv_data;      // 在本例中，关联到 AVIStream
124
125     AVRational time_base; // 由 av_set_pts_info()函数初始化
126
127     AVIndexEntry *index_entries; // only used if the format does not support seeking natively
128     int nb_index_entries;
129     int index_entries_allocated_size;
130
131     double frame_last_delay;     // 帧最后延迟
132 } AVStream;
133
```

AVFormatParameters 结构在瘦身后的 ffplay 中没有实际意义，为保证函数接口不变，没有删除。

```
134 typedef struct AVFormatParameters
135 {
136     int dbg; //only for debug
137 } AVFormatParameters;
138
```

AVInputFormat 定义输入文件容器格式，着重于功能函数，一种文件容器格式对应一个 AVInputFormat 结构，在程序运行时有多个实例，但瘦身后 ffplay 仅一个实例。

```
139 typedef struct AVInputFormat
140 {
141     const char *name;  // 文件容器格式名，用于人性化阅读，维护代码
142
143     int priv_data_size;// 程序运行时，文件容器格式对应的上下文结构大小，便于内存分配。
144
145     int(*read_probe)(AVProbeData*); // 功能性函数
146
147     int(*read_header)(struct AVFormatContext *, AVFormatParameters *ap);
```

```
148
149     int(*read_packet)(struct AVFormatContext *, AVPacket *pkt);
150
151     int(*read_close)(struct AVFormatContext*);
152
153     const char *extensions;      // 此种文件容器格式对应的文件扩展名，识别文件格式的最后办法。
154
155     struct AVInputFormat *next; // 用于把 ffplay 支持的所有文件容器格式链成一个链表。
156
157 } AVInputFormat;
158
```

AVFormatContext 结构表示程序运行的当前文件容器格式使用的上下文，着重于所有文件容器共有的属性，程序运行后仅一个实例。

```
159 typedef struct AVFormatContext  // format I/O context
160 {
161     struct AVInputFormat *iformat; // 关联程序运行时，实际的文件容器格式指针。
162
163     void *priv_data;               // 关联具体文件容器格式上下文的指针，在本例中是 AVIContext
164
165     ByteIOContext pb;              // 关联广义输入文件
166
167     int nb_streams;                // 广义输入文件中媒体流计数
168
169     AVStream *streams[MAX_STREAMS];// 关联广义输入文件中的媒体流
170
171 } AVFormatContext;
172
```

相关函数说明参考相应的 c 实现文件。

```
173 int avidec_init(void);
174
175 void av_register_input_format(AVInputFormat *format);
176
177 void av_register_all(void);
178
179 AVInputFormat *av_probe_input_format(AVProbeData *pd, int is_opened);
180 int match_ext(const char *filename, const char *extensions);
181
182 int av_open_input_stream(AVFormatContext **ic_ptr, ByteIOContext *pb, const char *filename,
183                          AVInputFormat *fmt, AVFormatParameters *ap);
```

```
184
185 int av_open_input_file(AVFormatContext **ic_ptr, const char *filename, AVInputFormat *fmt,
186                         int buf_size, AVFormatParameters *ap);
187
188 int av_read_frame(AVFormatContext *s, AVPacket *pkt);
189 int av_read_packet(AVFormatContext *s, AVPacket *pkt);
190 void av_close_input_file(AVFormatContext *s);
191 AVStream *av_new_stream(AVFormatContext *s, int id);
192 void av_set_pts_info(AVStream *s, int pts_wrap_bits, int pts_num, int pts_den);
193
194 int av_index_search_timestamp(AVStream *st, int64_t timestamp, int flags);
195 int av_add_index_entry(AVStream *st, int64_t pos, int64_t timestamp, int size, int distance, int flags);
196
197 int strstart(const char *str, const char *val, const char **ptr);
198 void pstrcpy(char *buf, int buf_size, const char *str);
199
200 #ifdef __cplusplus
201 }
202
203 #endif
204
205 #endif
```

3.3 allformat.c 文件

3.3.1 功能描述

简单的注册/初始化函数，把相应的协议，文件格式，解码器等用相应的链表串起来便于查找。

3.3.2 文件注释

```
1   #include "avformat.h"
2
3   extern URLProtocol file_protocol;
4
5   void av_register_all(void)
6   {
```

7 到 11 行，inited 变量声明成 static，做一下比较是为了避免此函数多次调用。
编程基本原则之一，初始化函数只调用一次，不能随意多次调用。

```
7       static int inited = 0;
8
9       if (inited != 0)
10          return ;
11      inited = 1;
12
```

ffplay 把 CPU 当做一个广义的 DSP。有些计算可以用 CPU 自带的加速指令来优化，ffplay 把这类函数独立出来放到 dsputil.h 和 dsputil.c 文件中，用函数指针的方法映射到各个 CPU 具体的加速优化实现函数，此处初始化这些函数指针。

```
13      avcodec_init();
14
```

把所有的解码器用链表的方式都串连起来，链表头指针是 first_avcodec。

```
15      avcodec_register_all()
16
```

把所有的输入文件格式用链表的方式都串连起来，链表头指针是 first_iformat。

```
17      avidec_init();
18
```

把所有的输入协议用链表的方式都串连起来，比如 tcp/udp/file 等，链表头指针是 first_protocol。

```
19      register_protocol(&file_protocol);
20  }
```

3.4 cutils.c 文件

3.4.1 功能描述

ffplay 文件格式分析模块使用的两个工具类函数，都是对字符串的操作。

3.4.2 文件注释

```
1   #include "avformat.h"
2
```

strstart 实际的功能就是在 str 字符串中搜索 val 字符串指示的头，并且去掉头后用*ptr 返回。
在本例中，在播本地文件时，在命令行输入时可能会在文件路径名前加前缀"file:",为调用系统的 open 函数，需要把这几个前导字符去掉，仅仅传入完整有效的文件路径名。
和 rtsp://等网络协议相对应，播本地文件时应加 file:前缀。

```
3   int strstart(const char *str, const char *val, const char **ptr)
4   {
5       const char *p,  *q;
6       p = str;
7       q = val;
8       while (*q != '\0')
9       {
10          if (*p !=  *q)
11              return 0;
12          p++;
13          q++;
14      }
15      if (ptr)
16          *ptr = p;
17      return 1;
18  }
19
```

字符串拷贝函数，拷贝的字符数由 buf_size 指定，更安全的字符串拷贝操作。
传统的 strcpy()函数是拷贝一个完整的字符串，如果目标字符串缓冲区小于源字符串长度，那么就会发生缓冲区溢出导致错误，并且这种错误很难发现。

```
20  void pstrcpy(char *buf, int buf_size, const char *str)
21  {
22      int c;
23      char *q = buf;
24
25      if (buf_size <= 0)
```

```
26          return ;
27
28      for (;;)
29      {
30          c =  *str++;
31          if (c == 0 || q >= buf + buf_size - 1)
32              break;
33          *q++ = c;
34      }
35      *q = '\0';
36  }
```

3.5 file.c 文件

3.5.1 功能描述

ffplay 把 file 当做类似于 rtsp，rtp，tcp 等协议的一种协议，用 file:前缀标示 file 协议。
URLContext 结构抽象统一表示这些广义上的协议，对外提供统一的抽象接口。
各具体的广义协议实现文件实现 URLContext 接口。此文件实现了 file 广义协议的 URLContext 接口。

3.5.2 文件注释

```
1   #include "../berrno.h"
2
3   #include "avformat.h"
4   #include <fcntl.h>
5
6   #ifndef CONFIG_WIN32
7   #include <unistd.h>
8   #include <sys/ioctl.h>
9   #include <sys/time.h>
10  #else
11  #include <io.h>
12  #define open(fname,oflag,pmode) _open(fname,oflag,pmode)
13  #endif
14
```

打开本地媒体文件，把本地文件句柄作为广义文件句柄存放在 priv_data 中。

```
15  static int file_open(URLContext *h, const char *filename, int flags)
16  {
17      int access;
18      int fd;
19
```

规整本地路径文件名，去掉前面可能的"file:"字符串。ffplay 把本地文件看做广义 URL 协议。

```
20      strstart(filename, "file:", &filename);
21
```

设置本地文件存取属性。

```
22      if (flags &URL_RDWR)
23          access = O_CREAT | O_TRUNC | O_RDWR;
24      else if (flags &URL_WRONLY)
25          access = O_CREAT | O_TRUNC | O_WRONLY;
26      else
```

```
27          access = O_RDONLY;
28  #if defined(CONFIG_WIN32) || defined(CONFIG_OS2) || defined(__CYGWIN__)
29      access |= O_BINARY;
30  #endif
```

调用 open()打开本地文件，并把本地文件句柄作为广义的 URL 句柄存放在 priv_data 变量中。

```
31      fd = open(filename, access, 0666);
32      if (fd < 0)
33          return  - ENOENT;
34      h->priv_data = (void*)(size_t)fd;
35      return 0;
36  }
37
```

转换广义 URL 句柄为本地文件句柄，调用 read()函数读本地文件。

```
38  static int file_read(URLContext *h, unsigned char *buf, int size)
39  {
40      int fd = (size_t)h->priv_data;
41      return read(fd, buf, size);
42  }
43
```

转换广义 URL 句柄为本地文件句柄，调用 wite()函数写本地文件，本播放器没实际使用此函数。

```
44  static int file_write(URLContext *h, unsigned char *buf, int size)
45  {
46      int fd = (size_t)h->priv_data;
47      return write(fd, buf, size);
48  }
49
```

转换广义 URL 句柄为本地文件句柄，调用 lseek()函数设置本地文件读指针。

```
50  static offset_t file_seek(URLContext *h, offset_t pos, int whence)
51  {
52      int fd = (size_t)h->priv_data;
53      return lseek(fd, pos, whence);
54  }
55
```

转换广义 URL 句柄为本地文件句柄，调用 close()函数关闭本地文件。

```
56  static int file_close(URLContext *h)
57  {
```

```
58      int fd = (size_t)h->priv_data;
59      return close(fd);
60  }
61
```

用 file 协议相应函数初始化 URLProtocol 结构。

```
62  URLProtocol file_protocol =
63  {
64      "file",
65      file_open,
66      file_read,
67      file_write,
68      file_seek,
69      file_close,
70  };
```

3.6 avio.h 文件

3.6.1 功能描述

文件读写模块定义的数据结构和函数声明，ffplay 把这些全部放到这个.h 文件中。

3.6.2 文件注释

```
1   #ifndef AVIO_H
2   #define AVIO_H
3
4   #define URL_EOF (-1)
5
6   typedef int64_t offset_t;
7
```

简单的文件存取宏定义

```
8   #define URL_RDONLY 0
9   #define URL_WRONLY 1
10  #define URL_RDWR   2
11
```

URLContext 结构表示程序运行的当前广义文件协议使用的上下文，着重于所有广义文件协议共有的属性(并且是在程序运行时才能确定其值)和关联其他结构的字段。

```
12  typedef struct URLContext
13  {
14      struct URLProtocol *prot; // 关联相应的广义输入文件协议。
15      int flags;
16      int max_packet_size;      // 如果非 0，表示最大包大小，用于分配足够的缓存。
17      void *priv_data;          // 在本例中，关联一个文件句柄
18      char filename[1];         // 在本例中，存取本地文件名，filename 仅指示本地文件名首地址。
19  } URLContext;
20
```

URLProtocol 定义广义的文件协议，着重于功能函数，一种广义的文件协议对应一个 URLProtocol 结构，本例删掉了 pipe，udp，tcp 等输入协议，仅保留一个 file 协议。

```
21  typedef struct URLProtocol
22  {
23      const char *name;           // 协议文件名，便于人性化阅读理解。
24      int(*url_open)(URLContext *h, const char *filename, int flags);
25      int(*url_read)(URLContext *h, unsigned char *buf, int size);
26      int(*url_write)(URLContext *h, unsigned char *buf, int size);
```

```
27      offset_t(*url_seek)(URLContext *h, offset_t pos, int whence);
28      int(*url_close)(URLContext *h);
29      struct URLProtocol *next;  // 把所有支持的输入协议串链起来，便于遍历查找。
30  } URLProtocol;
31
```

ByteIOContext 结构扩展 URLProtocol 结构成内部有缓冲机制的广泛意义上的文件，改善广义输入文件的 IO 性能。

主要变量间的逻辑位置关系简单示意如下：

注1：buffer和媒体文件的逻辑示意图，用于缓存的管理，各变量的理解和计算

整个长条代表媒体文件，

灰红表示缓存已使用数据，

亮红表示缓存未使用数据，

亮灰表示文件未读数据。

```
32  typedef struct ByteIOContext
33  {
34      unsigned char *buffer; // 缓存首地址
35      int buffer_size;       // 缓存大小
36      unsigned char *buf_ptr,  *buf_end; // 缓存读指针和末指针
37      void *opaque;          // 指向 URLContext 结构的指针，便于跳转
38      int(*read_packet)(void *opaque, uint8_t *buf, int buf_size);
39      int(*write_packet)(void *opaque, uint8_t *buf, int buf_size);
40      offset_t(*seek)(void *opaque, offset_t offset, int whence);
41      offset_t pos;          // position in the file of the current buffer
42      int must_flush;        // true if the next seek should flush
43      int eof_reached;       // true if eof reached
44      int write_flag;        // true if open for writing
45      int max_packet_size;   // 如果非 0，表示最大数据帧大小，用于分配足够的缓存。
46      int error;             // contains the error code or 0 if no error happened
47  } ByteIOContext;
48
```

相关函数说明参考相应的 c 实现文件。

```
49  int url_open(URLContext **h, const char *filename, int flags);
50  int url_read(URLContext *h, unsigned char *buf, int size);
```

```
51  int url_write(URLContext *h, unsigned char *buf, int size);
52  offset_t url_seek(URLContext *h, offset_t pos, int whence);
53  int url_close(URLContext *h);
54  int url_get_max_packet_size(URLContext *h);
55
56  int register_protocol(URLProtocol *protocol);
57
58  int init_put_byte(ByteIOContext *s,
59                    unsigned char *buffer,
60                    int buffer_size,
61                    int write_flag,
62                    void *opaque,
63                    int(*read_buf)(void *opaque, uint8_t *buf, int buf_size),
64                    int(*write_buf)(void *opaque, uint8_t *buf, int buf_size),
65                    offset_t(*seek)(void *opaque, offset_t offset, int whence));
66
67  offset_t url_fseek(ByteIOContext *s, offset_t offset, int whence);
68  void url_fskip(ByteIOContext *s, offset_t offset);
69  offset_t url_ftell(ByteIOContext *s);
70  offset_t url_fsize(ByteIOContext *s);
71  int url_feof(ByteIOContext *s);
72  int url_ferror(ByteIOContext *s);
73
74  int url_fread(ByteIOContext *s, unsigned char *buf, int size); // get_buffer
75  int get_byte(ByteIOContext *s);
76  unsigned int get_le32(ByteIOContext *s);
77  unsigned int get_le16(ByteIOContext *s);
78
79  int url_setbufsize(ByteIOContext *s, int buf_size);
80  int url_fopen(ByteIOContext *s, const char *filename, int flags);
81  int url_fclose(ByteIOContext *s);
82
83  int url_open_buf(ByteIOContext *s, uint8_t *buf, int buf_size, int flags);
84  int url_close_buf(ByteIOContext *s);
85
86  #endif
```

3.7 avio.c 文件

3.7.1 功能描述

此文件实现了 URLProtocol 抽象层广义文件操作函数，由于 URLProtocol 是底层其他具体文件(file,pipe 等)的简单封装，这一层只是一个中转站，大部分函数都是简单中转到底层的具体实现函数。

3.7.2 文件注释

```
1   #include "../berrno.h"
2   #include "avformat.h"
3
4   URLProtocol *first_protocol = NULL;
5
```

ffplay 抽象底层的 file，pipe 等为 URLProtocol，然后把这些 URLProtocol 串联起来做成链表，便于查找。register_protocol 实际就是串联的各个 URLProtocol，全局表头为 first_protocol。

```
6   int register_protocol(URLProtocol *protocol)
7   {
8       URLProtocol **p;
9       p = &first_protocol;
```

移动指针到 URLProtocol 链表末尾。

```
10      while (*p != NULL)
11          p = &(*p)->next;
```

在 URLProtocol 链表末尾直接挂接当前的 URLProtocol 指针。

```
12      *p = protocol;
13      protocol->next = NULL;
14      return 0;
15  }
16
```

打开广义输入文件。此函数主要有三部分逻辑，首先从文件路径名中分离出协议字符串到 proto_str 字符数组中，接着遍历 URLProtocol 链表查找匹配 proto_str 字符数组中的字符串来确定使用的协议，最后调用相应的文件协议的打开函数打开输入文件。

```
17  int url_open(URLContext **puc, const char *filename, int flags)
18  {
19      URLContext *uc;
20      URLProtocol *up;
21      const char *p;
22      char proto_str[128],  *q;
```

```
23      int err;
24
```

以冒号和结束符作为边界从文件名中分离出的协议字符串到 proto_str 字符数组。由于协议只能是字符，所以在边界前识别到非字符就断定是 file。

```
25      p = filename;
26      q = proto_str;
27      while (*p != '\0' &&  *p != ':')
28      {
29          if (!isalpha(*p))  // protocols can only contain alphabetic chars
30              goto file_proto;
31          if ((q - proto_str) < sizeof(proto_str) - 1)
32              *q++ =  *p;
33          p++;
34      }
35
```

如果协议字符串只有一个字符，我们就认为是 windows 下的逻辑盘符，断定是 file。

```
36      if (*p == '\0' || (q - proto_str) <= 1)
37      {
38  file_proto:
39          strcpy(proto_str, "file");
40      }
41      else
42      {
43          *q = '\0';
44      }
45
```

遍历 URLProtocol 链表匹配使用的协议，如果没有找到就返回错误码。

```
46      up = first_protocol;
47      while (up != NULL)
48      {
49          if (!strcmp(proto_str, up->name))
50              goto found;
51          up = up->next;
52      }
53      err =  - ENOENT;
54      goto fail;
55  found:
```

如果找到就分配 URLContext 结构内存，特别注意内存大小要加上文件名长度，文件名字符串结束标记 0 也要预先分配 1 个字节内存，这 1 个字节就是 URLContext 结构中的 char filename[1]。

```
56      uc = av_malloc(sizeof(URLContext) + strlen(filename));
57      if (!uc)
58      {
59          err =  - ENOMEM;
60          goto fail;
61      }
```

strcpy 函数会自动在 filename 字符数组后面补 0 作为字符串结束标记，所以不用特别赋值为 0。

```
62      strcpy(uc->filename, filename);
63      uc->prot = up;
64      uc->flags = flags;
65      uc->max_packet_size = 0; // default: stream file
```

接着调用相应协议的文件打开函数实质打开文件。如果文件打开错误，就需要释放 malloc 出来的内存，并返回错误码。

```
66      err = up->url_open(uc, filename, flags);
67      if (err < 0)
68      {
69          av_free(uc);    // 打开失败，释放刚刚分配的内存。
70          *puc = NULL;
71          return err;
72      }
73      *puc = uc;
74      return 0;
75  fail:
76      *puc = NULL;
77      return err;
78  }
79
```

简单的中转读操作到底层协议的读函数，完成读操作。

```
80  int url_read(URLContext *h, unsigned char *buf, int size)
81  {
82      int ret;
83      if (h->flags &URL_WRONLY)
84          return AVERROR_IO;
85      ret = h->prot->url_read(h, buf, size);
86      return ret;
```

```
87  }
88
```

简单的中转 seek 操作到底层协议的 seek 函数，完成 seek 操作。

```
89  offset_t url_seek(URLContext *h, offset_t pos, int whence)
90  {
91      offset_t ret;
92
93      if (!h->prot->url_seek)
94          return  - EPIPE;
95      ret = h->prot->url_seek(h, pos, whence);
96      return ret;
97  }
98
```

简单的中转关闭操作到底层协议的关闭函数，完成关闭操作，并释放在 url_open()函数中 malloc 出来的内存。

```
99  int url_close(URLContext *h)
100 {
101     int ret;
102
103     ret = h->prot->url_close(h);
104     av_free(h);
105     return ret;
106 }
107
```

取最大数据包大小，如果非 0，必须是实质有效的。

```
108 int url_get_max_packet_size(URLContext *h)
109 {
110     return h->max_packet_size;
111 }
```

3.8 aviobuf.c 文件

3.8.1 功能描述

有缓存的广义文件 ByteIOContext 相关的文件操作，比如 open，read，close，seek 等等。

3.8.2 文件注释

```
1   #include "../berrno.h"
2   #include "avformat.h"
3   #include "avio.h"
4   #include <stdarg.h>
5
6   #define IO_BUFFER_SIZE 32768
7
```

初始化广义文件 ByteIOContext 结构，一些简单的赋值操作。

```
8   int init_put_byte(ByteIOContext *s,
9                     unsigned char *buffer,
10                    int buffer_size,
11                    int write_flag,
12                    void *opaque,
13                    int(*read_buf)(void *opaque, uint8_t *buf, int buf_size),
14                    int(*write_buf)(void *opaque, uint8_t *buf, int buf_size),
15                    offset_t(*seek)(void *opaque, offset_t offset, int whence))
16  {
17      s->buffer = buffer;
18      s->buffer_size  = buffer_size;
19      s->buf_ptr = buffer;
20      s->write_flag = write_flag;
21      if (!s->write_flag)
22          s->buf_end = buffer; // 初始情况下，缓存中没有效数据，所以 buf_end 指向缓存首地址。
23      else
24          s->buf_end = buffer + buffer_size;
25      s->opaque = opaque;
26      s->write_buf = write_buf;
27      s->read_buf = read_buf;
28      s->seek = seek;
29      s->pos = 0;
30      s->must_flush = 0;
31      s->eof_reached = 0;
32      s->error = 0;
```

```
33      s->max_packet_size = 0;
34      return 0;
35  }
36
```

广义文件 ByteIOContext 的 seek 操作。
输入变量：s 为广义文件句柄，offset 为偏移量，whence 为定位方式。
输出变量：相对广义文件开始的偏移量。

```
37  offset_t url_fseek(ByteIOContext *s, offset_t offset, int whence)
38  {
39      offset_t offset1;
40
```

只支持 SEEK_CUR 和 SEEK_SET 定位方式，不支持 SEEK_END 方式。
SEEK_CUR:从文件当前读写位置为基准偏移 offset 字节。
SEEK_SET:从文件开始位置偏移 offset 字节。

```
41      if (whence != SEEK_CUR && whence != SEEK_SET)
42          return  - EINVAL;
43
```

ffplay 把 SEEK_CUR 和 SEEK_SET 统一成 SEEK_SET 方式处理，所以如果是 SEEK_CUR 方式就要转换成 SEEK_SET 的偏移量。
offset1 = s->pos - (s->buf_end - s->buffer) + (s->buf_ptr - s->buffer) 算式关系请参照 3.6 节的示意图，表示广义文件的当前实际偏移。

```
44      if (whence == SEEK_CUR)
45      {
46          offset1 = s->pos - (s->buf_end - s->buffer) + (s->buf_ptr - s->buffer);
47          if (offset == 0)
48              return offset1; // 如果偏移量为 0，返回实际偏移位置。
```

计算绝对偏移量，赋值给 offset。

```
49          offset += offset1;  // 加上实际偏移量，统一成相对广义文件开始的绝对偏移量
50      }
```

计算绝对偏移量相对当前缓存的偏移量，赋值给 offset1。

```
51      offset1 = offset - (s->pos - (s->buf_end - s->buffer));
```

判断绝对偏移量是否在当前缓存中，如果在当前缓存中，就简单的修改 buf_ptr 指针。

```
52      if (offset1 >= 0 && offset1 <= (s->buf_end - s->buffer))
53      {
54          s->buf_ptr = s->buffer + offset1; // can do the seek inside the buffer
```

```
55      }
56      else
57      {
```

判断当前广义文件是否可以 seek，如果不能 seek 就返回错误。

```
58          if (!s->seek)
59              return  - EPIPE;
```

调用底层具体的文件系统的 seek 函数完成实际的 seek 操作，此时缓存需重新初始化，buf_end 重新指向缓存首地址，并修改 pos 变量为广义文件当前实际偏移量。

```
60          s->buf_ptr = s->buffer;
61          s->buf_end = s->buffer;
62          if (s->seek(s->opaque, offset, SEEK_SET) == (offset_t) - EPIPE)
63              return  - EPIPE;
64          s->pos = offset;
65      }
66      s->eof_reached = 0;
67
```

返回广义文件当前的实际偏移量。

```
68      return offset;
69  }
70
```

广义文件 ByteIOContext 的当前实际偏移量再偏移 offset 字节，调用 url_fseek 实现。

```
71  void url_fskip(ByteIOContext *s, offset_t offset)
72  {
73      url_fseek(s, offset, SEEK_CUR);
74  }
75
```

返回广义文件 ByteIOContext 的当前实际偏移量。

```
76  offset_t url_ftell(ByteIOContext *s)
77  {
78      return url_fseek(s, 0, SEEK_CUR);
79  }
80
```

返回广义文件 ByteIOContext 的大小。

```
81  offset_t url_fsize(ByteIOContext *s)
82  {
```

```
83      offset_t size;
84
```

判断当前广义文件 ByteIOContext 是否能 seek，如果不能就返回错误

```
85      if (!s->seek)
86          return  - EPIPE;
```

调用底层的 seek 函数取得文件大小。

```
87      size = s->seek(s->opaque,  - 1, SEEK_END) + 1;
```

注意 seek 操作改变了读指针，所以要重新 seek 到当前读指针位置。

```
88      s->seek(s->opaque, s->pos, SEEK_SET);
89      return size;
90  }
91
```

判断当前广义文件 ByteIOContext 是否到末尾

```
92  int url_feof(ByteIOContext *s)
93  {
94      return s->eof_reached;
95  }
96
```

返回当前广义文件 ByteIOContext 操作错误码

```
97  int url_ferror(ByteIOContext *s)
98  {
99      return s->error;
100 }
101
102 // Input stream
103
```

填充广义文件 ByteIOContext 内部的数据缓存区。

```
104 static void fill_buffer(ByteIOContext *s)
105 {
106     int len;
107
```

如果到了广义文件 ByteIOContext 末尾就直接返回。

```
108     if (s->eof_reached)
109         return ;
```

```
110
```

调用底层文件系统的读函数实际读数据填到缓存，注意这里经过了好几次跳转才到底层读函数。首先跳转的 url_read_buf()函数，再跳转到 url_read()，再跳转到实际文件协议的读函数完成读操作。

```
111     len = s->read_buf(s->opaque, s->buffer, s->buffer_size); // url_read_buf //
112     if (len <= 0)
113     {
```

如果是到达文件末尾就不要改 buffer 参数，这样不用重新读数据就可以做 seek back 操作。

```
114         s->eof_reached = 1;
115
```

设置错误码，便于分析定位。

```
116         if (len < 0)
117             s->error = len;
118     }
119     else
120     {
```

如果正确读取，修改一下基本参数。参加 3.6 节中的示意图。

```
121         s->pos += len;
122         s->buf_ptr = s->buffer;
123         s->buf_end = s->buffer + len;
124     }
125 }
126
```

从广义文件 ByteIOContext 中读取一个字节。

```
127 int get_byte(ByteIOContext *s)
128 {
129     if (s->buf_ptr < s->buf_end)
130     {
```

如果广义文件 ByteIOContext 内部缓存有数据，就修改读指针，返回读取的数据。

```
131         return  *s->buf_ptr++;
132     }
133     else
134     {
```

如果广义文件 ByteIOContext 内部缓存没有数据，就先填充内部缓存。

```
135         fill_buffer(s);
```

如果广义文件 ByteIOContext 内部缓存有数据，就修改读指针，返回读取的数据。如果没有数据就是到了文件末尾，返回 0。
NOTE: return 0 if EOF, so you cannot use it if EOF handling is necessary

```
136         if (s->buf_ptr < s->buf_end)
137             return  *s->buf_ptr++;
138         else
139             return 0;
140     }
141 }
142
```

从广义文件 ByteIOContext 中以小端方式读取两个字节,实现代码充分复用 get_byte()函数。

```
143 unsigned int get_le16(ByteIOContext *s)
144 {
145     unsigned int val;
146     val = get_byte(s);
147     val |= get_byte(s) << 8;
148     return val;
149 }
150
```

从广义文件 ByteIOContext 中以小端方式读取四个字节,实现代码充分复用 get_le16()函数。

```
151 unsigned int get_le32(ByteIOContext *s)
152 {
153     unsigned int val;
154     val = get_le16(s);
155     val |= get_le16(s) << 16;
156     return val;
157 }
158
159 #define url_write_buf NULL
160
```

简单中转读操作函数。

```
161 static int url_read_buf(void *opaque, uint8_t *buf, int buf_size)
162 {
163     URLContext *h = opaque;
164     return url_read(h, buf, buf_size);
165 }
166
```

简单中转 seek 操作函数。

```
167 static offset_t url_seek_buf(void *opaque, offset_t offset, int whence)
168 {
169     URLContext *h = opaque;
170     return url_seek(h, offset, whence);
171 }
172
```

设置并分配广义文件 ByteIOContext 内部缓存的大小。更多的应用在修改内部缓存大小场合。

```
173 int url_setbufsize(ByteIOContext *s, int buf_size) // must be called before any I/O
174 {
175     uint8_t *buffer;
```

分配广义文件 ByteIOContext 内部缓存。

```
176     buffer = av_malloc(buf_size);
177     if (!buffer)
178         return  - ENOMEM;
179
```

释放掉原来广义文件 ByteIOContext 的内部缓存，这是一个保险的操作。

```
180     av_free(s->buffer);
```

设置广义文件 ByteIOContext 内部缓存相关参数。

```
181     s->buffer = buffer;
182     s->buffer_size = buf_size;
183     s->buf_ptr = buffer;
184     if (!s->write_flag)
185         s->buf_end = buffer; // 因为此时只是分配了内存,并没有读入数据,所以 buf_end 指向首地址
186     else
187         s->buf_end = buffer + buf_size;
188     return 0;
189 }
190
```

打开广义文件 ByteIOContext

```
191 int url_fopen(ByteIOContext *s, const char *filename, int flags)
192 {
193     URLContext *h;
194     uint8_t *buffer;
195     int buffer_size, max_packet_size;
```

```
196     int err;
197
```

调用底层文件系统的 open 函数实质性打开文件

```
198     err = url_open(&h, filename, flags);
199     if (err < 0)
200         return err;
201
```

读取底层文件系统支持的最大包大小。如果非 0，则设置为内部缓存的大小；否则内部缓存设置为默认大小 IO_BUFFER_SIZE(32768 字节)。

```
202     max_packet_size = url_get_max_packet_size(h);
203     if (max_packet_size)
204     {
205         buffer_size = max_packet_size; // no need to bufferize more than one packet
206     }
207     else
208     {
209         buffer_size = IO_BUFFER_SIZE;
210     }
211
```

分配广义文件 ByteIOContext 内部缓存，如果错误就关闭文件返回错误码。

```
212     buffer = av_malloc(buffer_size);
213     if (!buffer)
214     {
215         url_close(h);
216         return  - ENOMEM;
217     }
218
```

初始化广义文件 ByteIOContext 数据结构，如果错误就关闭文件，释放内部缓存，返回错误码

```
219     if (init_put_byte(s,
220                       buffer,
221                       buffer_size,
222                       (h->flags &URL_WRONLY || h->flags &URL_RDWR),
223                       h,
224                       url_read_buf,
225                       url_write_buf,
226                       url_seek_buf) < 0)
227     {
```

```
228         url_close(h);
229         av_free(buffer);
230         return AVERROR_IO;
231     }
232
```

保存最大包大小。

```
233     s->max_packet_size = max_packet_size;
234
235     return 0;
236 }
237
```

关闭广义文件 ByteIOContext，首先释放掉内部使用的缓存，再把自己的字段置 0，最后转入底层文件系统的关闭函数实质性关闭文件。

```
238 int url_fclose(ByteIOContext *s)
239 {
240     URLContext *h = s->opaque;
241
242     av_free(s->buffer);
243     memset(s, 0, sizeof(ByteIOContext));
244     return url_close(h);
245 }
246
```

广义文件 ByteIOContext 读操作，注意此函数从 get_buffer 改名而来，更贴切函数功能，也为了完备广义文件操作函数集。

```
247 int url_fread(ByteIOContext *s, unsigned char *buf, int size) // get_buffer
248 {
249     int len, size1;
250
```

考虑到 size 可能比缓存中的数据大得多，此时就多次读缓存，所以用 size1 保存要读取的总字节数，size 意义变更为还需要读取的字节数。

```
251     size1 = size;
```

如果还需要读的字节数大于 0，就进入循环继续读。

```
252     while (size > 0)
253     {
```

计算当次循环应该读取的字节数 len，首先设置 len 为内部缓存数据长度，再和需要读的字节数 size 比，有条件修正 len 的值。

```
254         len = s->buf_end - s->buf_ptr;
255         if (len > size)
256             len = size;
257         if (len == 0)
258         {
```

如果内部缓存没有数据。

```
259             if (size > s->buffer_size) // 读操作是否绕过内部缓存的判别条件
260             {
```

如果要读取的数据量比内部缓存数据量大，就调用底层函数读取数据绕过内部缓存直接到目标缓存。

```
261                 len = s->read_buf(s->opaque, buf, size);
262                 if (len <= 0)
263                 {
```

如果底层文件系统读错误，设置文件末尾标记和错误码，跳出循环，返回实际读到的字节数。

```
264                     s->eof_reached = 1;
265                     if (len < 0)
266                         s->error = len;
267                     break;
268                 }
269                 else
270                 {
```

如果底层文件系统正确读，修改相关参数，进入下一轮循环。特别注意此处读文件绕过了内部缓存。

```
271                     s->pos += len;
272                     size -= len;
273                     buf += len;   // 因为绕过了内部缓存，特别注意此处的修改
274                     s->buf_ptr = s->buffer;
275                     s->buf_end = s->buffer /* +len */;//因为绕过了内部缓存,特别注意此处
276                 }
277             }
278             else
279             {
```

如果要读取的数据量比内部缓存数据量小，就调用底层函数读取数据到内部缓存，判断读成果否。

```
280                 fill_buffer(s);
281                 len = s->buf_end - s->buf_ptr;
```

```
282                 if (len == 0)
283                     break;
284             }
285         }
286         else
287         {
```

如果内部缓存有数据，就拷贝 len 长度的数据到缓存区，并修改相关参数，进入下一个循环的条件判断。

```
288             memcpy(buf, s->buf_ptr, len);
289             buf += len;
290             s->buf_ptr += len;
291             size -= len;
292         }
293     }
```

返回实际读取的字节数。

```
294     return size1 - size;
295 }
```

3.9 utils_format.c 文件

3.9.1 功能描述

识别文件格式和媒体格式部分使用的一些工具类函数。

3.9.2 文件注释

```
1   #include "../berrno.h"
2   #include "avformat.h"
3   #include <assert.h>
4
5   #define UINT_MAX  (0xffffffff)
6
7   #define PROBE_BUF_MIN 2048
8   #define PROBE_BUF_MAX 131072
9
10  AVInputFormat *first_iformat = NULL;
11
```

注册文件容器格式。ffplay 把所有支持的文件容器格式用链表串联起来，表头是 first_iformat。

```
12  void av_register_input_format(AVInputFormat *format)
13  {
14      AVInputFormat **p;
15      p = &first_iformat;
```

循环移动节点指针到最后一个文件容器格式。

```
16      while (*p != NULL)
17          p = &(*p)->next;
```

直接挂接要注册的文件容器格式。

```
18      *p = format;
19      format->next = NULL;
20  }
21
```

比较文件的扩展名来识别文件类型。

```
22  int match_ext(const char *filename, const char *extensions)
23  {
24      const char *ext,  *p;
25      char ext1[32],  *q;
26
```

如果输入文件为空就直接返回。

```
27      if (!filename)
28          return 0;
29
```

用'.'号作为扩展名分割符，在文件名中找扩展名分割符。

```
30      ext = strrchr(filename, '.');
31      if (ext)
32      {
33          ext++;
34          p = extensions;
35          for (;;)
36          {
```

文件名中可能有多个标点符号，取两个标点符号间或一个标点和一个结束符间的字符串和扩展名比较来判断文件类型，所以可能要多次比较，所以这里有一个循环。

```
37              q = ext1;
```

定位下一个标点符号或字符串结束符，把这之间的字符拷贝到扩展名字符数组中。

```
38              while (*p != '\0' &&  *p != ',' && q - ext1 < sizeof(ext1) - 1)
39                  *q++ = *p++;
```

添加扩展名字符串结束标记 0。

```
40              *q = '\0';
```

比较识别的扩展名是否后给定的扩展名相同，如果相同就返回 1，否则继续。

```
41              if (!strcasecmp(ext1, ext))
42                  return 1;
```

判断是否到了文件名末尾，如果是就返回，否则进入下一个循环

```
43              if (*p == '\0')
44                  break;
45              p++;
46          }
47      }
```

如果在前面的循环中没有匹配到扩展名，就是不识别的文件类型，返回 0

```
48      return 0;
49  }
50
```

探测输入的文件容器格式，返回识别出来的文件格式。如果没有识别出来，就返回初始值 NULL。

```
51  AVInputFormat *av_probe_input_format(AVProbeData *pd, int is_opened)
52  {
53      AVInputFormat *fmt1,  *fmt;
54      int score, score_max;
55
56      fmt = NULL;
```

score，score_max 可以理解识别文件容器格式的正确级别。文件容器格式识别结果，如果完全正确可以设定为 100，如果可能正确可以设定为 50，没识别出来设定为 0。识别方法不同导致等级不同。

```
57      score_max = 0;
58      for (fmt1 = first_iformat; fmt1 != NULL; fmt1 = fmt1->next)
59      {
60          if (!is_opened)
61              continue;
62
63          score = 0;
64          if (fmt1->read_probe)
65          {
```

读取文件头，判断文件头的内容来识别文件容器格式，这种识别方法非常可靠，设定 score 为 100。

```
66              score = fmt1->read_probe(pd);
67          }
68          else if (fmt1->extensions)
69          {
```

通过文件扩展名来识别文件容器格式，因为文件扩展名任何人都可以改，如果改变扩展名，这种方法就错误，如果不改变扩展名，这种识别方法有点可靠，综合等级为 50。

```
70              if (match_ext(pd->filename, fmt1->extensions))
71                  score = 50;
72          }
```

如果识别出来的等级大于最大要求的等级，就认为正确识别，相关参数赋值后，进下一个循环，最后返回最高级别对应的文件容器格式。

```
73          if (score > score_max)
74          {
75              score_max = score;
76              fmt = fmt1;
77          }
78      }
```

返回文件容器格式，如果没有识别出来，返回的是初始值 NULL。

```
79      return fmt;
80  }
81
```

打开输入流，其中 AVFormatParameters *ap 参数在瘦身后的 ffplay 中没有用到，保留为了不改变接口。

```
82  int av_open_input_stream(AVFormatContext **ic_ptr, ByteIOContext *pb, const char *filename,
83                           AVInputFormat *fmt, AVFormatParameters *ap)
84  {
85      int err;
86      AVFormatContext *ic;
87      AVFormatParameters default_ap;
88
89      if (!ap)
90      {
91          ap = &default_ap;
92          memset(ap, 0, sizeof(default_ap));
93      }
94
```

分配 AVFormatContext 内存，部分成员变量在接下来的程序代码中赋值，部分成员变量在下面调用的 ic->iformat->read_header(ic, ap)函数中赋值。

```
95      ic = av_mallocz(sizeof(AVFormatContext));
96      if (!ic)
97      {
98          err = AVERROR_NOMEM;
99          goto fail;
100     }
```

关联 AVFormatContext 和 AVInputFormat

```
101     ic->iformat = fmt;
```

关联 AVFormatContext 和广义文件 ByteIOContext

```
102     if (pb)
103         ic->pb =  *pb;
104
105     if (fmt->priv_data_size > 0)
106     {
```

分配 priv_data 指向的内存。

```
107         ic->priv_data = av_mallocz(fmt->priv_data_size);
108         if (!ic->priv_data)
109         {
110             err = AVERROR_NOMEM;
111             goto fail;
112         }
113     }
114     else
115     {
116         ic->priv_data = NULL;
117     }
118
```

读取文件头，识别媒体流格式。

```
119     err = ic->iformat->read_header(ic, ap);
120     if (err < 0)
121         goto fail;
122
123     *ic_ptr = ic;
124     return 0;
125
```

简单常规的错误处理。

```
126 fail:
127     if (ic)
128         av_freep(&ic->priv_data);
129
130     av_free(ic);
131     *ic_ptr = NULL;
132     return err;
133 }
134
```

打开输入文件，并识别文件格式，然后调用函数识别媒体流格式。

```
135 int av_open_input_file(AVFormatContext **ic_ptr, const char *filename, AVInputFormat *fmt,
136                         int buf_size, AVFormatParameters *ap)
137 {
138     int err, must_open_file, file_opened, probe_size;
139     AVProbeData probe_data,  *pd = &probe_data;
140     ByteIOContext pb1,  *pb = &pb1;
141
```

```
142     file_opened = 0;
143     pd->filename = "";
144     if (filename)
145         pd->filename = filename;
146     pd->buf = NULL;
147     pd->buf_size = 0;
148
149     must_open_file = 1;
150
151     if (!fmt || must_open_file)
152     {
```

打开输入文件，关联 ByteIOContext，经过跳转几次后才实质调用文件系统 open()函数实质打开文件。

```
153         if (url_fopen(pb, filename, URL_RDONLY) < 0)
154         {
155             err = AVERROR_IO;
156             goto fail;
157         }
158         file_opened = 1;
```

如果程序指定 ByteIOContext 内部使用的缓存大小，就重新设置内部缓存大小。通常不指定大小。

```
159         if (buf_size > 0)
160             url_setbufsize(pb, buf_size);
161
```

先读 PROBE_BUF_MIN(2048)字节文件开始数据识别文件格式，如果不能识别文件格式，就把识别文件缓存以 2 倍的增长扩大再读文件开始数据识别，直到识别出文件格式或者超过 131072 字节缓存。

```
162         for (probe_size = PROBE_BUF_MIN; probe_size <= PROBE_BUF_MAX && !fmt; probe_size <<= 1)
163         {
```

重新分配缓存，重新读文件开始数据。

```
164             pd->buf = av_realloc(pd->buf, probe_size);
165             pd->buf_size = url_fread(pb, pd->buf, probe_size);
```

把文件读指针 seek 到文件开始处，便于下一次读。

```
166             if (url_fseek(pb, 0, SEEK_SET) == (offset_t) - EPIPE)
167             {
```

如果 seek 错误，关闭文件，再重新打开。

```
168                 url_fclose(pb);
169                 if (url_fopen(pb, filename, URL_RDONLY) < 0)
```

```
170                 {
```

重新打开文件出错，设置错误码，跳到错误处理。

```
171                     file_opened = 0;
172                     err = AVERROR_IO;
173                     goto fail;
174                 }
175             }
176
```

重新识别文件格式，因为一次比一次数据多，数据少的时候可能识别不出，数据多了可能就可以了。

```
177             fmt = av_probe_input_format(pd, 1);
178         }
179         av_freep(&pd->buf);
180     }
181
182     if (!fmt)
183     {
184         err = AVERROR_NOFMT;
185         goto fail;
186     }
187
```

识别出文件格式后，调用函数识别流 av_open_input_stream 格式。

```
188     err = av_open_input_stream(ic_ptr, pb, filename, fmt, ap);
189     if (err)
190         goto fail;
191     return 0;
192
193 fail:
```

简单的异常错误处理。

```
194     av_freep(&pd->buf);
195     if (file_opened)
196         url_fclose(pb);
197     *ic_ptr = NULL;
198     return err;
199 }
200
```

一次读取一个数据包，在瘦身后的 ffplay 中，一次读取一个完整的数据帧，数据包。

```
201 int av_read_packet(AVFormatContext *s, AVPacket *pkt)
202 {
203     return s->iformat->read_packet(s, pkt);
204 }
205
```

添加索引到索引表。有些媒体文件为便于 seek，有音视频数据帧有索引，ffplay 把这些索引以时间排序放到一个数据中。返回值添加项的索引。

```
206 int av_add_index_entry(AVStream *st, int64_t pos, int64_t timestamp, int size, int distance, int flags)
207 {
208     AVIndexEntry *entries,  *ie;
209     int index;
210
```

索引项越界判断，如果占有内存达到 UINT_MAX 时，返回。

```
211     if ((unsigned)st->nb_index_entries + 1 >= UINT_MAX / sizeof(AVIndexEntry))
212         return  - 1;
213
```

重新分配索引内存。注意 av_fast_realloc()函数并不是每次调用就一定会重新分配内存，那样效率就太低了。

```
214     entries = av_fast_realloc(st->index_entries, &st->index_entries_allocated_size,
215                               (st->nb_index_entries + 1) * sizeof(AVIndexEntry));
216     if (!entries)
217         return  - 1;
218
```

保持重新分配内存后，索引的首地址。

```
219     st->index_entries = entries;
220
```

以时间为顺序查找当前索引应该插在索引表的位置。

```
221     index = av_index_search_timestamp(st, timestamp, AVSEEK_FLAG_ANY);
222
223     if (index < 0)
224     {
```

续补，既接着最后一个插入，索引计算加 1，取得索引项指针，便于后面赋值操作。

```
225         index = st->nb_index_entries++;
226         ie = &entries[index];
227         assert(index == 0 || ie[ - 1].timestamp < timestamp);
```

```
228     }
229     else
230     {
```

中插，既插入索引表的中间，取得索引项指针，便于后面赋值操作。

```
231         ie = &entries[index];
232         if (ie->timestamp != timestamp)
233         {
234             if (ie->timestamp <= timestamp)
235                 return  - 1;
236
```

把索引项后面的项全部后移一项，空出当前索引项。

```
237             memmove(entries + index + 1, entries + index,
238                                 sizeof(AVIndexEntry)*(st->nb_index_entries - index));
239
```

索引项计数加 1。

```
240             st->nb_index_entries++;
241         }
242     }
243
```

修改索引项参数，完成排序添加。

```
244     ie->pos = pos;
245     ie->timestamp = timestamp;
246     ie->size  = size;
247     ie->flags = flags;
248
```

返回索引。

```
249     return index;
250 }
251
```

以时间为关键字查找当前索引应排在索引表中的位置。

```
252 int av_index_search_timestamp(AVStream *st, int64_t wanted_timestamp, int flags)
253 {
254     AVIndexEntry *entries = st->index_entries;
255     int nb_entries = st->nb_index_entries;
256     int a, b, m;
```

```
257     int64_t timestamp;
258
259     a =  - 1;
260     b = nb_entries;
261
```

以时间为关键字折半查找位置，请仔细理解。

```
262     while (b - a > 1)
263     {
264         m = (a + b) >> 1;
265         timestamp = entries[m].timestamp;
266         if (timestamp >= wanted_timestamp)
267             b = m;
268         if (timestamp <= wanted_timestamp)
269             a = m;
270     }
271
272     m = (flags &AVSEEK_FLAG_BACKWARD) ? a : b;
273
274     if (!(flags &AVSEEK_FLAG_ANY))
275     {
```

Seek 时，找关键帧，从关键帧开始解码，注意有些帧解码但不显示。

```
276         while (m >= 0 && m < nb_entries && !(entries[m].flags &AVINDEX_KEYFRAME))
277         {
278             m += (flags &AVSEEK_FLAG_BACKWARD) ?  - 1: 1;
279         }
280     }
281
282     if (m == nb_entries)
283         return  - 1;
284
```

返回找到的位置。

```
285     return m;
286 }
287
```

关闭输入媒体文件，一大堆的关闭释放操作。

```
288 void av_close_input_file(AVFormatContext *s)
289 {
```

```
290     int i;
291     AVStream *st;
292
293     if (s->iformat->read_close)
294         s->iformat->read_close(s);
295
296     for (i = 0; i < s->nb_streams; i++)
297     {
298         st = s->streams[i];
299         av_free(st->index_entries);
300         av_free(st->actx);
301         av_free(st);
302     }
303
304     url_fclose(&s->pb);
305
306     av_freep(&s->priv_data);
307     av_free(s);
308 }
309
```

new 一个新的媒体流，返回 AVStream 指针

```
310 AVStream *av_new_stream(AVFormatContext *s, int id)
311 {
312     AVStream *st;
313
```

判断媒体流的数目是否超限，如果超过就丢弃当前流返回 NULL。

```
314     if (s->nb_streams >= MAX_STREAMS)
315         return NULL;
316
```

分配一块 AVStream 内存。

```
317     st = av_mallocz(sizeof(AVStream));
318     if (!st)
319         return NULL;
320
```

通过 avcodec_alloc_context 分配一块 AVFormatContext 内存，并关联到 AVStream。

```
321     st->actx = avcodec_alloc_context();
322
```

关联 AVFormatContext 和 AVStream。

```
323     s->streams[s->nb_streams++] = st;
324     return st;
325 }
326
```

设置计算 pts 时钟的相关参数。

```
327 void av_set_pts_info(AVStream *s, int pts_wrap_bits, int pts_num, int pts_den)
328 {
329     s->time_base.num = pts_num;
330     s->time_base.den = pts_den;
331 }
```

3.10 avidec.c 文件

3.10.1 功能描述

AVI 文件解析的相关函数，注意有些地方有些技巧性代码。

注意 1：AVI 文件容器媒体数据有两种存放方式，非交织存放和交织存放。交织存放就是音视频数据以帧为最小连续单位，相互间隔存放，这样音视频帧互相交织在一起，并且存放的间隔没有特别规定；非交织存放就是把单一媒体的所有数据帧连续存放在一起，非交织存放的 avi 文件很少。

注意 2：AVI 文件索引结构 AVIINDEXENTRY 中的 dwChunkOffset 字段指示的偏移有的是相对文件开始字节的偏移，有的事相对文件数据块 chunk 的偏移。

注意 3：附带的 avi 测试文件是交织存放的。

3.10.2 文件注释

```
1   #include "avformat.h"
2
3   #include <assert.h>
4
```

几个简单的宏定义。

```
5   #define AVIIF_INDEX         0x10
6
7   #define AVIF_HASINDEX       0x00000010  // Index at end of file?
8   #define AVIF_MUSTUSEINDEX   0x00000020
9
10  #define INT_MAX 2147483647
11
12  #define MKTAG(a,b,c,d) (a | (b << 8) | (c << 16) | (d << 24))
13
14  #define FFMIN(a,b) ((a) > (b) ? (b) : (a))
15  #define FFMAX(a,b) ((a) > (b) ? (a) : (b))
16
17  static int avi_load_index(AVFormatContext *s);
18  static int guess_ni_flag(AVFormatContext *s);
19
```

AVI 文件中的流参数定义，和 AVStream 数据结构协作。

```
20  typedef struct AVIStream
21  {
22      int64_t frame_offset; // 帧偏移，视频用帧计数，音频用字节计数，用于计算 pts 表示时间
23      int remaining;        // 表示需要读的数据大小，初值是帧裸数组大小，全部读完后为 0。
24      int packet_size;      // 包大小，非交织和帧裸数据大小相同，交织比帧裸数据大小大 8 字节。
```

```
25
26      int scale;
27      int rate;
28      int sample_size; // size of one sample (or packet) (in the rate/scale sense) in bytes
29
30      int64_t cum_len; // temporary storage (used during seek)
31
32      int prefix;      // normally 'd'<<8 + 'c' or 'w'<<8 + 'b'
33      int prefix_count;
34  } AVIStream;
35
```

AVI 文件中的文件格式参数相关定义，和 AVFormatContext 协作。

```
36  typedef struct
37  {
38      int64_t riff_end;   // RIFF 块大小
39      int64_t movi_list;  // 媒体数据块开始字节相对文件开始字节的偏移
40      int64_t movi_end;   // 媒体数据块开始字节相对文件开始字节的偏移
41      int non_interleaved;// 指示是否是非交织 AVI
42      int stream_index_2; // 为了和 AVPacket 中的 stream_index 相区别加一个后缀。
                            // 指示当前应该读取的流的索引。初值为-1，表示没有确定应该读的流。
                            // 实际表示 AVFormatContext 结构中 AVStream *streams[]数组中的索引。
43  } AVIContext;
44
```

CodecTag 数据结构，用于关联具体媒体格式的 ID 和 Tag 标签。

```
45  typedef struct CodecTag
46  {
47      int id;           // ID 号码
48      unsigned int tag; // 标签
49  } CodecTag;
50
```

瘦身后的 ffplay 支持的一些视频媒体 ID 和 Tag 标签数组。

```
51  const CodecTag codec_bmp_tags[] =
52  {
53      {CODEC_ID_MSRLE, MKTAG('m', 'r', '1', 'e')},
54      {CODEC_ID_MSRLE, MKTAG(0x1, 0x0, 0x0, 0x0)},
55      {CODEC_ID_NONE,  0},
56  };
57
```

瘦身后的 ffplay 支持的一些音频媒体 ID 和 Tag 标签数组。

```
58  const CodecTag codec_wav_tags[] =
59  {
60      {CODEC_ID_TRUESPEECH, 0x22},
61      {0, 0},
62  };
63
```

以媒体 tag 标签为关键字，查找 codec_bmp_tags 或 codec_wav_tags 数组，返回媒体 ID。

```
64  enum CodecID codec_get_id(const CodecTag *tags, unsigned int tag)
65  {
66      while (tags->id != CODEC_ID_NONE)
67      {
```

比较 Tag 关键字，相等时返回对应媒体 ID。

```
68          if (toupper((tag >> 0) &0xFF) == toupper((tags->tag >> 0) &0xFF)
69           && toupper((tag >> 8) &0xFF) == toupper((tags->tag >> 8) &0xFF)
70           && toupper((tag >> 16)&0xFF) == toupper((tags->tag >> 16)&0xFF)
71           && toupper((tag >> 24)&0xFF) == toupper((tags->tag >> 24)&0xFF))
72              return tags->id;
73
```

比较 Tag 关键字，不等移到数组的下一项。

```
74          tags++;
75      }
```

所有关键字都不匹配，返回 CODEC_ID_NONE。

```
76      return CODEC_ID_NONE;
77  }
78
```

校验 AVI 文件，读取 AVI 文件媒体数据块的偏移大小信息，和 avi_probe()函数部分相同。

```
79  static int get_riff(AVIContext *avi, ByteIOContext *pb)
80  {
81      uint32_t tag;
82      tag = get_le32(pb);
83
```

校验 AVI 文件开始关键字串"RIFF"。

```
84      if (tag != MKTAG('R', 'I', 'F', 'F'))
```

```
85          return  - 1;
86
87      avi->riff_end = get_le32(pb); // RIFF chunk size
88      avi->riff_end += url_ftell(pb); // RIFF chunk end
89      tag = get_le32(pb);
90
```

校验 AVI 文件关键字串"AVI "或"AVIX"。

```
91      if (tag != MKTAG('A', 'V', 'I', ' ') && tag != MKTAG('A', 'V', 'I', 'X'))
92          return  - 1;
93
```

如果通过 AVI 文件关键字串"RIFF"和"AVI "或"AVIX"校验，就认为是 AVI 文件，这种方式非常可靠。

```
94      return 0;
95  }
96
```

排序建立 AVI 索引表，函数名为 clean_index,不准确，功能以具体的实现代码为准。

```
97  static void clean_index(AVFormatContext *s)
98  {
99      int i, j;
100
101     for (i = 0; i < s->nb_streams; i++)
102     {
```

对每个流都建一个独立的索引表。

```
103         AVStream *st = s->streams[i];
104         AVIStream *ast = st->priv_data;
105         int n = st->nb_index_entries;
106         int max = ast->sample_size;
107         int64_t pos, size, ts;
108
```

如果索引表项大于 1，则认为索引表已建好，不再排序重建。如果 sample_size 为 0,则没办法重建。

```
109         if (n != 1 || ast->sample_size == 0)
110             continue;
111
```

此种情况多半是用在非交织存储的 avi 音频流。不管交织还是非交织存储，视频流通常都有索引。

防止包太小需要太多的索引项占有大量内存，设定最小帧 size 阈值为 1024。比如有些音频流，最小解码帧只十多个字节，如果文件比较大则在索引上耗费太多内存。

```
112         while (max < 1024)
113             max += max;
114
```

取位置，大小，时钟等基本参数。

```
115         pos = st->index_entries[0].pos;
116         size = st->index_entries[0].size;
117         ts = st->index_entries[0].timestamp;
118
119         for (j = 0; j < size; j += max)
120         {
```

以 max 指定的字节打包成帧，添加到索引表。

```
121             av_add_index_entry(st, pos + j, ts + j / ast->sample_size, FFMIN(max, size - j), 0,
AVINDEX_KEYFRAME);
122         }
123     }
124 }
125
```

读取 AVI 文件头，读取 AVI 文件索引，并识别具体的媒体格式，关联一些数据结构。

```
126 static int avi_read_header(AVFormatContext *s, AVFormatParameters *ap)
127 {
128     AVIContext *avi = s->priv_data;
129     ByteIOContext *pb = &s->pb;
130     uint32_t tag, tag1, handler;
131     int codec_type, stream_index, frame_period, bit_rate;
132     unsigned int size, nb_frames;
133     int i, n;
134     AVStream *st;
135     AVIStream *ast;
136
```

当前应该读取的流的索引赋初值为-1，表示没有确定应该读的流。

```
137     avi->stream_index_2 = -1;
138
```

校验 AVI 文件，读取 AVI 文件媒体数据块的偏移大小信息。

```
139     if (get_riff(avi, pb) < 0)
140         return -1;
141
```

简单变量符初值。

```
142     stream_index =  - 1; // first list tag
143     codec_type =  - 1;
144     frame_period = 0;
145
146     for (;;)
147     {
```

AVI 文件的基本结构是块，一个文件有多个块，并且块还可以内嵌，在这里循环读文件头中的块。

```
148         if (url_feof(pb))
149             goto fail;
150
```

读取每个块的标签和大小。

```
151         tag = get_le32(pb);
152         size = get_le32(pb);
153
154         switch (tag)
155         {
156         case MKTAG('L', 'I', 'S', 'T'):  // ignored, except when start of video packets
157             tag1 = get_le32(pb);
158             if (tag1 == MKTAG('m', 'o', 'v', 'i'))
159             {
```

读取 movi 媒体数据块的偏移和大小。

```
160                 avi->movi_list = url_ftell(pb) - 4;
161                 if (size)
162                     avi->movi_end = avi->movi_list + size;
163                 else
164                     avi->movi_end = url_fsize(pb);
165
```

AVI 文件头后面是 movi 媒体数据块，所以到了 movi 块，文件头肯定读完，需要跳出循环。

```
166                 goto end_of_header; //
167             }
168             break;
169         case MKTAG('a', 'v', 'i', 'h'):  // avi header, using frame_period is bad idea
170             frame_period = get_le32(pb);
171             bit_rate = get_le32(pb) *8;
172             get_le32(pb);
```

读取 non_interleaved 的初值。

```
173             avi->non_interleaved |= get_le32(pb) & AVIF_MUSTUSEINDEX;
174
175             url_fskip(pb, 2 *4);
176             n = get_le32(pb);
177             for (i = 0; i < n; i++)
178             {
```

读取流数目 n 后，分配 AVStream 和 AVIStream 数据结构，在 187 行把它们关联起来。
特别注意 av_new_stream()函数关联 AVFormatContext 和 AVStream 结构，分配关联 AVCodecContext 结构

```
179                 AVIStream *ast;
180                 st = av_new_stream(s, i);
181                 if (!st)
182                     goto fail;
183
184                 ast = av_mallocz(sizeof(AVIStream));
185                 if (!ast)
186                     goto fail;
187                 st->priv_data = ast;
188
189                 st->actx->bit_rate = bit_rate;
190             }
191             url_fskip(pb, size - 7 * 4);
192             break;
193         case MKTAG('s', 't', 'r', 'h'):  // stream header
```

指示当前流在 AVFormatContext 结构中 AVStream *streams[MAX_STREAMS]数组中的索引。

```
194             stream_index++;
```

从 strh 块读取所有流共有的一些信息，跳过有些不用的字段，填写需要的字段。

```
195             tag1 = get_le32(pb);
196             handler = get_le32(pb);
197
198             if (stream_index >= s->nb_streams)
199             {
```

出现这种情况通常代表媒体文件数据有错，ffplay 简单的跳过。

```
200                 url_fskip(pb, size - 8);
201                 break;
202             }
```

```
203             st = s->streams[stream_index];
204             ast = st->priv_data;
205
206             get_le32(pb); // flags
207             get_le16(pb); // priority
208             get_le16(pb); // language
209             get_le32(pb); // initial frame
210             ast->scale = get_le32(pb);
211             ast->rate = get_le32(pb);
212             if (ast->scale && ast->rate)
213             {}
214             else if (frame_period)
215             {
216                 ast->rate = 1000000;
217                 ast->scale = frame_period;
218             }
219             else
220             {
221                 ast->rate = 25;
222                 ast->scale = 1;
223             }
```

设置当前流的时间信息，用于计算 pts 表示时间，进而同步。

```
224             av_set_pts_info(st, 64, ast->scale, ast->rate);
225
226             ast->cum_len = get_le32(pb); // start
227             nb_frames = get_le32(pb);
228
229             get_le32(pb); // buffer size
230             get_le32(pb); // quality
231             ast->sample_size = get_le32(pb); // sample ssize
232
233             switch (tag1)
234             {
235             case MKTAG('v', 'i', 'd', 's'): codec_type = CODEC_TYPE_VIDEO;
```

特别注意视频流的每一帧大小不同，所以 sample_size 设置为 0；对比音频流每一帧大小固定的情况。

```
236                 ast->sample_size = 0;
237                 break;
238             case MKTAG('a', 'u', 'd', 's'): codec_type = CODEC_TYPE_AUDIO;
239                 break;
```

```
240             case MKTAG('t', 'x', 't', 's'):  //FIXME
241                 codec_type = CODEC_TYPE_DATA; //CODEC_TYPE_SUB ?  FIXME
242                 break;
243             case MKTAG('p', 'a', 'd', 's'): codec_type = CODEC_TYPE_UNKNOWN;
```

如果是填充流，stream_index 减 1 就实现了简单的丢弃，不计入流数目总数。

```
244                 stream_index--;
245                 break;
246             default:
247                 goto fail;
248             }
249             ast->frame_offset = ast->cum_len * FFMAX(ast->sample_size, 1);
250             url_fskip(pb, size - 12 * 4);
251             break;
252         case MKTAG('s', 't', 'r', 'f'):  // stream header
```

从 strf 块读取流中编解码器的一些信息，跳过有些不用的字段，填写需要的字段。
注意有些编解码器需要的附加信息从此块中读出，保持至 extradata 并最终传给相应的编解码器。

```
253             if (stream_index >= s->nb_streams)
254             {
255                 url_fskip(pb, size);
256             }
257             else
258             {
259                 st = s->streams[stream_index];
260                 switch (codec_type)
261                 {
262                 case CODEC_TYPE_VIDEO:    // BITMAPINFOHEADER
263                     get_le32(pb); // size
264                     st->actx->width = get_le32(pb);
265                     st->actx->height = get_le32(pb);
266                     get_le16(pb); // panes
267                     st->actx->bits_per_sample = get_le16(pb); // depth
268                     tag1 = get_le32(pb);
269                     get_le32(pb); // ImageSize
270                     get_le32(pb); // XPelsPerMeter
271                     get_le32(pb); // YPelsPerMeter
272                     get_le32(pb); // ClrUsed
273                     get_le32(pb); // ClrImportant
274
275                     if (size > 10 *4 && size < (1 << 30))
```

```
276                     {
```

对视频，extradata 通常是保存的是 BITMAPINFO

```
277                         st->actx->extradata_size = size - 10 * 4;
278                         st->actx->extradata = av_malloc(st->actx->extradata_size +
279                                                         FF_INPUT_BUFFER_PADDING_SIZE);
280                         url_fread(pb, st->actx->extradata, st->actx->extradata_size);
281                     }
282
283                     if (st->actx->extradata_size &1)
284                         get_byte(pb);
285
286                     /* Extract palette from extradata if bpp <= 8 */
287                     /* This code assumes that extradata contains only palette */
288                     /* This is true for all paletted codecs implemented in ffmpeg */
289                     if (st->actx->extradata_size && (st->actx->bits_per_sample <= 8))
290                     {
291                         int min = FFMIN(st->actx->extradata_size, AVPALETTE_SIZE);
292
293                         st->actx->palctrl = av_mallocz(sizeof(AVPaletteControl));
294                         memcpy(st->actx->palctrl->palette, st->actx->extradata, min);
295                         st->actx->palctrl->palette_changed = 1;
296                     }
297
298                     st->actx->codec_type = CODEC_TYPE_VIDEO;
299                     st->actx->codec_id = codec_get_id(codec_bmp_tags, tag1);
300
301                     st->frame_last_delay = 1.0 * ast->scale / ast->rate;
302
303                     break;
304                 case CODEC_TYPE_AUDIO:
305                     {
306                         AVCodecContext *actx = st->actx;
307
308                         int id = get_le16(pb);
309                         actx->codec_type = CODEC_TYPE_AUDIO;
310                         actx->channels = get_le16(pb);
311                         actx->sample_rate = get_le32(pb);
312                         actx->bit_rate = get_le32(pb) *8;
313                         actx->block_align = get_le16(pb);
314                         if (size == 14)  // We're dealing with plain vanilla WAVEFORMAT
```

```
315                             actx->bits_per_sample = 8;
316                         else
317                             actx->bits_per_sample = get_le16(pb);
318                         actx->codec_id = codec_get_id(codec_wav_tags, id);
319
320                         if (size > 16)
321                         {
322                             actx->extradata_size = get_le16(pb);
323                             if (actx->extradata_size > 0)
324                             {
```

对音频，extradata 通常是保存的是 WAVEFORMATEX

```
325                                 if (actx->extradata_size > size - 18)
326                                     actx->extradata_size = size - 18;
327                                 actx->extradata = av_mallocz(actx->extradata_size +
328                                                              FF_INPUT_BUFFER_PADDING_SIZE);
329                                 url_fread(pb, actx->extradata, actx->extradata_size);
330                             }
331                             else
332                             {
333                                 actx->extradata_size = 0;
334                             }
335
336                             // It is possible for the chunk to contain garbage at the end
337                             if (size - actx->extradata_size - 18 > 0)
338                                 url_fskip(pb, size - actx->extradata_size - 18);
339                         }
340                     }
341
342                     if (size % 2) // 2-aligned (fix for Stargate SG-1 - 3x18 - Shades of
Grey.avi)
343                         url_fskip(pb, 1);
344
345                     break;
346                 default:
```

对其他流类型，ffplay 简单的设置为 data 流。常规的是音频流和视频流，其他的少见。

```
347                     st->actx->codec_type = CODEC_TYPE_DATA;
348                     st->actx->codec_id = CODEC_ID_NONE;
349                     url_fskip(pb, size);
350                     break;
```

```
351                 }
352             }
353             break;
354         default:  // skip tag
```

对其他不识别的块 chunk，跳过。

```
355             size += (size &1);
356             url_fskip(pb, size);
357             break;
358         }
359     }
360
361 end_of_header:
362     if (stream_index != s->nb_streams - 1)
363     {
364 fail:
```

校验流的数目，如果有误，释放相关资源，返回-1 错误。

```
365         for (i = 0; i < s->nb_streams; i++)
366         {
367             av_freep(&s->streams[i]->actx->extradata);
368             av_freep(&s->streams[i]);
369         }
370         return  - 1;
371     }
372
```

加载 AVI 文件索引。

```
373     avi_load_index(s);
374
```

判别是否是非交织 avi。

```
375     avi->non_interleaved |= guess_ni_flag(s);
376     if (avi->non_interleaved)
```

对那些非交织存储的媒体流，人工的补上索引，便于读取操作。

```
377         clean_index(s);
378
379     return 0;
380 }
381
```

avi 文件可以简单认为音视频媒体数据时间基相同，因此音视频数据需要同步读取，同步解码，播放才能同步。

交织存储的 avi 文件，临近存储的音视频帧解码时间表示时间相近，微小的解码时间表示时间差别可以用帧缓存队列抵消，所以可以简单的按照文件顺序读取媒体数据。

非交织存储的 avi 文件，视频和音频这两种媒体数据相隔甚远，小缓存简单的顺序读文件时，不能同时读到音频和视频数据，最后导致不同步，ffplay 采取按最近时间点来决定读音频还是视频数据。

```
382 int avi_read_packet(AVFormatContext *s, AVPacket *pkt)
383 {
384     AVIContext *avi = s->priv_data;
385     ByteIOContext *pb = &s->pb;
386     int n, d[8], size;
387     offset_t i, sync;
388
389     if (avi->non_interleaved)
390     {
```

如果是非交织 AVI，用最近时间点来决定读取视频还是音频数据。

```
391         int best_stream_index = 0;
392         AVStream *best_st = NULL;
393         AVIStream *best_ast;
394         int64_t best_ts = INT64_MAX;
395         int i;
396
397         for (i = 0; i < s->nb_streams; i++)
398         {
```

遍历所有媒体流，按照已经播放的流数据，计算下一个最近的时间点。

```
399             AVStream *st = s->streams[i];
400             AVIStream *ast = st->priv_data;
401             int64_t ts = ast->frame_offset;
402
```

把帧偏移换算成帧数。

```
403             if (ast->sample_size)
404                 ts /= ast->sample_size;
405
```

把帧数换算成 pts 表示时间。

```
406             ts = av_rescale(ts, AV_TIME_BASE *(int64_t)st->time_base.num, st->time_base.den);
407
```

取最小的时间点对应的时间，流指针，流索引作为要读取的最佳 (读取)流参数。

```
408             if (ts < best_ts) // 每次读取时间点(ast->frame_offset)最近的包
409             {
410                 best_ts = ts;
411                 best_st = st;
412                 best_stream_index = i;
413             }
414         }
```

保存最佳流对应的 AVIStream，便于 432 行赋值并传递参数 packet_size 和 remaining。

```
415         best_ast = best_st->priv_data;
```

换算最小的时间点，查找索引表取出对应的索引。在缓存足够大，一次性完整读取帧数据时，此时 best_ast->remaining 参数为 0。

```
416         best_ts = av_rescale(best_ts,best_st->time_base.den,AV_TIME_BASE *(int64_t)best_st->time_base.num);
417         if (best_ast->remaining)
418             i = av_index_search_timestamp(best_st, best_ts, AVSEEK_FLAG_ANY | AVSEEK_FLAG_BACKWARD);
419         else
420             i = av_index_search_timestamp(best_st, best_ts, AVSEEK_FLAG_ANY);
421
422         if (i >= 0)
423         {
```

找到最佳索引，取出其他参数，在 426 行 seek 到相应位置，在 430 行保存最佳流索引，在 432 行保存并传递要读取的数据大小(通过最佳流索引找到最佳流，再找到对应 AVIStream 结构，再找到数据大小)。

```
424             int64_t pos = best_st->index_entries[i].pos;
425             pos += best_ast->packet_size - best_ast->remaining;
426             url_fseek(&s->pb, pos + 8, SEEK_SET);
427
428             assert(best_ast->remaining <= best_ast->packet_size);
429
430             avi->stream_index_2 = best_stream_index;
431             if (!best_ast->remaining)
432                 best_ast->packet_size = best_ast->remaining = best_st->index_entries[i].size;
433         }
434     }
435
436 resync:
437
438     if (avi->stream_index_2 >= 0)
```

```
439     {
```

如果找到最佳流索引，以此为根参数，取出其他参数和读取媒体数据。

```
440         AVStream *st = s->streams[avi->stream_index_2];
441         AVIStream *ast = st->priv_data;
442         int size;
443
444         if (ast->sample_size <= 1) // minorityreport.AVI block_align=1024 sample_size=1  IMA-
ADPCM
445             size = INT_MAX;
446         else if (ast->sample_size < 32)
447             size = 64 * ast->sample_size;
448         else
449             size = ast->sample_size;
450
```

在缓存足够大，一次全部读取一帧媒体数据的情况下，451 行判断不成立，size 等于 ast->sample_size

```
451         if (size > ast->remaining)
452             size = ast->remaining;
453
```

调用 av_get_packet()函数实际读取媒体数据到 pkt 包中。

```
454         av_get_packet(pb, pkt, size);
455
```

修改媒体流的一些其他参数。

```
456         pkt->dts = ast->frame_offset;
457
458         if (ast->sample_size)
459             pkt->dts /= ast->sample_size;
460
461         pkt->stream_index = avi->stream_index_2;
462
```

在简单情况顺序播放时，463 行到 487 行没有什么实际意义。

```
463         if (st->actx->codec_type == CODEC_TYPE_VIDEO)
464         {
465             if (st->index_entries)
466             {
467                 AVIndexEntry *e;
468                 int index;
```

```
469
470                 index = av_index_search_timestamp(st, pkt->dts, 0);
471                 e = &st->index_entries[index];
472
473                 if (index >= 0 && e->timestamp == ast->frame_offset)
474                 {
475                     if (e->flags &AVINDEX_KEYFRAME)
476                         pkt->flags |= PKT_FLAG_KEY;
477                 }
478             }
479             else
480             {
```

如果没有索引，较好的办法是把所有帧都设为关键帧。

```
481                 pkt->flags |= PKT_FLAG_KEY;
482             }
483         }
484         else
485         {
486             pkt->flags |= PKT_FLAG_KEY;
487         }
488
```

修改帧偏移。

```
489         if (ast->sample_size)
490             ast->frame_offset += pkt->size;
491         else
492             ast->frame_offset++;
493
494         ast->remaining -= size;
495         if (!ast->remaining)
496         {
```

缓存足够大时，程序一定跑到这里，复位标志性参数。

```
497             avi->stream_index_2 =  - 1;
498             ast->packet_size = 0;
499             if (size &1)
500             {
501                 get_byte(pb);
502                 size++;
503             }
```

```
504         }
505
```

返回实际读到的数据大小。

```
506         return size;
507     }
508
509     memset(d,  - 1, sizeof(int) *8);
```

把数组 d[8]清为-1，为了在下面的流标记查找时不会出错。

```
510     for (i = sync = url_ftell(pb); !url_feof(pb); i++)
511     {
```

交织 avi 时顺序读取文件，媒体数据。

```
512         int j;
513
514         if (i >= avi->movi_end)
515             break;
516
```

首先要找到流标记，比如 00db,00dc,01wb 等。在 32bit CPU 上为存取数据方便，把 avi 文件中的帧标记和帧大小共 8 个字节对应赋值到 int 型数组 d[8]中，这样每次是整数操作。

```
517         for (j = 0; j < 7; j++)
518             d[j] = d[j + 1];
519
```

518 行把整型缓存前移一个单位。520 行从文件中读一个字节补充到整型缓存，计算包大小和流索引。

```
520         d[7] = get_byte(pb);
521
522         size = d[4] + (d[5] << 8) + (d[6] << 16) + (d[7] << 24);
523
524         if (d[2] >= '0' && d[2] <= '9' && d[3] >= '0' && d[3] <= '9')
525         {
526             n = (d[2] - '0') *10+(d[3] - '0');
527         }
528         else
529         {
530             n = 100; //invalid stream id
531         }
532
```

校验 size 大小，如果偏移位置加 size 超过数据块大小就不是有效的流标记。
校验流索引，如果<0 就不是有效的流标记。流索引从 0 开始计数，媒体文件通常不超过 10 个流。

```
533         if (i + size > avi->movi_end || d[0] < 0)
534             continue;
535
```

536 行到 541 行代码处理诸如 junk 等需要跳过的块。

```
536         if ((d[0] == 'i' && d[1] == 'x' && n < s->nb_streams)
537          || (d[0] == 'J' && d[1] == 'U' && d[2] == 'N' && d[3] == 'K'))
538         {
539             url_fskip(pb, size);
540             goto resync;
541         }
542
```

计算流索引号 n。

```
543         if (d[0] >= '0' && d[0] <= '9' && d[1] >= '0' && d[1] <= '9')
544         {
545             n = (d[0] - '0') *10+(d[1] - '0');
546         }
547         else
548         {
549             n = 100; //invalid stream id
550         }
551
552         //parse ##dc/##wb
553         if (n < s->nb_streams)
554         {
```

如果流索引号 n 比流总数小，认为有效。(我个人认为这个校验不太严格。)

```
555             AVStream *st;
556             AVIStream *ast;
557             st = s->streams[n];
558             ast = st->priv_data;
559
560             if (((ast->prefix_count < 5 || sync + 9 > i) && d[2] < 128 && d[3] < 128)
561                     || d[2] * 256 + d[3] == ast->prefix)
562             {
```

if(d[2]*256+d[3]==ast->prefix)为真表示 "db","dc","wb"等字串匹配，找到正确帧标记。
判断 d[2]<128 && d[3]<128 是因为 'd','b','c','w'等字符的 ascii 码小于 128。
判断 ast->prefix_count<5 || sync + 9 > i，是判断单一媒体的 5 帧内或找帧标记超过 9 个字节。
563 行到 569 行是单一媒体帧边界初次识别成功和以后识别成功的简单处理，计数自增或保存标记。

```
563                 if (d[2] * 256 + d[3] == ast->prefix)
564                     ast->prefix_count++;
565                 else
566                 {
567                     ast->prefix = d[2] *256+d[3];
568                     ast->prefix_count = 0;
569                 }
570
```

找到相应的流索引后，保存相关参数，跳转到实质性读媒体程序。

```
571                 avi->stream_index_2 = n;
572                 ast->packet_size = size + 8;
573                 ast->remaining = size;
574                 goto resync;
575             }
576         }
577         // palette changed chunk
578         if (d[0] >= '0' && d[0] <= '9' && d[1] >= '0' && d[1] <= '9'
579         && (d[2] == 'p' && d[3] == 'c') && n < s->nb_streams && i + size <= avi->movi_end)
580         {
```

处理调色板改变块数据，读取调色板数据到编解码器上下文的调色板数组中。

```
581             AVStream *st;
582             int first, clr, flags, k, p;
583
584             st = s->streams[n];
585
586             first = get_byte(pb);
587             clr = get_byte(pb);
588             if (!clr) // all 256 colors used
589                 clr = 256;
590             flags = get_le16(pb);
591             p = 4;
592             for (k = first; k < clr + first; k++)
593             {
594                 int r, g, b;
```

```
595                 r = get_byte(pb);
596                 g = get_byte(pb);
597                 b = get_byte(pb);
598                 get_byte(pb);
599                 st->actx->palctrl->palette[k] = b + (g << 8) + (r << 16);
600             }
601             st->actx->palctrl->palette_changed = 1;
602             goto resync;
603         }
604     }
605
606     return  - 1;
607 }
608
```

实质读取 AVI 文件的索引。

```
609 static int avi_read_idx1(AVFormatContext *s, int size)
610 {
611     AVIContext *avi = s->priv_data;
612     ByteIOContext *pb = &s->pb;
613     int nb_index_entries, i;
614     AVStream *st;
615     AVIStream *ast;
616     unsigned int index, tag, flags, pos, len;
617     unsigned last_pos =  - 1;
618
619     nb_index_entries = size / 16;
```

如果没有索引块 chunk，直接返回。

```
620     if (nb_index_entries <= 0)
621         return  - 1;
622
```

遍历整个索引项。

```
623     for (i = 0; i < nb_index_entries; i++)
624     {
625         tag = get_le32(pb);
626         flags = get_le32(pb);
627         pos = get_le32(pb);
628         len = get_le32(pb);
629
```

如果第一个索引指示的偏移量大于数据块的偏移量，则索引指示的偏移量是相对文件开始字节的偏移量。索引加载到内存后，如果是相对数据块的偏移量就要换算成相对于文件开始字节的偏移量，便于 seek 操作。在 631 行和 633 行统一处理这两个情况。

```
630         if (i == 0 && pos > avi->movi_list)
631             avi->movi_list = 0;
632
633         pos += avi->movi_list;
634
```

计算流 ID，如索引块中的 00dc，01wb 等关键字表示的流 ID 分别为数字 0 和 1。

```
635         index = ((tag &0xff) - '0') *10;
636         index += ((tag >> 8) &0xff) - '0';
637         if (index >= s->nb_streams)
638             continue;
639
640         st = s->streams[index];
641         ast = st->priv_data;
642
643         if (last_pos == pos)
644             avi->non_interleaved = 1;
645         else
646             av_add_index_entry(st, pos, ast->cum_len, len, 0, (flags&AVIIF_INDEX)?AVINDEX_KEYFRAME:0);
647
648         if (ast->sample_size)
649             ast->cum_len += len / ast->sample_size;
650         else
651             ast->cum_len++;
652         last_pos = pos;
653     }
654     return 0;
655 }
656
```

判断是否是非交织存放媒体数据，其中 ni 是 non_interleaved 的缩写，非交织的意思。如果是非交织存放返回 1，交织存放返回 0。

非交织存放的 avi 文件，如果有多个媒体流，肯定有某个流的开始字节文件偏移量大于其他某个流的末尾字节的文件偏移量。程序利用这个来判断是否是非交织存放，否则认定为交织存放。

```
657 static int guess_ni_flag(AVFormatContext *s)
658 {
659     int i;
```

```
660     int64_t last_start = 0;
661     int64_t first_end = INT64_MAX;
662
```

遍历 AVI 文件中所有的索引，取流开始偏移量的最大值和末尾偏移量的最小值判断。

```
663     for (i = 0; i < s->nb_streams; i++)
664     {
665         AVStream *st = s->streams[i];
666         int n = st->nb_index_entries;
667
```

如果某个流没有 index 项，认为这个流没有数据，这个流忽略不计。

```
668         if (n <= 0)
669             continue;
670
```

遍历 AVI 文件中所有的索引，取流开始偏移量的最大值。

```
671         if (st->index_entries[0].pos > last_start)
672             last_start = st->index_entries[0].pos;
673
```

遍历 AVI 文件中所有的索引，取流末尾偏移量的最小值。

```
674         if (st->index_entries[n - 1].pos < first_end)
675             first_end = st->index_entries[n - 1].pos;
676     }
```

如果某个流的开始最大值大于某个流的末尾最小值，认为是非交织存储，否则是交织存储。

```
677     return last_start > first_end;
678 }
679
```

加载 AVI 文件索引块 chunk，特别注意在 avi_read_idx1()函数调用的 av_add_index_entry()函数是分媒体类型按照时间顺序重新排序的。

```
680 static int avi_load_index(AVFormatContext *s)
681 {
682     AVIContext *avi = s->priv_data;
683     ByteIOContext *pb = &s->pb;
684     uint32_t tag, size;
685     offset_t pos = url_ftell(pb);
686
687     url_fseek(pb, avi->movi_end, SEEK_SET);
```

```
688
689     for (;;)
690     {
691         if (url_feof(pb))
692             break;
693         tag = get_le32(pb);
694         size = get_le32(pb);
695
696         switch (tag)
697         {
698         case MKTAG('i', 'd', 'x', '1'):
699             if (avi_read_idx1(s, size) < 0)
700                 goto skip;
701             else
702                 goto the_end;
703             break;
704         default:
705 skip:
706             size += (size &1);
707             url_fskip(pb, size);
708             break;
709         }
710     }
711 the_end:
712     url_fseek(pb, pos, SEEK_SET);
713     return 0;
714 }
715
```

关闭 AVI 文件，释放内存和其他相关资源。

```
716 static int avi_read_close(AVFormatContext *s)
717 {
718     int i;
719     AVIContext *avi = s->priv_data;
720
721     for (i = 0; i < s->nb_streams; i++)
722     {
723         AVStream *st = s->streams[i];
724         AVIStream *ast = st->priv_data;
725         av_free(ast);
726         av_free(st->actx->extradata);
```

```
727         av_free(st->actx->palctrl);
728     }
729
730     return 0;
731 }
732
```

AVI 文件判断，取 AVI 文件的关键字串"RIFF"和"AVI "判断，和 get_riff()函数部分相同。

```
733 static int avi_probe(AVProbeData *p)
734 {
735     if (p->buf_size <= 32) // check file header
736         return 0;
737     if (p->buf[0] == 'R' && p->buf[1] == 'I' && p->buf[2] == 'F' && p->buf[3] == 'F'
738      && p->buf[8] == 'A' && p->buf[9] == 'V' && p->buf[10] == 'I'&& p->buf[11] == ' ')
739         return AVPROBE_SCORE_MAX;
740     else
741         return 0;
742 }
743
```

初始化 AVI 文件格式 AVInputFormat 结构，直接的赋值操作。

```
744 AVInputFormat avi_iformat =
745 {
746     "avi",
747     sizeof(AVIContext),
748     avi_probe,
749     avi_read_header,
750     avi_read_packet,
751     avi_read_close,
752 };
753
```

注册 avi 文件格式，ffplay 把所有支持的文件格式用链表串联起来，表头是 first_iformat，便于查找。

```
754 int avidec_init(void)
755 {
756     av_register_input_format(&avi_iformat);
757     return 0;
758 }
759
```

第四章 libavcodec 剖析

4.1 文件列表

文件类型	文件名	大小(bytes)
h	avcodec.h	4943
c	allcodecs.c	310
h	dsputil.h	163
c	dsputil.c	350
h	imgconvert_template.h	22311
c	imgconvert.c	47834
c	msrle.c	8387
h	turespeech_data.h	4584
c	turespeech.c	9622
c	utils_codec.c	8973

4.2 avcodec.h 文件

4.2.1 功能描述

定义编解码器库使用的宏、数据结构和函数，通常这些宏、数据结构和函数在此模块内相对全局有效。

4.2.2 文件注释

```
1   #ifndef AVCODEC_H
2   #define AVCODEC_H
3
4   #ifdef __cplusplus
5   extern "C"
6   {
7   #endif
8
9   #include "../libavutil/avutil.h"
10  #include <sys/types.h> // size_t
11
```

和版本信息有关的几个宏定义

```
12  #define FFMPEG_VERSION_INT      0x000409
13  #define FFMPEG_VERSION          "CVS"
14
15  #define AV_STRINGIFY(s)         AV_TOSTRING(s)
16  #define AV_TOSTRING(s) #s
17
```

```
18  #define LIBAVCODEC_VERSION_INT  ((51<<16)+(8<<8)+0)
19  #define LIBAVCODEC_VERSION      51.8.0
20  #define LIBAVCODEC_BUILD        LIBAVCODEC_VERSION_INT
21
22  #define LIBAVCODEC_IDENT        "Lavc" AV_STRINGIFY(LIBAVCODEC_VERSION)
23
24  #define AV_NOPTS_VALUE          int64_t_C(0x8000000000000000)
25  #define AV_TIME_BASE            1000000
26
```

Codec ID 宏定义，瘦身后的 ffplay 只支持这两种 codec，其他的都删掉了。

```
27  enum CodecID
28  {
29      CODEC_ID_TRUESPEECH,
30      CODEC_ID_MSRLE,
31      CODEC_ID_NONE
32  };
33
```

Codec 类型定义，瘦身后的 ffplay 只支持视频和音频。

```
34  enum CodecType
35  {
36      CODEC_TYPE_UNKNOWN =  - 1,
37      CODEC_TYPE_VIDEO,
38      CODEC_TYPE_AUDIO,
39      CODEC_TYPE_DATA
40  };
41
42  #define AVCODEC_MAX_AUDIO_FRAME_SIZE 192000 // 1 second of 48khz 32bit audio
43
44  #define FF_INPUT_BUFFER_PADDING_SIZE 8
45
```

AVPicture 和 AVFrame 主要表示解码过程中的使用缓存，通常帧缓存是 YUV 格式，输出格式有 YUV 也有 RGB 格式，所以定义了 4 个 data 指针来表示分量。

```
46  typedef struct AVPicture
47  {
48      uint8_t *data[4];
49      int linesize[4];
50  } AVPicture;
51
```

```
52  typedef struct AVFrame
53  {
54      uint8_t *data[4]; // 有多重意义，其一用 NULL 来判断是否被占用
55      int linesize[4];
56      uint8_t *base[4]; // 有多重意义，其一用 NULL 来判断是否分配内存
57  } AVFrame;
58
```

程序运行时当前 Codec 使用的上下文，着重于所有 Codec 共有的属性(并且是在程序运行时才能确定其值)，codec 和 priv_data 关联其他结构的字段，便于在数据结构间跳转。

```
59  typedef struct AVCodecContext
60  {
61      int bit_rate;
62      int frame_number;          // audio or video frame number
63
64      unsigned char *extradata; // codec 的私有数据,对 Audio 是 WAVEFORMATEX 扩展结构。
65      int extradata_size;        //                    对 Video 是 BITMAPINFOHEADER 扩展结构
66
67      int width, height;         // video only
68
69      enum PixelFormat pix_fmt; // 输出像素格式/视频图像格式
70
71      int sample_rate;           // samples per sec  // audio only
72      int channels;
73      int bits_per_sample;
74      int block_align;
75
76      struct AVCodec *codec;     // 指向 Codec 的指针，
77      void *priv_data;           // 具体解码器属性，在本例中指向 MsrleContext 或 TSContext
78
79      enum CodecType codec_type;// see CODEC_TYPE_xxx
80      enum CodecID codec_id;     // see CODEC_ID_xxx
81
82      int(*get_buffer)(struct AVCodecContext *c, AVFrame *pic);
83      void(*release_buffer)(struct AVCodecContext *c, AVFrame *pic);
84      int(*reget_buffer)(struct AVCodecContext *c, AVFrame *pic);
85
86      int internal_buffer_count;
87      void *internal_buffer;
88
89      struct AVPaletteControl *palctrl;
```

```
90  }AVCodecContext;
91
```

类似 COM 接口的数据结构，表示音视频编解码器，着重于功能函数，一种媒体类型对应一个 AVCodec 结构，在程序运行时有多个实例串联成链表便于查找。

```
92  typedef struct AVCodec
93  {
94      const char *name;      // 便于阅读的友好字符串，表征编解码器名称，比如"msrle","truespeech"
95      enum CodecType type;   // 编解码器类型，有效取值为 CODEC_TYPE_VIDEO 或 CODEC_TYPE_AUDIO
96      enum CodecID id;       // 编解码器 ID 值，
97      int priv_data_size;    // 具体编解码属性结构的大小，取代很多的 if-else 语句
98      int(*init)(AVCodecContext*);
99      int(*encode)(AVCodecContext *, uint8_t *buf, int buf_size, void *data);
100     int(*close)(AVCodecContext*);
101     int(*decode)(AVCodecContext *, void *outdata, int *outdata_size, uint8_t *buf, int buf_size);
102     int capabilities;
103
104     struct AVCodec *next; // 把所有的编解码器串联成链表便于查找
105 }AVCodec;
106
```

调色板大小和大小宏定义，每个调色板四字节(R,G,B,α)。有很多的视频图像颜色种类比较少，用索引间接表示每个像素的颜色值，就可以用调色板和索引值实现简单的大约的 4:1 压缩比。

```
107 #define AVPALETTE_SIZE 1024
108 #define AVPALETTE_COUNT 256
109
```

调色板数据结构定义，保存调色板数据。

```
110 typedef struct AVPaletteControl
111 {
112     // demuxer sets this to 1 to indicate the palette has changed; decoder resets to 0
113     int palette_changed;
114
115     /* 4-byte ARGB palette entries, stored in native byte order; note that
116      * the individual palette components should be on a 8-bit scale; if
117      * the palette data comes from a IBM VGA native format, the component
118      * data is probably 6 bits in size and needs to be scaled */
119     unsigned int palette[AVPALETTE_COUNT];
120
121 } AVPaletteControl;
122
```

编解码库使用的函数声明。

```
123 int avpicture_alloc(AVPicture *picture, int pix_fmt, int width, int height);
124
125 void avpicture_free(AVPicture *picture);
126
127 int avpicture_fill(AVPicture *picture, uint8_t *ptr, int pix_fmt, int width, int height);
128 int avpicture_get_size(int pix_fmt, int width, int height);
129 void avcodec_get_chroma_sub_sample(int pix_fmt, int *h_shift, int *v_shift);
130
131 int img_convert(AVPicture *dst, int dst_pix_fmt, const AVPicture *src, int pix_fmt,
132                 int width, int height);
133
134 void avcodec_init(void);
135
136 void register_avcodec(AVCodec *format);
137 AVCodec *avcodec_find_decoder(enum CodecID id);
138
139 AVCodecContext *avcodec_alloc_context(void);
140
141 int avcodec_default_get_buffer(AVCodecContext *s, AVFrame *pic);
142 void avcodec_default_release_buffer(AVCodecContext *s, AVFrame *pic);
143 int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *pic);
144 void avcodec_align_dimensions(AVCodecContext *s, int *width, int *height);
145 int avcodec_check_dimensions(void *av_log_ctx, unsigned int w, unsigned int h);
146
147 int avcodec_open(AVCodecContext *avctx, AVCodec *codec);
148
149 int avcodec_decode_audio(AVCodecContext *avctx, int16_t *samples, int *frame_size_ptr,
150                          uint8_t *buf, int buf_size);
151 int avcodec_decode_video(AVCodecContext *avctx, AVFrame *picture, int *got_picture_ptr,
152                          uint8_t *buf, int buf_size);
153
154 int avcodec_close(AVCodecContext *avctx);
155
156 void avcodec_register_all(void);
157
158 void avcodec_default_free_buffers(AVCodecContext *s);
159
160 void *av_malloc(unsigned int size);
161 void *av_mallocz(unsigned int size);
```

```
162 void *av_realloc(void *ptr, unsigned int size);
163 void av_free(void *ptr);
164 void av_freep(void *ptr);
165 void *av_fast_realloc(void *ptr, unsigned int *size, unsigned int min_size);
166
167 void img_copy(AVPicture *dst, const AVPicture *src, int pix_fmt, int width, int height);
168
169 #ifdef __cplusplus
170 }
171
172 #endif
173
174 #endif
```

4.3 allcodec.c 文件

4.3.1 功能描述

简单的注册/初始化函数，把编解码器用相应的链表串起来便于查找识别。

4.3.2 文件注释

```
1   #include "avcodec.h"
2
3   extern AVCodec truespeech_decoder;
4   extern AVCodec msrle_decoder;
5
6   void avcodec_register_all(void)
7   {
```

8 到 13 行，inited 变量声明成 static，做一下比较是为了避免此函数多次调用。
编程基本原则之一，初始化函数只调用一次，不能随意多次调用。

```
8       static int inited = 0;
9
10      if (inited != 0)
11          return ;
12
13      inited = 1;
14
```

把 msrle_decoder 解码器串接到解码器链表，链表头指针是 first_avcodec。

```
15      register_avcodec(&msrle_decoder);
16
```

把 truespeech_decoder 解码器串接到解码器链表，链表头指针是 first_avcodec。

```
17      register_avcodec(&truespeech_decoder);
18  }
```

4.4 dsputil.h 文件

4.4.1 功能描述

定义 dsp 优化限幅运算使用的查找表及其初始化函数。

4.4.2 文件注释

```
1   #ifndef DSPUTIL_H
2   #define DSPUTIL_H
3
4   #define MAX_NEG_CROP 1024
5
6   extern uint8_t cropTbl[256+2 * MAX_NEG_CROP];
7
8   void dsputil_static_init(void);
9
10  #endif
```

4.5 dsputil.c 文件

4.5.1 功能描述

定义 dsp 优化限幅运算使用的查找表，实现其初始化函数。

4.5.2 文件注释

```
1   #include "avcodec.h"
2   #include "dsputil.h"
3
4   uint8_t cropTbl[256+2 * MAX_NEG_CROP] = {0, };
5
6   void dsputil_static_init(void)
7   {
8       int i;
9
```

初始化限幅运算查找表，最后的结果是：前 MAX_NEG_CROP 个数组项为 0，接着的 256 个数组项分别为 0 到 255，后面 MAX_NEG_CROP 个数组项为 255。用查表代替比较实现限幅运算。

```
10      for (i = 0; i < 256; i++)
11          cropTbl[i + MAX_NEG_CROP] = i;
12
13      for (i = 0; i < MAX_NEG_CROP; i++)
14      {
15          cropTbl[i] = 0;
16          cropTbl[i + MAX_NEG_CROP + 256] = 255;
17      }
18  }
```

4.6 utils_codec.c 文件

4.6.1 功能描述

编解码库使用的帮助和工具函数，

4.6.2 文件注释

```
1   #include "avcodec.h"
2   #include "dsputil.h"
3
1   #include <assert.h>
2   #include "avcodec.h"
3   #include "dsputil.h"
4
5   #define EDGE_WIDTH   16
6   #define STRIDE_ALIGN 16
7
8   #define INT_MAX 2147483647
9
10  #define FFMAX(a,b) ((a) > (b) ? (a) : (b))
11
```

内存动态分配函数，做一下简单参数校验后调用系统函数

```
12  void *av_malloc(unsigned int size)
13  {
14      void *ptr;
15
16      if (size > INT_MAX)
17          return NULL;
18      ptr = malloc(size);
19
20      return ptr;
21  }
22
```

内存动态重分配函数，做一下简单参数校验后调用系统函数

```
23  void *av_realloc(void *ptr, unsigned int size)
24  {
25      if (size > INT_MAX)
26          return NULL;
27
28      return realloc(ptr, size);
```

```
29  }
30
```

内存动态释放函数，做一下简单参数校验后调用系统函数

```
31  void av_free(void *ptr)
32  {
33      if (ptr)
34          free(ptr);
35  }
36
```

内存动态分配函数，复用 av_malloc()函数，再把分配的内存清 0.

```
37  void *av_mallocz(unsigned int size)
38  {
39      void *ptr;
40
41      ptr = av_malloc(size);
42      if (!ptr)
43          return NULL;
44
45      memset(ptr, 0, size);
46      return ptr;
47  }
48
```

快速内存动态分配函数，预分配一些内存来避免多次调用系统函数达到快速的目的。

```
49  void *av_fast_realloc(void *ptr, unsigned int *size, unsigned int min_size)
50  {
51      if (min_size <  *size)
52          return ptr;
53
54      *size  = FFMAX(17 *min_size  / 16+32, min_size);
55
56      return av_realloc(ptr,  *size);
57  }
58
```

动态内存释放函数，注意传入的变量的类型。

```
59  void av_freep(void *arg)
60  {
61      void **ptr = (void **)arg;
```

```
62      av_free(*ptr);
63      *ptr = NULL;
64  }
65
66  AVCodec *first_avcodec = NULL;
67
```

把编解码器串联成一个链表，便于查找。

```
68  void register_avcodec(AVCodec *format)
69  {
70      AVCodec **p;
71      p = &first_avcodec;
72      while (*p != NULL)
73          p = &(*p)->next;
74      *p = format;
75      format->next = NULL;
76  }
77
```

编解码库内部使用的缓存区，因为视频图像有 RGB 或 YUV 分量格式，所以每个数组有四个分量。

```
78  typedef struct InternalBuffer
79  {
80      uint8_t *base[4];
81      uint8_t *data[4];
82      int  linesize[4];
83  } InternalBuffer;
84
85  #define INTERNAL_BUFFER_SIZE 32
86
87  #define ALIGN(x, a) (((x)+(a)-1)&~((a)-1))
88
```

计算各种图像格式要求的图像长宽的字节对齐数，是 1 个还是 2 个，4 个，8 个，16 个字节对齐。

```
89  void avcodec_align_dimensions(AVCodecContext *s, int *width, int *height)
90  {
```

默认长宽是 1 个字节对齐。

```
91      int w_align = 1;
92      int h_align = 1;
93
94      switch (s->pix_fmt)
```

```
95      {
96      case PIX_FMT_YUV420P:
97      case PIX_FMT_YUV422:
98      case PIX_FMT_UYVY422:
99      case PIX_FMT_YUV422P:
100     case PIX_FMT_YUV444P:
101     case PIX_FMT_GRAY8:
102     case PIX_FMT_YUVJ420P:
103     case PIX_FMT_YUVJ422P:
104     case PIX_FMT_YUVJ444P: //FIXME check for non mpeg style codecs and use less alignment
105         w_align = 16;
106         h_align = 16;
107         break;
108     case PIX_FMT_YUV411P:
109     case PIX_FMT_UYVY411:
110         w_align = 32;
111         h_align = 8;
112         break;
113     case PIX_FMT_YUV410P:
114     case PIX_FMT_RGB555:
115     case PIX_FMT_PAL8:
116         break;
117     case PIX_FMT_BGR24:
118         break;
119     default:
120         w_align = 1;
121         h_align = 1;
122         break;
123     }
124
125     *width = ALIGN(*width, w_align);
126     *height = ALIGN(*height, h_align);
127 }
128
```

校验视频图像的长宽是否合法。

```
129 int avcodec_check_dimensions(void *av_log_ctx, unsigned int w, unsigned int h)
130 {
131     if ((int)w > 0 && (int)h > 0 && (w + 128)*(uint64_t)(h + 128) < INT_MAX / 4)
132         return 0;
133
```

```
134     return  - 1;
135 }
136
```

每次取 internal_buffer_count 数据项，用 base[0]来判断是否已分配内存，用 data[0]来判断是否已被占用。base[]和 data[]有多重意义。
在 avcodec_alloc_context 中已把 internal_buffer 各项清 0，所以可以用 base[0]来判断。

```
137 int avcodec_default_get_buffer(AVCodecContext *s, AVFrame *pic)
138 {
139     int i;
140     int w = s->width;
141     int h = s->height;
142     int align_off;
143     InternalBuffer *buf;
144
145     assert(pic->data[0] == NULL);
146     assert(INTERNAL_BUFFER_SIZE > s->internal_buffer_count);
147
```

校验视频图像的长宽是否合法。

```
148     if (avcodec_check_dimensions(s, w, h))
149         return  - 1;
150
```

如果没有分配内存，就分配动态内存并清 0。

```
151     if (s->internal_buffer == NULL)
152         s->internal_buffer = av_mallocz(INTERNAL_BUFFER_SIZE *sizeof(InternalBuffer));
153
```

取缓存中的第一个没有占用内存。

```
154     buf = &((InternalBuffer*)s->internal_buffer)[s->internal_buffer_count];
155
156     if (buf->base[0])
157     { /* 如果内存已分配就跳过 */ }
158     else
159     {
```

如果没有分配内存就按照图像格式要求分配内存，并设置一些标记和计算一些参数值。

```
160         int h_chroma_shift, v_chroma_shift;
161         int pixel_size, size[3];
162
```

```
163         AVPicture picture;
164
```

计算 CbCr 色度分量长宽的与 Y 亮度分量长宽的比，最后用移位实现。

```
165         avcodec_get_chroma_sub_sample(s->pix_fmt, &h_chroma_shift, &v_chroma_shift);
166
```

规整长宽满足特定图像像素格式的要求。

```
167         avcodec_align_dimensions(s, &w, &h);
168
```

把长宽放大一些，比如在 mpeg4 视频中编码算法中的运动估计要把原始图像做扩展来满足不受限制运动矢量的要求(运动矢量可以超出原始图像边界)。

```
169         w+= EDGE_WIDTH*2;
170         h+= EDGE_WIDTH*2;
171
```

计算特定格式的图像参数，包括各分量的大小，单行长度(linesize/stride)等等。

```
172         avpicture_fill(&picture, NULL, s->pix_fmt, w, h);
173         pixel_size = picture.linesize[0] * 8 / w;
174         assert(pixel_size >= 1);
175
176         if (pixel_size == 3*8)
177             w = ALIGN(w, STRIDE_ALIGN << h_chroma_shift);
178         else
179             w = ALIGN(pixel_size*w, STRIDE_ALIGN << (h_chroma_shift + 3)) / pixel_size;
180
181         size[1] = avpicture_fill(&picture, NULL, s->pix_fmt, w, h);
182         size[0] = picture.linesize[0] *h;
183         size[1] -= size[0];
184         if (picture.data[2])
185             size[1] = size[2] = size[1] / 2;
186         else
187             size[2] = 0;
188
```

注意 base[]和 data[]数组还有作为标记的用途，free()时的非 NULL 判断，这里要清 0。

```
189         memset(buf->base, 0, sizeof(buf->base));
190         memset(buf->data, 0, sizeof(buf->data));
191
192         for (i = 0; i < 3 && size[i]; i++)
```

```
193         {
194             const int h_shift = i == 0 ? 0 : h_chroma_shift;
195             const int v_shift = i == 0 ? 0 : v_chroma_shift;
196
197             buf->linesize[i] = picture.linesize[i];
198
```

实质性分配内存，并且在 202 行把内存清 0。

```
199             buf->base[i] = av_malloc(size[i] + 16); //FIXME 16
200             if (buf->base[i] == NULL)
201                 return -1;
202             memset(buf->base[i], 128, size[i]);
203
```

内存对齐计算。

```
204             align_off=ALIGN((buf->linesize[i]*EDGE_WIDTH>>v_shift)+(EDGE_WIDTH>>h_shift),STRIDE_ALIGN);
205
206             if ((s->pix_fmt == PIX_FMT_PAL8) || !size[2])
207                 buf->data[i] = buf->base[i];
208             else
209                 buf->data[i] = buf->base[i] + align_off;
210         }
211     }
212
213     for (i = 0; i < 4; i++)
214     {
```

把分配的内存参数赋值到 pic 指向的结构中，传递出去。

```
215         pic->base[i] = buf->base[i];
216         pic->data[i] = buf->data[i];
217         pic->linesize[i] = buf->linesize[i];
218     }
```

内存数组计数+1，注意释放时的操作，保证计数对应的内存数组是空闲的。

```
219     s->internal_buffer_count++;
220
221     return 0;
222 }
223
```

释放占用的内存数组项。保证从 0 到 internal_buffer_count-1 数据项为有效数据，其他是空闲数据项

```
224 void avcodec_default_release_buffer(AVCodecContext *s, AVFrame *pic)
225 {
226     int i;
227     InternalBuffer *buf,  *last, temp;
228
```

简单的参数校验，内存必须是已经分配过。

```
229     assert(s->internal_buffer_count);
230
231     buf = NULL;
232     for (i = 0; i < s->internal_buffer_count; i++)
233     {
```

遍历内存数组，查找对应 pic 的内存数组项，以 data[0]内存地址为比较判别标记。

```
234         buf = &((InternalBuffer*)s->internal_buffer)[i]; //just 3-5 checks so is not worth to optimize
235         if (buf->data[0] == pic->data[0])
236             break;
237     }
238     assert(i < s->internal_buffer_count);
```

内存数组计数-1, 删除最后一项.

```
239     s->internal_buffer_count--;
240     last = &((InternalBuffer*)s->internal_buffer)[s->internal_buffer_count];
241
```

把将要空闲的数组项和数组最后一项交换，保证 internal_buffer_count 计算正确无误。注意这里并没有内存释放的动作，便于下次复用已分配的内存。

```
242     temp =  *buf;
243     *buf =  *last;
244     *last = temp;
245
246     for (i = 0; i < 3; i++)
247     {
```

把 data[i]置空，指示本块内存没有被占用，实际分配的首地址保持在 base[]中。
整个程序最多分配 INTERNAL_BUFFER_SIZE 次 avframe，其他次循环使用。

```
248         pic->data[i] = NULL;
249     }
250 }
251
```

重新获得缓存。

```
252 int avcodec_default_reget_buffer(AVCodecContext *s, AVFrame *pic)
253 {
254     if (pic->data[0] == NULL)  // If no picture return a new buffer
255     {
256         return s->get_buffer(s, pic);
257     }
258
259     return 0;
260 }
261
```

释放内存数组项占用的内存。

```
262 void avcodec_default_free_buffers(AVCodecContext *s)
263 {
264     int i, j;
265
266     if (s->internal_buffer == NULL)
267         return ;
268
269     for (i = 0; i < INTERNAL_BUFFER_SIZE; i++)
270     {
271         InternalBuffer *buf = &((InternalBuffer*)s->internal_buffer)[i];
272         for (j = 0; j < 4; j++)
273         {
```

av_freep()函数调用的 av_free()函数做了非 NULL 判断，并且分配时已置 NULL，所以内循环可以到 4，外循环可以到 INTERNAL_BUFFER_SIZE。

```
274             av_freep(&buf->base[j]);
275             buf->data[j] = NULL;
276         }
277     }
278     av_freep(&s->internal_buffer);
279
280     s->internal_buffer_count = 0;
281 }
282
```

分配编解码器上下文占用的内存，清 0 后部分参数赋初值。

```
283 AVCodecContext *avcodec_alloc_context(void)
284 {
```

```
285     AVCodecContext *s = av_malloc(sizeof(AVCodecContext));
286
287     if (s == NULL)
288         return NULL;
289
```

注意这里的清 0。

```
290     memset(s, 0, sizeof(AVCodecContext));
291
292     s->get_buffer = avcodec_default_get_buffer;
293     s->release_buffer = avcodec_default_release_buffer;
294
295     s->pix_fmt = PIX_FMT_NONE;
296
297     s->palctrl = NULL;
298     s->reget_buffer = avcodec_default_reget_buffer;
299
300     return s;
301 }
302
```

打开编解码器，分配具体编解码器使用的上下文，简单变量赋初值，调用初始化函数初始化编解码器

```
303 int avcodec_open(AVCodecContext *avctx, AVCodec *codec)
304 {
305     int ret =  - 1;
306
307     if (avctx->codec)
308         goto end;
309
310     if (codec->priv_data_size > 0)
311     {
```

这里体现了 priv_data_size 参数的重大作用，如果没有这个参数，就要用 codec 结构的名字比较确定具体编解码器使用的上下文结构大小，超级长的 if-else 语句。

```
312         avctx->priv_data = av_mallocz(codec->priv_data_size);
313         if (!avctx->priv_data)
314             goto end;
315     }
316     else
317     {
318         avctx->priv_data = NULL;
```

```
319     }
320
321     avctx->codec = codec;
322     avctx->codec_id = codec->id;
323     avctx->frame_number = 0;
324     ret = avctx->codec->init(avctx);
325     if (ret < 0)
326     {
327         av_freep(&avctx->priv_data);
328         avctx->codec = NULL;
329         goto end;
330     }
331     ret = 0;
332 end:
333     return ret;
334 }
335
```

视频解码，简单的跳转

```
336 int avcodec_decode_video(AVCodecContext *avctx, AVFrame *picture, int *got_picture_ptr,
337                                                     uint8_t *buf, int buf_size)
338 {
339     int ret;
340
341     *got_picture_ptr = 0;
342
343     if (buf_size)
344     {
345         ret = avctx->codec->decode(avctx, picture, got_picture_ptr, buf, buf_size);
346
347         if (*got_picture_ptr)
348             avctx->frame_number++;
349     }
350     else
351         ret = 0;
352
353     return ret;
354 }
355
```

音频解码，简单的跳转

```
356 int avcodec_decode_audio(AVCodecContext *avctx, int16_t *samples, int *frame_size_ptr,
357                                                   uint8_t *buf, int buf_size)
358 {
359     int ret;
360
361     *frame_size_ptr = 0;
362     if (buf_size)
363     {
364         ret = avctx->codec->decode(avctx, samples, frame_size_ptr, buf, buf_size);
365         avctx->frame_number++;
366     }
367     else
368         ret = 0;
369     return ret;
370 }
371
```

关闭解码器，释放动态分配的内存

```
372 int avcodec_close(AVCodecContext *avctx)
373 {
374     if (avctx->codec->close)
375         avctx->codec->close(avctx);
376     avcodec_default_free_buffers(avctx);
377     av_freep(&avctx->priv_data);
378     avctx->codec = NULL;
379     return 0;
380 }
381
```

查找编解码器，在本例中，读 avi 文件头得到 codec FOURCC，再由 FOURCC 查找 codec_bmp_tags 或 codec_wav_tags 得到 CodecID 传给此函数。

```
382 AVCodec *avcodec_find_decoder(enum CodecID id)
383 {
384     AVCodec *p;
385     p = first_avcodec;
386     while (p)
387     {
388         if (p->decode != NULL && p->id == id)
389             return p;
390         p = p->next;
391     }
```

```
392     return NULL;
393 }
394
```

初始化编解码库，在本例中仅初始化限幅数组/查找表。

```
395 void avcodec_init(void)
396 {
397     static int inited = 0;
398
399     if (inited != 0)
400         return ;
401     inited = 1;
402
403     dsputil_static_init();
404 }
```

4.7 imgconvert_template.h 文件

4.7.1 功能描述

定义并实现图像颜色空间转换使用的函数和宏，此文件请各位自己仔细分析。

4.7.2 文件注释

```
1   #ifndef RGB_OUT
2   #define RGB_OUT(d, r, g, b) RGBA_OUT(d, r, g, b, 0xff)
3   #endif
4
5   #pragma warning (disable:4305 4244)
6
```

此文件请各位读者自行分析，都是些颜色空间转换函数。

```
7   static void glue(yuv420p_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
8   {
9       const uint8_t *y1_ptr,  *y2_ptr,  *cb_ptr,  *cr_ptr;
10      uint8_t *d,  *d1,  *d2;
11      int w, y, cb, cr, r_add, g_add, b_add, width2;
12      uint8_t *cm = cropTbl + MAX_NEG_CROP;
13      unsigned int r, g, b;
14
15      d = dst->data[0];
16      y1_ptr = src->data[0];
17      cb_ptr = src->data[1];
18      cr_ptr = src->data[2];
19      width2 = (width + 1) >> 1;
20
21      for (; height >= 2; height -= 2)
22      {
23          d1 = d;
24          d2 = d + dst->linesize[0];
25          y2_ptr = y1_ptr + src->linesize[0];
26          for (w = width; w >= 2; w -= 2)
27          {
28              YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
29
30              YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]);    /* output 4 pixels */
31              RGB_OUT(d1, r, g, b);
32
33              YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[1]);
```

```
34              RGB_OUT(d1 + BPP, r, g, b);
35
36              YUV_TO_RGB2_CCIR(r, g, b, y2_ptr[0]);
37              RGB_OUT(d2, r, g, b);
38
39              YUV_TO_RGB2_CCIR(r, g, b, y2_ptr[1]);
40              RGB_OUT(d2 + BPP, r, g, b);
41
42              d1 += 2 * BPP;
43              d2 += 2 * BPP;
44
45              y1_ptr += 2;
46              y2_ptr += 2;
47              cb_ptr++;
48              cr_ptr++;
49          }
50
51          if (w)      /* handle odd width */
52          {
53              YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
54              YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]);
55              RGB_OUT(d1, r, g, b);
56
57              YUV_TO_RGB2_CCIR(r, g, b, y2_ptr[0]);
58              RGB_OUT(d2, r, g, b);
59              d1 += BPP;
60              d2 += BPP;
61              y1_ptr++;
62              y2_ptr++;
63              cb_ptr++;
64              cr_ptr++;
65          }
66          d += 2 * dst->linesize[0];
67          y1_ptr += 2 * src->linesize[0] - width;
68          cb_ptr += src->linesize[1] - width2;
69          cr_ptr += src->linesize[2] - width2;
70      }
71
72      if (height)     /* handle odd height */
73      {
74          d1 = d;
```

```
75          for (w = width; w >= 2; w -= 2)
76          {
77              YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
78
79              YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]);    /* output 2 pixels */
80              RGB_OUT(d1, r, g, b);
81
82              YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[1]);
83              RGB_OUT(d1 + BPP, r, g, b);
84
85              d1 += 2 * BPP;
86
87              y1_ptr += 2;
88              cb_ptr++;
89              cr_ptr++;
90          }
91
92          if (w)    /* handle width */
93          {
94              YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
95
96              YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]);    /* output 2 pixels */
97              RGB_OUT(d1, r, g, b);
98              d1 += BPP;
99
100             y1_ptr++;
101             cb_ptr++;
102             cr_ptr++;
103         }
104     }
105 }
106
107 static void glue(yuvj420p_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
108 {
109     const uint8_t *y1_ptr,  *y2_ptr,  *cb_ptr,  *cr_ptr;
110     uint8_t *d,  *d1,  *d2;
111     int w, y, cb, cr, r_add, g_add, b_add, width2;
112     uint8_t *cm = cropTbl + MAX_NEG_CROP;
113     unsigned int r, g, b;
114
115     d = dst->data[0];
```

```
116     y1_ptr = src->data[0];
117     cb_ptr = src->data[1];
118     cr_ptr = src->data[2];
119     width2 = (width + 1) >> 1;
120
121     for (; height >= 2; height -= 2)
122     {
123         d1 = d;
124         d2 = d + dst->linesize[0];
125         y2_ptr = y1_ptr + src->linesize[0];
126         for (w = width; w >= 2; w -= 2)
127         {
128             YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
129
130             YUV_TO_RGB2(r, g, b, y1_ptr[0]);  /* output 4 pixels */
131             RGB_OUT(d1, r, g, b);
132
133             YUV_TO_RGB2(r, g, b, y1_ptr[1]);
134             RGB_OUT(d1 + BPP, r, g, b);
135
136             YUV_TO_RGB2(r, g, b, y2_ptr[0]);
137             RGB_OUT(d2, r, g, b);
138
139             YUV_TO_RGB2(r, g, b, y2_ptr[1]);
140             RGB_OUT(d2 + BPP, r, g, b);
141
142             d1 += 2 * BPP;
143             d2 += 2 * BPP;
144
145             y1_ptr += 2;
146             y2_ptr += 2;
147             cb_ptr++;
148             cr_ptr++;
149         }
150
151         if (w)    /* handle odd width */
152         {
153             YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
154             YUV_TO_RGB2(r, g, b, y1_ptr[0]);
155             RGB_OUT(d1, r, g, b);
156
```

```
157             YUV_TO_RGB2(r, g, b, y2_ptr[0]);
158             RGB_OUT(d2, r, g, b);
159             d1 += BPP;
160             d2 += BPP;
161             y1_ptr++;
162             y2_ptr++;
163             cb_ptr++;
164             cr_ptr++;
165         }
166         d += 2 * dst->linesize[0];
167         y1_ptr += 2 * src->linesize[0] - width;
168         cb_ptr += src->linesize[1] - width2;
169         cr_ptr += src->linesize[2] - width2;
170     }
171
172     if (height)    /* handle odd height */
173     {
174         d1 = d;
175         for (w = width; w >= 2; w -= 2)
176         {
177             YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
178
179             YUV_TO_RGB2(r, g, b, y1_ptr[0]);  /* output 2 pixels */
180             RGB_OUT(d1, r, g, b);
181
182             YUV_TO_RGB2(r, g, b, y1_ptr[1]);
183             RGB_OUT(d1 + BPP, r, g, b);
184
185             d1 += 2 * BPP;
186
187             y1_ptr += 2;
188             cb_ptr++;
189             cr_ptr++;
190         }
191
192         if (w)     /* handle width */
193         {
194             YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
195
196             YUV_TO_RGB2(r, g, b, y1_ptr[0]);  /* output 2 pixels */
197             RGB_OUT(d1, r, g, b);
```

```
198             d1 += BPP;
199
200             y1_ptr++;
201             cb_ptr++;
202             cr_ptr++;
203         }
204     }
205 }
206
207 static void glue(RGB_NAME, _to_yuv420p)(AVPicture *dst, const AVPicture *src, int width, int height)
208 {
209     int wrap, wrap3, width2;
210     int r, g, b, r1, g1, b1, w;
211     uint8_t *lum,  *cb,  *cr;
212     const uint8_t *p;
213
214     lum = dst->data[0];
215     cb = dst->data[1];
216     cr = dst->data[2];
217
218     width2 = (width + 1) >> 1;
219     wrap = dst->linesize[0];
220     wrap3 = src->linesize[0];
221     p = src->data[0];
222     for (; height >= 2; height -= 2)
223     {
224         for (w = width; w >= 2; w -= 2)
225         {
226             RGB_IN(r, g, b, p);
227             r1 = r;
228             g1 = g;
229             b1 = b;
230             lum[0] = RGB_TO_Y_CCIR(r, g, b);
231
232             RGB_IN(r, g, b, p + BPP);
233             r1 += r;
234             g1 += g;
235             b1 += b;
236             lum[1] = RGB_TO_Y_CCIR(r, g, b);
237             p += wrap3;
238             lum += wrap;
```

```
239
240             RGB_IN(r, g, b, p);
241             r1 += r;
242             g1 += g;
243             b1 += b;
244             lum[0] = RGB_TO_Y_CCIR(r, g, b);
245
246             RGB_IN(r, g, b, p + BPP);
247             r1 += r;
248             g1 += g;
249             b1 += b;
250             lum[1] = RGB_TO_Y_CCIR(r, g, b);
251
252             cb[0] = RGB_TO_U_CCIR(r1, g1, b1, 2);
253             cr[0] = RGB_TO_V_CCIR(r1, g1, b1, 2);
254
255             cb++;
256             cr++;
257             p +=  - wrap3 + 2 * BPP;
258             lum +=  - wrap + 2;
259         }
260         if (w)
261         {
262             RGB_IN(r, g, b, p);
263             r1 = r;
264             g1 = g;
265             b1 = b;
266             lum[0] = RGB_TO_Y_CCIR(r, g, b);
267             p += wrap3;
268             lum += wrap;
269             RGB_IN(r, g, b, p);
270             r1 += r;
271             g1 += g;
272             b1 += b;
273             lum[0] = RGB_TO_Y_CCIR(r, g, b);
274             cb[0] = RGB_TO_U_CCIR(r1, g1, b1, 1);
275             cr[0] = RGB_TO_V_CCIR(r1, g1, b1, 1);
276             cb++;
277             cr++;
278             p +=  - wrap3 + BPP;
279             lum +=  - wrap + 1;
```

```
280         }
281         p += wrap3 + (wrap3 - width * BPP);
282         lum += wrap + (wrap - width);
283         cb += dst->linesize[1] - width2;
284         cr += dst->linesize[2] - width2;
285     }
286
287     if (height)    /* handle odd height */
288     {
289         for (w = width; w >= 2; w -= 2)
290         {
291             RGB_IN(r, g, b, p);
292             r1 = r;
293             g1 = g;
294             b1 = b;
295             lum[0] = RGB_TO_Y_CCIR(r, g, b);
296
297             RGB_IN(r, g, b, p + BPP);
298             r1 += r;
299             g1 += g;
300             b1 += b;
301             lum[1] = RGB_TO_Y_CCIR(r, g, b);
302             cb[0] = RGB_TO_U_CCIR(r1, g1, b1, 1);
303             cr[0] = RGB_TO_V_CCIR(r1, g1, b1, 1);
304             cb++;
305             cr++;
306             p += 2 * BPP;
307             lum += 2;
308         }
309         if (w)
310         {
311             RGB_IN(r, g, b, p);
312             lum[0] = RGB_TO_Y_CCIR(r, g, b);
313             cb[0] = RGB_TO_U_CCIR(r, g, b, 0);
314             cr[0] = RGB_TO_V_CCIR(r, g, b, 0);
315         }
316     }
317 }
318
319 static void glue(RGB_NAME, _to_gray)(AVPicture *dst, const AVPicture *src, int width, int height)
320 {
```

```
321     const unsigned char *p;
322     unsigned char *q;
323     int r, g, b, dst_wrap, src_wrap;
324     int x, y;
325
326     p = src->data[0];
327     src_wrap = src->linesize[0] - BPP * width;
328
329     q = dst->data[0];
330     dst_wrap = dst->linesize[0] - width;
331
332     for (y = 0; y < height; y++)
333     {
334         for (x = 0; x < width; x++)
335         {
336             RGB_IN(r, g, b, p);
337             q[0] = RGB_TO_Y(r, g, b);
338             q++;
339             p += BPP;
340         }
341         p += src_wrap;
342         q += dst_wrap;
343     }
344 }
345
346 static void glue(gray_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
347 {
348     const unsigned char *p;
349     unsigned char *q;
350     int r, dst_wrap, src_wrap;
351     int x, y;
352
353     p = src->data[0];
354     src_wrap = src->linesize[0] - width;
355
356     q = dst->data[0];
357     dst_wrap = dst->linesize[0] - BPP * width;
358
359     for (y = 0; y < height; y++)
360     {
361         for (x = 0; x < width; x++)
```

```
362         {
363             r = p[0];
364             RGB_OUT(q, r, r, r);
365             q += BPP;
366             p++;
367         }
368         p += src_wrap;
369         q += dst_wrap;
370     }
371 }
372
373 static void glue(pal8_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
374 {
375     const unsigned char *p;
376     unsigned char *q;
377     int r, g, b, dst_wrap, src_wrap;
378     int x, y;
379     uint32_t v;
380     const uint32_t *palette;
381
382     p = src->data[0];
383     src_wrap = src->linesize[0] - width;
384     palette = (uint32_t*)src->data[1];
385
386     q = dst->data[0];
387     dst_wrap = dst->linesize[0] - BPP * width;
388
389     for (y = 0; y < height; y++)
390     {
391         for (x = 0; x < width; x++)
392         {
393             v = palette[p[0]];
394             r = (v >> 16) &0xff;
395             g = (v >> 8) &0xff;
396             b = (v) &0xff;
397 #ifdef RGBA_OUT
398             {
399                 int a;
400                 a = (v >> 24) &0xff;
401                 RGBA_OUT(q, r, g, b, a);
402             }
```

```
403 #else
404             RGB_OUT(q, r, g, b);
405 #endif
406             q += BPP;
407             p++;
408         }
409         p += src_wrap;
410         q += dst_wrap;
411     }
412 }
413
414 #if !defined(FMT_RGBA32) && defined(RGBA_OUT)
415 /* alpha support */
416
417 static void glue(rgba32_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
418 {
419     const uint8_t *s;
420     uint8_t *d;
421     int src_wrap, dst_wrap, j, y;
422     unsigned int v, r, g, b, a;
423
424     s = src->data[0];
425     src_wrap = src->linesize[0] - width * 4;
426
427     d = dst->data[0];
428     dst_wrap = dst->linesize[0] - width * BPP;
429
430     for (y = 0; y < height; y++)
431     {
432         for (j = 0; j < width; j++)
433         {
434             v = ((const uint32_t*)(s))[0];
435             a = (v >> 24) &0xff;
436             r = (v >> 16) &0xff;
437             g = (v >> 8) &0xff;
438             b = v &0xff;
439             RGBA_OUT(d, r, g, b, a);
440             s += 4;
441             d += BPP;
442         }
443         s += src_wrap;
```

```
444         d += dst_wrap;
445     }
446 }
447
448 static void glue(RGB_NAME, _to_rgba32)(AVPicture *dst, const AVPicture *src, int width, int height)
449 {
450     const uint8_t *s;
451     uint8_t *d;
452     int src_wrap, dst_wrap, j, y;
453     unsigned int r, g, b, a;
454
455     s = src->data[0];
456     src_wrap = src->linesize[0] - width * BPP;
457
458     d = dst->data[0];
459     dst_wrap = dst->linesize[0] - width * 4;
460
461     for (y = 0; y < height; y++)
462     {
463         for (j = 0; j < width; j++)
464         {
465             RGBA_IN(r, g, b, a, s);
466             ((uint32_t*)(d))[0] = (a << 24) | (r << 16) | (g << 8) | b;
467             d += 4;
468             s += BPP;
469         }
470         s += src_wrap;
471         d += dst_wrap;
472     }
473 }
474
475 #endif /* !defined(FMT_RGBA32) && defined(RGBA_IN) */
476
477 #ifndef FMT_RGB24
478
479 static void glue(rgb24_to_, RGB_NAME)(AVPicture *dst, const AVPicture *src, int width, int height)
480 {
481     const uint8_t *s;
482     uint8_t *d;
483     int src_wrap, dst_wrap, j, y;
484     unsigned int r, g, b;
```

```
485
486     s = src->data[0];
487     src_wrap = src->linesize[0] - width * 3;
488
489     d = dst->data[0];
490     dst_wrap = dst->linesize[0] - width * BPP;
491
492     for (y = 0; y < height; y++)
493     {
494         for (j = 0; j < width; j++)
495         {
496             r = s[0];
497             g = s[1];
498             b = s[2];
499             RGB_OUT(d, r, g, b);
500             s += 3;
501             d += BPP;
502         }
503         s += src_wrap;
504         d += dst_wrap;
505     }
506 }
507
508 static void glue(RGB_NAME, _to_rgb24)(AVPicture *dst, const AVPicture *src, int width, int height)
509 {
510     const uint8_t *s;
511     uint8_t *d;
512     int src_wrap, dst_wrap, j, y;
513     unsigned int r, g, b;
514
515     s = src->data[0];
516     src_wrap = src->linesize[0] - width * BPP;
517
518     d = dst->data[0];
519     dst_wrap = dst->linesize[0] - width * 3;
520
521     for (y = 0; y < height; y++)
522     {
523         for (j = 0; j < width; j++)
524         {
525             RGB_IN(r, g, b, s)d[0] = r;
```

```
526             d[1] = g;
527             d[2] = b;
528             d += 3;
529             s += BPP;
530         }
531         s += src_wrap;
532         d += dst_wrap;
533     }
534 }
535
536 #endif /* !FMT_RGB24 */
537
538 #ifdef FMT_RGB24
539
540 static void yuv444p_to_rgb24(AVPicture *dst, const AVPicture *src, int width, int height)
541 {
542     const uint8_t *y1_ptr,  *cb_ptr,  *cr_ptr;
543     uint8_t *d,  *d1;
544     int w, y, cb, cr, r_add, g_add, b_add;
545     uint8_t *cm = cropTbl + MAX_NEG_CROP;
546     unsigned int r, g, b;
547
548     d = dst->data[0];
549     y1_ptr = src->data[0];
550     cb_ptr = src->data[1];
551     cr_ptr = src->data[2];
552     for (; height > 0; height--)
553     {
554         d1 = d;
555         for (w = width; w > 0; w--)
556         {
557             YUV_TO_RGB1_CCIR(cb_ptr[0], cr_ptr[0]);
558
559             YUV_TO_RGB2_CCIR(r, g, b, y1_ptr[0]);
560             RGB_OUT(d1, r, g, b);
561             d1 += BPP;
562
563             y1_ptr++;
564             cb_ptr++;
565             cr_ptr++;
566         }
```

```
567         d += dst->linesize[0];
568         y1_ptr += src->linesize[0] - width;
569         cb_ptr += src->linesize[1] - width;
570         cr_ptr += src->linesize[2] - width;
571     }
572 }
573
574 static void yuvj444p_to_rgb24(AVPicture *dst, const AVPicture *src, int width, int height)
575 {
576     const uint8_t *y1_ptr,  *cb_ptr,  *cr_ptr;
577     uint8_t *d,  *d1;
578     int w, y, cb, cr, r_add, g_add, b_add;
579     uint8_t *cm = cropTbl + MAX_NEG_CROP;
580     unsigned int r, g, b;
581
582     d = dst->data[0];
583     y1_ptr = src->data[0];
584     cb_ptr = src->data[1];
585     cr_ptr = src->data[2];
586     for (; height > 0; height--)
587     {
588         d1 = d;
589         for (w = width; w > 0; w--)
590         {
591             YUV_TO_RGB1(cb_ptr[0], cr_ptr[0]);
592
593             YUV_TO_RGB2(r, g, b, y1_ptr[0]);
594             RGB_OUT(d1, r, g, b);
595             d1 += BPP;
596
597             y1_ptr++;
598             cb_ptr++;
599             cr_ptr++;
600         }
601         d += dst->linesize[0];
602         y1_ptr += src->linesize[0] - width;
603         cb_ptr += src->linesize[1] - width;
604         cr_ptr += src->linesize[2] - width;
605     }
606 }
607
```

```
608 static void rgb24_to_yuv444p(AVPicture *dst, const AVPicture *src, int width, int height)
609 {
610     int src_wrap, x, y;
611     int r, g, b;
612     uint8_t *lum,  *cb,  *cr;
613     const uint8_t *p;
614 
615     lum = dst->data[0];
616     cb = dst->data[1];
617     cr = dst->data[2];
618 
619     src_wrap = src->linesize[0] - width * BPP;
620     p = src->data[0];
621 
622     for (y = 0; y < height; y++)
623     {
624         for (x = 0; x < width; x++)
625         {
626             RGB_IN(r, g, b, p);
627             lum[0] = RGB_TO_Y_CCIR(r, g, b);
628             cb[0] = RGB_TO_U_CCIR(r, g, b, 0);
629             cr[0] = RGB_TO_V_CCIR(r, g, b, 0);
630             p += BPP;
631             cb++;
632             cr++;
633             lum++;
634         }
635         p += src_wrap;
636         lum += dst->linesize[0] - width;
637         cb += dst->linesize[1] - width;
638         cr += dst->linesize[2] - width;
639     }
640 }
641 
642 static void rgb24_to_yuvj420p(AVPicture *dst, const AVPicture *src, int width, int height)
643 {
644     int wrap, wrap3, width2;
645     int r, g, b, r1, g1, b1, w;
646     uint8_t *lum,  *cb,  *cr;
647     const uint8_t *p;
648 
```

```
649     lum = dst->data[0];
650     cb = dst->data[1];
651     cr = dst->data[2];
652 
653     width2 = (width + 1) >> 1;
654     wrap = dst->linesize[0];
655     wrap3 = src->linesize[0];
656     p = src->data[0];
657     for (; height >= 2; height -= 2)
658     {
659         for (w = width; w >= 2; w -= 2)
660         {
661             RGB_IN(r, g, b, p);
662             r1 = r;
663             g1 = g;
664             b1 = b;
665             lum[0] = RGB_TO_Y(r, g, b);
666 
667             RGB_IN(r, g, b, p + BPP);
668             r1 += r;
669             g1 += g;
670             b1 += b;
671             lum[1] = RGB_TO_Y(r, g, b);
672             p += wrap3;
673             lum += wrap;
674 
675             RGB_IN(r, g, b, p);
676             r1 += r;
677             g1 += g;
678             b1 += b;
679             lum[0] = RGB_TO_Y(r, g, b);
680 
681             RGB_IN(r, g, b, p + BPP);
682             r1 += r;
683             g1 += g;
684             b1 += b;
685             lum[1] = RGB_TO_Y(r, g, b);
686 
687             cb[0] = RGB_TO_U(r1, g1, b1, 2);
688             cr[0] = RGB_TO_V(r1, g1, b1, 2);
689 
```

```
690             cb++;
691             cr++;
692             p +=  - wrap3 + 2 * BPP;
693             lum +=  - wrap + 2;
694         }
695         if (w)
696         {
697             RGB_IN(r, g, b, p);
698             r1 = r;
699             g1 = g;
700             b1 = b;
701             lum[0] = RGB_TO_Y(r, g, b);
702             p += wrap3;
703             lum += wrap;
704             RGB_IN(r, g, b, p);
705             r1 += r;
706             g1 += g;
707             b1 += b;
708             lum[0] = RGB_TO_Y(r, g, b);
709             cb[0] = RGB_TO_U(r1, g1, b1, 1);
710             cr[0] = RGB_TO_V(r1, g1, b1, 1);
711             cb++;
712             cr++;
713             p +=  - wrap3 + BPP;
714             lum +=  - wrap + 1;
715         }
716         p += wrap3 + (wrap3 - width * BPP);
717         lum += wrap + (wrap - width);
718         cb += dst->linesize[1] - width2;
719         cr += dst->linesize[2] - width2;
720     }
721
722     if (height)    /* handle odd height */
723     {
724         for (w = width; w >= 2; w -= 2)
725         {
726             RGB_IN(r, g, b, p);
727             r1 = r;
728             g1 = g;
729             b1 = b;
730             lum[0] = RGB_TO_Y(r, g, b);
```

```
731
732             RGB_IN(r, g, b, p + BPP);
733             r1 += r;
734             g1 += g;
735             b1 += b;
736             lum[1] = RGB_TO_Y(r, g, b);
737             cb[0] = RGB_TO_U(r1, g1, b1, 1);
738             cr[0] = RGB_TO_V(r1, g1, b1, 1);
739             cb++;
740             cr++;
741             p += 2 * BPP;
742             lum += 2;
743         }
744         if (w)
745         {
746             RGB_IN(r, g, b, p);
747             lum[0] = RGB_TO_Y(r, g, b);
748             cb[0] = RGB_TO_U(r, g, b, 0);
749             cr[0] = RGB_TO_V(r, g, b, 0);
750         }
751     }
752 }
753
754 static void rgb24_to_yuvj444p(AVPicture *dst, const AVPicture *src, int width, int height)
755 {
756     int src_wrap, x, y;
757     int r, g, b;
758     uint8_t *lum,  *cb,  *cr;
759     const uint8_t *p;
760
761     lum = dst->data[0];
762     cb = dst->data[1];
763     cr = dst->data[2];
764
765     src_wrap = src->linesize[0] - width * BPP;
766     p = src->data[0];
767     for (y = 0; y < height; y++)
768     {
769         for (x = 0; x < width; x++)
770         {
771             RGB_IN(r, g, b, p);
```

```
772             lum[0] = RGB_TO_Y(r, g, b);
773             cb[0] = RGB_TO_U(r, g, b, 0);
774             cr[0] = RGB_TO_V(r, g, b, 0);
775             p += BPP;
776             cb++;
777             cr++;
778             lum++;
779         }
780         p += src_wrap;
781         lum += dst->linesize[0] - width;
782         cb += dst->linesize[1] - width;
783         cr += dst->linesize[2] - width;
784     }
785 }
786
787 #endif /* FMT_RGB24 */
788
789 #if defined(FMT_RGB24) || defined(FMT_RGBA32)
790
791 static void glue(RGB_NAME, _to_pal8)(AVPicture *dst, const AVPicture *src, int width, int height)
792 {
793     const unsigned char *p;
794     unsigned char *q;
795     int dst_wrap, src_wrap;
796     int x, y, has_alpha;
797     unsigned int r, g, b;
798
799     p = src->data[0];
800     src_wrap = src->linesize[0] - BPP * width;
801
802     q = dst->data[0];
803     dst_wrap = dst->linesize[0] - width;
804     has_alpha = 0;
805
806     for (y = 0; y < height; y++)
807     {
808         for (x = 0; x < width; x++)
809         {
810 #ifdef RGBA_IN
811             {
812                 unsigned int a;
```

```
813                 RGBA_IN(r, g, b, a, p);
814
815                 if (a < 0x80)   /* crude approximation for alpha ! */
816                 {
817                     has_alpha = 1;
818                     q[0] = TRANSP_INDEX;
819                 }
820                 else
821                 {
822                     q[0] = gif_clut_index(r, g, b);
823                 }
824             }
825 #else
826             RGB_IN(r, g, b, p);
827             q[0] = gif_clut_index(r, g, b);
828 #endif
829             q++;
830             p += BPP;
831         }
832         p += src_wrap;
833         q += dst_wrap;
834     }
835
836     build_rgb_palette(dst->data[1], has_alpha);
837 }
838
839 #endif /* defined(FMT_RGB24) || defined(FMT_RGBA32) */
840
841 #ifdef RGBA_IN
842
843 #define FF_ALPHA_TRANSP       0x0001 /* image has some totally transparent pixels */
844 #define FF_ALPHA_SEMI_TRANSP  0x0002 /* image has some transparent pixels */
845
846 static int glue(get_alpha_info_, RGB_NAME)(const AVPicture *src, int width, int height)
847 {
848     const unsigned char *p;
849     int src_wrap, ret, x, y;
850     unsigned int r, g, b, a;
851
852     p = src->data[0];
853     src_wrap = src->linesize[0] - BPP * width;
```

```
854     ret = 0;
855     for (y = 0; y < height; y++)
856     {
857         for (x = 0; x < width; x++)
858         {
859             RGBA_IN(r, g, b, a, p);
860             if (a == 0x00)
861             {
862                 ret |= FF_ALPHA_TRANSP;
863             }
864             else if (a != 0xff)
865             {
866                 ret |= FF_ALPHA_SEMI_TRANSP;
867             }
868             p += BPP;
869         }
870         p += src_wrap;
871     }
872     return ret;
873 }
874
875 #endif /* RGBA_IN */
876
877 #undef RGB_IN
878 #undef RGBA_IN
879 #undef RGB_OUT
880 #undef RGBA_OUT
881 #undef BPP
882 #undef RGB_NAME
883 #undef FMT_RGB24
884 #undef FMT_RGBA32
```

4.8 imgconvert.c 文件

4.8.1 功能描述

定义并实现图像颜色空间转换使用的函数和宏，此文件大部分请各位自己仔细分析。

4.8.2 文件注释

```
1   #include "avcodec.h"
2   #include "dsputil.h"
3
4   #define xglue(x, y) x ## y
5   #define glue(x, y) xglue(x, y)
6
7   #define FF_COLOR_RGB      0 // RGB color space
8   #define FF_COLOR_GRAY     1 // gray color space
9   #define FF_COLOR_YUV      2 // YUV color space. 16 <= Y <= 235, 16 <= U, V <= 240
10  #define FF_COLOR_YUV_JPEG 3 // YUV color space. 0 <= Y <= 255, 0 <= U, V <= 255
11
12  #define FF_PIXEL_PLANAR   0 // each channel has one component in AVPicture
13  #define FF_PIXEL_PACKED   1 // only one components containing all the channels
14  #define FF_PIXEL_PALETTE  2 // one components containing indexes for a palette
15
```

定义视频图像格式信息类型。

```
16  typedef struct PixFmtInfo
17  {
18      const char *name;
19      uint8_t nb_channels; // number of channels (including alpha)
20      uint8_t color_type; // color type (see FF_COLOR_xxx constants)
21      uint8_t pixel_type; // pixel storage type (see FF_PIXEL_xxx constants)
22      uint8_t is_alpha; // true if alpha can be specified
23      uint8_t x_chroma_shift; // X chroma subsampling factor is 2 ^ shift
24      uint8_t y_chroma_shift; // Y chroma subsampling factor is 2 ^ shift
25      uint8_t depth; // bit depth of the color components
26  } PixFmtInfo;
27
```

定义支持的视频图像格式信息。

```
28  // this table gives more information about formats
29  static PixFmtInfo pix_fmt_info[PIX_FMT_NB] =
30  {
31      { "yuv420p",  3, FF_COLOR_YUV, FF_PIXEL_PLANAR, 0, 1, 1, 8},
```

```
32      { "yuv422",    1, FF_COLOR_YUV,  FF_PIXEL_PACKED, 0, 1, 0, 8},
33      { "rgb24",     3, FF_COLOR_RGB,  FF_PIXEL_PACKED, 0, 0, 0, 8},
34      { "bgr24",     3, FF_COLOR_RGB,  FF_PIXEL_PACKED, 0, 0, 0, 8},
35      { "yuv422p",   3, FF_COLOR_YUV,  FF_PIXEL_PLANAR, 0, 1, 0, 8},
36      { "yuv444p",   3, FF_COLOR_YUV,  FF_PIXEL_PLANAR, 0, 0, 0, 8},
37      { "rgba32",    4, FF_COLOR_RGB,  FF_PIXEL_PACKED, 1, 0, 0, 8},
38      { "yuv410p",   3, FF_COLOR_YUV,  FF_PIXEL_PLANAR, 0, 2, 2, 8},
39      { "yuv411p",   3, FF_COLOR_YUV,  FF_PIXEL_PLANAR, 0, 2, 0, 8},
40      { "rgb565",    3, FF_COLOR_RGB,  FF_PIXEL_PACKED, 0, 0, 0, 5},
41      { "rgb555",    4, FF_COLOR_RGB,  FF_PIXEL_PACKED, 1, 0, 0, 5},
42      { "gray",      1, FF_COLOR_GRAY, FF_PIXEL_PLANAR, 0, 0, 0, 8},
43      { "monow",     1, FF_COLOR_GRAY, FF_PIXEL_PLANAR, 0, 0, 0, 1},
44      { "monob",     1, FF_COLOR_GRAY, FF_PIXEL_PLANAR, 0, 0, 0, 1},
45      { "pal8",      4, FF_COLOR_RGB,  FF_PIXEL_PALETTE, 1, 0, 0, 8},
46      { "yuvj420p", 3, FF_COLOR_YUV_JPEG, FF_PIXEL_PLANAR, 0, 1, 1, 8},
47      { "yuvj422p", 3, FF_COLOR_YUV_JPEG, FF_PIXEL_PLANAR, 0, 1, 0, 8},
48      { "yuvj444p", 3, FF_COLOR_YUV_JPEG, FF_PIXEL_PLANAR, 0, 0, 0, 8},
49      { "xvmcmc",  },
50      { "xvmcidct",},
51      { "uyvy422",   1, FF_COLOR_YUV,  FF_PIXEL_PACKED, 0, 1, 0, 8},
52      { "uyvy411",   1, FF_COLOR_YUV,  FF_PIXEL_PACKED, 0, 2, 0, 8},
53  };
54
```

读取视频图像格式信息中色度相对亮度采样比例(用移位的位数表示)。

```
55  void avcodec_get_chroma_sub_sample(int pix_fmt, int *h_shift, int *v_shift)
56  {
57      *h_shift = pix_fmt_info[pix_fmt].x_chroma_shift;
58      *v_shift = pix_fmt_info[pix_fmt].y_chroma_shift;
59  }
60
```

填充各种视频图像格式对应的 AVPicture 结构字段，返回图像大小。

```
61  // Picture field are filled with 'ptr' addresses. Also return size
62  int avpicture_fill(AVPicture *picture, uint8_t *ptr, int pix_fmt, int width, int height)
63  {
64      int size, w2, h2, size2;
65      PixFmtInfo *pinfo;
66
```

图像像素大小规整，比如 YUV420P 宽度和高度必须是 2 的整数倍，如果不符合，程序自动填充补足。

```
67      if (avcodec_check_dimensions(NULL, width, height))
68          goto fail;
69
70      pinfo = &pix_fmt_info[pix_fmt];
71      size  = width * height;
72      switch (pix_fmt)
73      {
```

按照图像格式，分别计算 AVPicture 结构字段的值。

```
74      case PIX_FMT_YUV420P:
75      case PIX_FMT_YUV422P:
76      case PIX_FMT_YUV444P:
77      case PIX_FMT_YUV410P:
78      case PIX_FMT_YUV411P:
79      case PIX_FMT_YUVJ420P:
80      case PIX_FMT_YUVJ422P:
81      case PIX_FMT_YUVJ444P:
82          w2 = (width + (1 << pinfo->x_chroma_shift) - 1) >> pinfo->x_chroma_shift;
83          h2 = (height + (1 << pinfo->y_chroma_shift) - 1) >> pinfo->y_chroma_shift;
84          size2  = w2 * h2;
85          picture->data[0] = ptr;
86          picture->data[1] = picture->data[0] + size;
87          picture->data[2] = picture->data[1] + size2;
88          picture->linesize[0] = width;
89          picture->linesize[1] = w2;
90          picture->linesize[2] = w2;
91          return size  + 2 * size2;
92      case PIX_FMT_RGB24:
93      case PIX_FMT_BGR24:
94          picture->data[0] = ptr;
95          picture->data[1] = NULL;
96          picture->data[2] = NULL;
97          picture->linesize[0] = width * 3;
98          return size *3;
99      case PIX_FMT_RGBA32:
100         picture->data[0] = ptr;
101         picture->data[1] = NULL;
102         picture->data[2] = NULL;
103         picture->linesize[0] = width * 4;
104         return size *4;
105     case PIX_FMT_RGB555:
```

```
106     case PIX_FMT_RGB565:
107     case PIX_FMT_YUV422:
108         picture->data[0] = ptr;
109         picture->data[1] = NULL;
110         picture->data[2] = NULL;
111         picture->linesize[0] = width * 2;
112         return size *2;
113     case PIX_FMT_UYVY422:
114         picture->data[0] = ptr;
115         picture->data[1] = NULL;
116         picture->data[2] = NULL;
117         picture->linesize[0] = width * 2;
118         return size *2;
119     case PIX_FMT_UYVY411:
120         picture->data[0] = ptr;
121         picture->data[1] = NULL;
122         picture->data[2] = NULL;
123         picture->linesize[0] = width + width / 2;
124         return size + size / 2;
125     case PIX_FMT_GRAY8:
126         picture->data[0] = ptr;
127         picture->data[1] = NULL;
128         picture->data[2] = NULL;
129         picture->linesize[0] = width;
130         return size;
131     case PIX_FMT_MONOWHITE:
132     case PIX_FMT_MONOBLACK:
133         picture->data[0] = ptr;
134         picture->data[1] = NULL;
135         picture->data[2] = NULL;
136         picture->linesize[0] = (width + 7) >> 3;
137         return picture->linesize[0] *height;
138     case PIX_FMT_PAL8:
139         size2 = (size + 3) &~3;
140         picture->data[0] = ptr;
141         picture->data[1] = ptr + size2; // palette is stored here as 256 32 bit words
142         picture->data[2] = NULL;
143         picture->linesize[0] = width;
144         picture->linesize[1] = 4;
145         return size2 + 256 * 4;
146     default:
```

```
147 fail:
148         picture->data[0] = NULL;
149         picture->data[1] = NULL;
150         picture->data[2] = NULL;
151         picture->data[3] = NULL;
152         return  - 1;
153     }
154 }
155
```

传入像素格式，图像长宽，计算图像大小。程序简单的复用 avpicture_fill()函数的返回值。

```
156 int avpicture_get_size(int pix_fmt, int width, int height)
157 {
158     AVPicture dummy_pict;
159     return avpicture_fill(&dummy_pict, NULL, pix_fmt, width, height);
160 }
161
```

初始化 AVPicture 结构。输入像素格式和长宽，计算图像大小，分配图像缓存，填充 AVPicture 结构。

```
162 int avpicture_alloc(AVPicture *picture, int pix_fmt, int width, int height)
163 {
164     unsigned int size;
165     void *ptr;
166
```

调用函数计算图像大小。

```
167     size  = avpicture_get_size(pix_fmt, width, height);
168     if (size < 0)
169         goto fail;
```

调用函数分配图像缓存。

```
170     ptr = av_malloc(size);
171     if (!ptr)
172         goto fail;
```

填充 AVPicture 结构。

```
173     avpicture_fill(picture, ptr, pix_fmt, width, height);
174     return 0;
175 fail:
176    memset(picture, 0, sizeof(AVPicture));
177     return  - 1;
```

```
178 }
179
```

释放 AVPicture 分配的内存，因为内存首地址在 picture->data[0]中，所以可以简单的释放。

```
180 void avpicture_free(AVPicture *picture)
181 {
182     av_free(picture->data[0]);
183 }
184
```

计算各种图像格式平均每个像素占用的 bit 位数。

```
185 static int avg_bits_per_pixel(int pix_fmt)
186 {
187     int bits;
188     const PixFmtInfo *pf;
189
190     pf = &pix_fmt_info[pix_fmt];
191     switch (pf->pixel_type)
192     {
193    case FF_PIXEL_PACKED:
194         switch (pix_fmt)
195         {
196         case PIX_FMT_YUV422:
197         case PIX_FMT_UYVY422:
198         case PIX_FMT_RGB565:
199         case PIX_FMT_RGB555:
200             bits = 16;
201             break;
202         case PIX_FMT_UYVY411:
203             bits = 12;
204             break;
205         default:
206             bits = pf->depth *pf->nb_channels;
207             break;
208         }
209         break;
210     case FF_PIXEL_PLANAR:
211          if (pf->x_chroma_shift == 0 && pf->y_chroma_shift == 0)
212          {
213              bits = pf->depth *pf->nb_channels;
214          }
```

```
215         else
216         {
217             bits = pf->depth + ((2 *pf->depth) >> (pf->x_chroma_shift + pf->y_chroma_shift));
218         }
219         break;
220     case FF_PIXEL_PALETTE:
221         bits = 8;
222         break;
223     default:
224         bits =  - 1;
225         break;
226     }
227     return bits;
228 }
229
230 ////////////////////////////
231
```

图像数据平面拷贝，由于宽度可能有差别，只能一行一行的拷贝。

```
232 void ff_img_copy_plane(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
233 {
234     if ((!dst) || (!src))
235         return ;
236     for (; height > 0; height--)
237     {
238         memcpy(dst, src, width);
239         dst += dst_wrap;
240         src += src_wrap;
241     }
242 }
243
```

各种图像格式的图像数据拷贝。

```
244 void img_copy(AVPicture *dst, const AVPicture *src, int pix_fmt, int width, int height)
245 {
246     int bwidth, bits, i;
247     PixFmtInfo *pf = &pix_fmt_info[pix_fmt];
248
249     pf = &pix_fmt_info[pix_fmt];
250     switch (pf->pixel_type)
251     {
```

```
252     case FF_PIXEL_PACKED:
253         switch (pix_fmt)
254         {
255         case PIX_FMT_YUV422:
256         case PIX_FMT_UYVY422:
257         case PIX_FMT_RGB565:
258         case PIX_FMT_RGB555:
259             bits = 16;
260             break;
261         case PIX_FMT_UYVY411:
262             bits = 12;
263             break;
264         default:
265             bits = pf->depth *pf->nb_channels;
266             break;
267         }
268         bwidth = (width *bits + 7) >> 3;
269         ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0], bwidth, height);
270         break;
271     case FF_PIXEL_PLANAR:
272          for (i = 0; i < pf->nb_channels; i++)
273          {
274              int w, h;
275              w = width;
276              h = height;
277              if (i == 1 || i == 2)
278              {
279                  w >>= pf->x_chroma_shift;
280                  h >>= pf->y_chroma_shift;
281              }
282              bwidth = (w *pf->depth + 7) >> 3;
283              ff_img_copy_plane(dst->data[i], dst->linesize[i], src->data[i], src->linesize[i], bwidth, h);
284          }
285          break;
286     case FF_PIXEL_PALETTE:
287          ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0], width, height);
288          // copy the palette
289          ff_img_copy_plane(dst->data[1], dst->linesize[1], src->data[1], src->linesize[1], 4,
256);
290          break;
291     }
```

```
292 }
293
```

本文件的后面部分请各位自行仔细分析。

```
294 static void yuv422_to_yuv420p(AVPicture *dst, const AVPicture *src, int width, int height)
295 {
296     const uint8_t *p,  *p1;
297     uint8_t *lum,  *cr,  *cb,  *lum1,  *cr1,  *cb1;
298     int w;
299
300     p1 = src->data[0];
301     lum1 = dst->data[0];
302     cb1 = dst->data[1];
303     cr1 = dst->data[2];
304
305     for (; height >= 1; height -= 2)
306     {
307         p = p1;
308         lum = lum1;
309         cb = cb1;
310         cr = cr1;
311         for (w = width; w >= 2; w -= 2)
312         {
313             lum[0] = p[0];
314             cb[0] = p[1];
315             lum[1] = p[2];
316             cr[0] = p[3];
317             p += 4;
318             lum += 2;
319             cb++;
320             cr++;
321         }
322         if (w)
323         {
324             lum[0] = p[0];
325             cb[0] = p[1];
326             cr[0] = p[3];
327             cb++;
328             cr++;
```

```
329         }
330         p1 += src->linesize[0];
331         lum1 += dst->linesize[0];
332         if (height > 1)
333         {
334             p = p1;
335             lum = lum1;
336             for (w = width; w >= 2; w -= 2)
337             {
338                 lum[0] = p[0];
339                 lum[1] = p[2];
340                 p += 4;
341                 lum += 2;
342             }
343             if (w)
344             {
345                 lum[0] = p[0];
346             }
347             p1 += src->linesize[0];
348             lum1 += dst->linesize[0];
349         }
350         cb1 += dst->linesize[1];
351         cr1 += dst->linesize[2];
352     }
353 }
354
355 static void uyvy422_to_yuv420p(AVPicture *dst, const AVPicture *src, int width, int height)
356 {
357     const uint8_t *p,  *p1;
358     uint8_t *lum,  *cr,  *cb,  *lum1,  *cr1,  *cb1;
359     int w;
360
361     p1 = src->data[0];
362
363     lum1 = dst->data[0];
364     cb1 = dst->data[1];
365     cr1 = dst->data[2];
366
367     for (; height >= 1; height -= 2)
368     {
369         p = p1;
```

```
370         lum = lum1;
371         cb = cb1;
372         cr = cr1;
373         for (w = width; w >= 2; w -= 2)
374         {
375             lum[0] = p[1];
376             cb[0] = p[0];
377             lum[1] = p[3];
378             cr[0] = p[2];
379             p += 4;
380             lum += 2;
381             cb++;
382             cr++;
383         }
384         if (w)
385         {
386             lum[0] = p[1];
387             cb[0] = p[0];
388             cr[0] = p[2];
389             cb++;
390             cr++;
391         }
392         p1 += src->linesize[0];
393         lum1 += dst->linesize[0];
394         if (height > 1)
395         {
396             p = p1;
397             lum = lum1;
398             for (w = width; w >= 2; w -= 2)
399             {
400                 lum[0] = p[1];
401                 lum[1] = p[3];
402                 p += 4;
403                 lum += 2;
404             }
405             if (w)
406             {
407                 lum[0] = p[1];
408             }
409             p1 += src->linesize[0];
410             lum1 += dst->linesize[0];
```

```
411         }
412         cb1 += dst->linesize[1];
413         cr1 += dst->linesize[2];
414     }
415 }
416
417 static void uyvy422_to_yuv422p(AVPicture *dst, const AVPicture *src, int width, int height)
418 {
419     const uint8_t *p,  *p1;
420     uint8_t *lum,  *cr,  *cb,  *lum1,  *cr1,  *cb1;
421     int w;
422
423     p1 = src->data[0];
424     lum1 = dst->data[0];
425     cb1 = dst->data[1];
426     cr1 = dst->data[2];
427     for (; height > 0; height--)
428     {
429         p = p1;
430         lum = lum1;
431         cb = cb1;
432         cr = cr1;
433         for (w = width; w >= 2; w -= 2)
434         {
435             lum[0] = p[1];
436             cb[0] = p[0];
437             lum[1] = p[3];
438             cr[0] = p[2];
439             p += 4;
440             lum += 2;
441             cb++;
442             cr++;
443         }
444         p1 += src->linesize[0];
445         lum1 += dst->linesize[0];
446         cb1 += dst->linesize[1];
447         cr1 += dst->linesize[2];
448     }
449 }
450
451 static void yuv422_to_yuv422p(AVPicture *dst, const AVPicture *src, int width, int height)
```

```
452 {
453     const uint8_t *p,  *p1;
454     uint8_t *lum,  *cr,  *cb,  *lum1,  *cr1,  *cb1;
455     int w;
456 
457     p1 = src->data[0];
458     lum1 = dst->data[0];
459     cb1 = dst->data[1];
460     cr1 = dst->data[2];
461     for (; height > 0; height--)
462     {
463         p = p1;
464         lum = lum1;
465         cb = cb1;
466         cr = cr1;
467         for (w = width; w >= 2; w -= 2)
468         {
469             lum[0] = p[0];
470             cb[0] = p[1];
471             lum[1] = p[2];
472             cr[0] = p[3];
473             p += 4;
474             lum += 2;
475             cb++;
476             cr++;
477         }
478         p1 += src->linesize[0];
479         lum1 += dst->linesize[0];
480         cb1 += dst->linesize[1];
481         cr1 += dst->linesize[2];
482     }
483 }
484 
485 static void yuv422p_to_yuv422(AVPicture *dst, const AVPicture *src, int width, int height)
486 {
487     uint8_t *p,  *p1;
488     const uint8_t *lum,  *cr,  *cb,  *lum1,  *cr1,  *cb1;
489     int w;
490 
491     p1 = dst->data[0];
492     lum1 = src->data[0];
```

```
493     cb1 = src->data[1];
494     cr1 = src->data[2];
495     for (; height > 0; height--)
496     {
497         p = p1;
498         lum = lum1;
499         cb = cb1;
500         cr = cr1;
501         for (w = width; w >= 2; w -= 2)
502         {
503             p[0] = lum[0];
504             p[1] = cb[0];
505             p[2] = lum[1];
506             p[3] = cr[0];
507             p += 4;
508             lum += 2;
509             cb++;
510             cr++;
511         }
512         p1 += dst->linesize[0];
513         lum1 += src->linesize[0];
514         cb1 += src->linesize[1];
515         cr1 += src->linesize[2];
516     }
517 }
518
519 static void yuv422p_to_uyvy422(AVPicture *dst, const AVPicture *src, int width, int height)
520 {
521     uint8_t *p,  *p1;
522     const uint8_t *lum,  *cr,  *cb,  *lum1,  *cr1,  *cb1;
523     int w;
524
525     p1 = dst->data[0];
526     lum1 = src->data[0];
527     cb1 = src->data[1];
528     cr1 = src->data[2];
529     for (; height > 0; height--)
530     {
531         p = p1;
532         lum = lum1;
533         cb = cb1;
```

```
534         cr = cr1;
535         for (w = width; w >= 2; w -= 2)
536         {
537             p[1] = lum[0];
538             p[0] = cb[0];
539             p[3] = lum[1];
540             p[2] = cr[0];
541             p += 4;
542             lum += 2;
543             cb++;
544             cr++;
545         }
546         p1 += dst->linesize[0];
547         lum1 += src->linesize[0];
548         cb1 += src->linesize[1];
549         cr1 += src->linesize[2];
550     }
551 }
552
553 static void uyvy411_to_yuv411p(AVPicture *dst, const AVPicture *src, int width, int height)
554 {
555     const uint8_t *p,  *p1;
556     uint8_t *lum,  *cr,  *cb,  *lum1,  *cr1,  *cb1;
557     int w;
558
559     p1 = src->data[0];
560     lum1 = dst->data[0];
561     cb1 = dst->data[1];
562     cr1 = dst->data[2];
563     for (; height > 0; height--)
564     {
565         p = p1;
566         lum = lum1;
567         cb = cb1;
568         cr = cr1;
569         for (w = width; w >= 4; w -= 4)
570         {
571             cb[0] = p[0];
572             lum[0] = p[1];
573             lum[1] = p[2];
574             cr[0] = p[3];
```

```
575                 lum[2] = p[4];
576                 lum[3] = p[5];
577                 p += 6;
578                 lum += 4;
579                 cb++;
580                 cr++;
581             }
582             p1 += src->linesize[0];
583             lum1 += dst->linesize[0];
584             cb1 += dst->linesize[1];
585             cr1 += dst->linesize[2];
586         }
587     }
588
589     static void yuv420p_to_yuv422(AVPicture *dst, const AVPicture *src, int width, int height)
590     {
591         int w, h;
592         uint8_t *line1,  *line2,  *linesrc = dst->data[0];
593         uint8_t *lum1,  *lum2,  *lumsrc = src->data[0];
594         uint8_t *cb1,  *cb2 = src->data[1];
595         uint8_t *cr1,  *cr2 = src->data[2];
596
597         for (h = height / 2; h--;)
598         {
599             line1 = linesrc;
600             line2 = linesrc + dst->linesize[0];
601
602             lum1 = lumsrc;
603             lum2 = lumsrc + src->linesize[0];
604
605             cb1 = cb2;
606             cr1 = cr2;
607
608             for (w = width / 2; w--;)
609             {
610                 *line1++ =  *lum1++;
611                 *line2++ =  *lum2++;
612                 *line1++ =  *line2++ =  *cb1++;
613                 *line1++ =  *lum1++;
614                 *line2++ =  *lum2++;
615                 *line1++ =  *line2++ =  *cr1++;
```

```
616         }
617
618         linesrc += dst->linesize[0] *2;
619         lumsrc += src->linesize[0] *2;
620         cb2 += src->linesize[1];
621         cr2 += src->linesize[2];
622     }
623 }
624
625 static void yuv420p_to_uyvy422(AVPicture *dst, const AVPicture *src, int width, int height)
626 {
627     int w, h;
628     uint8_t *line1,  *line2,  *linesrc = dst->data[0];
629     uint8_t *lum1,  *lum2,  *lumsrc = src->data[0];
630     uint8_t *cb1,  *cb2 = src->data[1];
631     uint8_t *cr1,  *cr2 = src->data[2];
632
633     for (h = height / 2; h--;)
634     {
635         line1 = linesrc;
636         line2 = linesrc + dst->linesize[0];
637
638         lum1 = lumsrc;
639         lum2 = lumsrc + src->linesize[0];
640
641         cb1 = cb2;
642         cr1 = cr2;
643
644         for (w = width / 2; w--;)
645         {
646             *line1++ =  *line2++ =  *cb1++;
647             *line1++ =  *lum1++;
648             *line2++ =  *lum2++;
649             *line1++ =  *line2++ =  *cr1++;
650             *line1++ =  *lum1++;
651             *line2++ =  *lum2++;
652         }
653
654         linesrc += dst->linesize[0] *2;
655         lumsrc += src->linesize[0] *2;
656         cb2 += src->linesize[1];
```

```
657         cr2 += src->linesize[2];
658     }
659 }
660
661 #define SCALEBITS 10
662 #define ONE_HALF  (1 << (SCALEBITS - 1))
663 #define FIX(x)    ((int) ((x) * (1<<SCALEBITS) + 0.5))
664
665 #define YUV_TO_RGB1_CCIR(cb1, cr1)\
666 {\
667     cb = (cb1) - 128;\
668     cr = (cr1) - 128;\
669     r_add = FIX(1.40200*255.0/224.0) * cr + ONE_HALF;\
670     g_add = - FIX(0.34414*255.0/224.0) * cb - FIX(0.71414*255.0/224.0) * cr + \
671     ONE_HALF;\
672     b_add = FIX(1.77200*255.0/224.0) * cb + ONE_HALF;\
673 }
674
675 #define YUV_TO_RGB2_CCIR(r, g, b, y1)\
676 {\
677     y = ((y1) - 16) * FIX(255.0/219.0);\
678     r = cm[(y + r_add) >> SCALEBITS];\
679     g = cm[(y + g_add) >> SCALEBITS];\
680     b = cm[(y + b_add) >> SCALEBITS];\
681 }
682
683 #define YUV_TO_RGB1(cb1, cr1)\
684 {\
685     cb = (cb1) - 128;\
686     cr = (cr1) - 128;\
687     r_add = FIX(1.40200) * cr + ONE_HALF;\
688     g_add = - FIX(0.34414) * cb - FIX(0.71414) * cr + ONE_HALF;\
689     b_add = FIX(1.77200) * cb + ONE_HALF;\
690 }
691
692 #define YUV_TO_RGB2(r, g, b, y1)\
693 {\
694     y = (y1) << SCALEBITS;\
695     r = cm[(y + r_add) >> SCALEBITS];\
696     g = cm[(y + g_add) >> SCALEBITS];\
697     b = cm[(y + b_add) >> SCALEBITS];\
```

```
698 }
699
700 #define Y_CCIR_TO_JPEG(y)\
701   cm[((y) * FIX(255.0/219.0) + (ONE_HALF - 16 * FIX(255.0/219.0))) >> SCALEBITS]
702
703 #define Y_JPEG_TO_CCIR(y)\
704   (((y) * FIX(219.0/255.0) + (ONE_HALF + (16 << SCALEBITS))) >> SCALEBITS)
705
706 #define C_CCIR_TO_JPEG(y)\
707   cm[(((y) - 128) * FIX(127.0/112.0) + (ONE_HALF + (128 << SCALEBITS))) >> SCALEBITS]
708
709 /* NOTE: the clamp is really necessary! */
710 static inline int C_JPEG_TO_CCIR(int y)
711 {
712     y = (((y - 128) *FIX(112.0 / 127.0) + (ONE_HALF + (128 << SCALEBITS))) >> SCALEBITS);
713     if (y < 16)
714         y = 16;
715     return y;
716 }
717
718 #define RGB_TO_Y(r, g, b) \
719   ((FIX(0.29900) * (r) + FIX(0.58700) * (g) + \
720   FIX(0.11400) * (b) + ONE_HALF) >> SCALEBITS)
721
722 #define RGB_TO_U(r1, g1, b1, shift)\
723   (((- FIX(0.16874) * r1 - FIX(0.33126) * g1 +         \
724   FIX(0.50000) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
725
726 #define RGB_TO_V(r1, g1, b1, shift)\
727   (((FIX(0.50000) * r1 - FIX(0.41869) * g1 -           \
728   FIX(0.08131) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
729
730 #define RGB_TO_Y_CCIR(r, g, b) \
731   ((FIX(0.29900*219.0/255.0) * (r) + FIX(0.58700*219.0/255.0) * (g) + \
732   FIX(0.11400*219.0/255.0) * (b) + (ONE_HALF + (16 << SCALEBITS))) >> SCALEBITS)
733
734 #define RGB_TO_U_CCIR(r1, g1, b1, shift)\
735   (((- FIX(0.16874*224.0/255.0) * r1 - FIX(0.33126*224.0/255.0) * g1 +         \
736   FIX(0.50000*224.0/255.0) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
737
738 #define RGB_TO_V_CCIR(r1, g1, b1, shift)\
```

```
739     (((FIX(0.50000*224.0/255.0) * r1 - FIX(0.41869*224.0/255.0) * g1 -           \
740     FIX(0.08131*224.0/255.0) * b1 + (ONE_HALF << shift) - 1) >> (SCALEBITS + shift)) + 128)
741
742 static uint8_t y_ccir_to_jpeg[256];
743 static uint8_t y_jpeg_to_ccir[256];
744 static uint8_t c_ccir_to_jpeg[256];
745 static uint8_t c_jpeg_to_ccir[256];
746
747 /* apply to each pixel the given table */
748 static void img_apply_table(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap,
749                             int width, int height, const uint8_t *table1)
750 {
751     int n;
752     const uint8_t *s;
753     uint8_t *d;
754     const uint8_t *table;
755
756     table = table1;
757     for (; height > 0; height--)
758     {
759         s = src;
760         d = dst;
761         n = width;
762         while (n >= 4)
763         {
764             d[0] = table[s[0]];
765             d[1] = table[s[1]];
766             d[2] = table[s[2]];
767             d[3] = table[s[3]];
768             d += 4;
769             s += 4;
770             n -= 4;
771         }
772         while (n > 0)
773         {
774             d[0] = table[s[0]];
775             d++;
776             s++;
777             n--;
778         }
779         dst += dst_wrap;
```

```
780         src += src_wrap;
781     }
782 }
783
784 /* XXX: use generic filter ? */
785 /* XXX: in most cases, the sampling position is incorrect */
786
787 /* 4x1 -> 1x1 */
788 static void shrink41(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
789 {
790     int w;
791     const uint8_t *s;
792     uint8_t *d;
793
794     for (; height > 0; height--)
795     {
796         s = src;
797         d = dst;
798         for (w = width; w > 0; w--)
799         {
800             d[0] = (s[0] + s[1] + s[2] + s[3] + 2) >> 2;
801             s += 4;
802             d++;
803         }
804         src += src_wrap;
805         dst += dst_wrap;
806     }
807 }
808
809 /* 2x1 -> 1x1 */
810 static void shrink21(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
811 {
812     int w;
813     const uint8_t *s;
814     uint8_t *d;
815
816     for (; height > 0; height--)
817     {
818         s = src;
819         d = dst;
820         for (w = width; w > 0; w--)
```

```
821         {
822             d[0] = (s[0] + s[1]) >> 1;
823             s += 2;
824             d++;
825         }
826         src += src_wrap;
827         dst += dst_wrap;
828     }
829 }
830
831 /* 1x2 -> 1x1 */
832 static void shrink12(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
833 {
834     int w;
835     uint8_t *d;
836     const uint8_t *s1,  *s2;
837
838     for (; height > 0; height--)
839     {
840         s1 = src;
841         s2 = s1 + src_wrap;
842         d = dst;
843         for (w = width; w >= 4; w -= 4)
844         {
845             d[0] = (s1[0] + s2[0]) >> 1;
846             d[1] = (s1[1] + s2[1]) >> 1;
847             d[2] = (s1[2] + s2[2]) >> 1;
848             d[3] = (s1[3] + s2[3]) >> 1;
849             s1 += 4;
850             s2 += 4;
851             d += 4;
852         }
853         for (; w > 0; w--)
854         {
855             d[0] = (s1[0] + s2[0]) >> 1;
856             s1++;
857             s2++;
858             d++;
859         }
860         src += 2 * src_wrap;
861         dst += dst_wrap;
```

```
862     }
863 }
864
865 /* 2x2 -> 1x1 */
866 void ff_shrink22(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
867 {
868     int w;
869     const uint8_t *s1,  *s2;
870     uint8_t *d;
871
872     for (; height > 0; height--)
873     {
874         s1 = src;
875         s2 = s1 + src_wrap;
876         d = dst;
877         for (w = width; w >= 4; w -= 4)
878         {
879             d[0] = (s1[0] + s1[1] + s2[0] + s2[1] + 2) >> 2;
880             d[1] = (s1[2] + s1[3] + s2[2] + s2[3] + 2) >> 2;
881             d[2] = (s1[4] + s1[5] + s2[4] + s2[5] + 2) >> 2;
882             d[3] = (s1[6] + s1[7] + s2[6] + s2[7] + 2) >> 2;
883             s1 += 8;
884             s2 += 8;
885             d += 4;
886         }
887         for (; w > 0; w--)
888         {
889             d[0] = (s1[0] + s1[1] + s2[0] + s2[1] + 2) >> 2;
890             s1 += 2;
891             s2 += 2;
892             d++;
893         }
894         src += 2 * src_wrap;
895         dst += dst_wrap;
896     }
897 }
898
899 /* 4x4 -> 1x1 */
900 void ff_shrink44(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
901 {
902     int w;
```

```
903     const uint8_t *s1,  *s2,  *s3,  *s4;
904     uint8_t *d;
905
906     for (; height > 0; height--)
907     {
908         s1 = src;
909         s2 = s1 + src_wrap;
910         s3 = s2 + src_wrap;
911         s4 = s3 + src_wrap;
912         d = dst;
913         for (w = width; w > 0; w--)
914         {
915             d[0] = (s1[0] + s1[1] + s1[2] + s1[3] + s2[0] + s2[1] + s2[2] + s2[3] +
916                     s3[0] + s3[1] + s3[2] + s3[3] + s4[0] + s4[1] + s4[2] + s4[3] + 8) >> 4;
917             s1 += 4;
918             s2 += 4;
919             s3 += 4;
920             s4 += 4;
921             d++;
922         }
923         src += 4 * src_wrap;
924         dst += dst_wrap;
925     }
926 }
927
928 static void grow21_line(uint8_t *dst, const uint8_t *src, int width)
929 {
930     int w;
931     const uint8_t *s1;
932     uint8_t *d;
933
934     s1 = src;
935     d = dst;
936     for (w = width; w >= 4; w -= 4)
937     {
938         d[1] = d[0] = s1[0];
939         d[3] = d[2] = s1[1];
940         s1 += 2;
941         d += 4;
942     }
943     for (; w >= 2; w -= 2)
```

```
944     {
945         d[1] = d[0] = s1[0];
946         s1++;
947         d += 2;
948     }
949     /* only needed if width is not a multiple of two */
950     /* XXX: veryfy that */
951     if (w)
952     {
953         d[0] = s1[0];
954     }
955 }
956
957 static void grow41_line(uint8_t *dst, const uint8_t *src, int width)
958 {
959     int w, v;
960     const uint8_t *s1;
961     uint8_t *d;
962
963     s1 = src;
964     d = dst;
965     for (w = width; w >= 4; w -= 4)
966     {
967         v = s1[0];
968         d[0] = v;
969         d[1] = v;
970         d[2] = v;
971         d[3] = v;
972         s1++;
973         d += 4;
974     }
975 }
976
977 /* 1x1 -> 2x1 */
978 static void grow21(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
979 {
980     for (; height > 0; height--)
981     {
982         grow21_line(dst, src, width);
983         src += src_wrap;
984         dst += dst_wrap;
```

```
985     }
986 }
987
988 /* 1x1 -> 2x2 */
989 static void grow22(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
990 {
991     for (; height > 0; height--)
992     {
993         grow21_line(dst, src, width);
994         if (height % 2)
995             src += src_wrap;
996         dst += dst_wrap;
997     }
998 }
999
1000 /* 1x1 -> 4x1 */
1001 static void grow41(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
1002 {
1003     for (; height > 0; height--)
1004     {
1005         grow41_line(dst, src, width);
1006         src += src_wrap;
1007         dst += dst_wrap;
1008     }
1009 }
1010
1011 /* 1x1 -> 4x4 */
1012 static void grow44(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
1013 {
1014     for (; height > 0; height--)
1015     {
1016         grow41_line(dst, src, width);
1017         if ((height &3) == 1)
1018             src += src_wrap;
1019         dst += dst_wrap;
1020     }
1021 }
1022
1023 /* 1x2 -> 2x1 */
1024 static void conv411(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap, int width, int height)
1025 {
```

```
1026     int w, c;
1027     const uint8_t *s1,  *s2;
1028     uint8_t *d;
1029
1030     width >>= 1;
1031
1032     for (; height > 0; height--)
1033     {
1034         s1 = src;
1035         s2 = src + src_wrap;
1036         d = dst;
1037         for (w = width; w > 0; w--)
1038         {
1039             c = (s1[0] + s2[0]) >> 1;
1040             d[0] = c;
1041             d[1] = c;
1042             s1++;
1043             s2++;
1044             d += 2;
1045         }
1046         src += src_wrap * 2;
1047         dst += dst_wrap;
1048     }
1049 }
1050
1051 /* XXX: add jpeg quantize code */
1052
1053 #define TRANSP_INDEX (6*6*6)
1054
1055 /* this is maybe slow, but allows for extensions */
1056 static inline unsigned char gif_clut_index(uint8_t r, uint8_t g, uint8_t b)
1057 {
1058     return ((((r) / 47) % 6) *6 * 6+(((g) / 47) % 6) *6+(((b) / 47) % 6));
1059 }
1060
1061 static void build_rgb_palette(uint8_t *palette, int has_alpha)
1062 {
1063     uint32_t *pal;
1064     static const uint8_t pal_value[6] = {0x00, 0x33, 0x66, 0x99, 0xcc, 0xff };
1065     int i, r, g, b;
1066
```

```
1067     pal = (uint32_t*)palette;
1068     i = 0;
1069     for (r = 0; r < 6; r++)
1070     {
1071         for (g = 0; g < 6; g++)
1072         {
1073             for (b = 0; b < 6; b++)
1074             {
1075                 pal[i++] = (0xff << 24) | (pal_value[r] << 16) | (pal_value[g] << 8) | pal_value[b];
1076             }
1077         }
1078     }
1079     if (has_alpha)
1080         pal[i++] = 0;
1081     while (i < 256)
1082         pal[i++] = 0xff000000;
1083 }
1084
1085 /* copy bit n to bits 0 ... n - 1 */
1086 static inline unsigned int bitcopy_n(unsigned int a, int n)
1087 {
1088     int mask;
1089     mask = (1 << n) - 1;
1090     return (a &(0xff &~mask)) | (( - ((a >> n) &1)) &mask);
1091 }
1092
1093 /* rgb555 handling */
1094
1095 #define RGB_NAME rgb555
1096
1097 #define RGB_IN(r, g, b, s)\
1098 {\
1099    unsigned int v = ((const uint16_t *)(s))[0];\
1100    r = bitcopy_n(v >> (10 - 3), 3);\
1101    g = bitcopy_n(v >> (5 - 3), 3);\
1102    b = bitcopy_n(v << 3, 3);\
1103 }
1104
1105 #define RGBA_IN(r, g, b, a, s)\
1106 {\
1107    unsigned int v = ((const uint16_t *)(s))[0];\
```

```
1108    r = bitcopy_n(v >> (10 - 3), 3);\
1109    g = bitcopy_n(v >> (5 - 3), 3);\
1110    b = bitcopy_n(v << 3, 3);\
1111    a = (-(v >> 15)) & 0xff;\
1112 }
1113
1114 #define RGBA_OUT(d, r, g, b, a)\
1115 {\
1116    ((uint16_t *)(d))[0] = ((r >> 3) << 10) | ((g >> 3) << 5) | (b >> 3) | \
1117    ((a << 8) & 0x8000);\
1118 }
1119
1120 #define BPP 2
1121
1122 #include "imgconvert_template.h"
1123
1124 /* rgb565 handling */
1125
1126 #define RGB_NAME rgb565
1127
1128 #define RGB_IN(r, g, b, s)\
1129 {\
1130    unsigned int v = ((const uint16_t *)(s))[0];\
1131    r = bitcopy_n(v >> (11 - 3), 3);\
1132    g = bitcopy_n(v >> (5 - 2), 2);\
1133    b = bitcopy_n(v << 3, 3);\
1134 }
1135
1136 #define RGB_OUT(d, r, g, b)\
1137 {\
1138    ((uint16_t *)(d))[0] = ((r >> 3) << 11) | ((g >> 2) << 5) | (b >> 3);\
1139 }
1140
1141 #define BPP 2
1142
1143 #include "imgconvert_template.h"
1144
1145 /* bgr24 handling */
1146
1147 #define RGB_NAME bgr24
1148
```

```
1149 #define RGB_IN(r, g, b, s)\
1150 {\
1151     b = (s)[0];\
1152     g = (s)[1];\
1153     r = (s)[2];\
1154 }
1155
1156 #define RGB_OUT(d, r, g, b)\
1157 {\
1158     (d)[0] = b;\
1159     (d)[1] = g;\
1160     (d)[2] = r;\
1161 }
1162
1163 #define BPP 3
1164
1165 #include "imgconvert_template.h"
1166
1167 #undef RGB_IN
1168 #undef RGB_OUT
1169 #undef BPP
1170
1171 /* rgb24 handling */
1172
1173 #define RGB_NAME rgb24
1174 #define FMT_RGB24
1175
1176 #define RGB_IN(r, g, b, s)\
1177 {\
1178     r = (s)[0];\
1179     g = (s)[1];\
1180     b = (s)[2];\
1181 }
1182
1183 #define RGB_OUT(d, r, g, b)\
1184 {\
1185     (d)[0] = r;\
1186     (d)[1] = g;\
1187     (d)[2] = b;\
1188 }
1189
```

```
1190 #define BPP 3
1191
1192 #include "imgconvert_template.h"
1193
1194 /* rgba32 handling */
1195
1196 #define RGB_NAME rgba32
1197 #define FMT_RGBA32
1198
1199 #define RGB_IN(r, g, b, s)\
1200 {\
1201     unsigned int v = ((const uint32_t *)(s))[0];\
1202     r = (v >> 16) & 0xff;\
1203     g = (v >> 8) & 0xff;\
1204     b = v & 0xff;\
1205 }
1206
1207 #define RGBA_IN(r, g, b, a, s)\
1208 {\
1209     unsigned int v = ((const uint32_t *)(s))[0];\
1210     a = (v >> 24) & 0xff;\
1211     r = (v >> 16) & 0xff;\
1212     g = (v >> 8) & 0xff;\
1213     b = v & 0xff;\
1214 }
1215
1216 #define RGBA_OUT(d, r, g, b, a)\
1217 {\
1218     ((uint32_t *)(d))[0] = (a << 24) | (r << 16) | (g << 8) | b;\
1219 }
1220
1221 #define BPP 4
1222
1223 #include "imgconvert_template.h"
1224
1225 static void mono_to_gray(AVPicture *dst, const AVPicture *src, int width, int height, int xor_mask)
1226 {
1227     const unsigned char *p;
1228     unsigned char *q;
1229     int v, dst_wrap, src_wrap;
1230     int y, w;
```

```
1231
1232    p = src->data[0];
1233    src_wrap = src->linesize[0] - ((width + 7) >> 3);
1234
1235    q = dst->data[0];
1236    dst_wrap = dst->linesize[0] - width;
1237    for (y = 0; y < height; y++)
1238    {
1239        w = width;
1240        while (w >= 8)
1241        {
1242            v =  *p++ ^ xor_mask;
1243            q[0] =  - (v >> 7);
1244            q[1] =  - ((v >> 6) &1);
1245            q[2] =  - ((v >> 5) &1);
1246            q[3] =  - ((v >> 4) &1);
1247            q[4] =  - ((v >> 3) &1);
1248            q[5] =  - ((v >> 2) &1);
1249            q[6] =  - ((v >> 1) &1);
1250            q[7] =  - ((v >> 0) &1);
1251            w -= 8;
1252            q += 8;
1253        }
1254        if (w > 0)
1255        {
1256            v =  *p++ ^ xor_mask;
1257            do
1258            {
1259                q[0] =  - ((v >> 7) &1);
1260                q++;
1261                v <<= 1;
1262            }
1263            while (--w);
1264        }
1265        p += src_wrap;
1266        q += dst_wrap;
1267    }
1268 }
1269
1270 static void monowhite_to_gray(AVPicture *dst, const AVPicture *src, int width, int height)
1271 {
```

```
1272     mono_to_gray(dst, src, width, height, 0xff);
1273 }
1274
1275 static void monoblack_to_gray(AVPicture *dst, const AVPicture *src, int width, int height)
1276 {
1277     mono_to_gray(dst, src, width, height, 0x00);
1278 }
1279
1280 static void gray_to_mono(AVPicture *dst, const AVPicture *src, int width, int height, int xor_mask)
1281 {
1282     int n;
1283     const uint8_t *s;
1284     uint8_t *d;
1285     int j, b, v, n1, src_wrap, dst_wrap, y;
1286
1287     s = src->data[0];
1288     src_wrap = src->linesize[0] - width;
1289
1290     d = dst->data[0];
1291     dst_wrap = dst->linesize[0] - ((width + 7) >> 3);
1292
1293     for (y = 0; y < height; y++)
1294     {
1295         n = width;
1296         while (n >= 8)
1297         {
1298             v = 0;
1299             for (j = 0; j < 8; j++)
1300             {
1301                 b = s[0];
1302                 s++;
1303                 v = (v << 1) | (b >> 7);
1304             }
1305             d[0] = v ^ xor_mask;
1306             d++;
1307             n -= 8;
1308         }
1309         if (n > 0)
1310         {
1311             n1 = n;
1312             v = 0;
```

```
1313             while (n > 0)
1314             {
1315                 b = s[0];
1316                 s++;
1317                 v = (v << 1) | (b >> 7);
1318                 n--;
1319             }
1320             d[0] = (v << (8-(n1 &7))) ^ xor_mask;
1321             d++;
1322         }
1323         s += src_wrap;
1324         d += dst_wrap;
1325     }
1326 }
1327
1328 static void gray_to_monowhite(AVPicture *dst, const AVPicture *src, int width, int height)
1329 {
1330     gray_to_mono(dst, src, width, height, 0xff);
1331 }
1332
1333 static void gray_to_monoblack(AVPicture *dst, const AVPicture *src, int width, int height)
1334 {
1335     gray_to_mono(dst, src, width, height, 0x00);
1336 }
1337
1338 typedef struct ConvertEntry
1339 {
1340     void(*convert)(AVPicture *dst, const AVPicture *src, int width, int height);
1341 } ConvertEntry;
1342
1343 /* Add each new convertion function in this table. In order to be able
1344 to convert from any format to any format, the following constraints must be satisfied:
1345
1346   - all FF_COLOR_RGB formats must convert to and from PIX_FMT_RGB24
1347
1348   - all FF_COLOR_GRAY formats must convert to and from PIX_FMT_GRAY8
1349
1350   - all FF_COLOR_RGB formats with alpha must convert to and from PIX_FMT_RGBA32
1351
1352   - PIX_FMT_YUV444P and PIX_FMT_YUVJ444P must convert to and from PIX_FMT_RGB24.
1353
```

```
1354   - PIX_FMT_422 must convert to and from PIX_FMT_422P.
1355
1356   The other conversion functions are just optimisations for common cases.
1357 */
1358
1359 static ConvertEntry convert_table[PIX_FMT_NB][PIX_FMT_NB];
1360
1361 static void img_convert_init(void)
1362 {
1363     int i;
1364     uint8_t *cm = cropTbl + MAX_NEG_CROP;
1365
1366     for (i = 0; i < 256; i++)
1367     {
1368         y_ccir_to_jpeg[i] = Y_CCIR_TO_JPEG(i);
1369         y_jpeg_to_ccir[i] = Y_JPEG_TO_CCIR(i);
1370         c_ccir_to_jpeg[i] = C_CCIR_TO_JPEG(i);
1371         c_jpeg_to_ccir[i] = C_JPEG_TO_CCIR(i);
1372     }
1373
1374     convert_table[PIX_FMT_YUV420P][PIX_FMT_YUV422].convert = yuv420p_to_yuv422;
1375     convert_table[PIX_FMT_YUV420P][PIX_FMT_YUV422].convert = yuv420p_to_yuv422;
1376     convert_table[PIX_FMT_YUV420P][PIX_FMT_RGB555].convert = yuv420p_to_rgb555;
1377     convert_table[PIX_FMT_YUV420P][PIX_FMT_RGB565].convert = yuv420p_to_rgb565;
1378     convert_table[PIX_FMT_YUV420P][PIX_FMT_BGR24].convert = yuv420p_to_bgr24;
1379     convert_table[PIX_FMT_YUV420P][PIX_FMT_RGB24].convert = yuv420p_to_rgb24;
1380     convert_table[PIX_FMT_YUV420P][PIX_FMT_RGBA32].convert = yuv420p_to_rgba32;
1381     convert_table[PIX_FMT_YUV420P][PIX_FMT_UYVY422].convert = yuv420p_to_uyvy422;
1382
1383     convert_table[PIX_FMT_YUV422P][PIX_FMT_YUV422].convert = yuv422p_to_yuv422;
1384     convert_table[PIX_FMT_YUV422P][PIX_FMT_UYVY422].convert = yuv422p_to_uyvy422;
1385
1386     convert_table[PIX_FMT_YUV444P][PIX_FMT_RGB24].convert = yuv444p_to_rgb24;
1387
1388     convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGB555].convert = yuvj420p_to_rgb555;
1389     convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGB565].convert = yuvj420p_to_rgb565;
1390     convert_table[PIX_FMT_YUVJ420P][PIX_FMT_BGR24].convert = yuvj420p_to_bgr24;
1391     convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGB24].convert = yuvj420p_to_rgb24;
1392     convert_table[PIX_FMT_YUVJ420P][PIX_FMT_RGBA32].convert = yuvj420p_to_rgba32;
1393
1394     convert_table[PIX_FMT_YUVJ444P][PIX_FMT_RGB24].convert = yuvj444p_to_rgb24;
```

```
1395
1396    convert_table[PIX_FMT_YUV422][PIX_FMT_YUV420P].convert = yuv422_to_yuv420p;
1397    convert_table[PIX_FMT_YUV422][PIX_FMT_YUV422P].convert = yuv422_to_yuv422p;
1398
1399    convert_table[PIX_FMT_UYVY422][PIX_FMT_YUV420P].convert = uyvy422_to_yuv420p;
1400    convert_table[PIX_FMT_UYVY422][PIX_FMT_YUV422P].convert = uyvy422_to_yuv422p;
1401
1402    convert_table[PIX_FMT_RGB24][PIX_FMT_YUV420P].convert = rgb24_to_yuv420p;
1403    convert_table[PIX_FMT_RGB24][PIX_FMT_RGB565].convert = rgb24_to_rgb565;
1404    convert_table[PIX_FMT_RGB24][PIX_FMT_RGB555].convert = rgb24_to_rgb555;
1405    convert_table[PIX_FMT_RGB24][PIX_FMT_RGBA32].convert = rgb24_to_rgba32;
1406    convert_table[PIX_FMT_RGB24][PIX_FMT_BGR24].convert = rgb24_to_bgr24;
1407    convert_table[PIX_FMT_RGB24][PIX_FMT_GRAY8].convert = rgb24_to_gray;
1408    convert_table[PIX_FMT_RGB24][PIX_FMT_PAL8].convert = rgb24_to_pal8;
1409    convert_table[PIX_FMT_RGB24][PIX_FMT_YUV444P].convert = rgb24_to_yuv444p;
1410    convert_table[PIX_FMT_RGB24][PIX_FMT_YUVJ420P].convert = rgb24_to_yuvj420p;
1411    convert_table[PIX_FMT_RGB24][PIX_FMT_YUVJ444P].convert = rgb24_to_yuvj444p;
1412
1413    convert_table[PIX_FMT_RGBA32][PIX_FMT_RGB24].convert = rgba32_to_rgb24;
1414    convert_table[PIX_FMT_RGBA32][PIX_FMT_RGB555].convert = rgba32_to_rgb555;
1415    convert_table[PIX_FMT_RGBA32][PIX_FMT_PAL8].convert = rgba32_to_pal8;
1416    convert_table[PIX_FMT_RGBA32][PIX_FMT_YUV420P].convert = rgba32_to_yuv420p;
1417    convert_table[PIX_FMT_RGBA32][PIX_FMT_GRAY8].convert = rgba32_to_gray;
1418
1419    convert_table[PIX_FMT_BGR24][PIX_FMT_RGB24].convert = bgr24_to_rgb24;
1420    convert_table[PIX_FMT_BGR24][PIX_FMT_YUV420P].convert = bgr24_to_yuv420p;
1421    convert_table[PIX_FMT_BGR24][PIX_FMT_GRAY8].convert = bgr24_to_gray;
1422
1423    convert_table[PIX_FMT_RGB555][PIX_FMT_RGB24].convert = rgb555_to_rgb24;
1424    convert_table[PIX_FMT_RGB555][PIX_FMT_RGBA32].convert = rgb555_to_rgba32;
1425    convert_table[PIX_FMT_RGB555][PIX_FMT_YUV420P].convert = rgb555_to_yuv420p;
1426    convert_table[PIX_FMT_RGB555][PIX_FMT_GRAY8].convert = rgb555_to_gray;
1427
1428    convert_table[PIX_FMT_RGB565][PIX_FMT_RGB24].convert = rgb565_to_rgb24;
1429    convert_table[PIX_FMT_RGB565][PIX_FMT_YUV420P].convert = rgb565_to_yuv420p;
1430    convert_table[PIX_FMT_RGB565][PIX_FMT_GRAY8].convert = rgb565_to_gray;
1431
1432    convert_table[PIX_FMT_GRAY8][PIX_FMT_RGB555].convert = gray_to_rgb555;
1433    convert_table[PIX_FMT_GRAY8][PIX_FMT_RGB565].convert = gray_to_rgb565;
1434    convert_table[PIX_FMT_GRAY8][PIX_FMT_RGB24].convert = gray_to_rgb24;
1435    convert_table[PIX_FMT_GRAY8][PIX_FMT_BGR24].convert = gray_to_bgr24;
```

```
1436     convert_table[PIX_FMT_GRAY8][PIX_FMT_RGBA32].convert = gray_to_rgba32;
1437     convert_table[PIX_FMT_GRAY8][PIX_FMT_MONOWHITE].convert = gray_to_monowhite;
1438     convert_table[PIX_FMT_GRAY8][PIX_FMT_MONOBLACK].convert = gray_to_monoblack;
1439
1440     convert_table[PIX_FMT_MONOWHITE][PIX_FMT_GRAY8].convert = monowhite_to_gray;
1441
1442     convert_table[PIX_FMT_MONOBLACK][PIX_FMT_GRAY8].convert = monoblack_to_gray;
1443
1444     convert_table[PIX_FMT_PAL8][PIX_FMT_RGB555].convert = pal8_to_rgb555;
1445     convert_table[PIX_FMT_PAL8][PIX_FMT_RGB565].convert = pal8_to_rgb565;
1446     convert_table[PIX_FMT_PAL8][PIX_FMT_BGR24].convert = pal8_to_bgr24;
1447     convert_table[PIX_FMT_PAL8][PIX_FMT_RGB24].convert = pal8_to_rgb24;
1448     convert_table[PIX_FMT_PAL8][PIX_FMT_RGBA32].convert = pal8_to_rgba32;
1449
1450     convert_table[PIX_FMT_UYVY411][PIX_FMT_YUV411P].convert = uyvy411_to_yuv411p;
1451 }
1452
1453 static inline int is_yuv_planar(PixFmtInfo *ps)
1454 {
1455     return (ps->color_type == FF_COLOR_YUV || ps->color_type == FF_COLOR_YUV_JPEG)
1456         && ps->pixel_type == FF_PIXEL_PLANAR;
1457 }
1458
1459 int img_convert(AVPicture *dst, int dst_pix_fmt, const AVPicture *src, int src_pix_fmt,
1460                 int src_width, int src_height)
1461 {
1462     static int inited;
1463     int i, ret, dst_width, dst_height, int_pix_fmt;
1464     PixFmtInfo *src_pix,  *dst_pix;
1465     ConvertEntry *ce;
1466     AVPicture tmp1,  *tmp = &tmp1;
1467
1468     if (src_pix_fmt < 0 || src_pix_fmt >= PIX_FMT_NB || dst_pix_fmt < 0 || dst_pix_fmt >= PIX_FMT_NB)
1469         return  - 1;
1470
1471     if (src_width <= 0 || src_height <= 0)
1472         return 0;
1473
1474     if (!inited)
1475     {
1476         inited = 1;
```

```
1477        img_convert_init();
1478    }
1479
1480    dst_width = src_width;
1481    dst_height = src_height;
1482
1483    dst_pix = &pix_fmt_info[dst_pix_fmt];
1484    src_pix = &pix_fmt_info[src_pix_fmt];
1485
1486    if (src_pix_fmt == dst_pix_fmt)  // no conversion needed: just copy
1487    {
1488        img_copy(dst, src, dst_pix_fmt, dst_width, dst_height);
1489        return 0;
1490    }
1491
1492    ce = &convert_table[src_pix_fmt][dst_pix_fmt];
1493    if (ce->convert)
1494    {
1495        ce->convert(dst, src, dst_width, dst_height); // specific conversion routine
1496        return 0;
1497    }
1498
1499    if (is_yuv_planar(dst_pix) && src_pix_fmt == PIX_FMT_GRAY8) // gray to YUV
1500    {
1501        int w, h, y;
1502        uint8_t *d;
1503
1504        if (dst_pix->color_type == FF_COLOR_YUV_JPEG)
1505        {
1506            ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1507                              dst_width, dst_height);
1508        }
1509        else
1510        {
1511            img_apply_table(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1512                            dst_width, dst_height, y_jpeg_to_ccir);
1513        }
1514
1515        w = dst_width;     // fill U and V with 128
1516        h = dst_height;
1517        w >>= dst_pix->x_chroma_shift;
```

```
1518        h >>= dst_pix->y_chroma_shift;
1519        for (i = 1; i <= 2; i++)
1520        {
1521            d = dst->data[i];
1522            for (y = 0; y < h; y++)
1523            {
1524                memset(d, 128, w);
1525                d += dst->linesize[i];
1526            }
1527        }
1528        return 0;
1529    }
1530
1531    if (is_yuv_planar(src_pix) && dst_pix_fmt == PIX_FMT_GRAY8)  // YUV to gray
1532    {
1533        if (src_pix->color_type == FF_COLOR_YUV_JPEG)
1534        {
1535            ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1536                              dst_width, dst_height);
1537        }
1538        else
1539        {
1540            img_apply_table(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1541                            dst_width, dst_height, y_ccir_to_jpeg);
1542        }
1543        return 0;
1544    }
1545
1546    if (is_yuv_planar(dst_pix) && is_yuv_planar(src_pix))   // YUV to YUV planar
1547    {
1548        int x_shift, y_shift, w, h, xy_shift;
1549        void(*resize_func)(uint8_t *dst, int dst_wrap, const uint8_t *src, int src_wrap,
1550                                         int width, int height);
1551
1552        // compute chroma size of the smallest dimensions
1553        w = dst_width;
1554        h = dst_height;
1555        if (dst_pix->x_chroma_shift >= src_pix->x_chroma_shift)
1556            w >>= dst_pix->x_chroma_shift;
1557        else
1558            w >>= src_pix->x_chroma_shift;
```

```
1559         if (dst_pix->y_chroma_shift >= src_pix->y_chroma_shift)
1560             h >>= dst_pix->y_chroma_shift;
1561         else
1562             h >>= src_pix->y_chroma_shift;
1563 
1564         x_shift = (dst_pix->x_chroma_shift - src_pix->x_chroma_shift);
1565         y_shift = (dst_pix->y_chroma_shift - src_pix->y_chroma_shift);
1566         xy_shift = ((x_shift &0xf) << 4) | (y_shift &0xf);
1567 
1568        // there must be filters for conversion at least from and to YUV444 format
1569         switch (xy_shift)
1570         {
1571        case 0x00:
1572            resize_func = ff_img_copy_plane;
1573            break;
1574        case 0x10:
1575            resize_func = shrink21;
1576            break;
1577        case 0x20:
1578            resize_func = shrink41;
1579            break;
1580        case 0x01:
1581            resize_func = shrink12;
1582            break;
1583        case 0x11:
1584            resize_func = ff_shrink22;
1585            break;
1586        case 0x22:
1587            resize_func = ff_shrink44;
1588            break;
1589        case 0xf0:
1590            resize_func = grow21;
1591            break;
1592        case 0xe0:
1593            resize_func = grow41;
1594            break;
1595        case 0xff:
1596            resize_func = grow22;
1597            break;
1598        case 0xee:
1599            resize_func = grow44;
```

```
1600            break;
1601        case 0xf1:
1602            resize_func = conv411;
1603            break;
1604        default:
1605            goto no_chroma_filter; // currently not handled
1606         }
1607
1608         ff_img_copy_plane(dst->data[0], dst->linesize[0], src->data[0], src->linesize[0],
1609                           dst_width, dst_height);
1610
1611         for (i = 1; i <= 2; i++)
1612             resize_func(dst->data[i], dst->linesize[i], src->data[i], src->linesize[i],
1613               dst_width >> dst_pix->x_chroma_shift, dst_height >> dst_pix->y_chroma_shift);
1614
1615        // if yuv color space conversion is needed, we do it here on the destination image
1616         if (dst_pix->color_type != src_pix->color_type)
1617         {
1618             const uint8_t *y_table,  *c_table;
1619             if (dst_pix->color_type == FF_COLOR_YUV)
1620             {
1621                 y_table = y_jpeg_to_ccir;
1622                 c_table = c_jpeg_to_ccir;
1623             }
1624             else
1625             {
1626                 y_table = y_ccir_to_jpeg;
1627                 c_table = c_ccir_to_jpeg;
1628             }
1629
1630             img_apply_table(dst->data[0], dst->linesize[0], dst->data[0], dst->linesize[0],
1631                             dst_width, dst_height, y_table);
1632
1633             for (i = 1; i <= 2; i++)
1634                 img_apply_table(dst->data[i], dst->linesize[i], dst->data[i], dst->linesize[i],
1635                   dst_width >> dst_pix->x_chroma_shift, dst_height >> dst_pix->y_chroma_shift, c_table);
1636         }
1637         return 0;
1638     }
1639
1640 no_chroma_filter:  // try to use an intermediate format
```

```
1641
1642    if (src_pix_fmt == PIX_FMT_YUV422 || dst_pix_fmt == PIX_FMT_YUV422)
1643    {
1644        int_pix_fmt = PIX_FMT_YUV422P; // specific case: convert to YUV422P first
1645    }
1646    else if (src_pix_fmt == PIX_FMT_UYVY422 || dst_pix_fmt == PIX_FMT_UYVY422)
1647    {
1648
1649        int_pix_fmt = PIX_FMT_YUV422P; // specific case: convert to YUV422P first
1650    }
1651    else if (src_pix_fmt == PIX_FMT_UYVY411 || dst_pix_fmt == PIX_FMT_UYVY411)
1652    {
1653
1654        int_pix_fmt = PIX_FMT_YUV411P; // specific case: convert to YUV411P first
1655    }
1656    else if ((src_pix->color_type == FF_COLOR_GRAY && src_pix_fmt != PIX_FMT_GRAY8)
1657         || (dst_pix->color_type == FF_COLOR_GRAY && dst_pix_fmt != PIX_FMT_GRAY8))
1658    {
1659
1660        int_pix_fmt = PIX_FMT_GRAY8;   // gray8 is the normalized format
1661    }
1662    else if ((is_yuv_planar(src_pix) && src_pix_fmt != PIX_FMT_YUV444P
1663                                     && src_pix_fmt != PIX_FMT_YUVJ444P))
1664    {
1665        if (src_pix->color_type == FF_COLOR_YUV_JPEG) // yuv444 is the normalized format
1666            int_pix_fmt = PIX_FMT_YUVJ444P;
1667        else
1668            int_pix_fmt = PIX_FMT_YUV444P;
1669    }
1670    else if ((is_yuv_planar(dst_pix) && dst_pix_fmt != PIX_FMT_YUV444P
1671                                     && dst_pix_fmt != PIX_FMT_YUVJ444P))
1672    {
1673        if (dst_pix->color_type == FF_COLOR_YUV_JPEG) // yuv444 is the normalized format
1674            int_pix_fmt = PIX_FMT_YUVJ444P;
1675        else
1676            int_pix_fmt = PIX_FMT_YUV444P;
1677    }
1678    else                                  // the two formats are rgb or gray8 or yuv[j]444p
1679    {
1680        if (src_pix->is_alpha && dst_pix->is_alpha)
1681            int_pix_fmt = PIX_FMT_RGBA32;
```

```
1682         else
1683             int_pix_fmt = PIX_FMT_RGB24;
1684     }
1685
1686     if (avpicture_alloc(tmp, int_pix_fmt, dst_width, dst_height) < 0)
1687         return  - 1;
1688
1689     ret =  - 1;
1690
1691     if (img_convert(tmp, int_pix_fmt, src, src_pix_fmt, src_width, src_height) < 0)
1692         goto fail1;
1693
1694     if (img_convert(dst, dst_pix_fmt, tmp, int_pix_fmt, dst_width, dst_height) < 0)
1695         goto fail1;
1696     ret = 0;
1697
1698 fail1:
1699    avpicture_free(tmp);
1700    return ret;
1701 }
1702
1703 #undef FIX
```

4.9 msrle.c 文件

4.9.1 功能描述

此文件实现微软行程长度压缩算法解码器，此文件请各位参考压缩算法自己仔细分析。

4.9.2 文件注释

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #include "../libavutil/common.h"
6   #include "avcodec.h"
7   #include "dsputil.h"
8
9   #define FF_BUFFER_HINTS_VALID    0x01 // Buffer hints value is meaningful (if 0 ignore)
10  #define FF_BUFFER_HINTS_READABLE 0x02 // Codec will read from buffer
11  #define FF_BUFFER_HINTS_PRESERVE 0x04 // User must not alter buffer content
12  #define FF_BUFFER_HINTS_REUSABLE 0x08 // Codec will reuse the buffer (update)
13
```

此文件请各位参考压缩算法自己仔细分析。

```
14  typedef struct MsrleContext
15  {
16      AVCodecContext *avctx;
17      AVFrame frame;
18
19      unsigned char *buf;
20      int size;
21
22  } MsrleContext;
23
24  #define FETCH_NEXT_STREAM_BYTE() \
25      if (stream_ptr >= s->size) \
26      { \
27          return; \
28      } \
29      stream_byte = s->buf[stream_ptr++];
30
31  static void msrle_decode_pal4(MsrleContext *s)
32  {
33      int stream_ptr = 0;
```

```
34      unsigned char rle_code;
35      unsigned char extra_byte, odd_pixel;
36      unsigned char stream_byte;
37      int pixel_ptr = 0;
38      int row_dec = s->frame.linesize[0];
39      int row_ptr = (s->avctx->height - 1) *row_dec;
40      int frame_size  = row_dec * s->avctx->height;
41      int i;
42
43      // make the palette available
44      memcpy(s->frame.data[1], s->avctx->palctrl->palette, AVPALETTE_SIZE);
45      if (s->avctx->palctrl->palette_changed)
46      {
47  //      s->frame.palette_has_changed = 1;
48          s->avctx->palctrl->palette_changed = 0;
49      }
50
51      while (row_ptr >= 0)
52      {
53          FETCH_NEXT_STREAM_BYTE();
54          rle_code = stream_byte;
55          if (rle_code == 0)
56          {
57              // fetch the next byte to see how to handle escape code
58              FETCH_NEXT_STREAM_BYTE();
59              if (stream_byte == 0)
60              {
61                  // line is done, goto the next one
62                  row_ptr -= row_dec;
63                  pixel_ptr = 0;
64              }
65              else if (stream_byte == 1)
66              {
67                  // decode is done
68                  return ;
69              }
70              else if (stream_byte == 2)
71              {
72                  // reposition frame decode coordinates
73                  FETCH_NEXT_STREAM_BYTE();
74                  pixel_ptr += stream_byte;
```

```
75                  FETCH_NEXT_STREAM_BYTE();
76                  row_ptr -= stream_byte * row_dec;
77              }
78              else
79              {
80                  // copy pixels from encoded stream
81                  odd_pixel = stream_byte &1;
82                  rle_code = (stream_byte + 1) / 2;
83                  extra_byte = rle_code &0x01;
84                  if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
85                  {
86                      return ;
87                  }
88
89                  for (i = 0; i < rle_code; i++)
90                  {
91                      if (pixel_ptr >= s->avctx->width)
92                          break;
93                      FETCH_NEXT_STREAM_BYTE();
94                      s->frame.data[0][row_ptr + pixel_ptr] = stream_byte >> 4;
95                      pixel_ptr++;
96                      if (i + 1 == rle_code && odd_pixel)
97                          break;
98                      if (pixel_ptr >= s->avctx->width)
99                          break;
100                     s->frame.data[0][row_ptr + pixel_ptr] = stream_byte &0x0F;
101                     pixel_ptr++;
102                 }
103
104                 // if the RLE code is odd, skip a byte in the stream
105                 if (extra_byte)
106                     stream_ptr++;
107             }
108         }
109         else
110         {
111             // decode a run of data
112             if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
113             {
114                 return ;
115             }
```

```
116             FETCH_NEXT_STREAM_BYTE();
117             for (i = 0; i < rle_code; i++)
118             {
119                 if (pixel_ptr >= s->avctx->width)
120                     break;
121                 if ((i &1) == 0)
122                     s->frame.data[0][row_ptr + pixel_ptr] = stream_byte >> 4;
123                 else
124                     s->frame.data[0][row_ptr + pixel_ptr] = stream_byte &0x0F;
125                 pixel_ptr++;
126             }
127         }
128     }
129
130     // one last sanity check on the way out
131     if (stream_ptr < s->size)
132     {
133         // error
134     }
135 }
136
137 static void msrle_decode_pal8(MsrleContext *s)
138 {
139     int stream_ptr = 0;
140     unsigned char rle_code;
141     unsigned char extra_byte;
142     unsigned char stream_byte;
143     int pixel_ptr = 0;
144     int row_dec = s->frame.linesize[0];
145     int row_ptr = (s->avctx->height - 1) *row_dec;
146     int frame_size = row_dec * s->avctx->height;
147
148     // make the palette available
149     memcpy(s->frame.data[1], s->avctx->palctrl->palette, AVPALETTE_SIZE);
150     if (s->avctx->palctrl->palette_changed)
151     {
152 //      s->frame.palette_has_changed = 1;
153         s->avctx->palctrl->palette_changed = 0;
154     }
155
156     while (row_ptr >= 0)
```

```
157     {
158         FETCH_NEXT_STREAM_BYTE();
159         rle_code = stream_byte;
160         if (rle_code == 0)
161         {
162             // fetch the next byte to see how to handle escape code
163             FETCH_NEXT_STREAM_BYTE();
164             if (stream_byte == 0)
165             {
166                 // line is done, goto the next one
167                 row_ptr -= row_dec;
168                 pixel_ptr = 0;
169             }
170             else if (stream_byte == 1)
171             {
172                 // decode is done
173                 return ;
174             }
175             else if (stream_byte == 2)
176             {
177                 // reposition frame decode coordinates
178                 FETCH_NEXT_STREAM_BYTE();
179                 pixel_ptr += stream_byte;
180                 FETCH_NEXT_STREAM_BYTE();
181                 row_ptr -= stream_byte * row_dec;
182             }
183             else
184             {
185                 // copy pixels from encoded stream
186                 if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
187                 {
188                     return ;
189                 }
190 
191                 rle_code = stream_byte;
192                 extra_byte = stream_byte &0x01;
193                 if (stream_ptr + rle_code + extra_byte > s->size)
194                 {
195                     return ;
196                 }
197 
```

```
198                 while (rle_code--)
199                 {
200                     FETCH_NEXT_STREAM_BYTE();
201                     s->frame.data[0][row_ptr + pixel_ptr] = stream_byte;
202                     pixel_ptr++;
203                 }
204 
205                 // if the RLE code is odd, skip a byte in the stream
206                 if (extra_byte)
207                     stream_ptr++;
208             }
209         }
210         else
211         {
212             // decode a run of data
213             if ((row_ptr + pixel_ptr + stream_byte > frame_size) || (row_ptr < 0))
214             {
215                 return ;
216             }
217 
218             FETCH_NEXT_STREAM_BYTE();
219 
220             while (rle_code--)
221             {
222                 s->frame.data[0][row_ptr + pixel_ptr] = stream_byte;
223                 pixel_ptr++;
224             }
225         }
226     }
227 
228     // one last sanity check on the way out
229     if (stream_ptr < s->size)
230     {
231         // error
232     }
233 }
234 
235 static int msrle_decode_init(AVCodecContext *avctx)
236 {
237     MsrleContext *s = (MsrleContext*)avctx->priv_data;
238 
```

```
239     s->avctx = avctx;
240
241     avctx->pix_fmt = PIX_FMT_PAL8;
242
243     s->frame.data[0] = NULL;
244
245     return 0;
246 }
247
248 static int msrle_decode_frame(AVCodecContext *avctx, void *data, int *data_size, uint8_t *buf, int buf_size)
249 {
250     MsrleContext *s = (MsrleContext*)avctx->priv_data;
251
252     s->buf = buf;
253     s->size = buf_size;
254
255     if (avctx->reget_buffer(avctx, &s->frame))
256         return -1;
257
258     switch (avctx->bits_per_sample)
259     {
260     case 8:
261         msrle_decode_pal8(s);
262         break;
263     case 4:
264         msrle_decode_pal4(s);
265         break;
266     default:
267         break;
268     }
269
270     *data_size = sizeof(AVFrame);
271     *(AVFrame*)data = s->frame;
272
273     // report that the buffer was completely consumed
274     return buf_size;
275 }
276
277 static int msrle_decode_end(AVCodecContext *avctx)
278 {
279     MsrleContext *s = (MsrleContext*)avctx->priv_data;
```

```
280
281     // release the last frame
282     if (s->frame.data[0])
283         avctx->release_buffer(avctx, &s->frame);
284
285     return 0;
286 }
287
288 AVCodec msrle_decoder =
289 {
290     "msrle",
291     CODEC_TYPE_VIDEO,
292     CODEC_ID_MSRLE,
293     sizeof(MsrleContext),
294     msrle_decode_init,
295     NULL,
296     msrle_decode_end,
297     msrle_decode_frame
298 };
```

4.10 turespeech_data.h 文件

4.10.1 功能描述

此文件定义 true speed 音频解码器使用的常数，此文件请各位参考 TrueSpeed 压缩算法自己仔细分析。

4.10.2 文件注释

```
1   #ifndef __TRUESPEECH_DATA__
2   #define __TRUESPEECH_DATA__
3
4   #pragma warning(disable:4305 )
5
```

此文件请各位参考 TrueSpeed 压缩算法自己仔细分析。

```
6   /* codebooks fo expanding input filter */
7   static const int16_t ts_cb_0[32] =
8   {
9       0x8240, 0x8364, 0x84CE, 0x865D, 0x8805, 0x89DE, 0x8BD7, 0x8DF4,
10      0x9051, 0x92E2, 0x95DE, 0x990F, 0x9C81, 0xA079, 0xA54C, 0xAAD2,
11      0xB18A, 0xB90A, 0xC124, 0xC9CC, 0xD339, 0xDDD3, 0xE9D6, 0xF893,
12      0x096F, 0x1ACA, 0x29EC, 0x381F, 0x45F9, 0x546A, 0x63C3, 0x73B5,
13  };
14
15  static const int16_t ts_cb_1[32] =
16  {
17      0x9F65, 0xB56B, 0xC583, 0xD371, 0xE018, 0xEBB4, 0xF61C, 0xFF59,
18      0x085B, 0x1106, 0x1952, 0x214A, 0x28C9, 0x2FF8, 0x36E6, 0x3D92,
19      0x43DF, 0x49BB, 0x4F46, 0x5467, 0x5930, 0x5DA3, 0x61EC, 0x65F9,
20      0x69D4, 0x6D5A, 0x709E, 0x73AD, 0x766B, 0x78F0, 0x7B5A, 0x7DA5,
21  };
22
23  static const int16_t ts_cb_2[16] =
24  {
25      0x96F8, 0xA3B4, 0xAF45, 0xBA53, 0xC4B1, 0xCECC, 0xD86F, 0xE21E,
26      0xEBF3, 0xF640, 0x00F7, 0x0C20, 0x1881, 0x269A, 0x376B, 0x4D60,
27  };
28
29  static const int16_t ts_cb_3[16] =
30  {
31      0xC654, 0xDEF2, 0xEFAA, 0xFD94, 0x096A, 0x143F, 0x1E7B, 0x282C,
32      0x3176, 0x3A89, 0x439F, 0x4CA2, 0x557F, 0x5E50, 0x6718, 0x6F8D,
33  };
```

```
34
35  static const int16_t ts_cb_4[16] =
36  {
37      0xABE7, 0xBBA8, 0xC81C, 0xD326, 0xDD0E, 0xE5D4, 0xEE22, 0xF618,
38      0xFE28, 0x064F, 0x0EB7, 0x17B8, 0x21AA, 0x2D8B, 0x3BA2, 0x4DF9,
39  };
40
41  static const int16_t ts_cb_5[8] = { 0xD51B, 0xF12E, 0x042E, 0x13C7, 0x2260, 0x311B, 0x40DE, 0x5385,};
42
43  static const int16_t ts_cb_6[8] = { 0xB550, 0xC825, 0xD980, 0xE997, 0xF883, 0x0752, 0x1811, 0x2E18,};
44
45  static const int16_t ts_cb_7[8] = { 0xCEF0, 0xE4F9, 0xF6BB, 0x0646, 0x14F5, 0x23FF, 0x356F, 0x4A8D,};
46
47  static const int16_t *ts_codebook[8] = {ts_cb_0, ts_cb_1, ts_cb_2, ts_cb_3,
48                                          ts_cb_4, ts_cb_5, ts_cb_6, ts_cb_7};
49  /* table used for decoding pulse positions */
50  static const int16_t ts_140[120] =
51  {
52      0x0E46, 0x0CCC, 0x0B6D, 0x0A28, 0x08FC, 0x07E8, 0x06EB, 0x0604,
53      0x0532, 0x0474, 0x03C9, 0x0330, 0x02A8, 0x0230, 0x01C7, 0x016C,
54      0x011E, 0x00DC, 0x00A5, 0x0078, 0x0054, 0x0038, 0x0023, 0x0014,
55      0x000A, 0x0004, 0x0001, 0x0000, 0x0000, 0x0000,
56
57      0x0196, 0x017A, 0x015F, 0x0145, 0x012C, 0x0114, 0x00FD, 0x00E7,
58      0x00D2, 0x00BE, 0x00AB, 0x0099, 0x0088, 0x0078, 0x0069, 0x005B,
59      0x004E, 0x0042, 0x0037, 0x002D, 0x0024, 0x001C, 0x0015, 0x000F,
60      0x000A, 0x0006, 0x0003, 0x0001, 0x0000, 0x0000,
61
62      0x001D, 0x001C, 0x001B, 0x001A, 0x0019, 0x0018, 0x0017, 0x0016,
63      0x0015, 0x0014, 0x0013, 0x0012, 0x0011, 0x0010, 0x000F, 0x000E,
64      0x000D, 0x000C, 0x000B, 0x000A, 0x0009, 0x0008, 0x0007, 0x0006,
65      0x0005, 0x0004, 0x0003, 0x0002, 0x0001, 0x0000,
66
67      0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001,
68      0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001,
69      0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001,
70      0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
71  };
72
73  /* filter for correlated input filter */
74  static const int16_t ts_230[8] = { 0x7F3B, 0x7E78, 0x7DB6, 0x7CF5, 0x7C35, 0x7B76, 0x7AB8, 0x79FC };
```

```
75
76  /* two-point filters table */
77  static const int16_t ts_240[25 * 2] =
78  {
79      0xED2F, 0x5239,
80      0x54F1, 0xE4A9,
81      0x2620, 0xEE3E,
82      0x09D6, 0x2C40,
83      0xEFB5, 0x2BE0,
84
85      0x3FE1, 0x3339,
86      0x442F, 0xE6FE,
87      0x4458, 0xF9DF,
88      0xF231, 0x43DB,
89      0x3DB0, 0xF705,
90
91      0x4F7B, 0xFEFB,
92      0x26AD, 0x0CDC,
93      0x33C2, 0x0739,
94      0x12BE, 0x43A2,
95      0x1BDF, 0x1F3E,
96
97      0x0211, 0x0796,
98      0x2AEB, 0x163F,
99      0x050D, 0x3A38,
100     0x0D1E, 0x0D78,
101     0x150F, 0x3346,
102
103     0x38A4, 0x0B7D,
104     0x2D5D, 0x1FDF,
105     0x19B7, 0x2822,
106     0x0D99, 0x1F12,
107     0x194C, 0x0CE6
108 };
109
110 /* possible pulse values */
111 static const int16_t ts_562[64] =
112 {
113     0x0002, 0x0006, 0xFFFE, 0xFFFA,
114     0x0004, 0x000C, 0xFFFC, 0xFFF4,
115     0x0006, 0x0012, 0xFFFA, 0xFFEE,
```

```
116     0x000A, 0x001E, 0xFFF6, 0xFFE2,
117     0x0010, 0x0030, 0xFFF0, 0xFFD0,
118     0x0019, 0x004B, 0xFFE7, 0xFFB5,
119     0x0028, 0x0078, 0xFFD8, 0xFF88,
120     0x0040, 0x00C0, 0xFFC0, 0xFF40,
121     0x0065, 0x012F, 0xFF9B, 0xFED1,
122     0x00A1, 0x01E3, 0xFF5F, 0xFE1D,
123     0x0100, 0x0300, 0xFF00, 0xFD00,
124     0x0196, 0x04C2, 0xFE6A, 0xFB3E,
125     0x0285, 0x078F, 0xFD7B, 0xF871,
126     0x0400, 0x0C00, 0xFC00, 0xF400,
127     0x0659, 0x130B, 0xF9A7, 0xECF5,
128     0x0A14, 0x1E3C, 0xF5EC, 0xE1C4
129 };
130
131 /* filters used in final output calculations */
132 static const int16_t ts_5E2[8] = { 0x4666, 0x26B8, 0x154C, 0x0BB6, 0x0671, 0x038B, 0x01F3, 0x0112 };
133
134 static const int16_t ts_5F2[8] = { 0x6000, 0x4800, 0x3600, 0x2880, 0x1E60, 0x16C8, 0x1116, 0x0CD1 };
135
136 #endif
```

4.11 turespeech.c 文件

4.11.1 功能描述

此文件实现 true speed 音频解码器，此文件请各位参考压缩算法自己仔细分析。

4.11.2 文件注释

```
1   #include "avcodec.h"
2
3   #include "truespeech_data.h"
4
5   // TrueSpeech decoder context
6
```

此文件请各位参考 TrueSpeed 压缩算法自己仔细分析。

```
7   typedef struct TSContext
8   {
9       // input data
10      int16_t vector[8];   // input vector: 5/5/4/4/4/3/3/3
11      int offset1[2];      // 8-bit value, used in one copying offset
12      int offset2[4];      // 7-bit value, encodes offsets for copying and for two-point filter
13      int pulseoff[4];     // 4-bit offset of pulse values block
14      int pulsepos[4];     // 27-bit variable, encodes 7 pulse positions
15      int pulseval[4];     // 7x2-bit pulse values
16      int flag;            // 1-bit flag, shows how to choose filters
17      // temporary data
18      int filtbuf[146];    // some big vector used for storing filters
19      int prevfilt[8];     // filter from previous frame
20      int16_t tmp1[8];     // coefficients for adding to out
21      int16_t tmp2[8];     // coefficients for adding to out
22      int16_t tmp3[8];     // coefficients for adding to out
23      int16_t cvector[8];  // correlated input vector
24      int filtval;         // gain value for one function
25      int16_t newvec[60];  // tmp vector
26      int16_t filters[32]; // filters for every subframe
27  } TSContext;
28
29  #if !defined(LE_32)
30  #define LE_32(x)  ((((uint8_t*)(x))[3] << 24)| (((uint8_t*)(x))[2] << 16) |  \
31                     (((uint8_t*)(x))[1] << 8) | ((uint8_t*)(x))[0])
32  #endif
33
```

```
34  static int truespeech_decode_init(AVCodecContext *avctx)
35  {
36      return 0;
37  }
38
39  static void truespeech_read_frame(TSContext *dec, uint8_t *input)
40  {
41      uint32_t t;
42
43      t = LE_32(input); // first dword
44      input += 4;
45
46      dec->flag = t &1;
47
48      dec->vector[0] = ts_codebook[0][(t >> 1) &0x1F];
49      dec->vector[1] = ts_codebook[1][(t >> 6) &0x1F];
50      dec->vector[2] = ts_codebook[2][(t >> 11) &0xF];
51      dec->vector[3] = ts_codebook[3][(t >> 15) &0xF];
52      dec->vector[4] = ts_codebook[4][(t >> 19) &0xF];
53      dec->vector[5] = ts_codebook[5][(t >> 23) &0x7];
54      dec->vector[6] = ts_codebook[6][(t >> 26) &0x7];
55      dec->vector[7] = ts_codebook[7][(t >> 29) &0x7];
56
57
58      t = LE_32(input); // second dword
59      input += 4;
60
61      dec->offset2[0] = (t >> 0) &0x7F;
62      dec->offset2[1] = (t >> 7) &0x7F;
63      dec->offset2[2] = (t >> 14) &0x7F;
64      dec->offset2[3] = (t >> 21) &0x7F;
65
66      dec->offset1[0] = ((t >> 28) &0xF) << 4;
67
68
69      t = LE_32(input); // third dword
70      input += 4;
71
72      dec->pulseval[0] = (t >> 0) &0x3FFF;
73      dec->pulseval[1] = (t >> 14) &0x3FFF;
74
```

```
75      dec->offset1[1] = (t >> 28) &0x0F;
76
77
78      t = LE_32(input); // fourth dword
79      input += 4;
80
81      dec->pulseval[2] = (t >> 0) &0x3FFF;
82      dec->pulseval[3] = (t >> 14) &0x3FFF;
83
84      dec->offset1[1] |= ((t >> 28) &0x0F) << 4;
85
86
87      t = LE_32(input); // fifth dword
88      input += 4;
89
90      dec->pulsepos[0] = (t >> 4) &0x7FFFFFF;
91
92      dec->pulseoff[0] = (t >> 0) &0xF;
93
94      dec->offset1[0] |= (t >> 31) &1;
95
96
97      t = LE_32(input); // sixth dword
98      input += 4;
99
100     dec->pulsepos[1] = (t >> 4) &0x7FFFFFF;
101
102     dec->pulseoff[1] = (t >> 0) &0xF;
103
104     dec->offset1[0] |= ((t >> 31) &1) << 1;
105
106
107     t = LE_32(input); // seventh dword
108     input += 4;
109
110     dec->pulsepos[2] = (t >> 4) &0x7FFFFFF;
111
112     dec->pulseoff[2] = (t >> 0) &0xF;
113
114     dec->offset1[0] |= ((t >> 31) &1) << 2;
115
```

```
116
117     t = LE_32(input); // eighth dword
118     input += 4;
119
120     dec->pulsepos[3] = (t >> 4) &0x7FFFFFF;
121
122     dec->pulseoff[3] = (t >> 0) &0xF;
123
124     dec->offset1[0] |= ((t >> 31) &1) << 3;
125 }
126
127 static void truespeech_correlate_filter(TSContext *dec)
128 {
129     int16_t tmp[8];
130     int i, j;
131
132     for (i = 0; i < 8; i++)
133     {
134         if (i > 0)
135         {
136             memcpy(tmp, dec->cvector, i *2);
137             for (j = 0; j < i; j++)
138                 dec->cvector[j] =((tmp[i-j-1]*dec->vector[i])+(dec->cvector[j]<< 15)+0x4000)>>15;
139         }
140         dec->cvector[i] = (8-dec->vector[i]) >> 3;
141     }
142
143     for (i = 0; i < 8; i++)
144         dec->cvector[i] = (dec->cvector[i] *ts_230[i]) >> 15;
145
146     dec->filtval = dec->vector[0];
147 }
148
149 static void truespeech_filters_merge(TSContext *dec)
150 {
151     int i;
152
153     if (!dec->flag)
154     {
155         for (i = 0; i < 8; i++)
156         {
```

```
157             dec->filters[i + 0] = dec->prevfilt[i];
158             dec->filters[i + 8] = dec->prevfilt[i];
159         }
160     }
161     else
162     {
163         for (i = 0; i < 8; i++)
164         {
165              dec->filters[i + 0] = (dec->cvector[i] *21846+dec->prevfilt[i] *10923+16384) >>
15;
166              dec->filters[i + 8] = (dec->cvector[i] *10923+dec->prevfilt[i] *21846+16384) >>
15;
167         }
168     }
169     for (i = 0; i < 8; i++)
170     {
171         dec->filters[i + 16] = dec->cvector[i];
172         dec->filters[i + 24] = dec->cvector[i];
173     }
174 }
175
176 static void truespeech_apply_twopoint_filter(TSContext *dec, int quart)
177 {
178     int16_t tmp[146+60],  *ptr0,  *ptr1,  *filter;
179     int i, t, off;
180
181     t = dec->offset2[quart];
182     if (t == 127)
183     {
184         memset(dec->newvec, 0, 60 *2);
185         return ;
186     }
187
188     for (i = 0; i < 146; i++)
189         tmp[i] = dec->filtbuf[i];
190
191     off = (t / 25) + dec->offset1[quart >> 1] + 18;
192     ptr0 = tmp + 145-off;
193     ptr1 = tmp + 146;
194     filter = (int16_t*)ts_240 + (t % 25) *2;
195     for (i = 0; i < 60; i++)
```

```
196     {
197         t = (ptr0[0] *filter[0] + ptr0[1] *filter[1] + 0x2000) >> 14;
198         ptr0++;
199         dec->newvec[i] = t;
200         ptr1[i] = t;
201     }
202 }
203
204 static void truespeech_place_pulses(TSContext *dec, int16_t *out, int quart)
205 {
206     int16_t tmp[7];
207     int i, j, t;
208     int16_t *ptr1,  *ptr2;
209     int coef;
210
211     memset(out, 0, 60 *2);
212     for (i = 0; i < 7; i++)
213     {
214         t = dec->pulseval[quart] &3;
215         dec->pulseval[quart] >>= 2;
216         tmp[6-i] = ts_562[dec->pulseoff[quart] *4+t];
217     }
218
219     coef = dec->pulsepos[quart] >> 15;
220     ptr1 = (int16_t*)ts_140 + 30;
221     ptr2 = tmp;
222     for (i = 0, j = 3; (i < 30) && (j > 0); i++)
223     {
224         t =  *ptr1++;
225         if (coef >= t)
226             coef -= t;
227         else
228         {
229             out[i] =  *ptr2++;
230             ptr1 += 30;
231             j--;
232         }
233     }
234     coef = dec->pulsepos[quart] &0x7FFF;
235     ptr1 = (int16_t*)ts_140;
236     for (i = 30, j = 4; (i < 60) && (j > 0); i++)
```

```
237     {
238         t =  *ptr1++;
239         if (coef >= t)
240             coef -= t;
241         else
242         {
243             out[i] =  *ptr2++;
244             ptr1 += 30;
245             j--;
246         }
247     }
248 }
249
250 static void truespeech_update_filters(TSContext *dec, int16_t *out, int quart)
251 {
252     int i;
253
254     for (i = 0; i < 86; i++)
255         dec->filtbuf[i] = dec->filtbuf[i + 60];
256
257     for (i = 0; i < 60; i++)
258     {
259         dec->filtbuf[i + 86] = out[i] + dec->newvec[i] - (dec->newvec[i] >> 3);
260         out[i] += dec->newvec[i];
261     }
262 }
263
264 static void truespeech_synth(TSContext *dec, int16_t *out, int quart)
265 {
266     int i, k;
267     int t[8];
268     int16_t *ptr0,  *ptr1;
269
270     ptr0 = dec->tmp1;
271     ptr1 = dec->filters + quart * 8;
272     for (i = 0; i < 60; i++)
273     {
274         int sum = 0;
275         for (k = 0; k < 8; k++)
276             sum += ptr0[k] *ptr1[k];
277         sum = (sum + (out[i] << 12) + 0x800) >> 12;
```

```
278         out[i] = clip(sum,  - 0x7FFE, 0x7FFE);
279         for (k = 7; k > 0; k--)
280             ptr0[k] = ptr0[k - 1];
281         ptr0[0] = out[i];
282     }
283
284     for (i = 0; i < 8; i++)
285         t[i] = (ts_5E2[i] *ptr1[i]) >> 15;
286
287     ptr0 = dec->tmp2;
288     for (i = 0; i < 60; i++)
289     {
290         int sum = 0;
291         for (k = 0; k < 8; k++)
292             sum += ptr0[k] *t[k];
293         for (k = 7; k > 0; k--)
294             ptr0[k] = ptr0[k - 1];
295         ptr0[0] = out[i];
296         out[i] = ((out[i] << 12) - sum) >> 12;
297     }
298
299     for (i = 0; i < 8; i++)
300         t[i] = (ts_5F2[i] *ptr1[i]) >> 15;
301
302     ptr0 = dec->tmp3;
303     for (i = 0; i < 60; i++)
304     {
305         int sum = out[i] << 12;
306         for (k = 0; k < 8; k++)
307             sum += ptr0[k] *t[k];
308         for (k = 7; k > 0; k--)
309             ptr0[k] = ptr0[k - 1];
310         ptr0[0] = clip((sum + 0x800) >> 12,  - 0x7FFE, 0x7FFE);
311
312         sum = ((ptr0[1]*(dec->filtval - (dec->filtval >> 2))) >> 4) + sum;
313         sum = sum - (sum >> 3);
314         out[i] = clip((sum + 0x800) >> 12,  - 0x7FFE, 0x7FFE);
315     }
316 }
317
318 static void truespeech_save_prevvec(TSContext *c)
```

```
319 {
320     int i;
321 
322     for (i = 0; i < 8; i++)
323         c->prevfilt[i] = c->cvector[i];
324 }
325 
326 int truespeech_decode_frame(AVCodecContext *avctx, void *data, int *data_size, uint8_t *buf, int buf_size)
327 {
328     TSContext *c = avctx->priv_data;
329 
330     int i;
331     short *samples = data;
332     int consumed = 0;
333     int16_t out_buf[240];
334 
335     if (!buf_size)
336         return 0;
337 
338     while (consumed < buf_size)
339     {
340         truespeech_read_frame(c, buf + consumed);
341         consumed += 32;
342 
343         truespeech_correlate_filter(c);
344         truespeech_filters_merge(c);
345 
346         memset(out_buf, 0, 240 *2);
347         for (i = 0; i < 4; i++)
348         {
349             truespeech_apply_twopoint_filter(c, i);
350             truespeech_place_pulses(c, out_buf + i * 60, i);
351             truespeech_update_filters(c, out_buf + i * 60, i);
352             truespeech_synth(c, out_buf + i * 60, i);
353         }
354 
355         truespeech_save_prevvec(c);
356 
357         for (i = 0; i < 240; i++)       // finally output decoded frame
358             *samples++ = out_buf[i];
359 
```

```
360     }
361
362     *data_size = consumed * 15;
363
364     return buf_size;
365 }
366
367 AVCodec truespeech_decoder =
368 {
369     "truespeech",
370      CODEC_TYPE_AUDIO,
371      CODEC_ID_TRUESPEECH,
372      sizeof(TSContext),
373      truespeech_decode_init,
374      NULL,
375      NULL,
376      truespeech_decode_frame,
377 };
```

第五章 ffplay 剖析

5.1 文件列表

文件类型	文件名	大小(bytes)
h	berrno.h	446
c	ffplay.c	17360

5.2 berrno.h 文件

5.2.1 功能描述

简单的错误码定义，用于描述错误类型。此文件来源于 \VC98\Include\ERRNO.H，做了删减。

5.2.2 文件注释

```
1   #ifndef BERRNO_H
2   #define BERRNO_H
3
4   #ifdef ENOENT
5   #undef ENOENT
6   #endif
7   #define ENOENT 2
8
9   #ifdef EINTR
10  #undef EINTR
11  #endif
12  #define EINTR  4
13
14  #ifdef EIO
15  #undef EIO
16  #endif
17  #define EIO    5
18
19  #ifdef EAGAIN
20  #undef EAGAIN
21  #endif
22  #define EAGAIN 11
23
24  #ifdef ENOMEM
25  #undef ENOMEM
26  #endif
27  #define ENOMEM 12
```

```
28
29  #ifdef EINVAL
30  #undef EINVAL
31  #endif
32  #define EINVAL 22
33
34  #ifdef EPIPE
35  #undef EPIPE
36  #endif
37  #define EPIPE  32
38
39  #endif
```

5.3 ffplay.c 文件

5.3.1 功能描述

主控文件，初始化运行环境，把各个数据结构和功能函数有机组织起来，协调数据流和功能函数，响应用户操作，启动并控制程序运行。

5.3.2 文件注释

```
1   #include "./libavformat/avformat.h"
2
3   #if defined(CONFIG_WIN32)
4   #include <sys/types.h>
5   #include <sys/timeb.h>
6   #include <windows.h>
7   #else
8   #include <fcntl.h>
9   #include <sys/time.h>
10  #endif
11
12  #include <time.h>
13
14  #include <math.h>
15  #include <SDL.h>
16  #include <SDL_thread.h>
17
```

SDL 里面定义了 main 函数，所以在这里取消 sdl 中的 main 定义，避免重复定义。

```
18  #ifdef CONFIG_WIN32
19  #undef main // We don't want SDL to override our main()
20  #endif
21
```

导入 SDL 库。

```
22  #pragma comment(lib, "SDL.lib")
23
```

简单的几个常数定义。

```
24  #define FF_QUIT_EVENT    (SDL_USEREVENT + 2)
25
26  #define MAX_VIDEOQ_SIZE (5 * 256 * 1024)
27  #define MAX_AUDIOQ_SIZE (5 * 16 * 1024)
28
```

```
29  #define VIDEO_PICTURE_QUEUE_SIZE 1
30
```

音视频数据包/数据帧队列数据结构定义，几个数据成员一看就明白，不详述。

```
31  typedef struct PacketQueue
32  {
33      AVPacketList *first_pkt,  *last_pkt;
34      int size;
35      int abort_request;
36      SDL_mutex *mutex;
37      SDL_cond *cond;
38  } PacketQueue;
39
```

视频图像数据结构定义，几个数据成员一看就明白，不详述。

```
40  typedef struct VideoPicture
41  {
42      SDL_Overlay *bmp;
43      int width, height; // source height & width
44  } VideoPicture;
45
```

总控数据结构，把其他核心数据结构整合在一起，起一个中转的作用，便于在各个子结构之间跳转。

```
46  typedef struct VideoState
47  {
48      SDL_Thread *parse_tid;   // Demux 解复用线程指针
49      SDL_Thread *video_tid;   // video 解码线程指针
50
51      int abort_request;       // 异常退出请求标记
52
53      AVFormatContext *ic;     // 输入文件格式上下文指针，和 iformat 配套使用
54
55      int audio_stream;        // 音频流索引，表示 AVFormatContext 中 AVStream *streams[]数组索引
56      int video_stream;        // 视频流索引，表示 AVFormatContext 中 AVStream *streams[]数组索引
57
58      AVStream *audio_st;      // 音频流指针
59      AVStream *video_st;      // 视频流指针
60
61      PacketQueue audioq;      // 音频数据帧/数据包队列
62      PacketQueue videoq;      // 视频数据帧/数据包队列
```

```
63
64      VideoPicture pictq[VIDEO_PICTURE_QUEUE_SIZE]; // 解码后视频图像队列数组
65      double frame_last_delay;                      // 视频帧延迟，可简单认为是显示间隔时间
66
67      uint8_t audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE *3) / 2];   // 输出音频缓存
68      unsigned int audio_buf_size;      // 解码后音频数据大小
69      int audio_buf_index;              // 已输出音频数据大小
70      AVPacket audio_pkt;               // 如果一个音频包中有多个帧，用于保存中间状态
71      uint8_t *audio_pkt_data;          // 音频包数据首地址，配合 audio_pkt 保存中间状态
72      int audio_pkt_size;               // 音频包数据大小，配合 audio_pkt 保存中间状态
73
74      SDL_mutex *video_decoder_mutex;  // 视频数据包队列同步操作而定义的互斥量指针
75      SDL_mutex *audio_decoder_mutex;  // 音频数据包队列同步操作而定义的互斥量指针
76
77      char filename[240];   // 媒体文件名
78
79  } VideoState;
80
81  static AVInputFormat *file_iformat;
82  static const char *input_filename;
83  static VideoState *cur_stream;
84
```

SDL 库需要的显示表面。

```
85  static SDL_Surface *screen;
86
```

取得当前时间，以 1/1000000 秒为单位，为便于在各个平台上移植，由宏开关控制编译的代码。

```
87  int64_t av_gettime(void)
88  {
89  #if defined(CONFIG_WINCE)
90      return timeGetTime() *int64_t_C(1000);
91  #elif defined(CONFIG_WIN32)
92      struct _timeb tb;
93      _ftime(&tb);
94      return ((int64_t)tb.time *int64_t_C(1000) + (int64_t)tb.millitm) *int64_t_C(1000);
95  #else
96      struct timeval tv;
97      gettimeofday(&tv, NULL);
98      return (int64_t)tv.tv_sec *1000000+tv.tv_usec;
99  #endif
```

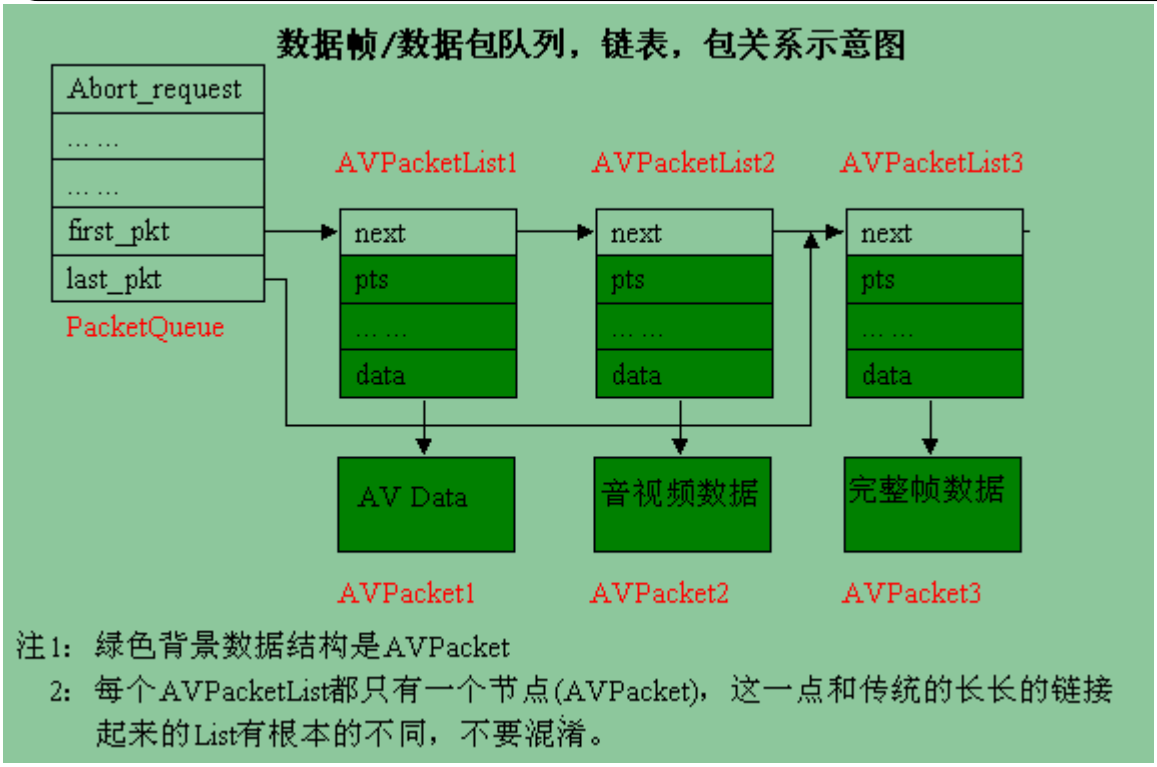
```
100 }
101
```

数据帧/数据包生命周期：

1：在 av_get_packet()函数中调用 av_malloc()函数分配内存，并调用 url_fread()填充媒体数据。

2：如果是视频包调用 packet_queue_put()进 is->videoq 队列，如果是音频包进 is->audioq 队列，如果是其他包，就调用 av_free_packet()函数直接释放内存。

3：进入队列的包，用 packet_queue_get()取出队列，用 av_free_packet()释放内存。



初始化队列，初始化为 0 后再创建线程同步使用的互斥和条件。

```
102 static void packet_queue_init(PacketQueue *q) // packet queue handling
103 {
104     memset(q, 0, sizeof(PacketQueue));
105     q->mutex = SDL_CreateMutex();
106     q->cond = SDL_CreateCond();
107 }
108
```

刷新队列，释放掉队列中所有动态分配的内存，包括音视频裸数据占用的内存和 AVPacketList 结构占用的内存，参考上面示意图。

```
109 static void packet_queue_flush(PacketQueue *q)
110 {
```

```
111     AVPacketList *pkt,  *pkt1;
112
```

由于是多线程程序，需要同步，所以在遍历队列释放所有动态分配内存前，加锁。

```
113     SDL_LockMutex(q->mutex);
114     for (pkt = q->first_pkt; pkt != NULL; pkt = pkt1)
115     {
116         pkt1 = pkt->next;
117         av_free_packet(&pkt->pkt); // 释放音视频数据内存
118         av_freep(&pkt);            // 释放 AVPacketList 结构
119     }
120     q->last_pkt = NULL;
121     q->first_pkt = NULL;
122     q->size = 0;
123     SDL_UnlockMutex(q->mutex);
124 }
125
```

释放队列占用所有资源，首先释放掉所有动态分配的内存，接着释放申请的互斥量和条件量。

```
126 static void packet_queue_end(PacketQueue *q)
127 {
128     packet_queue_flush(q);
129     SDL_DestroyMutex(q->mutex);
130     SDL_DestroyCond(q->cond);
131 }
132
```

往音视频队列中挂接音视频数据帧/数据包。

```
133 static int packet_queue_put(PacketQueue *q, AVPacket *pkt)
134 {
135     AVPacketList *pkt1;
136
```

先分配一个 AVPacketList 结构内存，接着，140 行从 AVPacket 浅复制数据，141 行链表尾置空。

```
137     pkt1 = av_malloc(sizeof(AVPacketList));
138     if (!pkt1)
139         return  - 1;
140     pkt1->pkt =  *pkt;
141     pkt1->next = NULL;
142
143     SDL_LockMutex(q->mutex);
```

144

往队列中挂接 AVPacketList，并且在 150 行统计缓存的媒体数据大小。

```
145     if (!q->last_pkt)
146         q->first_pkt = pkt1;
147     else
148         q->last_pkt->next = pkt1;
149     q->last_pkt = pkt1;
150     q->size  += pkt1->pkt.size;
151
```

设置条件量为有信号状态，如果解码线程因等待而睡眠就及时唤醒。

```
152     SDL_CondSignal(q->cond);
153
154     SDL_UnlockMutex(q->mutex);
155     return 0;
156 }
157
```

设置 异常请求退出 状态。

```
158 static void packet_queue_abort(PacketQueue *q)
159 {
160     SDL_LockMutex(q->mutex);
161
162     q->abort_request = 1; // 请求异常退出
163
164     SDL_CondSignal(q->cond);
165
166     SDL_UnlockMutex(q->mutex);
167 }
168
```

从队列中取出一帧/包数据。

```
169 /* return < 0 if aborted, 0 if no packet and > 0 if packet.  */
170 static int packet_queue_get(PacketQueue *q, AVPacket *pkt, int block)
171 {
172     AVPacketList *pkt1;
173     int ret;
174
175     SDL_LockMutex(q->mutex);
176
```

```
177     for (;;)
178     {
```

如果异常请求退出标记置位，就带错误码返回。

```
179         if (q->abort_request)
180         {
181             ret =  - 1;
182             break;
183         }
184 
185         pkt1 = q->first_pkt;
186         if (pkt1)
187         {
```

如果队列中有数据，就取第一个数据包，在 191 行修正缓存的媒体大小，在 192 行浅复制帧/包数据

```
188             q->first_pkt = pkt1->next;
189             if (!q->first_pkt)
190                 q->last_pkt = NULL;
191             q->size -= pkt1->pkt.size;
192             *pkt = pkt1->pkt;
```

释放掉 AVPacketList 结构，此结构在 packet_queue_put()函数中动态分配(137 行代码处)。

```
193             av_free(pkt1);
194             ret = 1;
195             break;
196         }
197         else if (!block)
198         {
```

如果是非阻塞模式，没数据就直接返回 0。

```
199             ret = 0;
200             break;
201         }
202         else
203         {
```

如果是阻塞模式，没数据就进入睡眠状态等待，packet_queue_put()中唤醒(152 行代码处)。

```
204             SDL_CondWait(q->cond, q->mutex);
205         }
206     }
207     SDL_UnlockMutex(q->mutex);
```

```
208     return ret;
209 }
210
```

分配 SDL 库需要的 Overlay 显示表面，并设置长宽属性。

```
211 static void alloc_picture(void *opaque)
212 {
213     VideoState *is = opaque;
214     VideoPicture *vp;
215
216     vp = &is->pictq[0];
217
218     if (vp->bmp)
219         SDL_FreeYUVOverlay(vp->bmp);
220
221     vp->bmp = SDL_CreateYUVOverlay(is->video_st->actx->width,
222                                    is->video_st->actx->height,
223                                    SDL_YV12_OVERLAY,
224                                    screen);
225
226     vp->width = is->video_st->actx->width;
227     vp->height = is->video_st->actx->height;
228 }
229
```

解码后的视频图像在等待显示间隔时间后，做颜色空间转换，调用 SDL 库显示。简单认为 cpu 耗在前面读文件，解复用，解码的时间为 0，做简单的同步处理逻辑。

```
230 static int queue_picture(VideoState *is, AVFrame *src_frame, double pts)
231 {
232     VideoPicture *vp;
233     int dst_pix_fmt;
234     AVPicture pict;
235
236     if (is->videoq.abort_request)
237         return  - 1;
238
239     vp = &is->pictq[0];
240
241     /* if the frame is not skipped, then display it */
242     if (vp->bmp)
243     {
```

```
244         SDL_Rect rect;
245
```

等待显示间隔时间，调用 Sleep() 函数简单实现。

```
246         if (pts)
247             Sleep((int)(is->frame_last_delay *1000));
248
249         /* get a pointer on the bitmap */
250         SDL_LockYUVOverlay(vp->bmp);
251
```

设置显示图像的属性。

```
252         dst_pix_fmt = PIX_FMT_YUV420P;
253         pict.data[0] = vp->bmp->pixels[0];
254         pict.data[1] = vp->bmp->pixels[2];
255         pict.data[2] = vp->bmp->pixels[1];
256
257         pict.linesize[0] = vp->bmp->pitches[0];
258         pict.linesize[1] = vp->bmp->pitches[2];
259         pict.linesize[2] = vp->bmp->pitches[1];
260
```

把解码后的颜色空间转换为显示颜色空间。

```
261         img_convert(&pict,
262                     dst_pix_fmt,
263                     (AVPicture*)src_frame,
264                     is->video_st->actx->pix_fmt,
265                     is->video_st->actx->width,
266                     is->video_st->actx->height);
267
268         SDL_UnlockYUVOverlay(vp->bmp); /* update the bitmap content */
269
270         rect.x = 0;
271         rect.y = 0;
272         rect.w = is->video_st->actx->width;
273         rect.h = is->video_st->actx->height;
```

实质性显示，刷屏操作。

```
274         SDL_DisplayYUVOverlay(vp->bmp, &rect);
275     }
276     return 0;
```

```
277 }
278
```

视频解码线程，主要功能是分配解码帧缓存和 SDL 显示缓存后进入解码循环(从队列中取数据帧，解码，计算时钟，显示)，释放视频数据帧/数据包缓存。

```
279 static int video_thread(void *arg)
280 {
281     VideoState *is = arg;
282     AVPacket pkt1,  *pkt = &pkt1;
283     int len1, got_picture;
284     double pts = 0;
285
```

分配解码帧缓存

```
286     AVFrame *frame = av_malloc(sizeof(AVFrame));
287     memset(frame, 0, sizeof(AVFrame));
288
```

分配 SDL 显示缓存

```
289     alloc_picture(is);
290
291     for (;;)
292     {
```

从队列中取数据帧/数据包

```
293         if (packet_queue_get(&is->videoq, pkt, 1) < 0)
294             break;
295
```

实质性解码

```
296         SDL_LockMutex(is->video_decoder_mutex);
297         len1 = avcodec_decode_video(is->video_st->actx, frame, &got_picture, pkt->data, pkt->size);
298         SDL_UnlockMutex(is->video_decoder_mutex);
299
```

计算同步时钟

```
300         if (pkt->dts != AV_NOPTS_VALUE)
301             pts = av_q2d(is->video_st->time_base) *pkt->dts;
302
303         if (got_picture)
304         {
```

判断得到图像，调用显示函数同步显示视频图像。

```
305             if (queue_picture(is, frame, pts) < 0)
306                 goto the_end;
307         }
```

释放视频数据帧/数据包内存，此数据包内存是在 av_get_packet()函数中调用 av_malloc()分配的。

```
308         av_free_packet(pkt);
309     }
310
311 the_end:
312     av_free(frame);
313     return 0;
314 }
315
```

解码一个音频帧，返回解压的数据大小。特别注意一个音频包可能包含多个音频帧，但一次只解码一个音频帧，所以一包可能要多次才能解码完。程序首先用 while 语句判断包数据是否全部解完，如果没有就解码当前包中的帧，修改状态参数；否则，释放数据包，再从队列中取，记录初始值，再进循环。

```
316 /* decode one audio frame and returns its uncompressed size */
317 static int audio_decode_frame(VideoState *is, uint8_t *audio_buf, double *pts_ptr)
318 {
319     AVPacket *pkt = &is->audio_pkt;
320     int len1, data_size;
321
322     for (;;)
323     {
324
```

特别注意，一个音频包可能包含多个音频帧，可能需多次解码，VideoState 用一个 AVPacket 型变量保存多次解码的中间状态。如果多次解码但不是最后次解码，audio_decode_frame 直接进 while 循环。

```
325         while (is->audio_pkt_size > 0)
326         {
```

调用解码函数解码，avcodec_decode_audio()函数返回解码用掉的字节数。

```
327             SDL_LockMutex(is->audio_decoder_mutex);
328             len1 = avcodec_decode_audio(is->audio_st->actx, (int16_t*)audio_buf,
329                         &data_size, is->audio_pkt_data, is->audio_pkt_size);
330
331             SDL_UnlockMutex(is->audio_decoder_mutex);
332             if (len1 < 0)
```

```
333             {
334
```

如果发生错误，跳过当前帧，跳出底层循环。

```
335                 is->audio_pkt_size = 0;
336                 break;
337             }
338
```

修正解码后的音频帧缓存首地址和大小。

```
339             is->audio_pkt_data += len1;
340             is->audio_pkt_size -= len1;
341             if (data_size <= 0)
```

如果没有得到解码后的数据，继续解码。可能有些帧第一次解码时只解一个帧头就返回，此时需要继续解码数据帧。

```
342                 continue;
343
```

返回解码后的数据大小。

```
344             return data_size;
345         }
346
```

程序到这里，可能是初始时 audio_pkt 没有赋值；或者一包已经解码完，此时需要释放包数据内存。

```
347         /* free the current packet */
348         if (pkt->data)
349             av_free_packet(pkt);
350
```

读取下一个数据包。

```
351         /* read next packet */
352         if (packet_queue_get(&is->audioq, pkt, 1) < 0)
353             return -1;
354
```

初始化数据包首地址和大小，用于一包中包含多个音频帧需多次解码的情况。

```
355         is->audio_pkt_data = pkt->data;
356         is->audio_pkt_size = pkt->size;
357     }
358 }
```

359

音频输出回调函数，每次音频输出缓存为空时，系统就调用此函数填充音频输出缓存。目前采用比较简单的同步方式，音频按照自己的节拍往前走即可，不需要 synchronize_audio()函数同步处理。

```
360 /* prepare a new audio buffer */
361 void sdl_audio_callback(void *opaque, Uint8 *stream, int len)
362 {
363     VideoState *is = opaque;
364     int audio_size, len1;
365     double pts = 0;
366
367     while (len > 0)
368     {
369         if (is->audio_buf_index >= is->audio_buf_size)
370         {
```

如果解码后的数据已全部输出，就进行音频解码，并在381 行保存解码数据大小，在383 行读索引置0。

```
371             audio_size = audio_decode_frame(is, is->audio_buf, &pts);
372             if (audio_size < 0)
373             {
374                 /* if error, just output silence */
375                 is->audio_buf_size = 1024;
376                 memset(is->audio_buf, 0, is->audio_buf_size);
377             }
378             else
379             {
380 //              audio_size = synchronize_audio(is, (int16_t*)is->audio_buf, audio_size, pts);
381                 is->audio_buf_size = audio_size;
382             }
383             is->audio_buf_index = 0;
384         }
```

385 到 391 行，拷贝适当的数据到输出缓存，并修改解码缓存的参数，进下一轮循环。
特别注意：由进下一轮循环可知，程序应填满 SDL 库给出的输出缓存。

```
385         len1 = is->audio_buf_size - is->audio_buf_index;
386         if (len1 > len)
387             len1 = len;
388         memcpy(stream, (uint8_t*)is->audio_buf + is->audio_buf_index, len1);
389         len -= len1;
390         stream += len1;
391         is->audio_buf_index += len1;
```

```
392     }
393 }
394
```

打开流模块，核心功能是打开相应 codec，启动解码线程(我们把音频回调函数看做一个广义的线程)。

```
395 /* open a given stream. Return 0 if OK */
396 static int stream_component_open(VideoState *is, int stream_index) // 核心功能 open codec
397 {
398     AVFormatContext *ic = is->ic;
399     AVCodecContext *enc;
400     AVCodec *codec;
401     SDL_AudioSpec wanted_spec, spec;
402
403     if (stream_index < 0 || stream_index >= ic->nb_streams)
404         return -1;
405
```

找到从文件格式分析中得到的解码器上下文指针，便于引用其中的参数。

```
406     enc = ic->streams[stream_index]->actx;
407
408     /* prepare audio output */
409     if (enc->codec_type == CODEC_TYPE_AUDIO)
410     {
```

初始化音频输出参数，并调用 SDL_OpenAudio()设置到 SDL 库。

```
411         wanted_spec.freq = enc->sample_rate;
412         wanted_spec.format = AUDIO_S16SYS;
413         /* hack for AC3. XXX: suppress that */
414         if (enc->channels > 2)
415             enc->channels = 2;
416         wanted_spec.channels = enc->channels;
417         wanted_spec.silence = 0;
418         wanted_spec.samples = 1024; //SDL_AUDIO_BUFFER_SIZE;
419         wanted_spec.callback = sdl_audio_callback; // 此处设定回调函数
420         wanted_spec.userdata = is;
421         if (SDL_OpenAudio(&wanted_spec, &spec) < 0)
422         {
```

wanted_spec 是应用程序设定给 SDL 库的音频参数，spec 是 SDL 库返回给应用程序它能支持的音频参数，通常是一致的。如果超过 SDL 支持的参数范围，会返回最相近的参数。

```
423             fprintf(stderr, "SDL_OpenAudio: %s\n", SDL_GetError());
```

```
424             return  - 1;
425         }
426     }
427
```

依照编解码上下文的 codec_id，遍历编解码器链表，找到相应的功能函数。

```
428     codec = avcodec_find_decoder(enc->codec_id);
429
```

核心功能之一,打开编解码器，初始化具体编解码器的运行环境。

```
430     if (!codec || avcodec_open(enc, codec) < 0)
431         return  - 1;
432
433     switch (enc->codec_type)
434     {
435     case CODEC_TYPE_AUDIO:
```

在 VideoState 中记录音频流参数。

```
436         is->audio_stream = stream_index;
437         is->audio_st = ic->streams[stream_index];
438         is->audio_buf_size  = 0;
439         is->audio_buf_index = 0;
440
```

初始化音频队列，并在 443 行启动广义的音频解码线程。

```
441         memset(&is->audio_pkt, 0, sizeof(is->audio_pkt));
442         packet_queue_init(&is->audioq);
443         SDL_PauseAudio(0);
444         break;
445     case CODEC_TYPE_VIDEO:
```

在 VideoState 中记录视频流参数。

```
446         is->video_stream = stream_index;
447         is->video_st = ic->streams[stream_index];
448
449         is->frame_last_delay = is->video_st->frame_last_delay;
450
```

初始化视频队列，并在 452 行直接启动视频解码线程。

```
451         packet_queue_init(&is->videoq);
452         is->video_tid = SDL_CreateThread(video_thread, is);
```

```
453         break;
454     default:
455         break;
456     }
457     return 0;
458 }
459
```

关闭流模块，停止解码线程，释放队列资源。
通过 packet_queue_abort()函数置 abort_request 标志位，解码线程判别此标志位并安全退出线程。

```
460 static void stream_component_close(VideoState *is, int stream_index)
461 {
462     AVFormatContext *ic = is->ic;
463     AVCodecContext *enc;
464
```

简单的流索引参数校验。

```
465     if (stream_index < 0 || stream_index >= ic->nb_streams)
466         return ;
```

找到从文件格式分析中得到的解码器上下文指针，便于引用其中的参数。

```
467     enc = ic->streams[stream_index]->actx;
468
469     switch (enc->codec_type)
470     {
```

停止解码线程，释放队列资源。

```
471     case CODEC_TYPE_AUDIO:
472         packet_queue_abort(&is->audioq);
473         SDL_CloseAudio();
474         packet_queue_end(&is->audioq);
475         break;
476     case CODEC_TYPE_VIDEO:
477         packet_queue_abort(&is->videoq);
478         SDL_WaitThread(is->video_tid, NULL);
479         packet_queue_end(&is->videoq);
480         break;
481     default:
482         break;
483     }
484
```

释放编解码器上下文资源

```
485     avcodec_close(enc);
486 }
487
```

文件解析线程，函数名有点不名副其实。完成三大功能，直接识别文件格式和间接识别媒体格式，打开具体的编解码器并启动解码线程，分离音视频媒体包并挂接到相应队列。

```
488 static int decode_thread(void *arg)
489 {
490     VideoState *is = arg;
491     AVFormatContext *ic;
492     int err, i, ret, video_index, audio_index;
493     AVPacket pkt1,  *pkt = &pkt1;
494     AVFormatParameters params,  *ap = &params;
495
496     int flags = SDL_HWSURFACE | SDL_ASYNCBLIT | SDL_HWACCEL | SDL_RESIZABLE;
497
```

498 到 502 行，初始化基本变量指示没有相应的流。

```
498     video_index =  - 1;
499     audio_index =  - 1;
500
501     is->video_stream =  - 1;
502     is->audio_stream =  - 1;
503
504     memset(ap, 0, sizeof(*ap));
505
```

调用函数直接识别文件格式，在此函数中再调用其他函数间接识别媒体格式。

```
506     err = av_open_input_file(&ic, is->filename, NULL, 0, ap);
507     if (err < 0)
508     {
509         ret =  - 1;
510         goto fail;
511     }
```

保存文件格式上下文，便于各数据结构间跳转。

```
512     is->ic = ic;
513
514     for (i = 0; i < ic->nb_streams; i++)
```

```
515     {
516         AVCodecContext *enc = ic->streams[i]->actx;
517         switch (enc->codec_type)
518         {
```

保存音视频流索引，并把显示视频参数设置到 SDL 库。

```
519         case CODEC_TYPE_AUDIO:
520             if (audio_index < 0)
521                 audio_index = i;
522             break;
523         case CODEC_TYPE_VIDEO:
524             if (video_index < 0)
525                 video_index = i;
526
527             screen = SDL_SetVideoMode(enc->width, enc->height, 0, flags);
528
529             SDL_WM_SetCaption("FFplay", "FFplay"); // 修改是为了适配视频大小
530
531 //          schedule_refresh(is, 40);
532             break;
533         default:
534             break;
535         }
536     }
537
```

如果有音频流，就调用函数打开音频编解码器并启动音频广义解码线程。

```
538     if (audio_index >= 0)
539         stream_component_open(is, audio_index);
540
```

如果有视频流，就调用函数打开视频编解码器并启动视频解码线程。

```
541     if (video_index >= 0)
542         stream_component_open(is, video_index);
543
544     if (is->video_stream < 0 && is->audio_stream < 0)
545     {
```

如果既没有音频流，又没有视频流，就设置错误码返回。

```
546         fprintf(stderr, "%s: could not open codecs\n", is->filename);
547         ret =  - 1;
```

```
548         goto fail;
549     }
550
551     for (;;)
552     {
553         if (is->abort_request)
```

如果异常退出请求置位，就退出文件解析线程。

```
554             break;
555
556         if (is->audioq.size > MAX_AUDIOQ_SIZE || is->videoq.size > MAX_VIDEOQ_SIZE || url_feof(&ic->pb))
557         {
```

如果队列满，就稍微延时一下。

```
558             SDL_Delay(10); // if the queue are full, no need to read more,wait 10 ms
559             continue;
560         }
```

从媒体文件中完整的读取一包音视频数据。

```
561         ret = av_read_packet(ic, pkt); //av_read_frame(ic, pkt);
562         if (ret < 0)
563         {
564             if (url_ferror(&ic->pb) == 0)
565             {
566                 SDL_Delay(100); // wait for user event
567                 continue;
568             }
569             else
570                 break;
571         }
```

判断包数据的类型，分别挂接到相应队列，如果是不识别的类型，就直接释放丢弃掉。

```
572         if (pkt->stream_index == is->audio_stream)
573         {
574             packet_queue_put(&is->audioq, pkt);
575         }
576         else if (pkt->stream_index == is->video_stream)
577         {
578             packet_queue_put(&is->videoq, pkt);
579         }
580         else
```

```
581         {
582             av_free_packet(pkt);
583         }
584     }
585
```

简单的延时，让后面的线程有机会把数据解码显示完。当然丢弃掉最后的一点点数据也可以。

```
586     while (!is->abort_request)   // wait until the end
587     {
588         SDL_Delay(100);
589     }
590
591     ret = 0;
592
```

释放掉在本线程中分配的各种资源，体现了谁申请谁释放的程序自封闭性。

```
593 fail:
594     if (is->audio_stream >= 0)
595         stream_component_close(is, is->audio_stream);
596
597     if (is->video_stream >= 0)
598         stream_component_close(is, is->video_stream);
599
600     if (is->ic)
601     {
602         av_close_input_file(is->ic);
603         is->ic = NULL;
604     }
605
606     if (ret != 0)
607     {
608         SDL_Event event;
609
610         event.type = FF_QUIT_EVENT;
611         event.user.data1 = is;
612         SDL_PushEvent(&event);
613     }
614     return 0;
615 }
616
```

打开流，这个名字也有点名不符实。主要功能是分配全局总控数据结构，初始化相关参数，启动文件解析线程。

```
617 static VideoState *stream_open(const char *filename, AVInputFormat *iformat)
618 {
619     VideoState *is;
620
621     is = av_mallocz(sizeof(VideoState));
622     if (!is)
623         return NULL;
624     pstrcpy(is->filename, sizeof(is->filename), filename);
625
626     is->audio_decoder_mutex = SDL_CreateMutex();
627     is->video_decoder_mutex = SDL_CreateMutex();
628
629     is->parse_tid = SDL_CreateThread(decode_thread, is);
630     if (!is->parse_tid)
631     {
632         av_free(is);
633         return NULL;
634     }
635     return is;
636 }
637
```

关闭流，这个名字也有点名不符实。主要功能是释放资源。

```
639 static void stream_close(VideoState *is)
640 {
641     VideoPicture *vp;
642     int i;
643
644     is->abort_request = 1;
645     SDL_WaitThread(is->parse_tid, NULL);
646
647     for (i = 0; i < VIDEO_PICTURE_QUEUE_SIZE; i++)
648     {
649         vp = &is->pictq[i];
650         if (vp->bmp)
651         {
652             SDL_FreeYUVOverlay(vp->bmp);
653             vp->bmp = NULL;
```

```
654         }
655     }
656
657     SDL_DestroyMutex(is->audio_decoder_mutex);
658     SDL_DestroyMutex(is->video_decoder_mutex);
659 }
660
```

程序退出时调用的函数，关闭释放一些资源。

```
661 void do_exit(void)
662 {
663     if (cur_stream)
664     {
665         stream_close(cur_stream);
666         cur_stream = NULL;
667     }
668
669     SDL_Quit();
670     exit(0);
671 }
672
```

SDL 库的消息事件循环。

```
673 void event_loop(void) // handle an event sent by the GUI
674 {
675     SDL_Event event;
676
677     for (;;)
678     {
679         SDL_WaitEvent(&event);
680         switch (event.type)
681         {
682         case SDL_KEYDOWN:
683             switch (event.key.keysym.sym)
684             {
685             case SDLK_ESCAPE:
686             case SDLK_q:
687                 do_exit();
688                 break;
689             default:
690                 break;
```

```
691             }
692             break;
693         case SDL_QUIT:
694         case FF_QUIT_EVENT:
695             do_exit();
696             break;
697         default:
698             break;
699         }
700     }
701 }
702
```

入口函数，初始化 SDL 库，注册 SDL 消息事件，启动文件解析线程，进入消息循环。

```
703 int main(int argc, char **argv)
704 {
705     int flags = SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER;
706
707     av_register_all();
708
709     input_filename = "d:/yuv/clocktxt_320.avi";
710
711     if (SDL_Init(flags))
712         exit(1);
713
714     SDL_EventState(SDL_ACTIVEEVENT, SDL_IGNORE);
715     SDL_EventState(SDL_MOUSEMOTION, SDL_IGNORE);
716     SDL_EventState(SDL_SYSWMEVENT, SDL_IGNORE);
717     SDL_EventState(SDL_USEREVENT, SDL_IGNORE);
718
719     cur_stream = stream_open(input_filename, file_iformat);
720
721     event_loop();
722
723     return 0;
724 }
725
```