Linux Decoder
开发指南

版本号: 1.4
发布日期: 2025.02.19

版本历史

版本号	日期	制/修订人	内容描述
1.0	2022.06.16	AWA1738	初始版本
1.1	2022.11.15	AWA1738	更新文档模板
1.2	2024.03.14	AWA2082	更新文档内容
1.3	2025.01.10	AWA1729	移除版权要求的视频编码格式
1.4	2025.02.19	AWA2082	复原相关配置说明

目　录

插　图

1 前言

1.1 文档简介

该文档用于介绍由全志所提供的视频解码库统一对外 API 接口，其适用于全志主流芯片系列。目的是为了上层应用开发人员不需要去考虑不同芯片间的差异，而可以直接使用统一的对外视频处理接口。

本文档描述了视频解码库的框架及组成模块，同时对 API 接口的基本使用场景进行了介绍。

1.2 目标读者

基于视频解码库开发和使用的相关人员。

1.3 适用范围

1.3.1 软件平台支持

VE 模块支持在各种版本的 Andorid 平台和 Linux 平台上运行。

1.3.2 硬件平台支持

支持全志主流的各种系列芯片平台：

A 系列，V 系列，T 系列，TV 系列等。

1.4 解码场景软件架构

以下解码场景软件架构以 Android 环境为基础进行描述。

解码场景软件架构

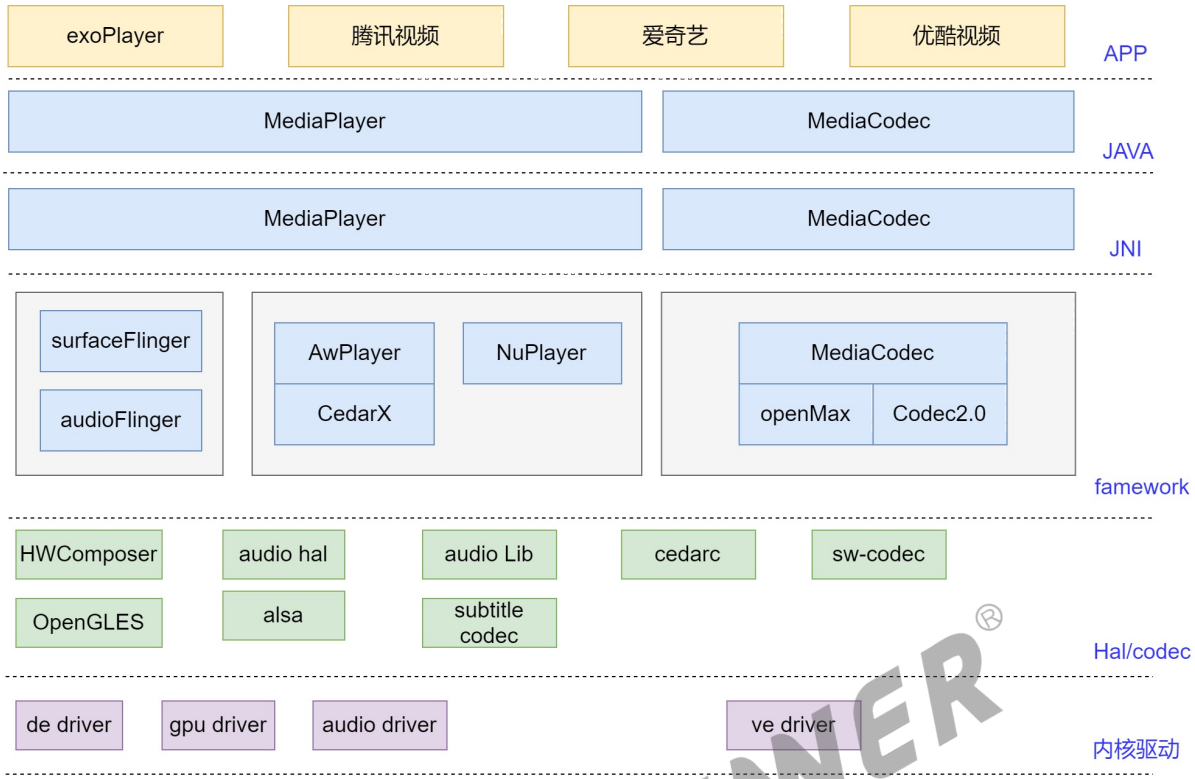


图 1-1: 解码场景软件架构

- App/Java：App 直接调用 android java 层提供的 mediaplayer 或者 mediacodec 接口实现视频播放的功能；
- JNI：由于 framework 层的代码是采用 C++ 编写的，JNI 的作用在于搭建起 JAVA 与 C++ 层的调用桥梁，把 JAVA 的接口转换为对 C++ 层的调用。
- Framework：为功能框架层，调用相关的资源组件（以 hal 或者 codec 的形式提供）实现特定的功能场景；
- Hal/codec：hal 为（Hardware Abstract Layer）硬件抽象层，对底层的驱动进行抽象与封装，屏蔽不同硬件设备之间的差异，为上层软件提供统一的 api 接口；codec 为软件解码模块，采用软件算法实现，主要消耗 cpu 的计算资源；
- Kernel：即为驱动层，通过配置寄存器操作硬件，并对硬件提供的一系列功能特性进行聚合，形成软件 api 供上层软件调用。

1.5 相关术语介绍

1.5.1 软件术语

术语	解释说明
Stream	码流，被压缩编码过的视频数据；
Stream Buffer Manager	负责管理 Stream 的程序模块；
SBM	Stream Buffer Manager；
Frame Buffer	负责管理 Frame Buffer 的程序模块；
FBM	Frame Buffer Manager；
VE	芯片中负责解码视频的硬件模块，也叫 VPU；
Video Engine	通过控制 VE 或通过软件代码解码视频码流的程序模块；
PTS	时间戳，表示图像的显示时间，presentation time stamp 的缩写；
Aspect Ratio	图像中一个像素的宽、高比值，和图像的分辨率一起控制显示时图像的宽高比例。

2 模块介绍

2.1 解码驱动架构

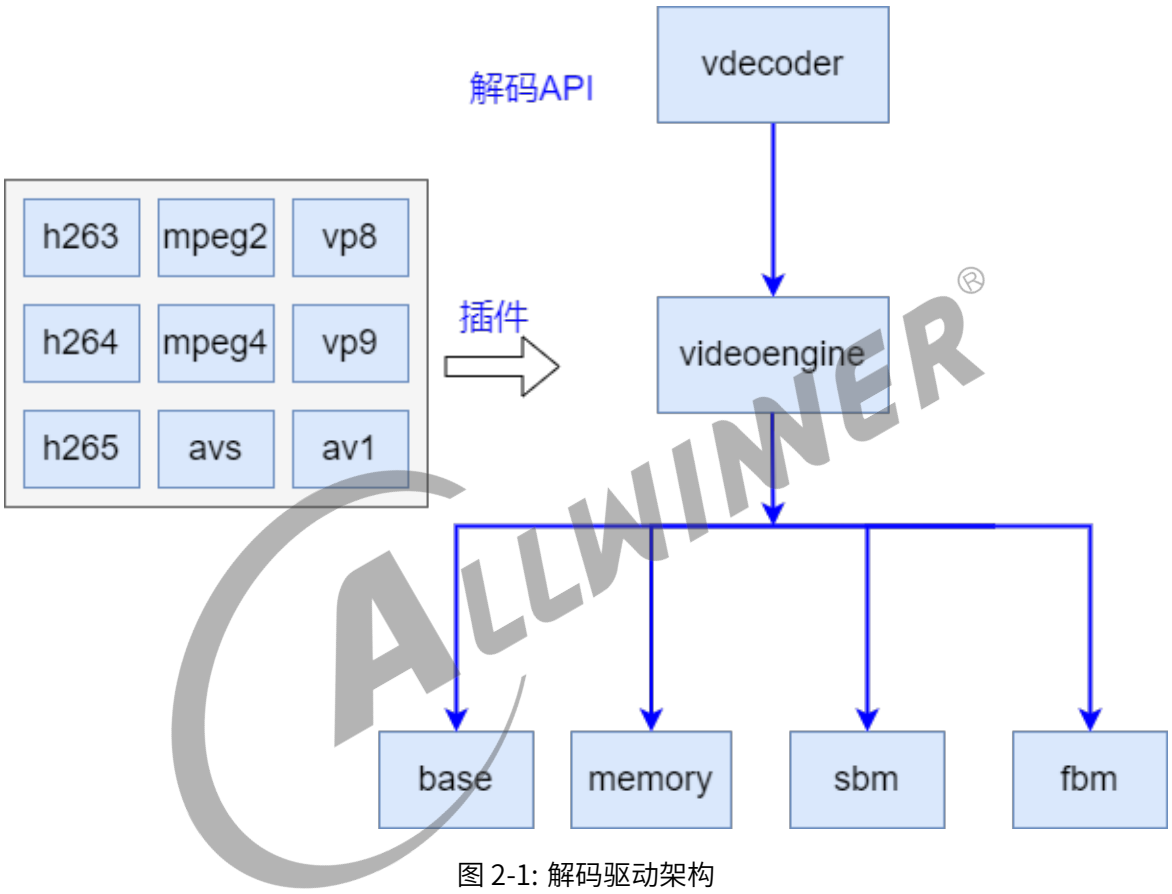


图 2-1: 解码驱动架构

内核层的 ve_driver 是一层影子驱动，只负责初始化与 ve 硬件相关的系统资源如 clk 时钟、响应 ve 中断等，并不对 ve 硬件寄存器进行实质的逻辑处理，而是将 ve 硬件寄存器的操作地址映射到用户态的 cedarc，由 cedarc 实现 ve 驱动的代码处理逻辑。

Cedarc 为 ve 编解码硬件模块的驱动代码工程，包含 ve 解码驱动和 ve 编码驱动，此文仅分析 ve 解码驱动，即 vdecoder。解码驱动 vdecoder 主要由 videoengine、base、memory、hw_adapter、sbm、fbm 组成。

- videoengine 定义一套 sub-vdecoder 接口，所支持的解码器为：H265、H264、MJPEG、MPEG2、MPEG1、MSMPEG4V1、MSMPEG4V2、XVID、H263、SORENSSON_H263、VP6、VP8、VP9、AVS/AVS+。而 h264/h265 等子解码模块则实现对 spec 语法解析、参考帧管理、参数计算、寄存器配置等操作，且以插件的方式对接到 videoengine。

- base 为基础函数功能模式，该模块主要用于实现一些调试手段，如 log 日志，输入码流输出 yuv 保存，内存检测等功能。
- memory 为连续物理内存管理模块，该模块主要用于物理内存的管理，如申请，释放，转换等操作。
- sbm 为输入 buffer 管理模块，用于管理解码前的码流数据，负责视频码流 Buffer 的管理以及码流数据的管理。
- fbm 为输出 buffer 管理模块，用于管理解码后的图像帧数据。

2.1.1 Videoengine 模块

videoengine 模块用于对接具体的子解码器，提供了统一的接口给用户进行使用，可以调用其结构体中提供的函数指针实现视频解码功能，其结构如下所示。

首先对函数指针会用到的输入参数进行解释：

名称	类型	描述
pConfig	VConfig*	初始化解码器时的基本配置信息
pVideoInfo	VideoStreamInfo*	初始化解码器时码流相关的基本信息
pFbmInfo	VideoFbmInfo*	FBM 模块相关信息
pSelf	DecoderInterface*	解码器句柄
pSbm	SbmInterface*	sbm 模块句柄

VideoStreamInfo

- 功能描述：初始化解码器时码流相关的基本信息

属性	类型	描述
eCodecFormat	int	视频源编码格式，取值如下 enumEVIDEOCODECFORMAT{ VIDEO_CODEC_FORMAT_UNKNOWN VIDEO_CODEC_FORMAT_MJPEG VIDEO_CODEC_FORMAT_MPEG1 VIDEO_CODEC_FORMAT_MPEG2 VIDEO_CODEC_FORMAT_MPEG4 VIDEO_CODEC_FORMAT_MSMPEG4V1 VIDEO_CODEC_FORMAT_MSMPEG4V2 VIDEO_CODEC_FORMAT_XVID VIDEO_CODEC_FORMAT_H263 VIDEO_CODEC_FORMAT_SORENSSON_H263 VIDEO_CODEC_FORMAT_RXG2

属性	类型	描述
		VIDEO_CODEC_FORMAT_VP6 VIDEO_CODEC_FORMAT_VP8 VIDEO_CODEC_FORMAT_VP9 VIDEO_CODEC_FORMAT_H264 VIDEO_CODEC_FORMAT_H265 VIDEO_CODEC_FORMAT_AVS };
nWidth	int	视频源宽度
nHeight	int	视频源高度
nFrameRate	int	视频源帧率
nFrameDuration	int	视频帧持续时间
nAspectRatio	int	视频源像素宽高比
bIs3DStream	int	视频源是否为 3D 双流片源
nCodecSpecificDataLen	int	视频源对应的 specialData 的长度
pCodecSpecificData	Char*	视频源对应的 specialData 的 buffer
bSecureStreamFlag	int	视频源是否为加密视频
bSecureStreamFlagLevel1	int	视频源加密等级
bIsFramePackage	int	视频源是否为帧封装格式
h265ReferencePictureNum	int	H265 视频参考帧个数
bReOpenEngine	int	视频数据是否为分辨率切换后的视频
bIsFrameCtsTestFlag	int	视频源是否为 cts 测试片源

VConfig

- 功能描述：初始化解码器时的基本配置信息

属性	类型	描述
bScaleDownEn	int	解码器缩放功能使能标记
bRotationEn	int	解码器旋转功能使能标记
bSecOutputEn	int	MJPEG 解码器第二路输出标记
nHorizonScaleDownRatio	int	视频帧水平方向缩放比例，取值 0,1,2,3
nVerticalScaleDownRatio	int	视频帧竖直方向多方比例，取值 0,1,2,3
nSecHorizonScaleDownRatio	int	第二路输出的视频帧水平方向缩放比例，取值 0,1,2,3
nSecVerticalScaleDownRatio	int	第二路输出的视频帧竖直方向缩放比例，取值 0,1,2,3
nRotateDegree	int	输出视频帧旋转角度，取值 0,1,2,3
bThumbnailMode	int	缩略图模式标记
eOutputPixelFormat	int	输出视频帧 yuv 排列方式，取值如下 PIXEL_FORMAT_DEFAULT

属性	类型	描述
		PIXEL_FORMAT_YUV_PLANER_420 PIXEL_FORMAT_YUV_PLANER_422 PIXEL_FORMAT_YUV_PLANER_444 PIXEL_FORMAT_YV12 PIXEL_FORMAT_NV21 PIXEL_FORMAT_NV12 PIXEL_FORMAT_YUV_MB32_420 PIXEL_FORMAT_YUV_MB32_422 PIXEL_FORMAT_YUV_MB32_444 PIXEL_FORMAT_RGBA PIXEL_FORMAT_ARGB PIXEL_FORMAT_ABGR PIXEL_FORMAT_BGRA PIXEL_FORMAT_YUYV PIXEL_FORMAT_YVYU PIXEL_FORMAT_UYVY PIXEL_FORMAT_VYUY PIXEL_FORMAT_PLANARUV_422 PIXEL_FORMAT_PLANARVU_422 PIXEL_FORMAT_PLANARVU_444
eSecOutputPixelFormat	int	第二路输出视频帧 yuv 排列方式，取值同 eOutputPixelFormat
bNoBFrames	int	视频源没有 B 帧标记
bDisable3D	int	解码器不支持 3D 双流标记
bDispErrorFrame	int	不显示错误帧标记
nVbvBufferSize	int	设置 vbv buffer 的取值
nFrameBufferNum	int	设置 frame buffer 个数
bSecureosEn	int	加密视频使能标记
bGpuBufValid	int	Frame buffer 由 GPU 统一申请的标记
nAlignStride	int	Frame buffer 的对齐值
bIsSoftDecoderFlag	int	是否选择软解的标记
bVirMallocSbm	int	Vbv buffer 是否采用开辟虚拟内存的标记
bSupportPallocBuf-BeforeDecode	int	支持在解码前开辟 frame buffer 的标记
nDeInterlaceHolding-FrameBufferNum	int	离线 interlace 模块占用 frame buffer 个数
nDisplayHoldingFrame-BufferNum	int	显示模块占用 frame buffer 个数
nRotateHoldingFrame-	int	离线旋转模块专用 frame

属性	类型	描述
BufferNum		buffer 个数
nDecodeSmoothFrame-BufferNum	int	解码器保持平滑度占用 frame buffer 个数
bIsTvStream	int	视频片源是电视信号标记
memops	struct ScMemOpsS *	memory 管理器接口
eCtlAfbcMode	eControlAfbcMode	设置 afbc 功能的模式，取值如下： DISABLE_AFBC_ALL_SIZE ENABLE_AFBC_JUST_BIG_SIZE ENABLE_AFBC_ALL_SIZE
eCtlIptvMode	eControlIptvMode	设置 “iptv 场景下获取特定信息” 功能的模式，取值如下： DISABLE_IPTV_ALL_SIZE ENABLE_IPTV_JUST_SMALL_SIZE ENABLE_IPTV_ALL_SIZE
veOpsS	VeOpsS*	Ve 模块的操作句柄
pVeOpsSelf	void*	Ve 模块的内部结构体数据
bConvertVp910bitTo8bit	int	在 vp9 10bit 的码流场景下，是否将 10bit 的图像数据转换成 8bi 的标志位
nVeFreq	unsigned int	设置 Ve 频率值
bCalledByOmxFlag	int	上层调用者是否为 omx 的标志位
bSetProcInfoEnable	int	是否设置 proc 信息的标志位
nSetProcInfoFreq	int	设置 proc 信息的频率，如每隔 1s 设置一次
nChannelNum	int	设置 proc 信息的通道数

DecoderInterface

- 功能描述：解码器句柄

名称	成员类型	描述
Init	函数指针	(Init)(DecoderInterface pSelf, VConfig* pConfig, VideoStreamInfo* pVideoInfo, VideoFbmInfo* pFbmInfo) 视频解码器初始化接口，用于初始化子解码器。
Reset	函数指针	(Reset)(DecoderInterface pSelf); 重置解码器。
SetSbm	函数指针	(SetSbm)(DecoderInterface pSelf, SbmInterface* pSbm, int nIndex); 配置 Sbm 模块。
GetFbmNum	函数指针	(GetFbmNum)(DecoderInterface pSelf);

名称	成员类型	描述
GetFbm	函数指针	获取 Fbm 模块数。 (GetFbm)(DecoderInterface pSelf, int nIndex); 获取当前使用的 Fbm 模块。
Decode	函数指针	(Decode)(DecoderInterface pSelf,int bEndOfStream,int bDecodeKeyFrameOnly,int bSkipBFrameIfDelay,int64_t nCurrentTimeUs); 解码器进行解码。
Destroy	函数指针	(Destroy)(DecoderInterface pSelf); 销毁解码器。
SetSpecialData	函数指针	(SetSpecialData)(DecoderInterface pSelf, void *pArg); 设置特殊参数。
SetExtraScaleInfo	函数指针	(SetExtraScaleInfo)(DecoderInterface pSelf,int nWidthTh,int nHeightTh,int nHorizonScaleRatio,int nVerticalScaleRatio); 设置 VE 的相关缩放参数。
SetPerformCmd	函数指针	(SetPerformCmd)(DecoderInterface pSelf,enum EVDECODERSETPERFORMCMD performCmd); 设置解码器开始或停止统计丢帧信息的命令。
GetPerformInfo	函数指针	(GetPerformInfo)(DecoderInterface pSelf,enum EVDECODERGETPERFORMCMD performCmd, VDecodePerformaceInfo** performInfo; 从解码器中获取丢帧相关的信息

2.1.2 Memory 模块

Memory 模块主要用于进行连续物理内存的管理，同样提供了对应的指针函数给予用户进行调用，其结构如下所示。

名称	成员类型	描述
open	函数指针	int (*open)(void); 创建公共物理内存管理模块。
open2	函数指针	int (open2)(void veops, void* veself); 创建私有物理内存管理模块。
close	函数指针	void (*close)(void); 关闭物理内存管理模块。
total_size	函数指针	int (*total_size)(void); 获取物理内存整体大小。
palloc	函数指针	void (palloc)(int /size/, void , void); 申请 size 大小物理内存，放入 cache 中。

名称	成员类型	描述
palloc_no_cache	函数指针	void (palloc_no_cache)(int /size/, void , void); 申请 size 大小物理内存。
pfree	函数指针	void (pfree)(void /mem/, void , void); 释放对应物理内存。
flush_cache	函数指针	void (flush_cache)(void /mem/, int /size/); 将对应物理内存刷入 cache 中。
ve_get_phyaddr	函数指针	void (ve_get_phyaddr)(void * /viraddr/); 获取 ve 申请的虚拟地址映射的物理地址。
ve_get_viraddr	函数指针	void (ve_get_viraddr)(void * /phyaddr/); 获取 ve 申请的物理地址对应的虚拟地址。
cpu_get_phyaddr	函数指针	void (cpu_get_phyaddr)(void * /viraddr/); 获取 cpu 申请的虚拟地址映射的物理地址。
cpu_get_viraddr	函数指针	void (cpu_get_viraddr)(void * /phyaddr/); 获取 cpu 申请的物理地址对应的虚拟地址。
mem_set	函数指针	int (mem_set)(void /s/, int /c/, size_t /n/); 用于将一块内存区域的内容全部设置为特定的值。
mem_cpy	函数指针	int (mem_cpy)(void /dest/, void * /src/, size_t /n/); 用于在内存之间进行字节级的数据拷贝。
mem_read	函数指针	int (mem_read)(void /dest /, void * /src/, size_t /n/); 用于在内存之间进行字节级的数据拷贝。
mem_write	函数指针	int (mem_write)(void /dest/, void * /src/, size_t /n/); 用于在内存之间进行字节级的数据拷贝。
setup	函数指针	int (*setup)(void);

名称	成员类型	描述
shutdown	函数指针	int (*shutdown)(void);
get_ve_addr_offset	函数指针	unsigned int (*get_ve_addr_offset)(void); 获取 ve 的物理地址偏移值。
get_debug_info	函数指针	int (get_debug_info)(char, int); 获取申请的物理 buffer 的总大小信息。
get_vir_by_fd	函数指针	暂无作用。
get_phy_by_fd	函数指针	int (get_phy_by_fd)(int/dmafd/, void /phy/); 通过 fd 获取对应物理地址。
free_phy_by_fd	函数指针	int (free_phy_by_fd)(int/dmafd/, unsigned long /phy*/); 通过 fd 释放对应物理地址。
get_fd_by_vir	函数指针	int (get_fd_by_vir)(void /viraddr/); 通过虚拟地址获得对应物理地址的 fd。

2.1.3 Sbm 模块

Stream Buffer Manager 负责管理码流 Buffer 和码流数据。初始化时，该模块申请一片物理连续的内存（4MB~12MB 不等），用于存储视频码流。

Stream Buffer Manager 按照帧结构管理码流，模块内部的 frameFifo 结构体记录了每一笔码流的内存地址、长度、时间戳等信息。各种视频编码标准都定义了数据单元的概念，例如 H264 标准定义的 NALU。Stream Buffer Manager 中的一帧码流，是指包含整数个数据单元的一笔数据。

由于需要被硬件解码模块访问，码流 Buffer 是物理连续的，它被当做循环 Buffer 使用。frameFifo 结构体内部使用一个循环数组记录每笔码流信息，按照先进先出的规则向解码器提供码流数据。

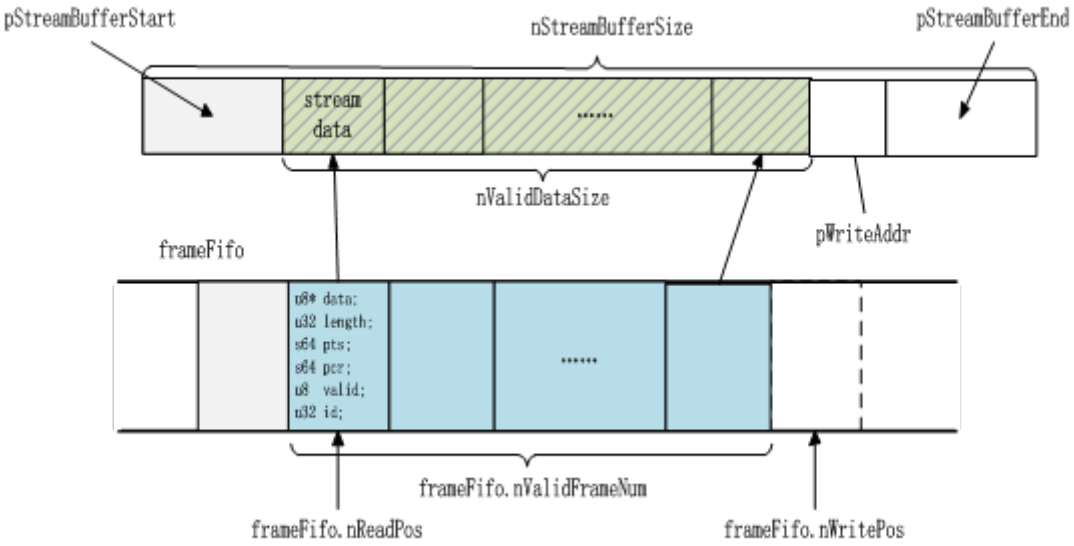


图 2-2: 码流管理模块数据结构设计

在 Sbm 模块中会用到数据结构如下所示:

名称	类型	描述
SbmConfig	struct SbmConfig	初始化 Sbm 时的基本配置信息
pDataInfo	VideoStreamDataInfo*	传到解码器的每笔数据结构信息
SbmInterface	struct SbmInterface	Sbm 模块的句柄

SbmConfig

- 功能描述：初始化 Sbm 时的基本配置信息

属性	类型	描述
bVirFlag	int	虚拟地址申请 sbmbuffer 标志位
nSbmBufferTotalSize	int	sbmbuffer 总 size
memops	struct ScMemOpsS *	memory 模块函数指针结构体
veOpsS	VeOpsS*	Hw_adapter 模块函数指针结构体
pVeOpsSelf	void*	私有数据
bSecureVideoFlag	int	安全模式标志位
nWidth	int	图像像素宽度
nConfigSbmBufferSize	int	sbmbuffersize 配置大小
nNaluLength	int	该笔码流 Nalu 数据长度

VideoStreamDataInfo

- 功能描述：传到解码器的每笔数据结构信息

属性	类型	描述
pData	Char*	传输的一笔视频数据所用的 buffer 指针
nLength	int	传输的一笔视频数据的长度
nPts	int64_t	传输的一笔视频数据所携带的时间戳
nPcr	int64_t	传输的一笔视频数据所携带的 PCR
bIsFirstPart	int	传输的一笔视频数据的第一部分的标记
bIsLastPart	int	传输的一笔视频数据的最后一部分的标记
nID	int	传输的视频数据所携带的 ID 号（目前没用）
nStreamIndex	int	传输的视频数据属于第几路视频的标记，（主流为 0，从流为 1）
bValid	int	视频数据有效的标记
bVideoInfoFlag	Unsigned int	传输的视频帧数据是否携带额外信息的标记
pVideoInfo	int	传输的视频帧数据携带额外信息所在的 buffer 指针

SbmInterface

- 功能描述：Sbm 模块的句柄，提供给用户进行调用

名称	成员类型	描述
init	函数指针	int (init)(SbmInterface pSelf, SbmConfig* pSbmConfig); Sbm 模块初始化。
destroy	函数指针	void (destroy)(SbmInterface pSelf); 销毁对应 Sbm 模块。
reset	函数指针	void (reset)(SbmInterface pSelf); 重置对应 Sbm 模块。
getBufferAddress	函数指针	void* (getBufferAddress)(SbmInterface pSelf); 获取当前 Sbm 模块 buffer 的起始地址。
getBufferSize	函数指针	int (getBufferSize)(SbmInterface pSelf); 获取当前 Sbm 模块 buffer 的 size。
getStreamFrameNum	函数指针	int (getStreamFrameNum)(SbmInterface pSelf);

名称	成员类型	描述
		获取当前 Sbm 模块中未被使用的码流 buffer 个数。
getStreamDataSize	函数指针	int (getStreamDataSize)(SbmInterface pSelf); 获取当前 Sbm 模块中未被使用的 buffersize。
requestBuffer	函数指针	(requestBuffer)(SbmInterface pSelf, int nRequireSize, char** ppBuf, int* pBufSize); 获取当前 Sbm 模块中可用的 buffer。
addStream	函数指针	int (addStream)(SbmInterface pSelf, VideoStreamDataInfo* pDataInfo); 添加一笔码流至 Sbm 模块中。
requestStream	函数指针	VideoStreamDataInfo* (requestStream)(SbmInterface pSelf); 从当前 Sbm 模块中获取一笔码流。
returnStream	函数指针	int (returnStream)(SbmInterface pSelf, VideoStreamDataInfo* pDataInfo); 将当前这笔码流返回至 Sbm 模块中，保持同之前的排序。
flushStream	函数指针	int (flushStream)(SbmInterface pSelf, VideoStreamDataInfo* pDataInfo); 丢弃当前这笔码流。
getBufferWritePointer	函数指针	char* (getBufferWritePointer)(SbmInterface pSelf); 获取下一笔要解码的视频数据信息所在的地址。
getBufferDataInfo	函数指针	void* (getBufferDataInfo)(SbmInterface pSelf); 获取当前 Sbm 模块正在读取的码流 buffer。
setEos	函数指针	int (setEos)(SbmInterface pSelf, int nEosFlag); 配置 Eos 标志，即是否为最后一笔码流。
nType	int	

名称	成员类型	描述
bUseNewVeMemoryProgram	int	码流类型。 暂不使用。
updateProcInfo	函数指针	int (updateProcInfo)(SbmInterface pSelf); 更新 vedebug 调试节点 Sbm 相关信息。

2.1.4 Fbm 模块

Frame Buffer Manager 负责管理图像 Buffer。初始化时，该模块申请指定个数的图像 Buffer，每个图像 Buffer 的信息放在其内部数组 frames[] 中。当解码器需要图像 Buffer 时，Frame Buffer Manager 从空 Buffer 队列取出一个 Buffer 给解码器，解码器完成解码后，图像 Buffer 被放入显示队列 pValidPictureQueue 等待显示。当 Render 需要获取图像用于显示时，Frame Buffer Manager 从 pValidPictureQueue 取出一个图像给 Render，Render 完成显示归还 Buffer 时，如果解码器没有在使用该 Buffer，图像 Buffer 被放入 pEmptyBufferQueue。

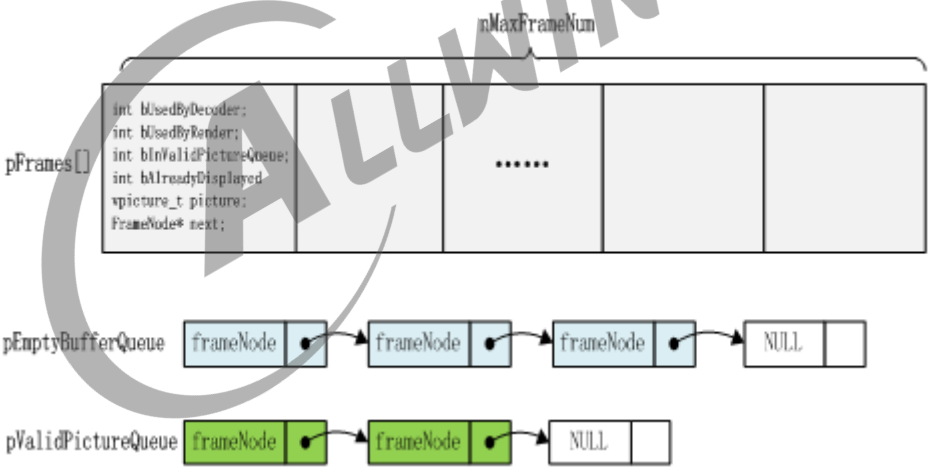


图 2-3: frame_buffer 管理模块数据结构设计

Fbm 模块中使用到的数据结构如下所示。

VideoPicture - 功能描述：解码器解码输出的视频帧的结构体信息

属性	类型	描述
nID	int	视频帧所用 buffer 的 ID 号
nStreamIndex	int	视频帧对应码流的标记号，（主流为 0，从流为 1）
ePixelFormat	int	输出视频帧 yuv 排列方式，取值如下

属性	类型	描述
		PIXEL_FORMAT_DEFAULT
		PIXEL_FORMAT_YUV_PLANER_420
		PIXEL_FORMAT_YUV_PLANER_422
		PIXEL_FORMAT_YUV_PLANER_444
		PIXEL_FORMAT_YV12
		PIXEL_FORMAT_NV21
		PIXEL_FORMAT_NV12
		PIXEL_FORMAT_YUV_MB32_420
		PIXEL_FORMAT_YUV_MB32_422
		PIXEL_FORMAT_YUV_MB32_444
		PIXEL_FORMAT_RGBA
		PIXEL_FORMAT_ARGB
		PIXEL_FORMAT_ABGR
		PIXEL_FORMAT_BGRA
		PIXEL_FORMAT_YUYV
		PIXEL_FORMAT_YVYU
		PIXEL_FORMAT_UYVY
		PIXEL_FORMAT_VYUY
		PIXEL_FORMAT_PLANARUV_422
		PIXEL_FORMAT_PLANARVU_422
		PIXEL_FORMAT_PLANARUV_444
		PIXEL_FORMAT_PLANARVU_444
nWidth	int	视频帧的宽度（做对齐后的宽度）
nHeight	int	视频帧的高度（做对齐后的高度）
nLineStride	int	视频帧宽度对齐要求值
nTopOffset	int	视频帧有效显示区域顶端开始值
nLeftOffset	int	视频帧有效显示区域左侧开始值
nBottomOffset	int	视频帧有效显示区域底端结束值
nRightOffset	int	视频帧有效显示区域右侧结束值
nFrameRate	int	视频帧率
nAspectRatio	int	视频帧宽高像素比
bIsProgressive	int	视频帧 interlace 格式标记
bTopFieldFirst	int	视频帧顶场优先标记
bRepeatTopField	int	视频帧顶场重复显示标记
nPts	int64_t	视频帧所携带的显示时间戳
nPcr	int64_t	视频帧多携带的 pcr
pData0	Char*	视频帧 y buffer 的地址指针
pData1	Char*	视频帧 u buffer 的地址指针
pData2	Char*	视频帧 v buffer 的地址指针
pData3	Char*	视频帧其它 buffer 的地址指针
bMafValid	int	Maf 有效标记

属性	类型	描述
pMafData	Char*	Maf 信息所在的 buffer 地址
bPreFrmValid	int	视频帧前面帧有效标记
nBufId	int	视频帧 Buffer 所携带的 buf id
phyYBufAddr	size_addr	视频帧亮度 buffer 对应的物理地址
phyCBufAddr	size_addr	视频帧色度 buffer 对应的物理地址
pPrivate	void*	视频帧私有信息 buffer 指针
nBufStatus	int	视频帧 buffer 的状态
bTopFieldError	int	视频帧顶场有错的标记
bBottomFieldError	int	视频帧底场有错的标记
nColorPrimary	int	解码器给出的 ColorPrimary 值
bFrameErrorFlag	int	图像是否包含错误数据的标志位
pMetaData	Void*	用于存放 afbc info 和 hdr info 的 buffer
video_full_range_flag	VIDEO_FULL_RANGE_FLAG	与 hdr 相关的参数
transfer_characteristics	VIDEO_TRANSFER	与 hdr 相关的参数
matrix_coeffs	VIDEO_MATRIX_COEFFS	与 hdr 相关的参数
colour_primaries	u8	与 hdr 相关的参数
nLower2BitBufSize	int	低 2bit 的 buffer 大小
nLower2BitBufOffset	int	低 2bit 的 buffer 的偏移值
nLower2BitBufStride	int	低 2bit 的 buffer 线宽值
b10BitPicFlag	int	10bit 图像标志位
bEnableAfbcFlag	int	是否开启 afbc 功能的标志位
nBufSize	int	Buffer size
nAfbcSize	int	Afbc buffer size
nDebugCount	int	用于 debug
nCurFrameInfo	VIDEO_FRM_STATUS_INFO	与码流特性相关的信息

FbmBufInfo - 功能描述：Frame buffer 相关信息

属性	类型	描述
nBufNum	int	开辟 frame buffer 的个数
nBufWidth	int	开辟 frame buffer 的宽度
nBufHeight	int	开辟 frame buffer 的高度
ePixelFormat	int	视频帧的 yuv 排列方式
nAlignValue	int	视频帧 buffer 的宽度对齐方式
bProgressiveFlag	int	视频码流是 progressive 片源的标记
bIsSoftDecoderFlag	int	视频片源格式对应的解码器是否为软解格式
bHdrVideoFlag	int	是否为 hdr 视频的标志位
b10bitVideoFlag	int	是否为 10bit 码流的标志位
bAfbcModeFlag	int	开启 afbc 功能的模式参数

属性	类型	描述
nTopOffset	int	视频帧有效显示区域顶端开始值
nBottomOffset	int	视频帧有效显示区域左侧开始值
nLeftOffset	int	视频帧有效显示区域底端结束值
nRightOffset	int	视频帧有效显示区域右侧结束值

VideoFbmInfo - 功能描述：Fbm 模块相关信息

属性	类型	描述
nValidBufNum	unsigned int	可使用的 Fbm buffer 数
pFbmFirst	void*	第一路 Fbm 模块句柄
pFbmSecond	void*	第二路 Fbm 模块句柄
pFbmBufInfo	FbmBufInfo	Frame buffer 相关信息
bIs3DStream	unsigned int	3D 码流标志位
bTwoStreamShareOneFbm	unsigned int	是否为两笔码流共享一个 fbmbuffer
pMajorDispFrame	VideoPicture*	当前显示图像 buffer
pMajorDecoderFrame	VideoPicture*	当前解码图像 buffer
nMinorYBufOffset	unsigned int	次路 fbmbufferY 分量偏移值
nMinorCBufOffset	unsigned int	次路 fbmbufferC 分量偏移值
bIsFrameCtsTestFlag	int	是否为 cts 测试 flag
nExtraFbmBufferNum	int	除解码必备 buffer 外的额外 buffer 个数
nDecoderNeededMiniFbmNum	int	解码最小需要 Fbm buffer 数
nDecoderNeededMiniFbmNumSD	int	解码最小需要 Fbm buffer 数
bIsSoftDecoderFlag	int	是否为软解标志位

3 接口说明

3.1 Sbm 模块相关接口

由于上文介绍数据结构时，已经将相关接口以函数指针的方式描述过了，接口只是将其在封装了一层，方便用户调用。故在下述阐述接口介绍时，将会结合使用场景对常用的接口进行阐述。

3.1.1 解码模块 API

序号	函数名	说明
NO.1	GetSbmInterface	获取码流对应的 Sbm 句柄
NO.2	SbmInit	初始化对应 Sbm 模块
NO.3	SbmBufferSize	获取对应 Sbmbuffer 所申请内存的总大小
NO.4	SbmStreamDataSize	获取对应 Sbmbuffer 已使用的内存大小
NO.5	SbmRequestBuffer	获取可用的 Streambuffer
NO.6	SbmAddStream	提供一笔码流进入 Sbm 模块中
NO.7	SbmRequestStream	获取一笔可用的码流
NO.8	SbmReturnStream	返还该笔码流
NO.9	SbmFlushStream	丢弃该笔码流
NO.10	SbmDestroy	销毁对应的 Sbm 模块

3.1.1.1 GetSbmInterface

函数原型	SbmInterface* GetSbmInterface(int nType)
功能	获取 Sbm 句柄
参数	nType: 码流类型
返回值	成功：返回句柄；失败：返回 Null；
调用说明	无

3.1.1.2 SbmBufferSize

函数原型	static inline int SbmBufferSize(SbmInterface* pSelf)
功能	获取对应 Sbmbuffer 所申请内存的总大小
参数	pSelf: 对应 Sbm 句柄
返回值	成功：返回 size；失败：返回-1；
调用说明	无

3.1.1.3 SbmInit

函数原型	static inline int SbmInit(SbmInterface* pSelf, SbmConfig* pSbmConfig)
功能	Sbm 模块初始化
参数	pSelf: 对应 Sbm 句柄
	pSbmConfig: Sbm 初始化配置参数
返回值	成功：返回 0；失败：返回-1；
调用说明	无

3.1.1.4 SbmStreamDataSize

函数原型	static inline int SbmStreamDataSize(SbmInterface* pSelf)
功能	获取对应 Sbmbuffer 已使用的内存大小
参数	pSelf: 对应 Sbm 句柄
返回值	成功：返回 size；失败：返回-1；
调用说明	无

3.1.1.5 SbmRequestBuffer

函数原型	static inline int SbmRequestBuffer(SbmInterface* pSelf, int nRequireSize,char** ppBuf, int* pBufSize)
功能	获取可用的 Streambuffer
参数	pSelf: 对应 Sbm 句柄
	nRequireSize：获取 buffer 长度
	ppBuf：获取 buffer 的起始地址
	pBufSize：Sbmbuffer 剩余 size 大小
返回值	成功：返回 0；失败：返回-1；
调用说明	无

3.1.1.6 SbmAddStream

函数原型	static inline int SbmAddStream(SbmInterface* pSelf, VideoStreamDataInfo* pDataInfo)
功能	提供一笔码流进入 Sbm 模块中
参数	pSelf: 对应 Sbm 句柄 pDataInfo: 传到解码器的每笔数据结构信息
返回值	成功：返回 0；失败：返回-1；
调用说明	无

3.1.1.7 SbmRequestStream

函数原型	static inline VideoStreamDataInfo* SbmRequestStream(SbmInterface* pSelf)
功能	获取一笔可用的码流
参数	pSelf: 对应 Sbm 句柄
返回值	成功：返回码流；失败：返回 Null；
调用说明	无

3.1.1.8 SbmReturnStream

函数原型	static inline int SbmReturnStream(SbmInterface* pSelf, VideoStreamDataInfo* pDataInfo)
功能	返还该笔码流
参数	pSelf: 对应 Sbm 句柄 pDataInfo: 传到解码器的每笔数据结构信息
返回值	成功：返回 0；失败：返回-1；
调用说明	无

3.1.1.9 SbmFlushStream

函数原型	static inline int SbmFlushStream(SbmInterface* pSelf, VideoStreamDataInfo* pDataInfo)
功能	丢弃该笔码流
参数	pSelf: 对应 Sbm 句柄

函数原型	static inline int SbmFlushStream(SbmInterface* pSelf, VideoStreamDataInfo* pDataInfo)
	pDataInfo: 传到解码器的每笔数据结构信息
返回值	成功：返回 0；失败：返回-1；
调用说明	无

3.1.1.10 SbmDestroy

函数原型	static inline void SbmDestroy(SbmInterface* pSelf)
功能	销毁对应 Sbm 模块
参数	pSelf: 对应 Sbm 句柄
返回值	无
调用说明	无

3.1.2 使用流程

```
type = SBM_TYPE_FRAME_AVC;
SbmInterface *pSbm = GetSbmInterface(type); //通过调用该接口获取对应的Sbm接口，以H264码流为例；
SbmInit(pSbm, &mSbmConfig); //初始化对应Sbm接口；
while（！文件没结束）
{
nValidSize = SbmBufferSize() - SbmStreamDataSize(); //判断当前Sbm剩余空间。确保有足够空间存放码流；
nRet = SbmRequestBuffer(&buffer); //获取可用的sbmbuffer；
SbmAddStream(&buffer); //即先从Sbm模块中获取可用buffer，再将码流数据写入该buffer中放入Sbm里；
}
SbmDestroy();
```

以上只是简略的 Sbm 模块调用流程，并不代表可以直接使用，若想要参考具体实现代码，可参考 vdecoderDemo.c 文件中 parserThreadFunc 部分实现。

3.2 Fbm 模块相关接口

由于 Fbm 模块的接口基本嵌套于子解码器内部实现，无需客户在外层调用，故只对用户会接触到的接口进行说明。

3.2.1 Fbm 模块 API

序号	函数名	说明
NO.1	FbmNextPictureInfo	获取下一帧输出图像的信息
NO.2	FbmRequestPicture	通过 FbmCreate 函数创建的 Fbm 模块指针；
NO.3	FbmReturnPicture	归还通过 RequestPicture 获取的视频图像给解码器
NO.4	FbmAllocatePictureBuffer	申请一个图像 Buffer。
NO.5	FbmFreePictureBuffer	释放一个由 AllocatePictureBuffer 函数申请的图像 Buffer。
NO.6	FbmReturnPicture	将使用过的图像 buffer 进行返还给解码器。
NO.7	FbmSetFbmBufAddress	将外部申请的 buffer 转化进 Fbm 模块中。

3.2.1.1 FbmNextPictureInfo

函数原型	VideoPicture* FbmNextPictureInfo(Fbm* pFbm)
功能	获取下一帧输出图像的信息
参数	pFbm: 通过 FbmCreate 函数创建的 Fbm 模块指针；
返回值	成功: 返回下一帧待显示图像 Buffer 的指针；失败: 返回 NULL；
调用说明	与 RequestPicture 函数相比，本函数只是获取视频图像的信息，不会使该图像从解码器的显示队列中删除。

3.2.1.2 FbmRequestPicture

函数原型	VideoPicture* FbmRequestPicture(Fbm* pFbm)
功能	获取一帧图像
参数	pFbm: 通过 FbmCreate 函数创建的 Fbm 模块指针；
返回值	成功: 返回图像 Buffer 指针；失败: 返回 NULL；
调用说明	无

3.2.1.3 FbmReturnPicture

函数原型	int FbmReturnPicture(Fbm* pFbm, VideoPicture* pVPicture)
功能	归还通过 RequestPicture 获取的视频图像给解码器
参数	pFbm: 通过 FbmCreate 函数创建的 Fbm 模块指针； pPicture: 通过 RequestPicture 函数获取的图像 Buffer；
返回值	0: 表示成功；-1: 失败；
调用说明	无

3.2.1.4 FbmAllocatePictureBuffer

函数原型	int FbmAllocatePictureBuffer(Fbm* pFbm, VideoPicture* pPicture,int* nAlignValue, int nWidth, int nHeight)
功能	申请一个图像 Buffer。
参数	pFbm: 通过 FbmCreate 函数创建的 Fbm 模块指针； pPicture: 需要进行申请的图像 Buffer 指针； nAlignValue: 图像在内存中存放的行宽对齐值，以像素为单位； nWidth: 图像像素宽度； nHeight: 图像像素高度；
返回值	0: 表示成功；-1: 失败；
调用说明	无

3.2.1.5 FbmFreePictureBuffer

函数原型	int FbmFreePictureBuffer(Fbm* pFbm, VideoPicture* pPicture)
功能	释放一个由 AllocatePictureBuffer 函数申请的图像 Buffer。
参数	pFbm: 通过 FbmCreate 函数创建的 Fbm 模块指针； pPicture: 通过 AllocatePictureBuffer 函数申请的图像 Buffer；
返回值	成功: 返回图像 Buffer 指针；失败: 返回 NULL
调用说明	无

3.2.1.6 FbmReturnPicture

函数原型	int FbmReturnPicture(Fbm* pFbm, VideoPicture* pPicture)
功能	将使用过的图像 buffer 进行返还给解码器。
参数	pFbm: 通过 FbmCreate 函数创建的 Fbm 模块指针； pPicture: 通过 AllocatePictureBuffer 函数申请的图像 Buffer；
返回值	成功: 返回 0；失败: 返回-1
调用说明	无

3.2.1.7 FbmSetFbmBufAddress

函数原型	VideoPicture* FbmSetFbmBufAddress(VideoFbmInfo* pFbmInfo,VideoPicture* pPicture, int bForbidUseFlag)
功能	将外部申请的 buffer 转化进 Fbm 模块中。
参数	pFbmInfo: 通过 FbmCreate 函数创建的 Fbm 模块指针； pFbmInfo: Fbm 模块相关信息； pPicture: 外部申请的 buffer； bForbidUseFlag: 是否禁止解码器使用；
返回值	成功: 返回转化后的 Fbm 模块 buffer；失败: 返回 Null
调用说明	无

3.2.2 Fbm 模块使用场景

对于解码模块在处理数据时，Sbm 模块和 Fbm 模块的内存申请都应为物理内存，这样才能最大程度保证数据处理的稳定性与高效性。Sbm 模块只由解码模块内部使用，故都由内部进行申请。但 Fbm 模块不同，其图像数据并不仅只给解码器内部使用，而是需要提供给用户进行使用，用户再使用完后返还给解码器，再进行参考等操作，再关闭解码器时将所有申请过的内存进行释放。

而在这种使用场景下，若用户使用的图像 buffer 与解码器内部使用的图像 buffer 不是同一块的话，则会造成内存资源浪费及时间上的浪费，因为需要将数据从一块内存拷贝到另外一块内存。

鉴于以上情况，Fbm 模块提供了两种内存申请方式，供用户进行使用。

第一种：解码器内部申请内存方式

图像由解码器内部进行申请，通过调用上述 FbmAllocatePictureBuffer 接口进行图像 buffer 的申请，用户得到解码器输出的图像，在使用完后通过调用 FbmFreePictureBuffer 进行销毁。同时需注意，Fbm 模块中的 buffer 并不是一次性使用的，用户在使用完之后需归还给解码器，即调用 FbmReturnPicture 接口进行返还。

这种方式优点是使用起来简单方便，在单纯测试解码功能或使用 demo 时可以使用该方式；但在复杂的功能场景，即需要将图像进行显示时，则最好不要使用该方式，因为解码器内部申请的 buffer 不一定与用户的显示系统兼容，所以对于零拷贝的实现会十分困难。

第二种：外部申请内存方式

图像由外部模块进行申请，通过调用上述 FbmSetFbmBufAddress 接口将外部申请的图像 buffer 转化进入 Fbm 模块中，这样直接使用外部申请的 buffer，在对于需要进行显示的场景比较适用，在数据的转移上不再需要进行拷贝，而是共享相同的物理地址。

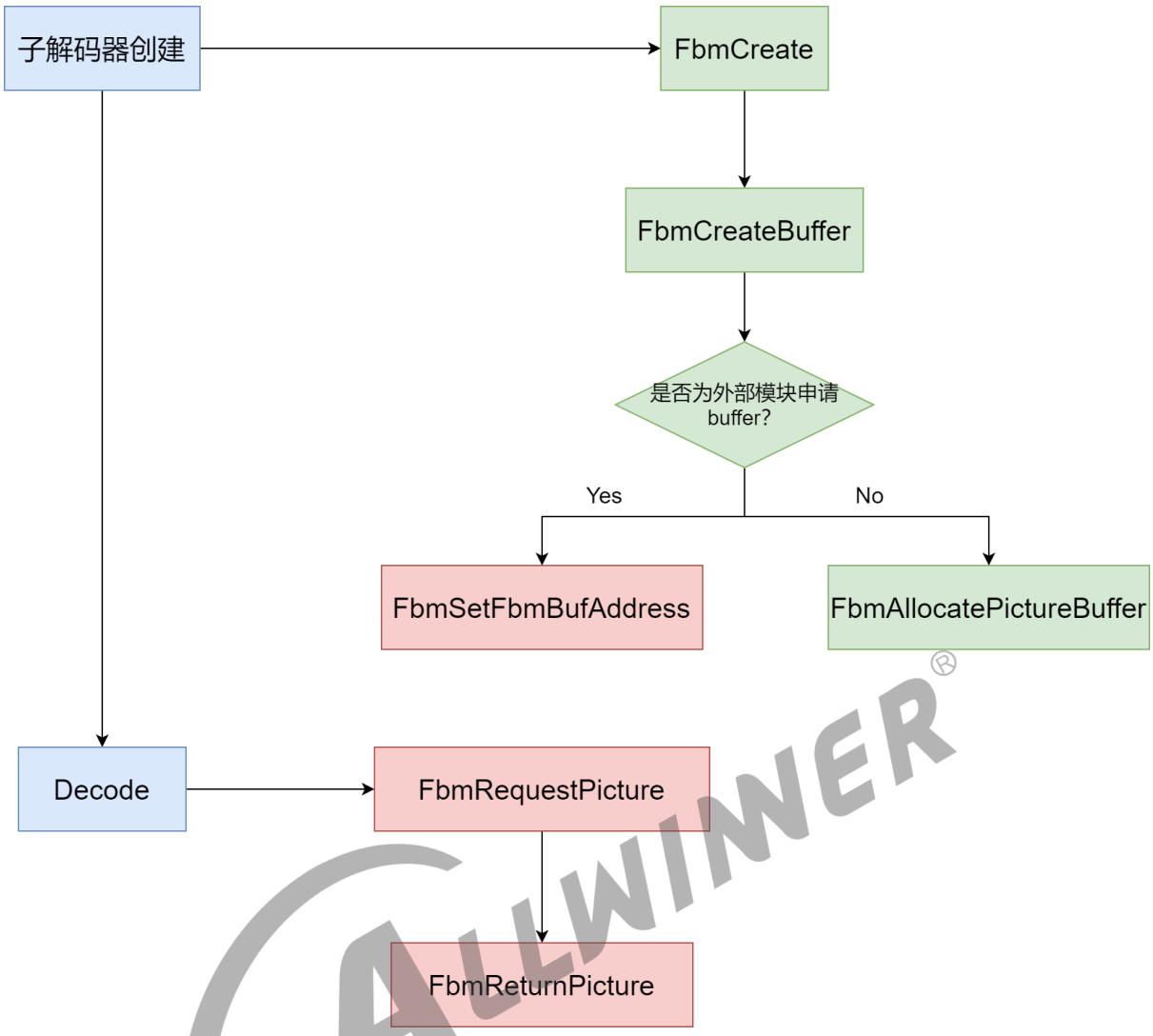


图 3-1: Fbm 模块接口调用流程

如上图所示，实际上大部分有关 Fbm 模块的轮转使用操作都在解码器内部完成了，对于用户来说只需要关心如何将外部模块申请的 buffer 对接上 Fbm 模块 buffer，以及在解码结束后调用接口获取到图像，再使用完之后将图像返还回解码器。

3.3 解码模块相关接口

3.3.1 解码模块使用方式

开始解码前，应用程序首先调用 CreateVideoDecoder 函数创建一个解码器，然后调用 InitializeVideoDecoder 函数，将视频基本信息作为参数，初始化解码器。

初始化后，解码器可以开始解码视频流。

应用程序通过 RequestVideoStreamBuffer 函数从解码器获取码流 Buffer，将数据填入后，通

过 SubmitVideoStreamData 函数将码流提交给解码器。

应用程序通过调用 DecodeVideoStream 函数解码视频码流。

应用程序通过调用 RequestPicture 函数获取视频图像，视频图像显示完毕后，应用程序调用 ReturnPicture 将图像 Buffer 归还解码器。

视频解码库支持多线程操作，码流的传送、解码和视频图像的输出工作可以在不同的线程中进行。一般来说，播放器会有 Demux 线程、视频解码线程和视频渲染（Render）等三个线程处理视频相关的工作。Demux 线程不断调用 RequestVideoStreamBuffer 函数和 SubmitVideoStream-Data 函数传送数据；视频解码线程通过调用 DecodeVideoStream 函数解码视频流；视频渲染线程不断调用 RequestPicture 函数获取图像用于显示，调用 ReturnPicture 归还已经显示的图像。

视频解码库还支持图像缩放、旋转等功能，这些功能需要调用其他 API 进行配置。下文详细介绍各个 API 函数。

3.3.2 解码模块 API

视频解码库的 API 接口如下表所示。

序号	函数名	说明
NO.1	AddVDPlugin	加载所有的子解码库
NO.2	CreateVideoDecoder	创建一个视频解码器
NO.3	DestroyVideoDecoder	销毁视频解码器
NO.4	InitializeVideoDecoder	初始化视频解码器
NO.5	ResetVideoDecoder	重置视频解码器
NO.6	ReopenVideoEngine	在遇到分辨率发生改变的情况时，解码器返回 VDECODE_RESULT_RESOLUTION_CHANGE 通知中间件，对分辨率变化后的码流调用解码之前，需要先调用 ReopenVideoEngine 重新启动解码器
NO.7	DecodeVideoStream	视频码流解码函数
NO.8	RequestVideoStreamBuffer	从视频解码器获取传输视频码流的 buffer
NO.9	SubmitVideoStreamData	将视频码流数据传输到解码器中
NO.10	NextPictureInfo	获取下一个视频解码图像的信息
NO.11	RequestPicture	获取视频解码图像
NO.12	ReturnPicture	还回视频解码图像
NO.13	RotatePicture	对解码后的视频图像调用软件接口进行旋转
NO.14	RotatePictureHw	对解码后的视频图像调用硬件接口进行旋转
NO.15	GetVideoStreamInfo	获取 VideoStreamInfo 的信息
NO.16	VideoStreamBufferSize	获取视频解码器开辟的视频 buffer size
NO.17	VideoStreamDataSize	获取视频解码器中有效的数据 size
NO.18	VideoStreamFrameNum	获取视频解码器中还未解码的数据笔数
NO.19	VideoStreamDataInfoPointer	获取下一笔要解码的视频数据信息所在的地址

序号	函数名	说明
NO.20	TotalPictureBufferNum	获取解码器开辟的图像 buffer 个数
NO.21	EmptyPictureBufferNum	获取解码器空闲的图像 buffer 个数
NO.22	ValidPictureNum	获取解码器已经解完，但还未显示的图像个数
NO.23	AllocatePictureBuffer	申请物理地址连续的视频帧 buffer
NO.24	FreePictureBuffer	释放调用 AllocatePictureBuffer 申请的视频帧 buffer
NO.25	VideoDecoderPallocIonBuf	申请 ion buffer
NO.26	VideoDecoderFreeIonBuf	释放 ion buffer
NO.27	GetVideoFbmBufInfo	获取解码器申请的 fbm buffer 的信息
NO.28	SetVideoFbmBufAddress	将中间件申请的视频图像 buffer 地址设置到解码器
NO.29	SetVideoFbmBufRelease	在新显架构下，nativeWindow 发生变化时，在旧的 nativewindow 中申请的视频帧 buffer 将逐渐被在新 nativewindow 中申请的视频帧 buffer 所替代，通过调用此函数，解码器将原来的 buffer 全部设置为 release 状态
NO.30	RequestReleasePicture	中间件从解码器调用标记为 release 状态的视频帧图像
NO.31	ReturnRelasePicture	还回标记为 release 状态的图像，此图像对应的地址 buffer 已经被重新分配
NO.32	ConfigExtraScaleInfo	若中间件在调用解码器 InitializeVideoDecoder 函数时，无法确定 VE 模块缩放的参数时，可以在调用完 InitializeVideoDecoder 后，在调用 DecodeVideoStream 函数前，通过调用 ConfigExtraScaleInfo 设置 VE 的相关缩放参数
NO.33	DecoderSetSpecialData	设置特殊参数
NO.34	SetDecodePerformCmd	设置解码器开始或停止统计丢帧信息的命令
NO.35	GetDecodePerformInfo	从解码器中获取丢帧相关的信息

3.3.2.1 AddVDPlugin

函数原型	void AddVDPlugin(void)
功能	加载所有的子解码库, 如 libawh264.so, libawh264.so 等。
参数	无
返回值	无
调用说明	上层调用者可自行设计按需加载子解码库的函数，如可设计只加载 libawh264.so 这个子解码库的函数；若上层没有按需加载子解码库的需求，则可以调用此函数。

3.3.2.2 CreateVideoDecoder

函数原型	VideoDecoder* CreateVideoDecoder(void)
功能	创建一个视频解码器
参数	无
返回值	成功：视频解码器指针。失败：返回 NULL。
调用说明	无

3.3.2.3 DestroyVideoDecoder

函数原型	void DestroyVideoDecoder(VideoDecoder* pDecoder)
功能	销毁视频解码器
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针。
返回值	无
调用说明	无

3.3.2.4 InitializeVideoDecoder

函数原型	int InitializeVideoDecoder(VideoDecoder* pDecoder, VideoStreamInfo* pVideoInfo, VConfig* pVconfig)
功能	初始化视频解码器
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针 pVconfig：解码器初始化信息 pVideoInfo：视频片源的相关信息
返回值	成功：返回 0；失败：返回-1，不支持的编码格式或内存资源不足
调用说明	pVconfig：编码器基本初始化信息； 1. nInputWidth：输入图像帧的宽度，以像素为单位； 2. nInputHeight：输入图像帧的高度，以像素为单位； 3. nDstWidth：编码前对输入图像做 scale 后的宽度，以像素为单位；如果不需要做 scale，nDstWidth 的值保持和 nInputWidth 一致； 4. nDstHeight：编码前对输入图像做 scale 后的高度，以像素为单位；如果不需要做 scale，nDstHeight 的值保持和 nInputHeight 一致； 5. eInputFormat：输入的颜色格式； 6. nStride: 输入图像帧在内存中的行宽，以像素为单位，编码器要求 nStride 必须 16 对齐； 7. Memops：编码器内部内存管理的数据结构，该数据结构由调用者初始化，其定义在 memory 模块中，具体请参看 memory 相关文档；

3.3.2.5 ResetVideoDecoder

函数原型	void ResetVideoDecoder(VideoDecoder* pDecoder)
功能	重置视频编码器
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针
返回值	成功：返回 0；失败：返回-1；
调用说明	无

3.3.2.6 ReopenVideoEngine

函数原型	int ReopenVideoEngine (VideoDecoder* pDecoder)
功能	重新打开解码器内 Video Engine 模块
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针；
返回值	成功：返回 0；失败：返回-1；
调用说明	如果视频分辨率发生改变，DecodeVideoStream 函数会返回对应的值 VDECODE_RESULT_RESOLUTION_CHANGE，并将码流归还到码 Buffer；此时应用应该调用本函数重新打开 Video Engine 模块；重新打开 Video Engine 模块会导致图像 Buffer 被释放，图像 Buffer 管理模块重新初始化。这一点显示控制需要注意。

3.3.2.7 DecodeVideoStream

函数原型	int DecodeVideoStream(VideoDecoder* pDecoder, int bEndOfStream,int bDecodeKeyFrameOnly,int bDropBFrameIfDelay, int64_t nCurrentTimeUs)
功能	视频码流解码函数
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； bEndOfStream：最后一笔码流数据，取值为 0/1 bDropBFrameIfDelay：在解码过时的情况下是否允许丢 B 帧 nCurrentTimeUs：当前系统时间，单位 us
返回值	(1) VDECODE_RESULT_UNSUPPORTED = -1, 视频格式或规格不支持 (2) VDECODE_RESULT_OK = 0, 解码器中间状态 (3)VDECODE_RESULT_FRAME_DECODED = 1, 解码器解完一个 P 帧或 B 帧 (4)VDECODE_RESULT_CONTINUE = 2, 解码一帧未完成 (5)VDECODE_RESULT_KEYFRAME_DECODED = 3, 解完一个关键帧（I 帧）

函数原型	int DecodeVideoStream(VideoDecoder* pDecoder, int bEndOfStream,int bDecodeKeyFrameOnly,int bDropBFrameIfDelay, int64_t nCurrentTimeUs)
调用说明	(6)VDECODE_RESULT_NO_FRAME_BUFFER = 4, 解码器申请不到 frame buffer (7)VDECODE_RESULT_NO_BITSTREAM = 5, 解码器中没有可以解码的码流 (8)VDECODE_RESULT_RESOLUTION_CHANGE = 6, 码流的分辨率发生改变 无

3.3.2.8 RequestVideoStreamBuffer

函数原型	int RequestVideoStreamBuffer(VideoDecoder* pDecoder, int nRequireSize, char** ppBuf, int* pBufSize, char** ppRingBuf, int* pRingBufSize, int nStreamBufIndex)
功能	向解码器申请传输视频码流的 buffer；
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nRequireSize: 请求 Buffer 的大小，以字节为单位； ppBuf: 输出参数，码流 Buffer 起始地址，等于 NULL 表示失败； pBufSize: 输出参数，码流 Buffer ppBuf 的大小； ppRingBuf: 输出参数，第二块 Buffer 的起始地址，等于 NULL 表示没有； pRingBufSize: 第二块 Buffer ppRingBuf 的大小； nStreamBufIndex: 对于蓝光 MVC 等 3D 视频，解码器需要处理两路码流，nStreamBufIndex 指定从第几路视频码流 Buffer 获取 Buffer，0 表示第 0 路（MVC 主码流）、1 表示第 1 路（MVC 从码流）。
返回值	成功：返回 0；失败：返回-1；
调用说明	码流 Buffer 是一块循环 Buffer，当 Buffer 回头时，外部请求的 Buffer 被分成两段，ppBuf 和 pBufSize 返回第一段 Buffer 的地址和大小，ppRingBuf 和 pRingBufSize 返回第二段 Buffer 的地址和大小。Buffer 没有回头时，ppRingBuf 和 pRingBufSize 返回 NULL。

3.3.2.9 SubmitVideoStreamData

函数原型	int SubmitVideoStreamData(VideoDecoder* pDecoder, VideoStreamDataInfo* pDataInfo, int nStreamBufIndex)
功能	向解码器提交码流数据
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pDataInfo: 码流数据信息，包括地址、长度、时间戳、边界信息等；
返回值	成功：返回 0；失败：返回-1；

函数原型	int SubmitVideoStreamData(VideoDecoder* pDecoder, VideoStreamDataInfo* pDataInfo, int nStreamBufIndex)
调用说明	提交数据时，数据可以是一笔包含整数个数据单元的完整码流帧，也可以只包含一个数据单元的部分数据，只需将 VideoStreamDataInfo 结构体中的两个表示数据边界信息的变量正确填写即可。两个边界信息变量为 bIsFirstPart: 表示该笔数据第一个字节是否是一个数据单元的开始；bIsLastPart: 表示该笔数据最后一个有效字节是否是一个数据单元的结束；

3.3.2.10 NextPictureInfo

函数原型	VideoPicture* NextPictureInfo(VideoDecoder* pDecoder，int nStreamIndex)
功能	获取下一帧输出图像的信息
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamIndex: 对于蓝光 MVC 等 3D 视频，解码器有两路视频可供显示；nStreamIndex 指定从获取第几路视频的图像。
返回值	成功: 返回下一帧待显示图像 Buffer 的指针；失败: 返回 NULL；
调用说明	与 RequestPicture 函数相比，本函数只是获取视频图像的信息，不会使该图像从解码器的显示队列中删除。

3.3.2.11 RequestPicture

函数原型	VideoPicture* RequestPicture(VideoDecoder* pDecoder,int nStreamIndex)
功能	获取一帧图像
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamIndex: 对于蓝光 MVC 等 3D 视频，解码器有两路视频可供显示，nStreamIndex 指定从获取第几路视频的图像。
返回值	成功: 返回图像 Buffer 指针；失败: 返回 NULL；
调用说明	无

3.3.2.12 ReturnPicture

函数原型	int ReturnPicture(VideoDecoder* pDecoder,VideoPicture* pPicture)
功能	归还通过 RequestPicture 获取的视频图像给解码器
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pPicture: 通过 RequestPicture 函数获取的图像 Buffer；

函数原型	int ReturnPicture(VideoDecoder* pDecoder,VideoPicture* pPicture)
返回值	0: 表示成功；-1: 失败；
调用说明	无

3.3.2.13 RotatePicture

函数原型	int RotatePicture (VideoDecoder* pDecoder，VideoPicture* pPictureIn,VideoPicture* pPictureOut, int nRotateDegree)
功能	把图像 pPictureIn 旋转输出到图像 Buffer pPictureOut
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pPictureIn: 输入图像； pPictureOut: 输出图像； nRotateDegree: 顺时针旋转角度，0 表示不旋转、1 表示旋转 90 度、2 表示 180 度、3 表示 270 度；
返回值	0: 表示成功；-1: 表示失败；
调用说明	除旋转图像外，本函数还支持像素格式的转换，输出像素格式由 pPictureOut 图像的 ePixelFormat 参数指定；目前已经支持的像素格式转换有： (1) PIXEL_FORMAT_YUV_MB32_420 转为 PIXEL_FORMAT_YV12 (2) PIXEL_FORMAT_YUV_MB32_422 转为 PIXEL_FORMAT_YV12 (3) PIXEL_FORMAT_YUV_MB32_420 转为 PIXEL_FORMAT_NV21 (4) PIXEL_FORMAT_YUV_MB32_422 转为 PIXEL_FORMAT_NV21 (5) PIXEL_FORMAT_YV12 转为 PIXEL_FORMAT_NV21 (6) PIXEL_FORMAT_NV21 转为 PIXEL_FORMAT_YV12

3.3.2.14 RotatePictureHw

函数原型	int RotatePictureHw(VideoDecoder* pDecoder, VideoPicture* pPictureIn, VideoPicture* pPictureOut, int nRotateDegree)
功能	采用硬件旋转模，把图像 pPictureIn 旋转输出到图像 Buffer pPictureOut
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pPictureIn: 输入图像； pPictureOut: 输出图像； nRotateDegree: 顺时针旋转角度，0 表示不旋转、1 表示旋转 90 度、2 表示 180 度、3 表示 270 度；
返回值	0: 表示成功；-1: 表示失败；
调用说明	无

3.3.2.15 GetVideoStreamInfo

函数原型	int GetVideoStreamInfo(VideoDecoder* pDecoder, VideoStreamInfo* pVideoInfo);
功能	获取 VideoStreamInfo 结构体的信息
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pVideoInfo：VideoStreamInfo 结构体指针；
返回值	成功：0 失败：-1
调用说明	无

3.3.2.16 VideoStreamBufferSize

函数原型	int VideoStreamBufferSize(VideoDecoder* pDecoder,int nStreamBufIndex)
功能	查询码流 Buffer 的总大小，单位为字节
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamBufIndex: 对于蓝光 MVC 等 3D 视频，解码器需要处理两路码流，nStreamBufIndex 指定从第几路视频码流 Buffer 获取 Buffer，0 表示第 0 路（MVC 主码流）、1 表示第 1 路（MVC 从码流）。
返回值	成功：返回 0；失败：返回-1；
调用说明	在解码器初始化后才能正确返回码流 Buffer 的大小，否则返回 0.

3.3.2.17 VideoStreamDataSize

函数原型	int VideoStreamDataSize(VideoDecoder* pDecoder,int nStreamBufIndex)
功能	查询码流 Buffer 中有效数据（未解码的数据）大小，单位为字节
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamBufIndex: 对于蓝光 MVC 等 3D 视频，解码器需要处理两路码流，nStreamBufIndex 指定从第几路视频码流 Buffer 获取 Buffer。
返回值	码流 Buffer 中未解码的数据量，单位为字节。
调用说明	无

3.3.2.18 VideoStreamFrameNum

函数原型	int VideoStreamFrameNum (VideoDecoder* pDecoder,int nStreamBufIndex)
功能	查询码流 Buffer 中有效数据（未解码的数据）有多少帧；
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamBufIndex: 对于蓝光 MVC 等 3D 视频，解码器需要处理两路码流，nStreamBufIndex 指定从第几路视频码流 Buffer 获取 Buffer，0 表示第 0 路（MVC 主码流）、1 表示第 1 路（MVC 从码流）。
返回值	码流 Buffer 中未解码的数据有多少帧。
调用说明	无

3.3.2.19 VideoStreamDataInfoPointer

函数原型	void* VideoStreamDataInfoPointer(VideoDecoder* pDecoder, int nStreamBufIndex)
功能	获取码流 Buffer 中将要解码的数据所携带的 videoinfo 信息的指针；
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamBufIndex: 码流所在 buffer 的 index 标签
返回值	无
调用说明	无

3.3.2.20 TotalPictureBufferNum

函数原型	int TotalPictureBufferNum(VideoDecoder* pDecoder，int nStreamIndex)
功能	询问解码器内总共有多少个图像 Buffer
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamIndex: 对于蓝光 MVC 等 3D 视频，解码器有两路视频可供显示，nStreamIndex 指定从获取第几路视频的图像。
返回值	图像 Buffer 个数
调用说明	某些视频格式（H264 和 MPEG2）需要解码部分码流（SPS/PPS、Sequence Header）信息后才申请图像 Buffer，在此之前，本函数返回 0。

3.3.2.21 EmptyPictureBufferNum

函数原型	int EmptyPictureBufferNum(VideoDecoder* pDecoder,int nStreamIndex)
功能	询问解码器内有多少个空闲的图像 Buffer，即未被解码器和显示占用的图像 Buffer 个数。

函数原型	int EmptyPictureBufferNum(VideoDecoder* pDecoder,int nStreamIndex)
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamIndex: 对于蓝光 MVC 等 3D 视频，解码器有两路视频可供显示，nStreamIndex 指定从获取第几路视频的图像。
返回值	空闲图像 Buffer 个数
调用说明	无

3.3.2.22 ValidPictureNum

函数原型	int ValidPictureNum(VideoDecoder* pDecoder, int nStreamIndex)
功能	询问解码器内显示队列中有多少个待显示的图像。
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nStreamIndex: 对于蓝光 MVC 等 3D 视频，解码器有两路视频可供显示，nStreamIndex 指定从获取第几路视频的图像。
返回值	待显示的图像个数
调用说明	无

3.3.2.23 AllocatePictureBuffer

函数原型	VideoPicture* AllocatePictureBuffer (struct ScMemOpsS* memOps, int nWidth, int nHeight, int nLineStride, int ePixelFormat)
功能	申请一个图像 Buffer。
参数	memOps: memory 管理器接口； nWidth: 图像像素宽度； nHeight: 图像像素高度； nLineStride: 图像在内存中存放的行宽，以像素为单位； ePixelFormat: 图像像素格式；
返回值	0: 表示成功；-1: 失败；
调用说明	本函数独立于 VideoDecoder 模块，是全局函数。

3.3.2.24 FreePictureBuffer

函数原型	int FreePictureBuffer (struct ScMemOpsS* memOps, VideoPicture* pPicture)
功能	释放一个由 AllocatePictureBuffer 函数申请的图像 Buffer。

函数原型	int FreePictureBuffer (struct ScMemOpsS* memOps, VideoPicture* pPicture)
参数	memOps: memory 管理器接口 pPicture: 通过 AllocatePictureBuffer 函数申请的图像 Buffer；
返回值	成功: 返回图像 Buffer 指针；失败: 返回 NULL
调用说明	本函数独立于 VideoDecoder 模块，是全局函数。

3.3.2.25 VideoDecoderPallocIonBuf

函数原型	void VideoDecoderPallocIonBuf(VideoDecoder pDecoder, int nSize);
功能	获取 ion buffer（物理连续的 buffer）
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nSize：buffer 大小
返回值	成功：buffer 地址失败：NULL
调用说明	无

3.3.2.26 VideoDecoderFreeIonBuf

函数原型	void VideoDecoderFreeIonBuf(VideoDecoder* pDecoder, void *pIonBuf);
功能	释放 ion buffer（物理连续的 buffer）
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pIonBuf：buffer 地址；
返回值	无
调用说明	无

3.3.2.27 GetVideoFbmBufInfo

函数原型	FbmBufInfo* GetVideoFbmBufInfo(VideoDecoder* pDecoder)
功能	获取解码器开辟的 fbm buffer 的结构体信息
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针；
返回值	成功：返回 FbmBufInfo 结构体失败：返回 NULL
调用说明	无

3.3.2.28 SetVideoFbmBufAddress

函数原型	VideoPicture* SetVideoFbmBufAddress(VideoDecoder* pDecoder, VideoPicture* pVideoPicture, int bForbidUseFlag)
功能	在新显架构下，将显示模块开辟的视频帧 buffer 的地址信息传到解码器
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pVideoPicture：显示模块开辟的视频帧 buffer； bForbidUseFlag：此 buffer 此时刻是否允许使用
返回值	成功：返回 VideoPicture 结构体失败：返回 NULL
调用说明	此函数将新显模式下显示模块开辟的视频帧 buffer 地址传到解码器，然后从解码器获取相应视频帧 buffer 的全部信息

3.3.2.29 SetVideoFbmBufRelease

函数原型	VideoPicture* SetVideoFbmBufRelease(VideoDecoder* pDecoder)
功能	在新显架构下，在遇到 nativeWindow 发生改变时，需要通知解码器将设置到解码器的显示 buffer 设置成 release 状态
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针；
返回值	成功：返回 VideoPicture 结构体失败：返回 NULL
调用说明	解码器会将设成 release 状态且空闲的 buffer 添加到 release buffer 队列，供 RequestReleasePicture 函数调用

3.3.2.30 RequestReleasePicture

函数原型	VideoPicture* RequestReleasePicture (VideoDecoder* pDecoder)
功能	在新显架构下，获取解码器设置为 release 状态的视频帧 buffer
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针；
返回值	成功：返回 VideoPicture 结构体失败：返回 NULL
调用说明	无

3.3.2.31 ReturnRelasePicture

函数原型	VideoPicture* ReturnReleasePicture (VideoDecoder* pDecoder, VideoPicture* pVpicture, int bForbidUseFlag)
功能	在新显架构下，将获取到的标记为 release 的 picture 中的 buffer 相关信息修改后重新将 picture 传到解码器
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pVpicture: 包含新 buffer 信息的 pVpicture 指针；

函数原型	VideoPicture* ReturnReleasePicture (VideoDecoder* pDecoder, VideoPicture* pVpicture, int bForbidUseFlag)
返回值	bForbidUseFlag: pVpicture 包含的 buffer 是否可用的标记； 成功：返回 VideoPicture 结构体失败：返回 NULL
调用说明	中间件获取到 release 状态的 buffer 后，会将原先申请的 buffer 内存空间释放，在新的 nativeWIndow 下重新申请 buffer，然后将包含新 buffer 的内存信息的 pVpicture 设置到解码器中

3.3.2.32 ConfigExtraScaleInf

函数原型	int ConfigExtraScaleInfo(VideoDecoder* pDecoder, int nWidthTh, int nHeightTh, int nHorizonScaleRatio, int nVerticalScaleRatio)
功能	设置 scaleDown 相关信息到解码器
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； nWidthTh：要进行缩放的图像的阈值（对 >=nWidthTh 的图像进行缩放）； nHeightTh：要进行缩放的图像的阈值（对 >= nHeightTh 的图像进行缩放）； nHorizonScaleRatio：水平缩放比例； nVerticalScaleRatio：竖直缩放比例；
返回值	成功: 返回 0；失败: 返回-1；
调用说明	通常设置解码器的 scaledown 信息是通过调用 InitializeVideoDecoder 函数进行设置，但当在调用 InitializeVideoDecoder 阶段无法确定 scaleDown 参数时，可以通过调用 ConfigExtraScaleInfo 函数进行设置，但要注意此函数一定在 DecodeVideoStream 函数调用之前调用

3.3.2.33 DecoderSetSpecialData

函数原型	int DecoderSetSpecialData(VideoDecoder* pDecoder, void *pArg);
功能	设置特殊的参数，目前用于设置 jpeg 解码器里关于 skipConfig 的参数
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； pArg：参数结构体的指针
返回值	成功：0 失败：-1
调用说明	无

3.3.2.34 SetDecodePerformCmd

函数原型	Int SetDecodePerformCmd(VideoDecoder* pDecoder, enum EVDECODERSETPERFORMCMD performCmd)
功能	在视频播放过程中，开启或关闭统计丢帧信息的功能
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； performCmd：默认取值 VDECODE_SETCMD_DEFAULT 取值为：VDECODE_SETCMD_START_CALDROPFRAME 表示开始统计丢帧信息 取值为：VDECODE_SETCMD_STOP_CALDROPFRAME 表示结束统计丢帧信息
返回值	成功：返回 0 失败：返回 -1
调用说明	无

3.3.2.35 GetDecodePerformInfo

函数原型	int GetDecodePerformInfo(VideoDecoder* pDecoder, enum EVDECODERGETPERFORMCMD performCmd, VDecodePerformaceInfo** performInfo)
功能	在视频播放过程中，获取统计的丢帧信息
参数	pDecoder: 通过 CreateVideoDecoder 函数创建的视频解码器指针； performCmd：默认取值 VDECODE_SETCMD_DEFAULT 取值为：VDECODE_GETCMD_DROPFRAME_INFO 表示获取统计丢帧信息
返回值	成功：返回 0 失败：返回 -1
调用说明	无

4 其他使用场景

4.1 同编同解场景

目前主流的同编同解使用场景一般为投屏场景，即一路编码录屏，一路解码解析出图像进行显示。

同编同解场景主流芯片都是默认支持的，一般不需要做其他操作就可以正常使用。

以 T507 与 T7 两块芯片进行举例：

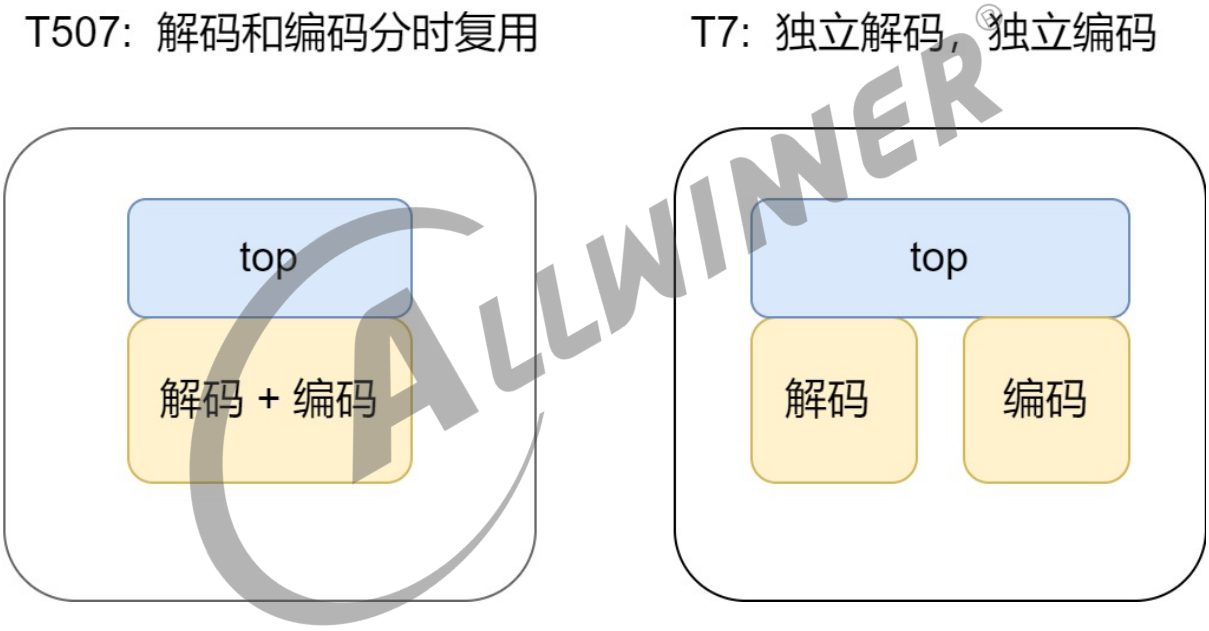


图 4-1: 编解码硬件区分

如上图所示，如果解码和编码在硬件上属于分时复用（T507 等大多数芯片），则解码和编码的任务是串行执行的，在评估性能时，需将解码和编码的性能加总。如解码和编码的性能规格都是 1080p60fps，而使用场景为解码 1080p30fps + 编码 1080p45fps，则属于超规格场景。如果解码和编码在硬件上时独立的（如 T7 等少数芯片），可并行执行任务，则可独立评估解码 + 编码的性能，如解码和编码的性能规格都是 1080p60fps，而使用场景为解码 1080p30fps + 编码 1080p45fps，则属于在性能规格范围内。

但在多线程的场景时，则会出现性能损耗。多路解码或多路编码场景（或者多路解码 + 多路编码）的性能评估不单要考虑 ve 硬件的性能，还需考虑多线程场景的线程调度效率、中断响应效率；如编码性能为 1080p120fps，而使用场景为 4 路 1080p30fps，由于多线程的软件调度的损耗会带来 ve 实际性能的下降，所以这种场景肯定是超规格的。目前针对多路编解码的使用场景的性能评估

并没有比较准确的理论评估方式，只能以实际测试为准，按照以往的经验：6 路编码场景，多线程的损耗会带来 10% ~ 20% 的 ve 性能的损失。

4.1.1 芯片差异

目前全志的大部分主流芯片都是属于分时复用，即编解码任务为串行执行。只有如 T7,V5,V200 的编解码任务可以并行进行。

5 配置说明

5.1 T527 支持 h265 解码 4K60FPS 的配置说明

T527 的 ve 默认使用的是 system 域的电压，使用 system 域的电压，h265 解码的最高性能为 4K30FPS，若要将 h265 解码的性能提高至 4K60FPS，则需要进入如下操作：

a. 硬件上将 ve 的供电改为 “外挂独立供电”，且电压值为 1.05 V；

b. 通过打开 board.dts 的配置将 ve 的频率设置为 624 MHz，如下图所示：

图 5-1: 通过 dts 配置 ve 频率

board.dts 的路径说明：longan/device/config/chips/t527/configs/方案名称/linux 内核版本，方案名称与 “./build.sh config” 里的 board 配置项一致，如方案名称为 demo_car，linux 内核为 linux-5.15，则 board.dts 的具体路径为：longan/device/config/chips/t527/configs/demo_car/linux-5.15/board.dts

```
wangxiwang@AwExdroid84:longan$ pwd
/home/wangxiwang/workspace/sdk/android-platform/t527-aiot/sdk-aiot/android13/longan
wangxiwang@AwExdroid84:longan$
wangxiwang@AwExdroid84:longan$ ./build.sh config
========ACTION List: mk_config ;========
options :
All available platform:
   0. android
   1. linux
Choice [android]:
All available ic:
   0. a523
   1. a527
   2. t527
Choice [t527]:
All available board:
   0. demo
   1. demo_battery
   2. demo_car
   3. demo_fastboot
   4. demo_linux_aiot
   5. demo_linux_car
Choice [demo_car]:
All available flash:
   0. default
   1. nor
Choice [default]:
```

“方案名称”与这里的配置一致

图 5-2: 方案名称的确认

可以通过如下方式确认 VE 的频率设置是否生效：

```
console:/ $
console:/ $
console:/ $
console:/ $
console:/ $ cat  /sys/kernel/debug/clk/clk_summary | grep ve
    ve-mbus-gate                0        1        0    24000000          0     0  50000
    bus-ve                      0        1        0    24000000          0     0  50000
    pll-ve                      0        1        0   624000000          0     0  50000
        ve                      0        1        0   624000000          0     0  50000
console:/ $
console:/ $
console:/ $
console:/ $
```

播放视频的过程中，在串口终端敲此命令

若此值与board.dts的一致，则配置成功

图 5-3: 确认 ve 频率是否设置成功

6 解码器 demo 说明

由于解码器 demo 程序的迭代速度较快，且会针对不同产品做出特性功能开发，故以下说明仅供参考，具体情况请以实际情况为准。

6.1 demo 路径

解码 demo 在 Android 的 sdk 中的具体路径为：android/frameworks/av/media/libcedarc/demo/vdecoderDemo;

若开发环境为 linux，则位于编解码相关代码 libcedarc 中的 demo 文件夹下。

6.2 基本参数介绍

在运行解码器 demo 时，即 vdecoderDemo（以下简称 demo），通过输入不同参数来实现不同的解码功能，目前 demo 中主流的参数配置如下所示：（以下所属环境为 Android64 位系统）

```
134|diana-p1:/system/bin # vdecoderDemo -h
Usage:
-h --help                          Print this help
-i --input                         Input file
-n --decode_frame_num              After display n frames, decoder stop
-ss --save_frame_start              After display ss frames, saving pictures begin
-sn --save_frame_num                After sn frames saved, stop saving picture
-o --save_frame_file               saving picture file
-cn --cost_dram_thread_num          create cn threads to cost dram, or disturb cpu, test decoder robust
-sha --save_sha                     save sha file
-vefreq --setup_ve_freq               setup ve freq
-cmpsha --compare_sha                 compare sha value
-codFmat --nCodecFormat               input file codecformat
-decNum --decoder_num                 decoder number
-outFmat --output_format              output file PixelFormat
```

图 6-1: 解码 demo 参数

在无法确定当前所使用解码器支持哪些参数输入时，则可以运行 demo 加上参数-h，即

```
./vdecoderdemo -h
```

就会将当前 demo 所支持的输入参数打印出来，可以根据该提示进行使用。

其中，i 代表输入文件的路径，即需解码的码流文件，n 代表所需解码的帧数，ss 代表从哪一帧开始保存解码得到的 yuv，sn 代表需要保存几帧解码得到的 yuv，o 代表输出文件的路径，即 yuv 的保存路径，codFmat 代表输入码流格式，目前 demo 仅支持 H264/H265/AVI 格式的码流输入，

decNum 代表解码线程数，故 demo 本身是自带多线程解码功能，outFmat 则是定义输出 yuv 的采样格式。

6.3 实例分析

接下来将以实际使用案例进行分析，使用的输入码流路径为 sdcard/tmp.264，故输入命令：

```
vdecoderDemo -i tmp.264 -o output.yuv -ss 5 -sn 10 -n 20 -codFmat 1
```

其中，-i 代表输入文件，-o 代表输出文件，-ss 5 代表从第五帧开始保存，-sn 10 代表保存 10 帧，-n 20 代表解码 20 帧，-codFmat 1 代表使用的是 H264 解码器。

若运行正常，则会得到如下结果：

```
<displayPictureThreadFunc:689>: □[40;31m display thread get enungh frame, exit ...□[0m
<displayPictureThreadFunc:734>:  display thread exit....disp frame num: 20
<DecodeThread:530>:  decoer thread recieve a finish singnal,  exit.....
<DecodeThread:564>:  decoder thread exit....
<ChannelThread:1089>: □[40;31m demoDecoder finish.decode frame: 21, display frame: 20, cost 0 s □[0m
<DestroyVideoDecoder:363>: DestroyVideoDecoder, pDecoder=0xb4000072f743b000
<ProcessThread:1712>:  exit sbm thread
<VeRelease:1740>: not malloc locks
<VeRelease:1762>: ve release ok
<ChannelThread:1100>:  demo decoder exit successful
<VeRelease:1740>: not malloc locks
<VeRelease:1762>: ve release ok
```

图 6-2: 解码 demo 示例

可以看到，若是正确解析码流后，log 日志中会打印出本次解码的实际帧数和显示帧数，显示帧数就是 n 的值，而解码实际帧数则是实际上解码器硬件所解析的码流帧数，若出现解码帧数比显示帧数多的情况，则是因为所解码的码流中有 B 帧的存在。

7 解码库编译与使用

7.1 代码路径

在 Android 系统下，ve 源码路径为：android/frameworks/av/media/libcedarc；

在 Tina 系统下，ve 源码路径为：platform/allwinner/multimedia/libcedarc；

在 Longan 系统下，ve 源码路径为：longan/platform/framework/libcedarc；

由于 SDK 的结构更新可能会导致源码路径更变，故请以实际源码路径为准，具体情况可以联系相关人员进行确认。

7.2 代码编译

以下将会对代码编译进行步骤描述，具体过程可以参考源码中 readMe.txt 文档。

7.2.1 Linux 相关环境编译

1. 首先根据当前小机端的工作环境选择对应的编译工具链，通过 export PATH 的方式设置好编译工具链的环境变量；
2. 运行 automake 相关工具：./bootstrap；
3. 配置 makefile 相关参数：./configure –prefix=INSTALL_PATH –host=HOST_NAME LDFLAGS=“-LSO_PATH”

示例：./configure –prefix=/home/user/workspace/libcedarc/install –host=arm-linux LDFLAGS=“-L/home/user/workspace/libcedarc/lib/arm926-uclibc”

即 prefix 填写的是编译出的 lib 或 bin 的保存路径，host 填写的是编译工具链的名称。

同时，FLAG 中需要带上 linux 的版本号，如当前使用的 linux 版本为 5.15，则需要带上相关 flag：CFLAGS=“-DCONF_KERNEL_VERSION_5_15” CPPFLAGS=“-DCONF_KERNEL_VERSION_5_15”

即：./configure –prefix=/home/user/workspace/libcedarc/install –host=arm-linux CFLAGS=“-DCONF_KERNEL_VERSION_5_15” CPPFLAGS=“-DCONF_KERNEL_VERSION_5_15” LDFLAGS=“-L/home/user/workspace/libcedarc/lib/arm926-uclibc”

7.2.2 Android 环境编译

Android 平台编译较为简易，只需在 SDK 中配置好相关配置，就可以进行编译；

1. 在 Android 主目录下，运行脚本 source ./build/envsetup.sh；
2. 输入 lunch 选择对应的产品配置；
3. 进入到 libcedarc 路径下，输入 mm 即可进行编译。

8 常见问题 FAQ

Q：在播放视频的过程中，如何保存输入的码流或输出的 yuv 图像?

A：libcedarc 提供了配置文件的方式来进行输入输出的保存，linux 系统下打开 etc 中的 cedarc.conf，Android 系统则可以打开 vendor/etc/cedarc.conf 进行修改。

若想要打开保存输入码流的功能，则需要在 cedarc.conf 中将两个参数进行配置，分别是 vdecoder_save_bitstream 需要配置为 1，vdecoder_save_bitstream_path 配置为输入码流保存的文件路径。

例：

```
vdecoder_save_bitstream = 0  vdecoder_save_bitstream_path = /sdcard/bs.dat
```

而若想要打开保存输出图像的功能，则需要调整其他四个参数，分别是 vdecoder_save_picture_en 配置为 1，vdecoder_save_picture_start_num 配置需要从第几帧图像开始保存，vdecoder_save_picture_total_num 配置总共要保存多少帧，vdecoder_save_picture_path 配置为输出图像的保存文件路径。

例：

```
vdecoder_save_picture_en = 0
vdecoder_save_picture_start_num = 0
vdecoder_save_picture_total_num = 2000
vdecoder_save_picture_path =/tmp/pic.dat
```

或直接进行代码修改，重新编译库：

代码路径：libcedarc/vdecoder/vdecoder.c

相关宏的修改：

DEBUG_SAVE_PICTURE：置为 1，使能此功能

PIC_SAVE_FILE：设置数据保存路径

DEBUG_SAVE_BITSTREAM：置为 1，使能此功能

BIT_STREAM_FILE：设置数据保存路径

修改完代码后，重新编译代码、推库；

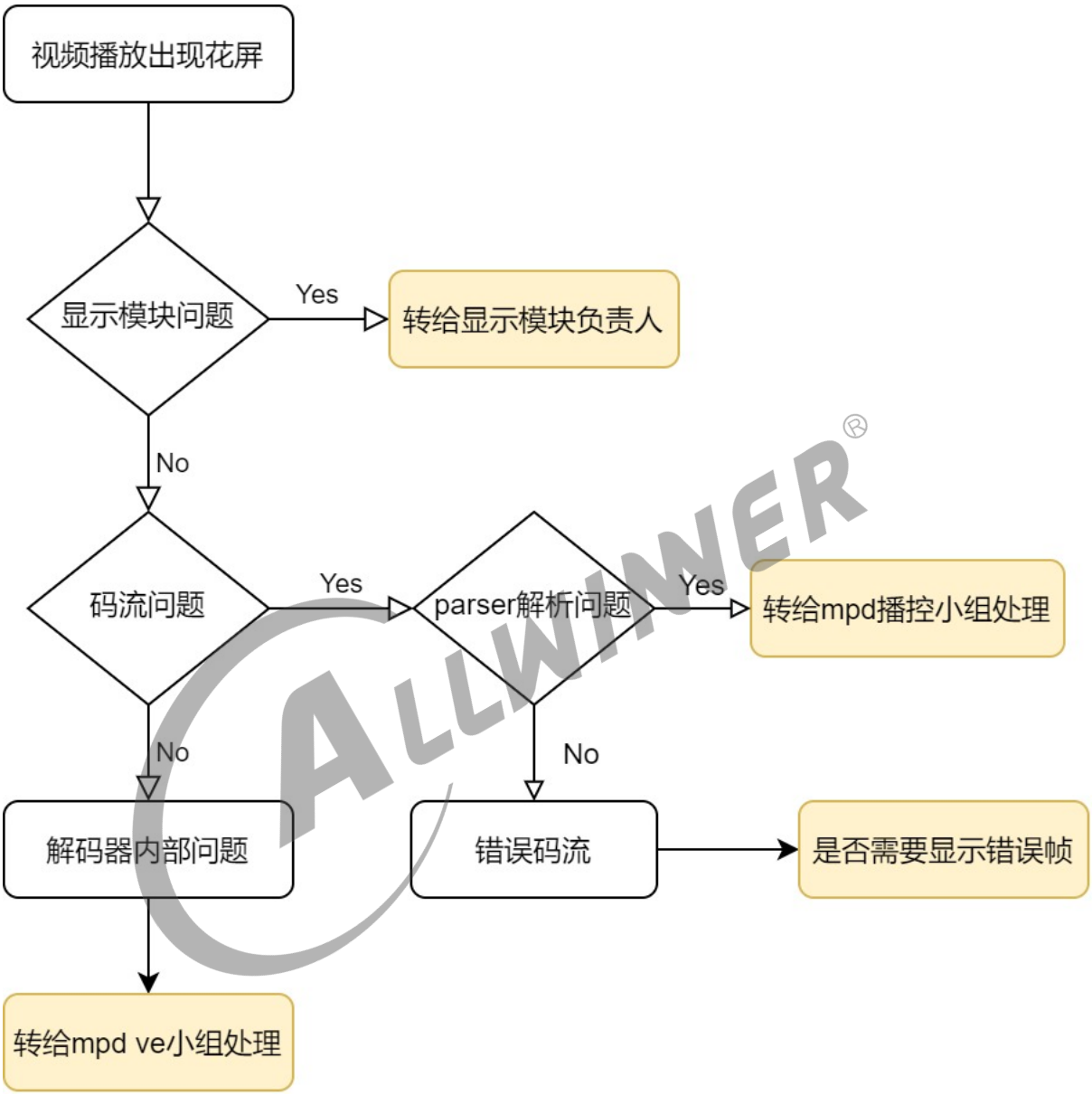
Q：错误码流的错误帧显示策略如何打开?

A：当视频文件的码流为错误码流时，解码器解码出来的图像为错误帧（即花屏帧），且会对错

误帧进行标记，AE 可结合使用场景决定是否需要显示错误帧。如何设置是否显示错误帧？调用 InitializeVideoDecoder 函数时，设置入参 VConfig 的 bDispErrorFrame 变量即可；若设置为 1，则解码器会将错误帧输出到显示模块进行显示，若设置为 0，则解码器不输出错误帧。

Q：解码时视频出现花屏现象如何排查？

A：



针对视频解码出现花屏现象的问题，排查流程如上图所示，依次排查是否属于显示端问题、码流问题、解码器内部问题，如下分别对 3 种原因的排查方法进行说明：

1. 显示端问题排查方法：保存解码后的视频帧数据（即 yuv 数据），用工具（常用工具：7yuv）查看视频帧数据是否存在花屏，如果存在花屏，则说明是解码器的问题，需进一步排查，如果没有花屏，则说明是显示端问题，需转给显示模块的负责人处理；视频帧数据的保存方式可参考上述 FAQ。
2. 码流问题排查方法：保存传递到解码器的码流数据（保存方法可参考上述 FAQ。），然后用 ffmpeg 播放码流数据，若没有花屏现象，则说明码流数据本身没有问题，需进一步排查解码器

内部问题；若存在花屏现象，则说明码流数据有问题，需进一步分析：直接使用 ffmpeg 播放视频文件，若没有花屏现象，则说明视频码流本身是没有问题的，是 parser 模块解析出了问题，导致解析出来的码流是错误的，应进一步分析 cedarx 的 parser 模块；若出现了花屏现象，则说明视频的码流本身存在错误数据，即为错误码流，需判断是否需要显示错误帧（可参考上述 FAQ）；若 parser 模块解析码流时出现了错误，应进一步分析 cedarx 的 parser 模块，可转给 mpd 部门的播控小组处理。

3. 解码器内部问题若通过前面的排查方式确认属于解码器内部问题，则转给 mpd 部门 ve 编解码驱动小组进行处理。

著作权声明

商标声明

免责声明