

# MPP模块设计文档

V0.1

David

## 概述

### 编写目的

主要介绍MPP模块的体系结构，关键流程，API接口等内容。

### 适用范围

适用于进迭时空K1系列SOC。

### 相关人员

- 应用开发工程师
- 多媒体中间件开发及维护工程师

## 模块介绍

### 背景介绍

MPP(Multimedia Processing Platform, 多媒体处理平台)属于自研操作系统Bianbu，其目的是封装多平台硬件编解码的差异，提供统一的API供开发者使用。

### 概念术语

- MPP(Multimedia Processing Platform)**: 多媒体处理平台
- MPI(Multimedia Processing Interface)**: 多媒体处理平台提供给上层的API调用
- MPP AL**: 抽象层，对不同IP，不同SOC，不同方案的多媒体接口进行抽象
- VPU(Video Processing Unit)**: 视频处理器，一般包括解码器和编码器，解码器主要有H.264/H.265/MJPEG/VP9/AV1解码器，编码器主要有H.264/H.265编码器
- G2D**: 2D图形处理硬件加速，包括图像的格式转换，缩放，旋转，裁剪，拼接等

- **Packet**: 数据包，主要表示经过压缩后的数据，即解码前或者编码后的数据，如H.264/H.265的视频流
- **Frame**: 数据帧，主要表示未经压缩的数据，即解码后或者编码前的数据，如YUV420的图像

## 模块功能

目前MPP主要包含下面几个部分：

- **VDEC**: 视频解码子模块及开放API，主要用于数据流packet的解码
- **VENC**: 视频编码子模块及开放API，主要用于数据帧frame的编码
- **G2D**: 2D图形处理加速子模块及开放API，主要进行数据帧frame的格式转换，缩放，旋转，裁剪，拼接等操作
- **BIND系统**: 支持多模块动态绑定
- **AL(Abstract Layer)**: 支持多平台

未包含部分：

- **AI/AO**: 音频的输入输出，走标准的pulseaudio->alsa-lib->alsa driver
- **VI**: 视频的输入，走标准的V4L2框架
- **VO**: 走标准的DRM显示框架
- **AENC/ADEC**: 纯软件实现，不构成性能瓶颈，Gstreamer/FFmpeg等开源框架都有全面支持，暂不支持

## 配置说明

暂无需要配置内容，待补充

## 源码结构

```

1   └── al                                ; AL(Abstract Layer)层代码，对接各平台解码库
2       ├── CMakeLists.txt
3       ├── ffmpegdec.c                   ; 对接ffmpeg的解码接口avcodec
4       ├── ffmpgenc.c                   ; 对接ffmpeg的编码接口avcodec
5       ├── ffmpegswscale.c              ; 对接ffmpeg的图像转换接口avcodec
6       └── include
7           ├── al_interface_base.h      ; AL层接口基类
8           ├── al_interface_dec.h     ; AL层解码接口基类，继承于base
9           ├── al_interface_enc.h     ; AL层编码接口基类，继承于base
10          └── al_interface_g2d.h    ; AL层图像转换接口基类，继承于base
11          └── linlonv5v7.c          ; 对接ARM LINON V5V7编解码接口，预留
12          └── openh264dec.cpp      ; 对接开源H264编解码库openh264的解码API
13          └── openh264enc.cpp      ; 对接开源H264编解码库openh264的解码API

```

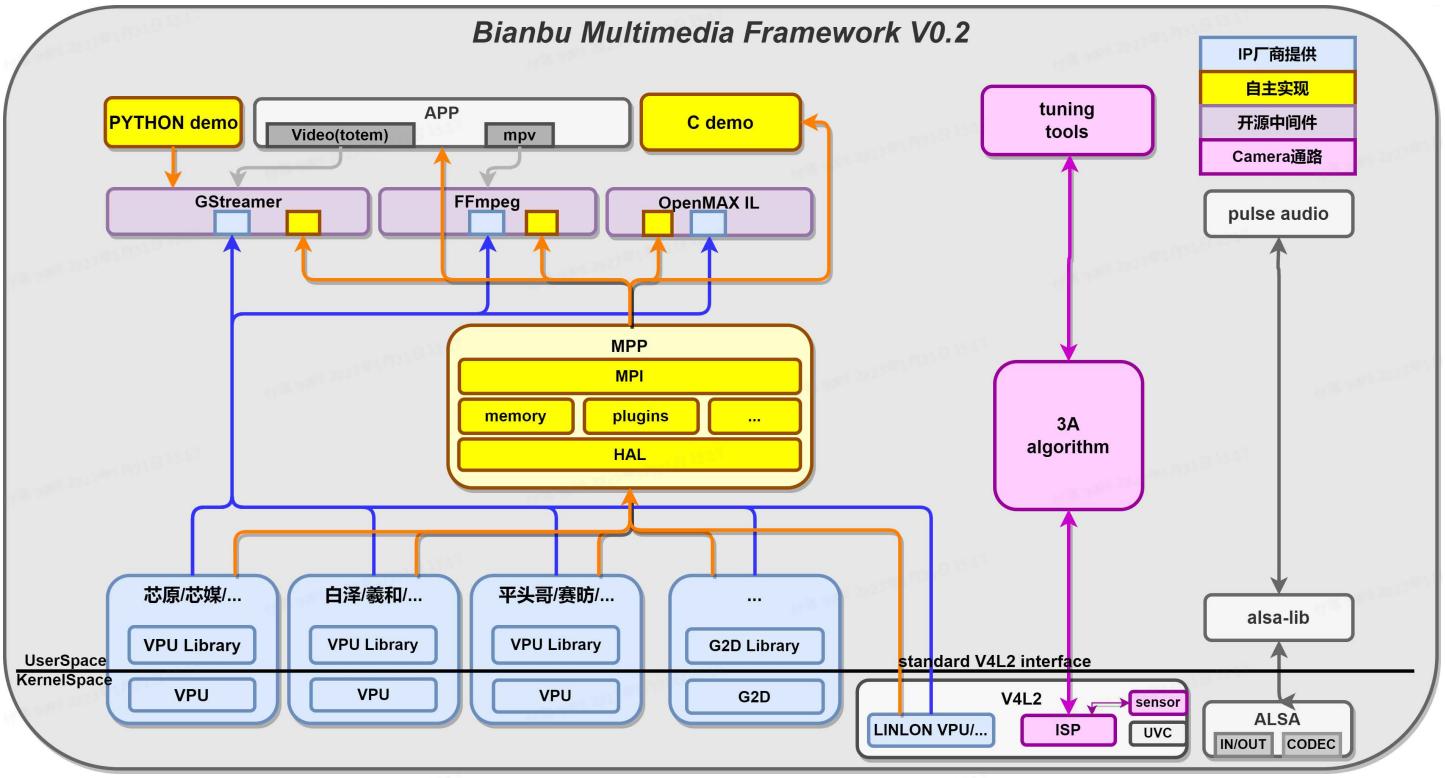
```
14 | └── sfdec_plugin.c           ; 对接StarFive赛昉惊鸿7110解码器接口，预留
15 | └── sfenc_plugin.c          ; 对接StarFive赛昉惊鸿7110编码器接口，预留
16 | └── sfomxil_dec_plugin.c   ; 对接StarFive赛昉惊鸿7110 openmaxIL层解码接口
17 | └── sfomxil_enc_plugin.c   ; 对接StarFive赛昉惊鸿7110 openmaxIL层编码接口
18 | └── v4l2dec.c               ; 对接V4L2 Codec解码接口
19 | └── v4l2enc.c               ; 对接V4L2 Codec编码接口
20 | └── vc8000.c                ; 对接阿里平头哥th1520编解码接口，预留
21 └── cmake
22     └── version.in
23     └── modules
24         ├── Findlibavcodec.cmake ; 系统中寻找ffmpeg编解码库的脚本
25         ├── Findlibopenh264.cmake ; 系统中寻找openh264编解码库的脚本
26         ├── Findlibsfdec.cmake   ; 系统中寻找StarFive赛昉惊鸿7110解码库的脚本
27         ├── Findlibsfenc.cmake   ; 系统中寻找StarFive赛昉惊鸿7110编码库的脚本
28         └── Findlibsf-omx-il.cmake ; 系统中寻找StarFive赛昉惊鸿7110 openmaxIL层编解码库的脚本
29 └── CMakeLists.txt           ; 根目录编译构建脚本
30 └── compile_install_completely.sh ; 一键执行编译安装脚本
31 └── doc
32     └── C_naming_conventions.md ; 命名规范说明
33 └── do_test.sh               ; 一键自动测试脚本
34 └── format.sh                ; 一键代码文本格式化脚本
35 └── include      ; API头文件
36     ├── base.h                ; 基础结构体和数据结构
37     ├── data.h                ; 数据封装基类MppData
38     ├── dataqueue.h            ; 数据队列实现
39     ├── frame.h                ; 数据帧类，继承于MppData
40     ├── framequeue.h           ; 无用
41     ├── g2d.h                 ; 图像转化API
42     ├── module.h               ; 动态加载编解码插件API
43     ├── packet.h               ; 数据包类，继承于MppData
44     ├── packetqueue.h          ; 无用
45     ├── para.h                ; 编解码相关的struct和enum
46     ├── sys.h                 ; BIND API
47     ├── vdec.h                ; 视频解码API
48     └── venc.h                ; 视频编码API
49 └── mpi        ; API接口实现
50     └── CMakeLists.txt
51     ├── g2d.c                 ; 图像转换接口实现
52     ├── module.c              ; 编解码插件动态加载接口实现
53     ├── sys.c                 ; BIND接口实现
54     ├── vdec.c                ; 视频解码接口实现
55     └── venc.c                ; 视频编码接口实现
56 └── mpp.cppcheck             ; cppcheck代码静态检查工具工程文件
57 └── pack_to_tar_gz.sh        ; mpp代码打包脚本
58 └── pkgconfig
59     └── spacemit_mpp.pc.cmake
60 └── test        ; 测试代码和测试流
```

```
61 |     └── CMakeLists.txt
62 |     └── include          ; 测试相关头文件
63 |         ├── argument.h    ; 测试程序输入参数处理
64 |         ├── const.h        ; 一些常量
65 |         ├── defaultparse.h ; 默认视频流解析分包数据结构
66 |         ├── h264parse.h    ; H264视频流解析分包数据结构
67 |         ├── h265parse.h    ; H265视频流解析分包数据结构
68 |         ├── mjpegparse.h   ; MJPEG视频流解析分包数据结构
69 |         └── parse.h        ; 视频流解析分包接口定义
70 |     └── parse            ; 视频流解析分包实现
71 |         ├── defaultparse.c ; 默认视频流解析分包实现
72 |         ├── h264parse.c    ; H264视频流解析分包实现
73 |         ├── h265parse.c    ; H265视频流解析分包实现
74 |         ├── mjpegparse.c   ; MJPEG视频流解析分包实现
75 |         └── parse.c        ; parse的创建和销毁
76 |     └── test_g2d_one_frame.c ; g2d测试
77 |     └── test_mpp_env.c      ; env接口测试
78 |     └── test_mpp_info.c      ; info接口测试
79 |     └── test_script          ; 测试脚本
80 |         └── cases           ; 测试用例
81 |             └── vdec.csv       ; 视频解码测试用例
82 |         └── streams          ; 测试流
83 |             ├── avs            ; avs测试流
84 |                 └── foreman_128x64.avs
85 |             ├── avs2           ; avs2测试流
86 |                 └── foreman_128x64.avs
87 |             ├── h264            ; H264测试流
88 |                 ├── Cisco_Men_whisper_640x320_CAVLC_Bframe_9.264
89 |                 ├── foreman_128x64.264
90 |                 ├── VID_1920x1080_cabac_temporal_direct.264
91 |                 ├── VID_1920x1080_cavlc_temporal_direct.264
92 |                 └── Zhling_1280x720.264
93 |             ├── h265            ; H265测试流
94 |                 ├── foreman_128x64.265
95 |                 ├── stream-1080P.h265
96 |                 └── tractorHDCrop_x0y220_960x128_3f_10b.hevc
97 |                     └── tractorHDCrop_x0y220_960x128_3f_8b.hevc
98 |             ├── mjpeg           ; MJPEG测试流
99 |                 ├── foreman_128x64.jpg
100 |                 └── outm.mjpeg
101 |             ├── vp8              ; VP8测试流
102 |                 └── foreman_128x64_vp8.ivf
103 |             ├── vp9              ; VP9测试流
104 |                 └── foreman_128x64_vp9.ivf
105 |             └── yuv420p          ; YUV420测试流
106 |                 ├── Cisco_Absolute_Power_1280x720_30fps.yuv
107 |                 └── foreman_128x64_3frames.yuv
```

```
108 |     |         └ tractorHDCrop_x0y220_960x128_3f_2plane.yuv
109 |     |         └ vdec_test.sh      ; 视频解码自动化测试脚本
110 |     └ test_sys_vdec_venc_one_frame.c    ; 解码-编码测试程序
111 |     └ test_sys_vdec_venc_vdec_one_frame.c  ; 解码-编码-解码测试程序
112 |     └ test_sys_venc_vdec_one_frame.c    ; 编码-解码测试程序
113 |     └ vdec_test.c          ; 解码测试程序
114 |     └ venc_test.c          ; 编码测试程序
115 └ thirdparty      ; 第三方库相关
116     └ ffmpeg_compile_install.md      ; ffmpeg编译安装说明
117     └ openh264_compile_install.md    ; openh264编译安装说明
118 └ utils      ; 工具相关
119     └ CMakeLists.txt
120     └ dataqueue.c      ; 数据队列实现
121     └ env.cpp          ; 环境变量读写实现
122     └ frame.c          ; 数据帧操作
123     └ include
124         └ env.h          ; 环境变量接口
125         └ info.h          ; 信息获取接口
126         └ log.h          ; 日志输出接口
127         └ os_env.h        ; 操作系统环境变量操作接口
128         └ os_log.h        ; 操作系统环境日志输出接口
129         └ type.h          ; 变量，返回值命名
130         └ v4l2_utils.h    ; V4L2操作接口
131     └ version.h        ; 版本信息
132     └ info.c          ; 信息获取实现
133     └ log.c          ; 日志输出实现
134     └ os      ; 区分操作系统的实现
135         └ linux          ; linux的实现
136             └ os_env.c
137             └ os_log.cpp
138     └ packet.c        ; 数据包操作
139     └ utils.c          ; 通用工具接口
140     └ v4l2_utils.c    ; V4L2通用工具接口
```

## 体系结构

### Bianbu多媒体系统框架结构图



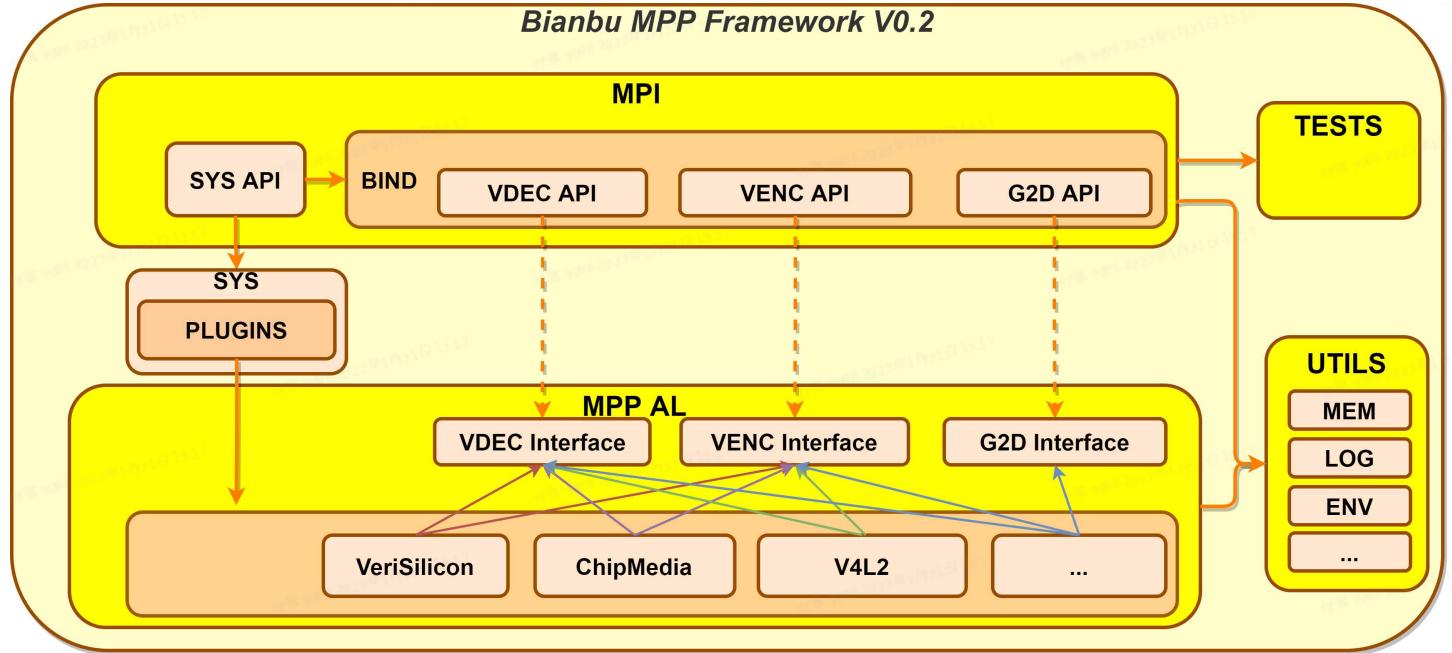
从框架结构上看分4层，从上到下依次为：

- **APP层：**包括第三方APP和自研APP，第三方APP基本上是通过GStreamer和FFmpeg来实现视频的编解码，例如Ubuntu默认集成的Video (totem)，还有我们常用的mpv player等，自研APP目前主要是我们提供的API使用参考demo，包括对接到GStreamer的PYTHON demo和对接到MPP的C demo
- **开源多媒体框架层：**常见的就是GStreamer, FFmpeg和openMAX IL，其中，GStreamer和FFmpeg是完整的多媒体解决方案，全面包含了muxer/demuxer/decoder/encoder/display的各种实现，是可以直接使用的开源框架。这一层，对于我们需要做的就是实现多个插件把硬件编解码库和MPP对接上。
- **MPP：**对上提供统一多媒体API，对下动态加载不同平台的编解码库插件来调用编解码库
- **Driver&Library：**IP厂商提供的驱动和动态库，需要我们集成到系统中

从功能上来看，分为：

- **MPP及demo：**完全自主实现的多媒体中间层
- **各开源多媒体框架：**需要实现我们自主硬件平台的插件
- **各编解码IP厂商的驱动包：**需要进行集成测试
- **V4L2驱动框架：**无改动
- **ISP IP厂商提供的驱动，算法库和调试工具：**需要进行集成测试
- **sensor厂商提供的驱动：**需要进行集成测试
- **AUDIO通路：**暂无涉及

## MPP框架结构图



从框架结构上，主要分2层，如下：

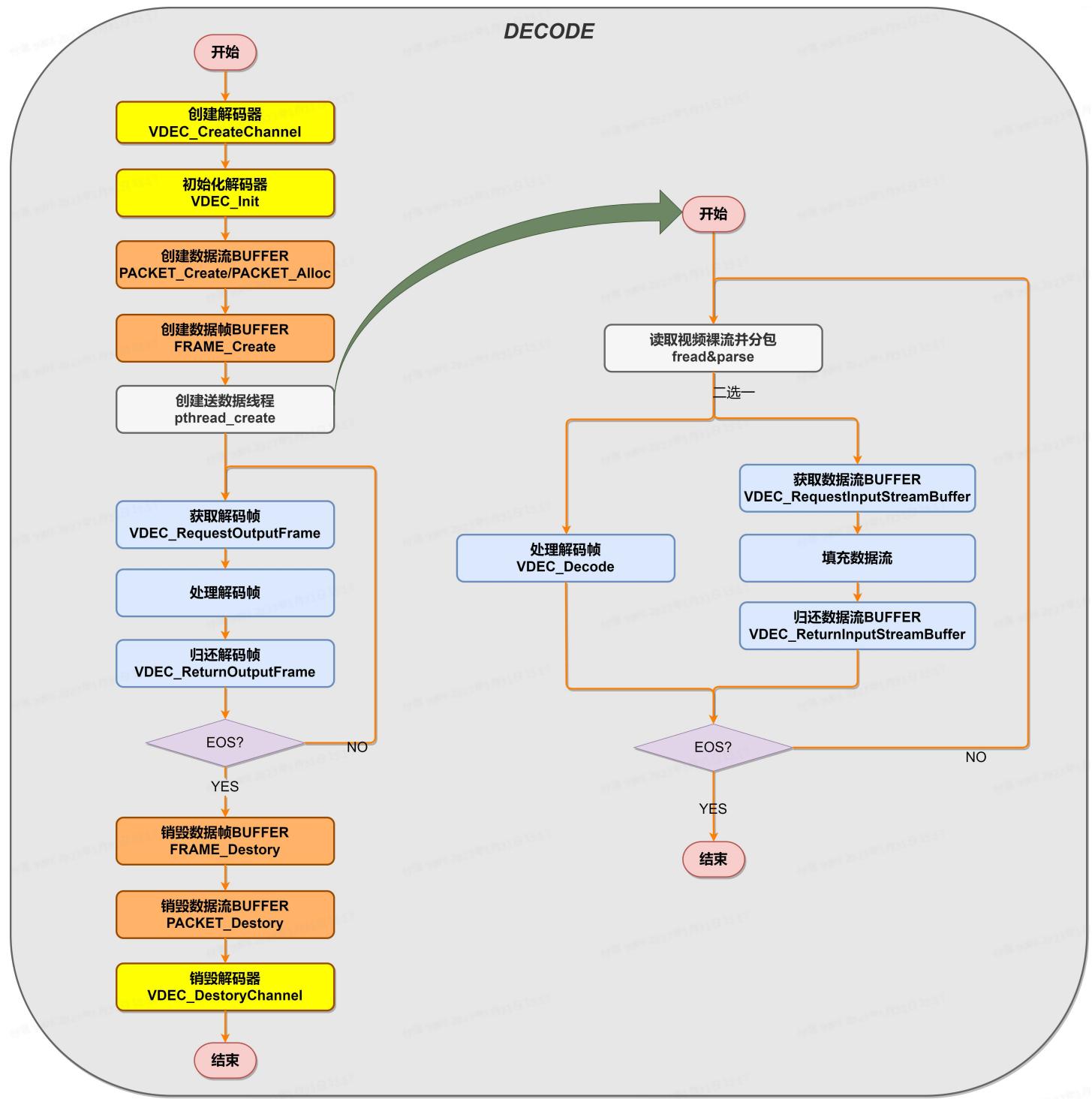
- **MPI**: 接口层，主要包含对上层的API及其实现
- **AL**: 抽象层，屏蔽不同编解码器的差异

从功能上来看，分为：

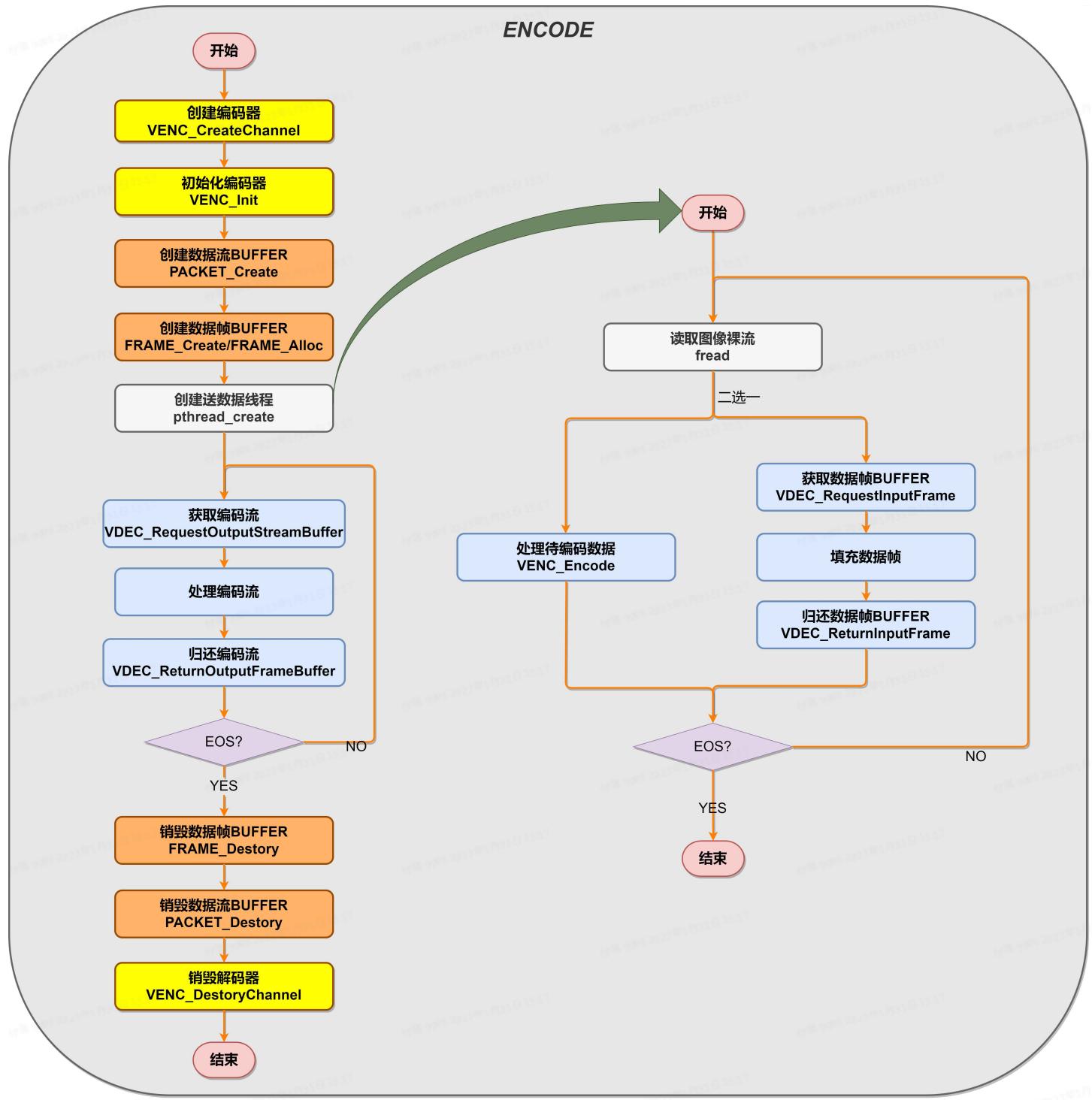
- **MPI**: 接口层
- **AL**: 抽象层
- **TESTS**: 测试程序，测试用例及测试流
- **UTILS**: 工具包，基础功能实现，包括日志输出，环境变量读写等
- **SYS**: 主要实现动态加载插件和BIND系统

## 关键流程

### 解码流程



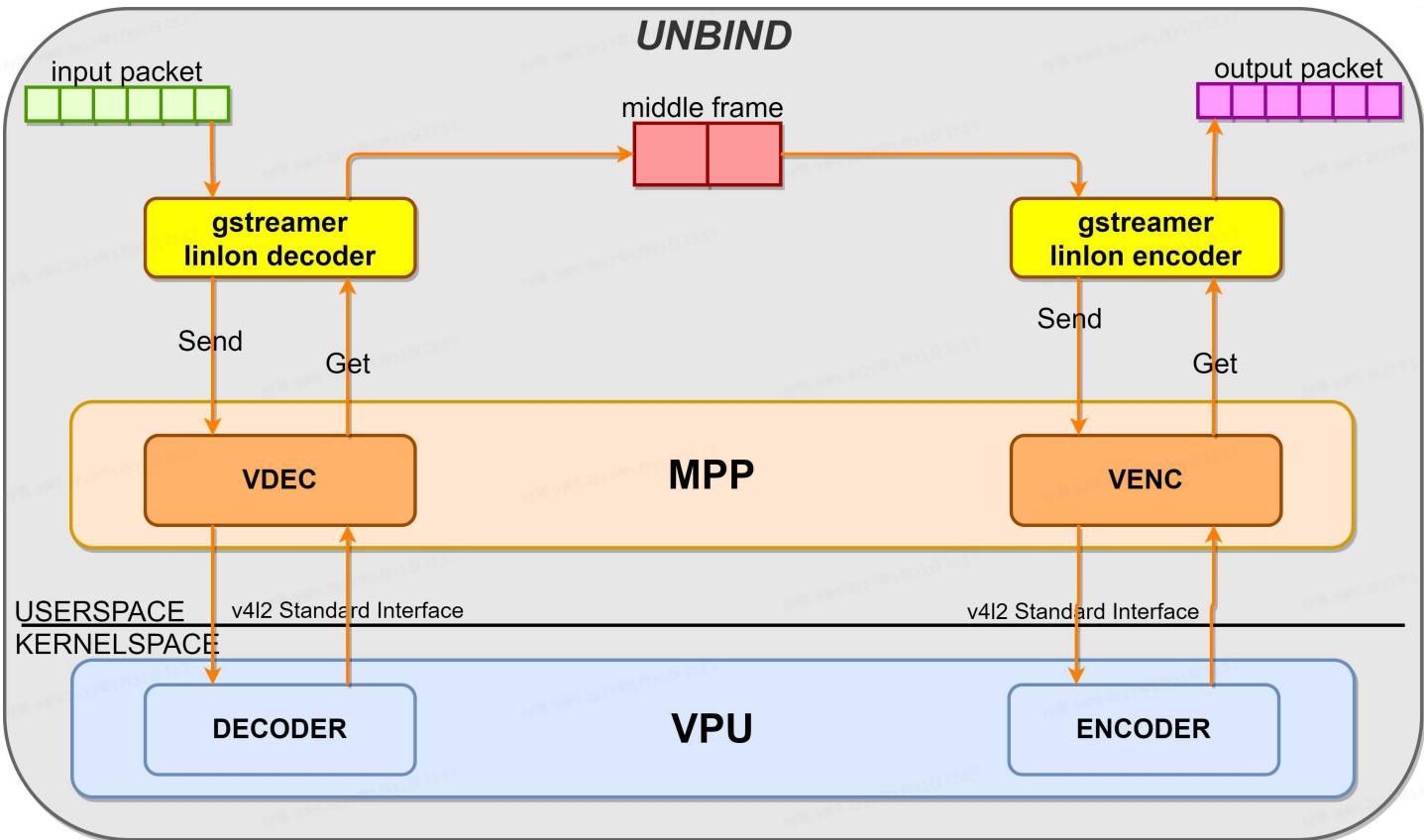
## 编码流程



## BIND

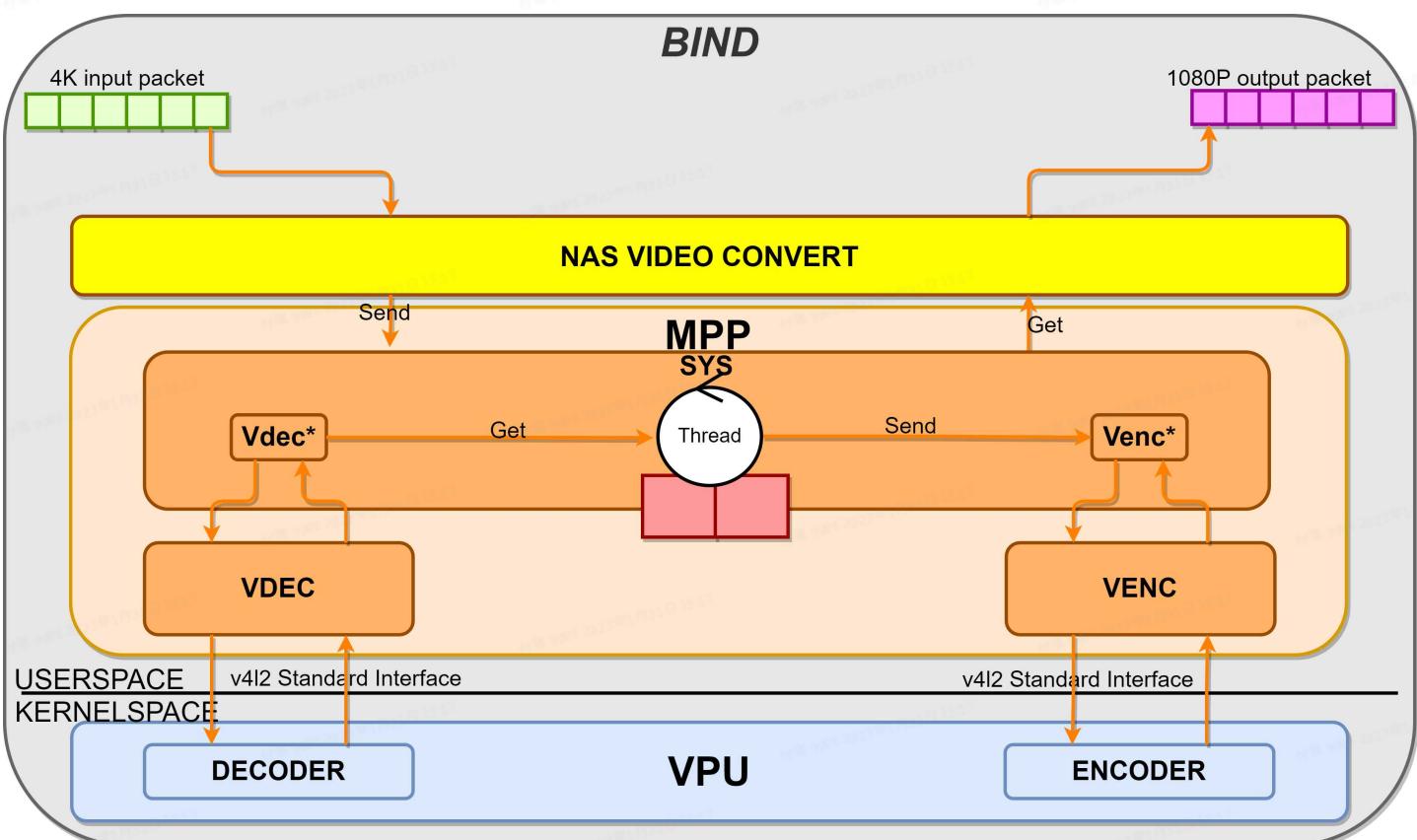
## UNBIND流程

- VDEC 和 VENC 各干各的，互不干扰，BUFFER管理传递在应用层进行

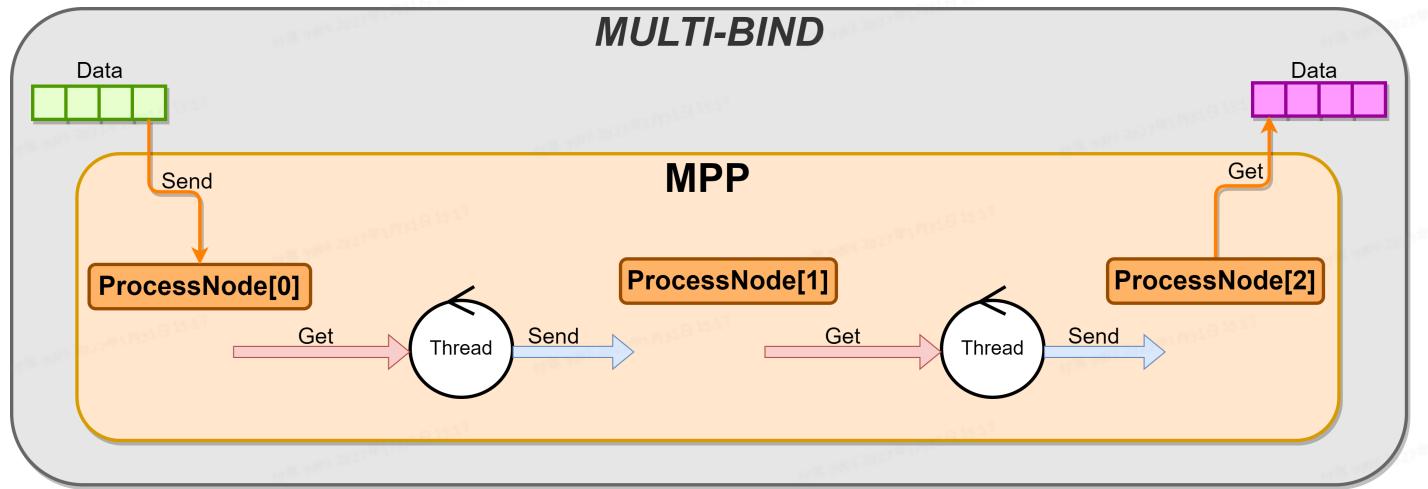


## BIND流程

- VDEC 和 VENC 作为一个整体，BUFFER管理传递在MPP进行
- BIND：关键点是SYS的context中按顺序存储了需要bind的所有node的context并将其通过thread贯穿起来



# 多节点BIND



## 插件管理

### 1. AL层接口统一，所有插件实现该接口

```
1 ALBaseContext* al_dec_create();
2 ST_RETURN al_dec_init(ALBaseContext* ctx);
3 ST_S32 al_dec_decode(ALBaseContext* ctx, MppData *sink_data);
4 ST_S32 al_dec_request_output_frame(ALBaseContext* ctx, MppData *src_data);
5 void al_dec_destory(ALBaseContext* ctx);
```

### 2. 通过节点信息等内容进行判断，得到当前平台型号，加载对应插件库，并赋值函数指针

```
1 #define OPENH264_PATH "libsoft_openh264.so"
2 #define FFMPEGCODEC_PATH "libffmpegcodec.so"
3
4 ALBaseContext* (*dec_create)();
5 void (*dec_init)(ALBaseContext* ctx);
6 ST_S32 (*dec_decode)(ALBaseContext* ctx, MppData *sink_data);
7 ST_S32 (*dec_request_output_frame)(ALBaseContext* ctx, MppData *src_data);
8 void (*dec_destory)(ALBaseContext* ctx);
9
10 ...
11
12 load_so = dlopen(FFMPEGCODEC_PATH, RTLD_LAZY | RTLD_GLOBAL);
13 if(!load_so)
14 {
15     //TO DO
16 }
```

```
18 dec_create = (ALBaseContext* (*)())dlsym(load_so, "al_dec_create");
19 dec_init = (void (*)(ALBaseContext* ctx))dlsym(load_so, "al_dec_init");
20 dec_decode = (ST_S32 (*) (ALBaseContext* ctx, MppData *sink_data))dlsym(load_so,
21 dec_request_output_frame = (ST_S32 (*) (ALBaseContext* ctx, MppData *src_data))dl
22 dec_destory = (void (*)(ALBaseContext* ctx))dlsym(load_so, "al_dec_destory");
```

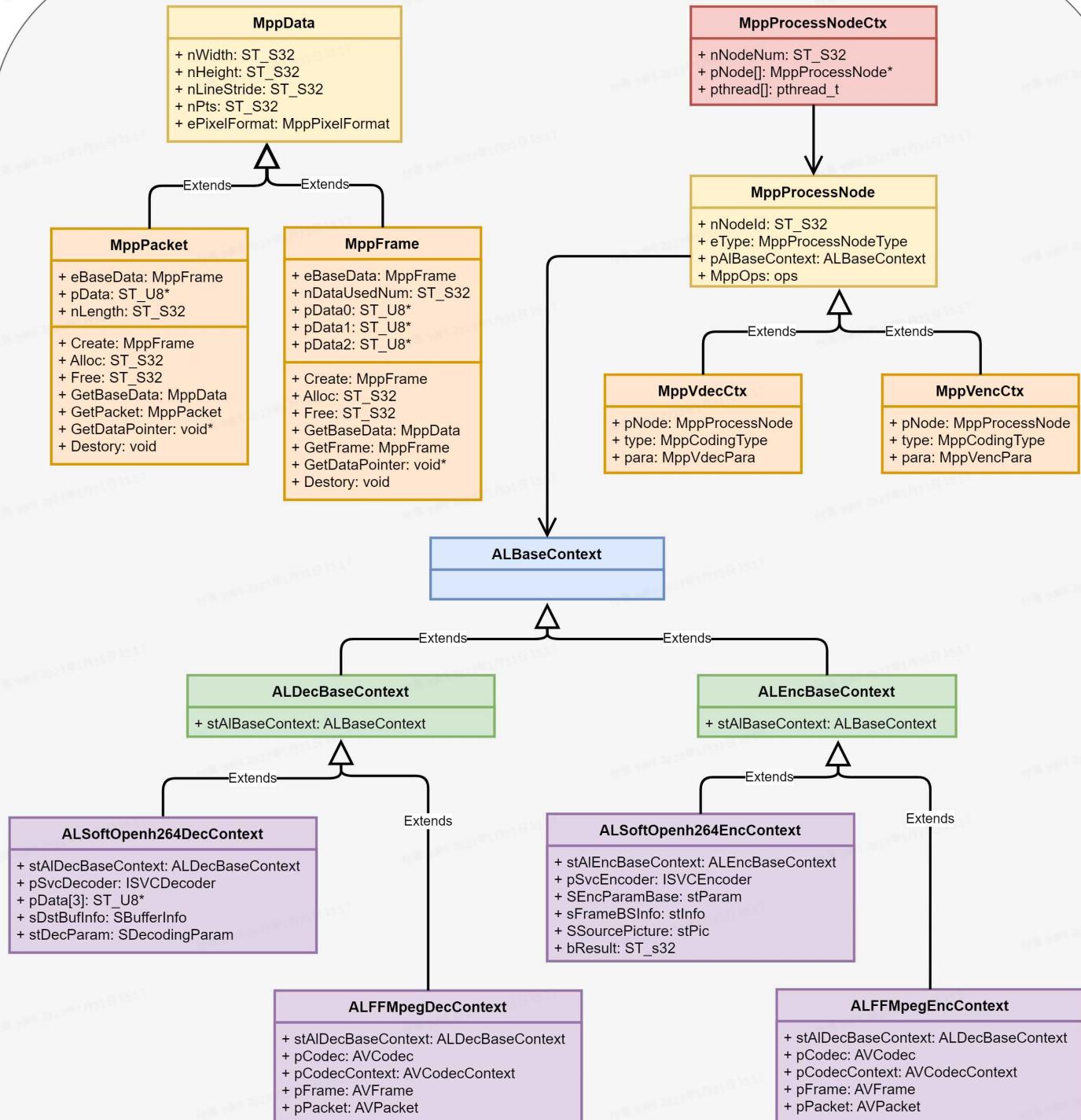
### 3. MPI调用上面的接口进行数据处理

---

## 数据结构

### 抽象

## Data Abstraction



### MppOps

```

typedef struct _MppOps {
    ST_S32 (* handle_data)(ALBaseContext *base_context, MppData *sink_data);
    ST_S32 (* get_result)(ALBaseContext *base_context, MppData *src_data);
} MppOps;
  
```

## 通用数据结构

### MppCodecType

该枚举类型定义了支持的编解码器类型，可以通过该枚举来选择编解码器，AUTO表示按照默认优先级来选择编解码器。

```
1 typedef enum _MppCodecType {
2     CODEC_AUTO = 0,    //default priority
3     CODEC_OPENH264,   //openh264 soft codec api
4     CODEC_FFMPEG,    //ffmpeg avcodec api
5     CODEC_SFDEC,     //starfive wave511 dec vpu api
6     CODEC_SFENC,     //starfive wave420l enc vpu api
7     CODEC_CODADEC,   //starfive codaj12 dec vpu api
8     CODEC_SFOMX,     //starfive omx-il api
9     CODEC_V4L2,      //V4L2 standard codec interface, ARM LINLON VPU use it
10    CODEC_UNKNOWN = 1023,
11 } MppCodecType;
```

代码中通过下面接口来动态加载特定编解码器的插件库。

```
1 /**
2  * @description: dlopen the video codec library by codec_type
3  * @param {MppCodecType} codec_type : input, the codec need to be opened
4  * @return {MppModule*} : the module context
5  */
6 MppModule* module_init(MppCodecType codec_type)
```

## MppCodingType

该枚举类型定义了支持的编码格式，包含解码器和编码器支持的所有格式，每种编解码器可能只支持其中的部分类型，比如openh264仅支持H264的编码和解码。

```
1 typedef enum _MppCodingType {
2     CODING_UNKNOWN = 0,
3     CODING_H264,
4     CODING_H265,
5     CODING_MJPEG,
6     CODING_VP8,
7     CODING_VP9,
8     CODING_AV1,
9     CODING_AV3,
10    CODING_AV32,
11    CODING_MPEG1,
12    CODING_MPEG2,
13    CODING_MPEG4,
```

```
14     CODING_FWHT,  
15 } MppCodingType;
```

需要特别指出的是：每一种编解码器都有自己定义的格式类型，需要进行格式的对应，下面是ffmpeg的格式对应示例：

```
1 #define CODING_TYPE_MAPPING_DEFINE(Type, format) |\  
2     typedef struct _AL##Type##CodingTypeMapping { |  
3         MppCodingType eMppCodingType; |  
4         format e##Type##CodingType; |  
5     } AL##Type##CodingTypeMapping;  
6  
7 #define CODING_TYPE_MAPPING_CONVERT(Type, type, format) |\  
8     static MppCodingType get_##type##_mpp_coding_type(format src_type) |\  
9     { |\  
10        S32 i = 0; |\  
11        S32 mapping_length = NUM_OF(stAL##Type##CodingTypeMapping); |\  
12        for(i = 0; i < mapping_length; i++) |\  
13        { |\  
14            if(src_type == stAL##Type##CodingTypeMapping[i].e##Type##CodingType)  
15                return stAL##Type##CodingTypeMapping[i].eMppCodingType; |\  
16        } |\  
17    |\  
18    mpp_loge("Can not find the mapping format, please check it !"); |\  
19    return CODING_UNKNOWN; |\  
20 } |\  
21 |\  
22 static format get_##type##_codec_coding_type(MppCodingType src_type) |\  
23 { |\  
24    S32 i = 0; |\  
25    S32 mapping_length = NUM_OF(stAL##Type##CodingTypeMapping); |\  
26    for(i = 0; i < mapping_length; i++) |\  
27    { |\  
28        if(src_type == stAL##Type##CodingTypeMapping[i].eMppCodingType) |\  
29            return stAL##Type##CodingTypeMapping[i].e##Type##CodingType; |\  
30    } |\  
31    |\  
32    mpp_loge("Can not find the mapping coding type, please check it !"); |\  
33    return CODING_UNKNOWN; |\  
34 } |\  
35  
36 ...  
37  
38 CODING_TYPE_MAPPING_DEFINE(FFMpegDec, enum AVCodecID)  
39 static const ALFFMpegDecCodingTypeMapping stALFFMpegDecCodingTypeMapping[] = {
```

```
40     {CODING_H264, AV_CODEC_ID_H264},  
41     {CODING_H265, AV_CODEC_ID_H265},  
42     {CODING_MJPEG, AV_CODEC_ID_MJPEG},  
43     {CODING_VP8, AV_CODEC_ID_VP8},  
44     {CODING_VP9, AV_CODEC_ID_VP9},  
45     {CODING_AV1, AV_CODEC_ID_NONE},  
46     {CODING_AVS, AV_CODEC_ID_AVVS},  
47     {CODING_AVS2, AV_CODEC_ID_AVSS2},  
48     {CODING_MPEG1, AV_CODEC_ID_MPEG1VIDEO},  
49     {CODING_MPEG2, AV_CODEC_ID_MPEG2VIDEO},  
50     {CODING_MPEG4, AV_CODEC_ID_MPEG4},  
51 };  
52 CODING_TYPE_MAPPING_CONVERT(FFMpegDec, ffmpgdec, enum AVCodecID)
```

## MppPixelFormat

该枚举类型定义了支持的像素格式，包含解码器和编码器支持的所有格式，每种编解码器可能只支持其中的部分类型。

```
1 typedef enum _MppPixelFormat {  
2     PIXEL_FORMAT_DEFAULT = 0,  
3     PIXEL_FORMAT_YUV_PLANER_420,  
4     PIXEL_FORMAT_YUV_PLANER_422,  
5     PIXEL_FORMAT_YUV_PLANER_444,  
6     PIXEL_FORMAT_YV12,  
7     PIXEL_FORMAT_I420,  
8     PIXEL_FORMAT_NV21,  
9     PIXEL_FORMAT_NV12,  
10    PIXEL_FORMAT_YUV_MB32_420,  
11    PIXEL_FORMAT_YUV_MB32_422,  
12    PIXEL_FORMAT_YUV_MB32_444,  
13    PIXEL_FORMAT_RGBA,  
14    PIXEL_FORMAT_ARGB,  
15    PIXEL_FORMAT_ABGR,  
16    PIXEL_FORMAT_BGRA,  
17    PIXEL_FORMAT_YUYV,  
18    PIXEL_FORMAT_YVYU,  
19    PIXEL_FORMAT_UYVY,  
20    PIXEL_FORMAT_VYUY,  
21  
22    PIXEL_FORMAT_UNKNOWN = 1023,  
23 } MppPixelFormat;
```

需要特别指出的是：每一种编解码器都有自己定义的格式类型，需要进行格式的对应，下面是ffmpeg的格式对应示例：

```
1 #define PIXEL_FORMAT_MAPPING_DEFINE(Type, format) \
2     typedef struct _AL##Type##PixelFormatMapping { \
3         MppPixelFormat eMppPixelFormat; \
4         format e##Type##PixelFormat; \
5     } AL##Type##PixelFormatMapping;
6
7 #define PIXEL_FORMAT_MAPPING_CONVERT(Type, type, format) \
8     static MppPixelFormat get_##type##_mpp_pixel_format(format src_format) \
9     { \
10         S32 i = 0; \
11         S32 mapping_length = NUM_OF(stAL##Type##PixelFormatMapping); \
12         for(i = 0; i < mapping_length; i++) \
13         { \
14             if(src_format == stAL##Type##PixelFormatMapping[i].e##Type##PixelFormat) \
15                 return stAL##Type##PixelFormatMapping[i].eMppPixelFormat; \
16         } \
17     \
18         mpp_loge("Can not find the mapping format, please check it !"); \
19         return PIXEL_FORMAT_UNKNOWN; \
20     } \
21 \
22     static format get_##type##_codec_pixel_format(MppPixelFormat src_format) \
23     { \
24         S32 i = 0; \
25         S32 mapping_length = NUM_OF(stAL##Type##PixelFormatMapping); \
26         for(i = 0; i < mapping_length; i++) \
27         { \
28             if(src_format == stAL##Type##PixelFormatMapping[i].eMppPixelFormat) \
29                 return stAL##Type##PixelFormatMapping[i].e##Type##PixelFormat; \
30         } \
31     \
32         mpp_loge("Can not find the mapping format, please check it !"); \
33         return (format)0; \
34     }
35
36 ...
37
38 PIXEL_FORMAT_MAPPING_DEFINE(FFMpegDec, enum AVPixelFormat)
39 static const ALFFMpegDecPixelFormatMapping stALFFMpegDecPixelFormatMapping[] = {
40     {PIXEL_FORMAT_I420, AV_PIX_FMT_YUV420P},
41     {PIXEL_FORMAT_NV12, AV_PIX_FMT_NV12},
42     {PIXEL_FORMAT_YVYU, AV_PIX_FMT_YVYU422},
43     {PIXEL_FORMAT_UYVY, AV_PIX_FMT_UYVY422},
```

```
44     {PIXEL_FORMAT_YUYV, AV_PIX_FMT_YUYV422},  
45     {PIXEL_FORMAT_RGBA, AV_PIX_FMT_RGBA},  
46     {PIXEL_FORMAT_BGRA, AV_PIX_FMT_BGRA},  
47     {PIXEL_FORMAT_ARGB, AV_PIX_FMT_ARGB},  
48     {PIXEL_FORMAT_ABGR, AV_PIX_FMT_ABGR},  
49 };  
50 PIXEL_FORMAT_MAPPING_CONVERT(FFMpegDec, ffmpgdec, enum AVPixelFormat)
```

## MppData

数据类型基类，MppPacket和MppFrame继承于MppData。

```
1 typedef struct _MppData {  
2     MppDataType         eType;  
3     MppPixelFormat      ePixelFormat;  
4     S32                 nWidth;  
5     S32                 nHeight;  
6     S32                 nLineStride;  
7     S32                 nFrameRate;  
8     S64                 nPts;  
9 }MppData;
```

## MppDataType

该枚举定义了数据类型，目前有2种类型，数据流类型和数据帧类型，数据流类型就是H264/H265等解码前/编码后数据类型，数据帧类型就是YUV/RGB等解码后/编码前数据类型。

```
1 typedef enum _MppDataType {  
2     MPP_DATA_STREAM     = 1,  
3     MPP_DATA_FRAME      = 2,  
4  
5     MPP_DATA_UNKNOWN    = 1023  
6 } MppDataType;
```

## 解码数据结构

### MppVdecCtx

视频解码器上下文，通过VDEC\_CreateChannel和VDEC\_Init进行创建和初始化。

```
1 typedef struct _MppVdecCtx {
```

```
2     MppProcessNode pNode;
3     MppCodecType eCodecType;
4     MppModule *pModule;
5     MppVdecPara para;
6     //MppOps *ops;
7     /*for bind system*/
8     //S32 bIsBind;
9     //void *bind_ctx;
10 } MppVdecCtx;
```

## MppVdecPara

解码器参数结构体。

```
1 typedef struct _MppVdecPara {
2     MppCodingType eCodingType;
3     S32 nWidth;
4     S32 nHeight;
5     S32 bScaleDownEn;
6     S32 bRotationEn;
7     S32 bSecOutputEn;
8     S32 nHorizonScaleDownRatio;
9     S32 nVerticalScaleDownRatio;
10    S32 nSecHorizonScaleDownRatio;
11    S32 nSecVerticalScaleDownRatio;
12    S32 nRotateDegree;
13    S32 bThumbnailMode;
14    S32 eOutputPixelFormat;
15    S32 eSecOutputPixelFormat;
16    S32 bNoBFrames;
17    S32 bDisable3D;
18    S32 bSupportMaf;
19    S32 bDispErrorFrame;
20 } MppVdecPara;
```

## 编码数据结构

### MppVencCtx

视频编码器上下文，通过VENC\_CreateChannel和VENC\_Init进行创建和初始化。

```
1 typedef struct _MppVencCtx {
2     MppProcessNode pNode;
```

```
3     MppCodecType eCodecType;
4     MppVencPara para;
5     MppModule *pModule;
6     S32 bIsBind;
7     void *bind_ctx;
8 } MppVencCtx;
```

## MppVencPara

编码器参数结构体。

```
1 typedef struct _MppVencPara {
2     MppCodingType eCodingType;
3     S32 nWidth;
4     S32 nHeight;
5     S32 nStride;
6     S32 nBitrate;
7     S32 nFrameRate;
8 } MppVencPara;
```

## G2D数据结构

### MppG2dCtx

图像转换器上下文。

```
1 typedef struct _MppG2dCtx {
2     MppProcessNode pNode;
3     MppCodecType eCodecType;
4     MppModule *pModule;
5 } MppG2dCtx;
```

### MppG2dPara (待完善)

```
1 typedef struct _MppG2dPara {
2     MppCodingType eCodingType;
3     S32 nWidth;
4     S32 nHeight;
5     S32 nStride;
6     S32 nBitrate;
7     S32 nFrameRate;
```

```
8 } MppG2dPara;
```

## SYS数据结构

### MppProcessFlowCtx

BIND系统pipeline上下文。

```
1 typedef struct _MppProcessFlowCtx {  
2     S32 nNodeNum;  
3     MppProcessNode* pNode[MAX_NODE_NUM];  
4     pthread_t pthread[MAX_NODE_NUM];  
5 } MppProcessFlowCtx;
```

### MppProcessNode

BIND系统pipeline的每一个node节点的定义。

```
1 typedef struct _MppProcessNode {  
2     S32 nNodeId;  
3     MppProcessNodeType eType;  
4     ALBaseContext *pAlBaseContext;  
5     MppOps *ops;  
6 } MppProcessNode;
```

### MppOps

接口抽象。

```
1 typedef struct _MppOps {  
2     S32 (* handle_data)(ALBaseContext *base_context, MppData *sink_data);  
3     S32 (* get_result)(ALBaseContext *base_context, MppData *src_data);  
4 } MppOps;
```

### MppProcessNodeType

该枚举定义了node节点的类型。

```
1 typedef enum _MppProcessNodeType {  
2     VDEC = 1,
```

```
3     VENC      = 2,  
4     G2D       = 3,  
5 } MppProcessNodeType;
```

## MppProcessNodeTypeMapping

```
1 typedef struct _MppProcessNodeTypeMapping {  
2     MppProcessNodeType eType;  
3     S8 snodeName[MAX_NODE_NAME_LENGTH];  
4 } MppProcessNodeTypeMapping;
```

## MppProcessNodeBindCouple

```
1 typedef struct _MppProcessNodeBindCouple {  
2     MppProcessNodeType eSrcNodeType;  
3     MppProcessNodeType eSinkNodeType;  
4 } MppProcessNodeBindCouple;
```

## 内部关键数据结构

### MppFrame

```
1 struct _MppFrame {  
2     MppData    eBaseData;  
3     S32       nDataUsedNum;  
4     S32       nID;  
5     U8*       pData0;  
6     U8*       pData1;  
7     U8*       pData2;  
8     U8*       pData3;  
9     void*     pMetaData;  
10 };
```

### MppPacket

```
1 struct _MppPacket {  
2     MppData    eBaseData;  
3     U8*       pData;
```

```
4     S32    nLength;
5     void    *pMetaData;
6 }
```

## ALBaseContext

```
1 typedef struct _ALBaseContext ALBaseContext;
2
3 struct _ALBaseContext {
4 }
```

## ALDecBaseContext/ALEncBaseContext/ALG2dBBaseContext

```
1 typedef struct _ALDecBaseContext ALDecBaseContext;
2 typedef struct _ALEncBaseContext ALEncBaseContext;
3 typedef struct _ALG2dBBaseContext ALG2dBBaseContext;
4
5 struct _ALDecBaseContext {
6     ALBaseContext stAlBaseContext;
7 }
8
9 struct _ALEncBaseContext {
10    ALBaseContext stAlBaseContext;
11 }
12
13 struct _ALG2dBBaseContext {
14    ALBaseContext stAlBaseContext;
15 }
```

## 接口说明

### 外部接口

### VDEC

接口	说明	参数	返回值
----	----	----	-----

VDEC_CreateChannel	创建解码器	无	MppVdecCtx*: 解码器上下文
VDEC_Init	初始化解码器	MppVdecCtx *ctx: 解码器上下文	0: 成功 非0: 错误码
VDEC_SetParam	设置解码器参数	MppVdecCtx *ctx: 解码器上下文	0: 成功 非0: 错误码
VDEC_GetParam	获取解码器参数	MppVdecCtx *ctx: 解码器上下文	0: 成功 非0: 错误码
VDEC_GetDefaultParam	获取默认解码器参数	MppVdecCtx *ctx: 解码器上下文	0: 成功 非0: 错误码
VDEC_RequestInputStreamBuffer	从解码器获取空的 streambuffer	MppVdecCtx *ctx: 解码器上下文 MppData *sink_data: buffer	0: 成功 非0: 错误码
VDEC_ReturnInputStreamBuffer	填充码流后将 buffer 送回解码器	MppVdecCtx *ctx: 解码器上下文 MppData *sink_data: buffer	0: 成功 非0: 错误码
VDEC_Decode	传送码流给解码器并进行解码	MppVdecCtx *ctx: 解码器上下文 MppData *sink_data: buffer	0: 成功 非0: 错误码
VDEC_RequestOutputFrame	获取解码帧	MppVdecCtx *ctx: 解码器上下文 MppData *src_data: 解码出来的帧	0: 成功 非0: 错误码
VDEC_ReturnOutputFrame	归还解码帧	MppVdecCtx *ctx: 解码器上下文 MppData *src_data: 解码出来的帧	0: 成功 非0: 错误码
VDEC_DestroyChannel	销毁解码器	MppVdecCtx *ctx: 解码器上下文	0: 成功 非0: 错误码
VDEC_ResetChannel	重置解码器	MppVdecCtx *ctx: 解码器上下文	0: 成功 非0: 错误码

🌟 VDEC\_RequestInputStreamBuffer/VDEC\_ReturnInputStreamBuffer和VDEC\_Decode是2种不同的送input buffer方式，只能2选1，使用场景的区别在于：是谁在驱动解码，如果解码器内部本身有线程在跑，一直在等待input buffer，则使用第1种，如果需要上层来驱动解码，传1笔数据解1笔数据，则使用第2种

# VENC

接口	说明	参数	返回值
VENC_CreateChannel	创建编码器	无	MppVencCtx*: 编码器上下文
VENC_Init	初始化编码器	MppVencCtx *ctx: 编码器上下文	0: 成功 非0: 错误码
VENC_SetParam	设置编码器参数	MppVencCtx *ctx: 编码器上下文 MppVencPara *para: 编码器参数	0: 成功 非0: 错误码
VENC_GetParam	获取编码器参数	MppVencCtx *ctx: 编码器上下文 MppVencPara *para: 编码器参数	0: 成功 非0: 错误码
VENC_RequestInputFrame	从编码器获取空的帧	MppVencCtx *ctx: 编码器上下文 MppData *sink_data: 编码帧	0: 成功 非0: 错误码
VENC_ReturnInputFrame	填充后归还编码帧	MppVencCtx *ctx: 编码器上下文 MppData *sink_data: 编码帧	0: 成功 非0: 错误码
VENC_Encode	传送码流给编码器并进行编码	MppVencCtx *ctx: 编码器上下文 MppData *sink_data: 编码帧	0: 成功 非0: 错误码
VENC_RequestOutputStreamBuffer	获取编码后码流	MppVencCtx *ctx: 编码器上下文 MppData *sink_data: 编码出来的码流	0: 成功 非0: 错误码
VENC_ReturnOutputStreamBuffer	归还编码后的码流	MppVencCtx *ctx: 编码器上下文 MppData *sink_data: 编码出来的码流	0: 成功 非0: 错误码
VENC_DestroyChannel	销毁编码器	MppVencCtx *ctx: 编码器上下文	0: 成功 非0: 错误码
VENC_ResetChannel	重置编码器	MppVencCtx *ctx: 编码器上下文	0: 成功 非0: 错误码

 **VENC\_RequestInputFrame/VENC\_ReturnInputFrame**和**VENC\_Encode**是2种不同的送input buffer方式，只能2选1，使用场景的区别在于：是谁在驱动编码，如果编码器内部本身有线程在跑，一直在等待input buffer，则使用第1种，如果需要上层来驱动编码，传1笔数据编1笔数据，则使用第2种

## G2D（待完善）

接口	说明	参数	返回值
G2D_Create	创建G2D	MppCtxType type: 硬件类型 MppCodingType coding: 编码类型	MppCtx *ctx: G2D上下文
G2D_SetParam	设置G2D参数	MppCtx *ctx: G2D上下文 MppG2dPara *para: G2D参数	0: 成功 非0: 错误码
G2D_GetParam	获取G2D参数	MppCtx *ctx: G2D上下文 MppG2dPara *para: G2D参数	
G2D_Start	开始图像处理	MppCtx *ctx: G2D上下文	0: 成功 非0: 错误码
G2D_Ctrl	控制参数设置通道	MppCtx *ctx: G2D上下文 MppG2dCtrl *ctrl: G2D控制参数	0: 成功 非0: 错误码
G2D_SendPic	传送待处理帧	MppCtx *ctx: G2D上下文 MppPic *pic: 待处理帧	0: 成功 非0: 错误码
G2D_ReturnPic	归还待处理帧帧	MppCtx *ctx: G2D上下文 MppPic *pic: 待处理帧	0: 成功 非0: 错误码
G2D_GetPic	获取处理后的帧	MppCtx *ctx: G2D上下文 MppPacket *packet: 处理后的帧	0: 成功 非0: 错误码
G2D_ReleasePic	释放处理后的帧	MppCtx *ctx: G2D上下文 MppPacket *packet: 处理后的帧	0: 成功 非0: 错误码
G2D_Stop	结束图像处理	MppCtx *ctx: G2D上下文	0: 成功 非0: 错误码
G2D_Destroy	销毁G2D	MppCtx *ctx: G2D上下文	无

接口	说明	参数	返回值
SYS_GetVersion	获取MPP版本号	MppVersion *version: MPP版本号	0: 成功 非0: 错误码
SYS_CreateFlow	创建BIND flow	无	MppProcessFlow Ctx*: flow上下文
SYS_CreateNode	创建BIND node (节点)	MppProcessNodeType type: 节点类型	MppProcessNode e*: node上下文
SYS_Init	初始化BIND flow	MppProcessFlowCtx *ctx: flow上下文	无
SYS_Destory	销毁BIND flow	MppProcessFlowCtx *ctx: flow上下文	无
SYS_Bind	数据源绑定数据接收者	MppProcessFlowCtx *ctx: flow上下文 MppProcessNode *src_ctx: 数据源 MppProcessNode *sink_ctx: 数据接受者	0: 成功 非0: 错误码
SYS_UnBind	解绑所有数据源和数据接收者	MppProcessFlowCtx *ctx: flow上下文	无
SYS_Handledata	处理数据	MppProcessFlowCtx *ctx: flow上下文 MppData *sink_data: 待处理数据	无
SYS_Getresult	返回结果	MppProcessFlowCtx *ctx: flow上下文 MppData *src_data: 处理完成的数据	无

## 内部接口

待补充

# 使用说明

待补充

**总结**