MPP 开发参考

| File Status： | Project： | MPP |
|---|---|---|
| [ ] Draft | Version： | 0.5 |
| [ ] Beta | Author： | 陈恒明 |
| [ √ ] Release | Date： | 06/08/2020 |

| Revision | Date | Description | Author |
|---|---|---|---|
| 0.1 | 04/18/2018 | 初始版本 | 陈恒明 |
| 0.2 | 05/07/2018 | 增加解码器 control 命令说明，编码器部分说明和 demo 部分说明 | 陈恒明 |
| 0.3 | 05/22/2018 | 修正一些笔误和说明错误，重新编排页码 | 陈恒明<br>谢雄斌(精英智通) |
| 0.4 | 11/28/2018 | 更新编码器输入图像的内存排布说明。<br>修正编码器流程图笔误 | 陈恒明<br>研果(科通) |
| 0.5 | 06/08/2020 | 更新编码器新配置接口<br>不再支持 RK3188 | 陈恒明 |
| | | | |
| | | | |
| | | | |
| | | | |

# 文档目录

**文档目录 ........ 2**

**图表目录 ........ 4**

**第一章 MPP 介绍 ........ 5**

1.1 概述........ 5

1.2 系统架构........ 5

1.3 平台支持........ 6

*1.3.1 软件平台支持........ 6*

*1.3.2 硬件平台支持........ 6*

1.4 功能支持........ 6

1.5 注意事项........ 6

**第二章接口设计说明 ........ 7**

2.1 接口结构概述........ 7

2.1 内存封装 MppBuffer........ 8

2.2 码流封装 MppPacket ........ 10

2.3 图像封装 MppFrame ........ 11

2.4 高级任务封装 MppTask........ 13

2.5 实例上下文封装 MppCtx........ 14

2.6 API 封装 MppApi（MPI） ........ 14

**第三章 MPI 接口使用说明 ........ 17**

3.1 解码器数据流接口 ........ 17

*3.1.1 decode_put_packet........ 17*

*3.1.2 decode_get_frame........ 18*

*3.1.3 decode........ 19*

3.2 解码器控制接口 ........ 20

*3.2.1 control........ 20*

*3.2.2 reset ........ 21*

3.3 解码器使用要点........ 22

*3.3.1 解码器单/多线程使用方式........ 22*

*3.3.2 图像内存分配以及交互模式 ........ 22*

3.4 编码器数据流接口 ........ 25

*3.4.1 encode_put_frame........ 25*

*3.4.2 encode_get_packet........ 25*

*3.4.3 encode........ 26*

3.5 编码器控制接口 ........ 26

*3.5.1 control 与 MppEncCfg........ 26*

*3.5.2 control 其他命令........ 30*

3.6 编码器使用要点........ 33

*3.6.1 输入图像的宽高与 stride ........ 33*

*3.6.2 编码器控制信息输入方式以及扩展 ........ 33*

*3.6.3 编码器输入输出流程 ........ 33*

*3.6.4 插件式自定义码率控制策略机制 ........ 33*

**第四章 MPP DEMO 说明 ........ 34**

4.1 解码器 DEMO ........ 34
4.2 编码器 DEMO ........ 35
4.3 实用工具 ........ 36

**第五章 MPP 库编译与使用 ........ 37**

5.1 下载源代码 ........ 37
5.2 编译 ........ 37
*5.2.1 Android 平台交叉编译 ........ 37*
*5.2.2 Unix/Linux 平台编译 ........ 37*

**第六章 常见问题 FAQ ........ 38**

# 图表目录

图表 1 MPP 系统框架 ..... 5
图表 2 MPI 接口使用的数据结构 ..... 7
图表 3 使用简单接口实现视频解码 ..... 7
图表 4 MppBuffer 的常规使用方式 ..... 8
图表 5 MppBuffer 外部导入使用方式 ..... 9
图表 6 MppPacket 重要参数说明 ..... 10
图表 7 MppFrame 重要参数说明 ..... 11
图表 8 使用 MppTask 来进行输入输出 ..... 13
图表 9 MppTask 支持的数据类型与关键字类型 ..... 14
图表 10 MppCtx 使用过程 ..... 14
图表 11 MPI 接口范围 ..... 17
图表 12 解码器单线程与多线程使用方式 ..... 22
图表 13 解码器图像内存纯内部分配模式示意图 ..... 22
图表 14 解码器图像内存纯内部分配模式代码流程 ..... 22
图表 15 解码器图像内存半内部分配模式代码流程 ..... 23
图表 16 解码器图像内存纯外部分配模式示意图 ..... 23
图表 17 解码器图像内存纯外部分配模式代码流程 ..... 24
图表 18 编码器输入帧内存排布 ..... 33

# 第一章 MPP 介绍

## 1.1 概述

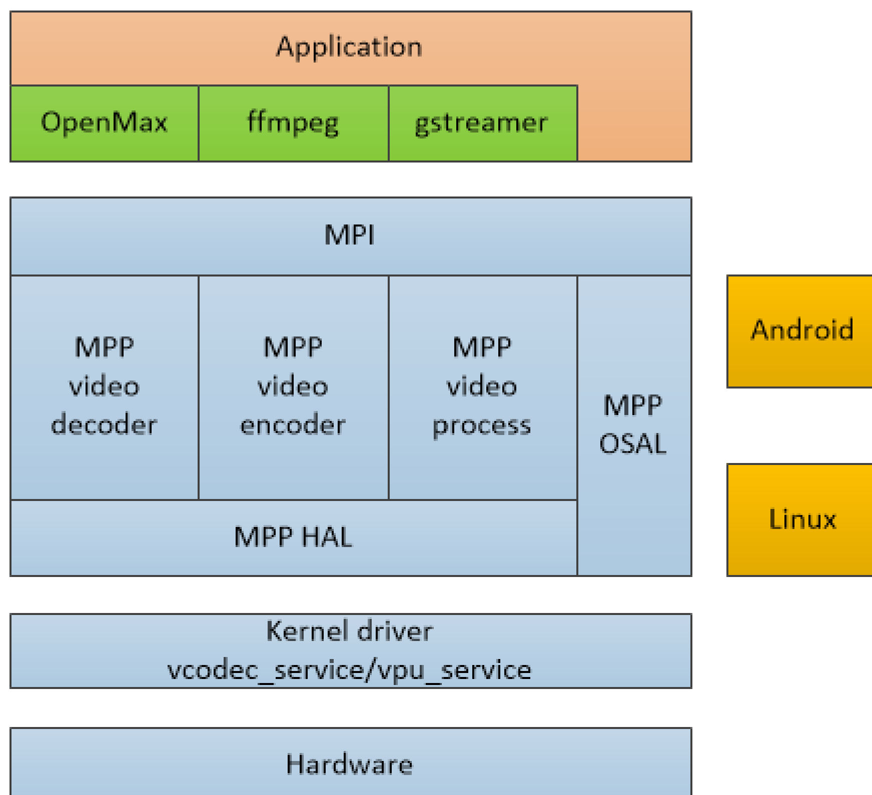
瑞芯微提供的媒体处理软件平台（Media Process Platform，简称 MPP）是适用于瑞芯微芯片系列的通用媒体处理软件平台。该平台对应用软件屏蔽了芯片相关的复杂底层处理，其目的是为了屏蔽不同芯片的差异，为使用者提供统一的视频媒体处理接口（Media Process Interface，缩写 MPI）。MPP 提供的功能包括：

- 视频解码
  - H.265 / H.264 / H.263 / VP9 / VP8 / MPEG-4 / MPEG-2 / MPEG-1 / VC1 / MJPEG
- 视频编码
  - H.264 / VP8 / MJPEG
- 视频处理
  - 视频拷贝，缩放，色彩空间转换，场视频解交织（Deinterlace）

本文档描述了 MPP 框架以及组成模块，以及供用户使用的 MPI 接口。本文档适合于上层应用开发人员以及技术支持人员阅读。

## 1.2 系统架构

MPP 平台在系统架构的层次图如下图：




图表 1 MPP 系统框架

- 硬件层 Hardware

硬件层是瑞芯微系列芯片平台的视频编解码硬件加速模块，包括 VPU，rkvdec，rkvenc 等不同类型，不同功能的硬件加速器。

- 内核驱动层 Kernel driver

Linux 内核的编码器硬件设备驱动，以及相关的 mmu，内存，时钟，电源管理模块等。支持的平台主要是 Linux kernel 3.10 和 4.4 两个版本。MPP 库对于内核驱动有依赖。

- MPP 层

用户态的 MPP 层屏蔽了不同操作系统和不同芯片平台的差异，为上层使用者提供统一的 MPI 接口。MPP 层包括 MPI 模块，OSAL 模块，HAL 模块以及视频编解码器（Video Decoder / Video Encoder）和视频处理功能模块（Video Process）。

- 操作系统层

MPP 用户态的运行平台，如 Android 以及 Debian 等 Linux 发行版

- 应用层

MPP 层通过 MPI 对接各种中间件软件，如 OpenMax，ffmpeg 和 gstreamer，或者直接对接客户的上层应用。

## 1.3 平台支持

### 1.3.1 软件平台支持

MPP 支持在各种版本的 Android 平台和纯 Linux 平台上运行。
支持瑞芯微 Linux 内核 3.10 和 4.4 版本，需要有 vcodec_service 设备驱动支持以及相应的 DTS 配置支持。

### 1.3.2 硬件平台支持

支持瑞芯微主流的各种系列芯片平台：
RK3288 系列，RK3368 系列，RK3399 系列
RK30xx 系列，RK312x 系列芯片，RK322x 系列芯片，RK332x 系列
RV1109 / RV1126 系列（注：RV1107/RV1108 会逐步退出支持）

## 1.4 功能支持

MPP 支持的编解码功能随运行的芯片平台规格不同区别很大，请查询对应芯片的《Multimedia Benchmark》。

## 1.5 注意事项

如果想快速了解 MPP 的使用和 demo，请直接转至——第四章 MPP demo 说明。

如果想快速编译和使用 MPP 代码，请直接转至——第五章 MPP 库编译与使用。

如果想了解 MPP 设计细节与设计思路，请参考 MPP 代码根目录的 readme.txt，doc 目录下的 txt 文档以及头文件的注释说明。

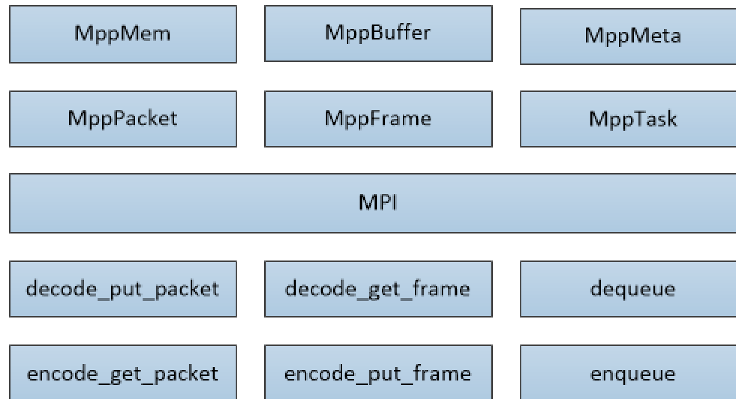
# 第二章接口设计说明

本章节描述了用户在使用 MPP 过程中会直接接触到的数据结构，以及这些数据结构的使用说明。

由于视频编解码与视频处理过程需要处理大量的数据交互，包括码流数据，图像数据以及内存数据，同时还要处理与上层应用以及内核驱动的交叉关系，所以 MPP 设计了 MPI 接口，用于与上层交互。本章节说明了 MPI 接口使用的数据结构，以及设计思路。

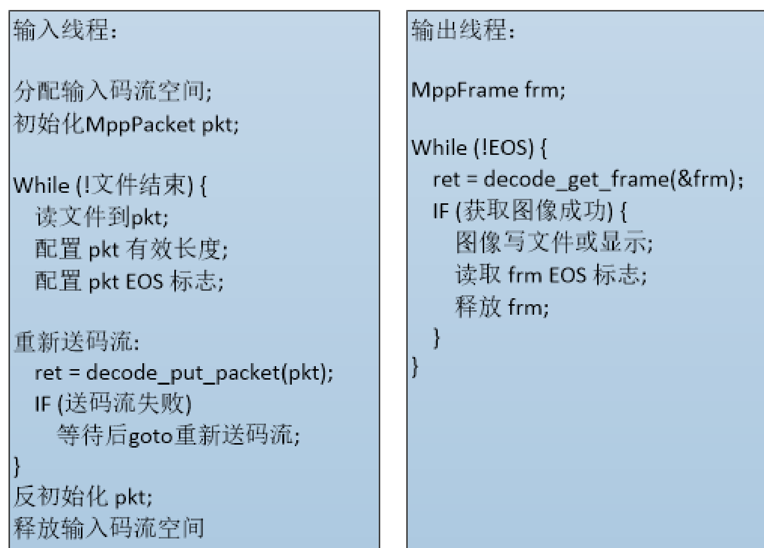
## 2.1 接口结构概述

下图为 MPI 接口使用的主要数据结构：




图表 2 MPI 接口使用的数据结构

MppMem 为 C 库 malloc 内存的封装。
MppBuffer 为硬件用的 dmabuf 内存的封装。
MppPacket 为一维缓存封装，可以从 MppMem 和 MppBuffer 生成，主要用于表示码流数据。
MppFrame 为二维帧数据封装，可以从 MppMem 和 MppBuffer 生成，主要用于表示图像数据。
使用 MppPacket 和 MppFrame 就可以简单有效的完成一般的视频编解码工作。
以视频解码为例，码流输入端把地址和大小赋值给 MppPacket，通过 put_packet 接口输入，在输出端通过 get_frame 接口得到输入图像 MppFrame，即可完成最简单的视频解码过程。




图表 3 使用简单接口实现视频解码

MppMeta 和 MppTask 为输入输出用任务的高级组合接口，可以支持指定输入输出方式等复杂使用方式，较少使用。

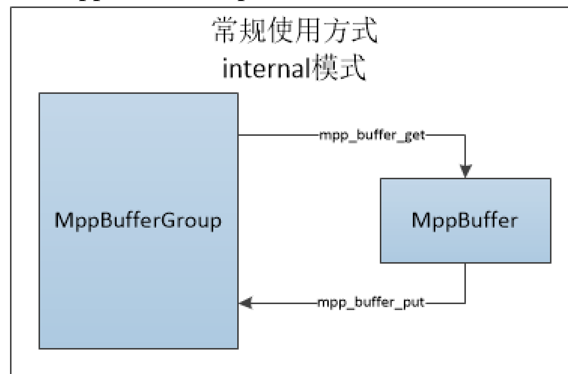
**注意：**以上这些接口数据结构都是使用 void*句柄来引用使用，其目的是为了方便扩展和前向兼容。本段落中的提到的成员都是通过形如 mpp_xxx_set/get_xxx 的接口来访问。

## 2.1 内存封装 MppBuffer

MppBuffer 主要用于描述供硬件使用的内存块，提供了内存块的分配，释放，加减引用等功能。目前支持 ion/drm 两种分配器，几个重要的参数成员如下：

| 成员名称 | 成员类型 | 描述说明 |
| --- | --- | --- |
| ptr | void * | 表示内存块的起始虚拟地址。 |
| size | size_t | 表示内存块的大小。 |
| fd | int | 表示内存块的用户态空间文件句柄。 |

在解码过程中，解码图像的缓存通常需要在固定的缓存池里进行轮转，为了实现这一点，MPP 在 MppBuffer 基础之上又定义了 MppBufferGroup，两者的使用方式有两种，如下图：



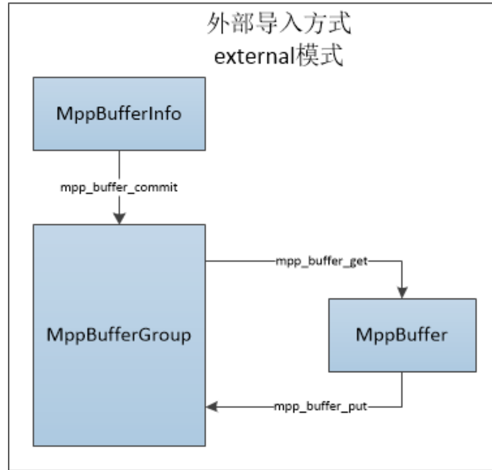

图表 4 MppBuffer 的常规使用方式

其流程伪代码如下图：

```
MppBuffer 常规使用方式

MppBufferGroup group = NULL;

// 获取缓存池并限制其大小和数量
mpp_buffer_group_get_internal(&group, type);
mpp_buffer_group_limit_config(group, size, count);

// 把缓存池配置给解码器，让解码器从缓存池中取缓存解码
mpi->control(dec, MPP_DEC_SET_EXT_BUF_GROUP, group);

// 开始解码过程
while (!end_of_decoding) {
  {  // 括号内为 MPP 解码器行为，解码器使用缓存数据
     MppBuffer buffer_in_mpp_decoder;
     // 解码器内部从缓存池取缓存解码
     mpp_buffer_get(group, &bufer_in_mpp_decoder);
     // 把图像解码到 buffer 内
     ...
  }
  mpi->decode_get_frame(&frame);
  // 输出 MppBuffer 给外部用户
  MppBuffer buffer_of_user = mpp_frame_get_buffer(frame);
  // 对图像数据做处理
  ......
  // 用户释放缓存引用和图像引用
  mpp_buffer_put(buffer_of_user);
  mpp_frame_deinit(frame);
}

// 释放缓存池
mpp_buffer_group_put(group);
```

这种方式可以实现解码器在解码过程中**零拷贝输出**（解码器的输出帧与解码器内部使用的参考帧用是同一帧），但不容易实现**零拷贝显示**（解码器的输出帧不一定在显示端可以直接显示），同时要求用户知道需要给解码器开多大的空间进行解码。

另一种使用方式是把 MppBufferGroup 完全做为一个缓存的管理器，可以管理外部导入的缓存。其使用方式如下图：




图表 5 MppBuffer 外部导入使用方式

其流程伪代码如下图：

```
MppBuffer 导入外部缓存使用方式（解码器零拷贝显示）

MppBufferGroup group = NULL;
MppBufferInfo info[16];

// 获取缓存池
mpp_buffer_group_get_external(&group, type);

// 向缓存池中导入外部缓存
mpp_buffer_commit(group, &info[0]);
mpp_buffer_commit(group, &info[1]);
...
...

// 把缓存池配置给解码器，让解码器从缓存池中取缓存解码
mpi->control(dec, MPP_DEC_SET_EXT_BUF_GROUP, group);

// 开始解码过程
while (!end_of_decoding) {
  {  // 括号内为 MPP 解码器行为，解码器使用缓存数据
     MppBuffer buffer_in_mpp_decoder;
     // 解码器内部从缓存池取缓存解码
     mpp_buffer_get(group, &bufer_in_mpp_decoder);
     // 把图像解码到 buffer 内
     ...
  }
  mpi->decode_get_frame(&frame);
  // 输出 MppBuffer 给外部用户
  MppBuffer buffer_of_user = mpp_frame_get_buffer(frame);
  // 对图像数据做处理
  ......
  // 用户释放缓存引用和图像引用
  mpp_buffer_put(buffer_of_user);
  mpp_frame_deinit(frame);
}

// 释放缓存池
mpp_buffer_group_put(group);
```

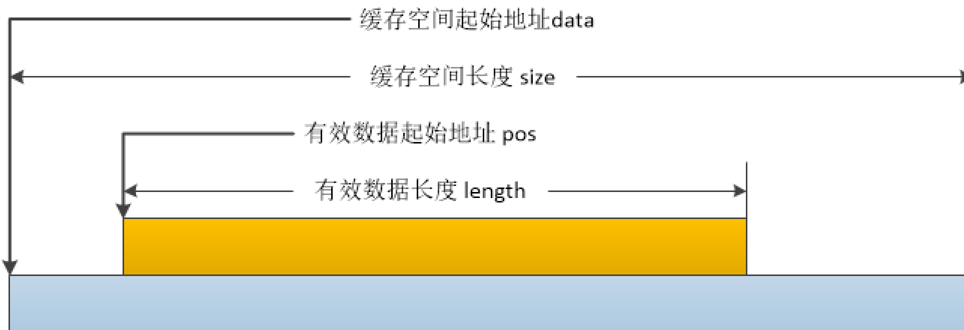
这种方式可以使得解码器使用外部的缓存，可以对接 OpenMax/ffmpeg/gstreamer 这样的中间件，也方便对接用户已有的上层代码，便于实现零拷贝显示。

## 2.2 码流封装 MppPacket

MppPacket 主要用于描述一维码流的相关信息，特别是有效数据的位置与长度。MppPacket 几个重要的参数成员如下：

| 成员名称 | 成员类型 | 描述说明 |
|---|---|---|
| data | void * | 表示缓存空间的起始地址。 |
| size | size_t | 表示缓存空间的大小。 |
| pos | void * | 表示缓存空间内有效数据的起始地址。 |
| length | size_t | 表示缓存空间内有效数据的长度。如果在 decode_put_packet 调用之后 length 变为 0，说明此包码流已消耗完成。 |

其关系如下图所示：




图表 6 MppPacket 重要参数说明

MppPacket 的其他配置参数成员如下：

| 成员名称 | 成员类型 | 描述说明 |
|---|---|---|
| pts | RK_U64 | 表示显示时间戳（Present Time Stamp）。 |
| dts | RK_U64 | 表示解码时间戳（Decoding Time Stamp）。 |
| eos | RK_U32 | 表示码流结束标志（End Of Stream）。 |
| buffer | MppBuffer | 表示 MppPacket 对应的 MppBuffer。 |
| flag | RK_U32 | 表示 MPP 内部使用的标志位，包含如下标志：<br>#define MPP_PACKET_FLAG_EOS (0x00000001)<br>#define MPP_PACKET_FLAG_EXTRA_DATA (0x00000002)<br>#define MPP_PACKET_FLAG_INTERNAL (0x00000004)<br>#define MPP_PACKET_FLAG_INTRA (0x00000008) |

MppPacket 做为描述一维内存的结构体，在使用时需要使用 malloc 出来的内存或者使用 MppBuffer 的内存进行初始化。在释放 MppPacket 时有几种情况：

如果是外部 malloc 地址配置到 MppPacket，不会做 free 释放处理，如下示例；

```
void *data = malloc(size);
MppPacket pkt = NULL;

mpp_packet_init(&pkt, data, size);
mpp_pakcet_deinit(&pkt); // <<-- 不会释放 data

free(data);
```

如果是拷贝产生的 MppPacket，会做 free 释放内存，如下示例；

```
void *data = malloc(size);
MppPacket pkt = NULL;
MppPacket pkt_copy = NULL;

mpp_packet_init(&pkt, data, size);
mpp_packet_copy_init(&pkt_copy, pkt);

mpp_pakcet_deinit(&pkt); // <<-- 不会释放 data
mpp_pakcet_deinit(&pkt_copy); // <<-- 会释放自动分配的内存

free(data);
```

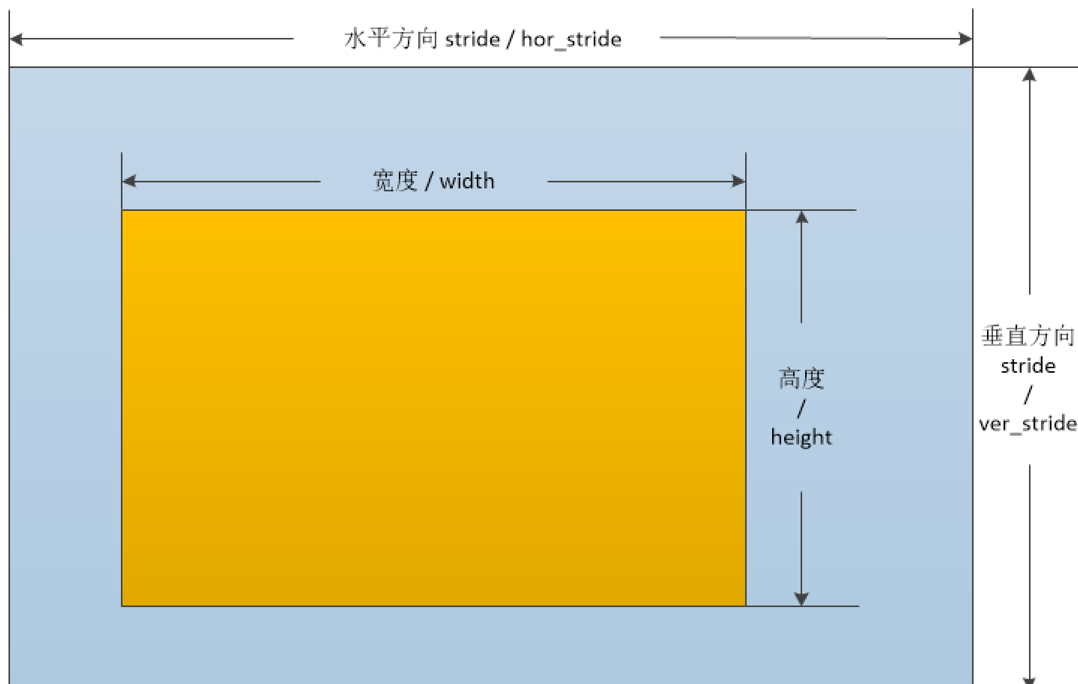
如果是 MppBuffer 产生的 MppPacket，会在生成时对 MppBuffer 加引用，在释放时对 MppPacket 减引用。

```
MppBuffer buffer;
MppPacket pkt = NULL;

mpp_buffer_get(NULL, &buffer, size);

mpp_packet_init_with_buffer(&pkt, buffer); // <<-- 自动加引用

mpp_pakcet_deinit(&pkt); // <<-- 自动减引用

mpp_buffer_put(buffer);
```

## 2.3 图像封装 MppFrame

MppFrame 主要用于定义二维图像缓存的相关信息，有效数据的位置与长度。MppFrame 几个重要的参数成员如下：

| 成员名称 | 成员类型 | 描述说明 |
|---|---|---|
| width | RK_U32 | 表示水平方向像素数，单位为像素个数。 |
| height | RK_U32 | 表示垂直方向像素数，单位为像素个数。 |
| hor_stride | RK_U32 | 表示垂直方向相邻两行之间的距离，单位为 byte 数。 |
| ver_stride | RK_U32 | 表示图像分量之间的以行数间隔数，单位为 1。 |




图表 7 MppFrame 重要参数说明

MppFrame 的其他配置参数成员如下：

<table>
<tr><th>成员名称</th><th>成员类型</th><th>描述说明</th></tr>
<tr><td>mode</td><td>RK_U32</td><td>表示图像数据帧场属性：
<pre>
/*
 * bit definition for mode flag in MppFrame
 */
/* progressive frame */
#define MPP_FRAME_FLAG_FRAME            (0x00000000)
/* top field only */
#define MPP_FRAME_FLAG_TOP_FIELD        (0x00000001)
/* bottom field only */
#define MPP_FRAME_FLAG_BOT_FIELD        (0x00000002)
/* paired field */
#define MPP_FRAME_FLAG_PAIRED_FIELD     (MPP_FRAME_FLAG_TOP_FIELD|MPP_FRAME_FLAG_BOT_FIELD)
/* paired field with field order of top first */
#define MPP_FRAME_FLAG_TOP_FIRST        (0x00000004)
/* paired field with field order of bottom first */
#define MPP_FRAME_FLAG_BOT_FIRST        (0x00000008)
/* paired field with unknown field order (MBAFF) */
#define MPP_FRAME_FLAG_DEINTERLACED     (MPP_FRAME_FLAG_TOP_FIRST|MPP_FRAME_FLAG_BOT_FIRST)
#define MPP_FRAME_FLAG_FIELD_ORDER_MASK (0x0000000C)
// for multiview stream
#define MPP_FRAME_FLAG_VIEW_ID_MASK     (0x000000f0)
</pre></td></tr>
<tr><td>pts</td><td>RK_U64</td><td>表示图像的显示时间戳（Present Time Stamp）。</td></tr>
<tr><td>dts</td><td>RK_U64</td><td>表示图像的解码时间戳（Decoding Time Stamp）。</td></tr>
<tr><td>eos</td><td>RK_U32</td><td>表示图像的结束标志（End Of Stream）。</td></tr>
<tr><td>errinfo</td><td>RK_U32</td><td>表示图像的错误标志，是否图像内有解码错误。</td></tr>
<tr><td>discard</td><td>RK_U32</td><td>表示图像的丢弃标志，如果图像解码时的参考关系不满足要求，则这帧图像会被标记为需要丢弃，不被显示。</td></tr>
<tr><td>buf_size</td><td>size_t</td><td>表示图像需要分配的缓存大小，与图像的格式相关，也与解码数据的格式相关。</td></tr>
<tr><td>info_change</td><td>RK_U32</td><td>如果为真，表示当前 MppFrame 是一个用于标记码流信息变化的描述结构，说明了新的宽高，stride，以及图像格式。<br>可能的 info_change 原因有：<br>1. 图像序列宽高变化。<br>2. 图像序列格式变化，如 8bit 变为 10bit。<br>一旦 info_change 产生，需要重新分析解码器使用的内存池。</td></tr>
<tr><td>fmt</td><td>MppFrame Format</td><td>表示图像色彩空间格式以及内存排布方式：
<pre>
typedef enum {
    MPP_FMT_YUV420SP        = MPP_FRAME_FMT_YUV,        /* YYYY... UV... (NV12)     */
    /*
     * A rockchip specific pixel format, without gap between pixel aganist
     * the P010_10LE/P010_10BE
     */
    MPP_FMT_YUV420SP_10BIT,
    MPP_FMT_YUV422SP,                                   /* YYYY... UVUV... (NV24)   */
    MPP_FMT_YUV422SP_10BIT,                             ///< Not part of ABI
    MPP_FMT_YUV420P,                                    /* YYYY... U...V...  (I420) */
    MPP_FMT_YUV420SP_VU,                                /* YYYY... VUVUVU... (NV21) */
    MPP_FMT_YUV422P,                                    /* YYYY... UU...VV...(422P) */
    MPP_FMT_YUV422SP_VU,                                /* YYYY... VUVUVU... (NV42) */
    MPP_FMT_YUV422_YUYV,                                /* YUYVYUYV... (YUY2)       */
    MPP_FMT_YUV422_UYVY,                                /* UYVYUYVY... (UYVY)       */
    MPP_FMT_YUV400SP,                                   /* YYYY...                  */
    MPP_FMT_YUV440SP,                                   /* YYYY... UVUV...          */
    MPP_FMT_YUV411SP,                                   /* YYYY... UV...            */
    MPP_FMT_YUV444SP,                                   /* YYYY... UVUVUVUV...      */
    MPP_FMT_YUV_BUTT,
    MPP_FMT_RGB565          = MPP_FRAME_FMT_RGB,        /* 16-bit RGB               */
    MPP_FMT_BGR565,                                     /* 16-bit RGB               */
    MPP_FMT_RGB555,                                     /* 15-bit RGB               */
    MPP_FMT_BGR555,                                     /* 15-bit RGB               */
    MPP_FMT_RGB444,                                     /* 12-bit RGB               */
    MPP_FMT_BGR444,                                     /* 12-bit RGB               */
    MPP_FMT_RGB888,                                     /* 24-bit RGB               */
    MPP_FMT_BGR888,                                     /* 24-bit RGB               */
    MPP_FMT_RGB101010,                                  /* 30-bit RGB               */
    MPP_FMT_BGR101010,                                  /* 30-bit RGB               */
    MPP_FMT_ARGB8888,                                   /* 32-bit RGB               */
    MPP_FMT_ABGR8888,                                   /* 32-bit RGB               */
    MPP_FMT_RGB_BUTT,
    /* simliar to I420, but Pixels are grouped in macroblocks of 8x4 size  */
    MPP_FMT_YUV420_8Z4      = MPP_FRAME_FMT_COMPLEX,
    /* The end of the formats have a complex layout */
    MPP_FMT_COMPLEX_BUTT,
    MPP_FMT_BUTT            = MPP_FMT_COMPLEX_BUTT,
} ? end MppFrameFormat ?  MppFrameFormat;
</pre></td></tr>
<tr><td>color_range</td><td>MppFrame ColorRange</td><td>表示图像数据彩色空间范围：<br>YUV full range：0 ~ 255（8bit）</td></tr>
</table>

<table>
<tr><td></td><td></td><td>YUV limit range：16 ~ 235（8bit）

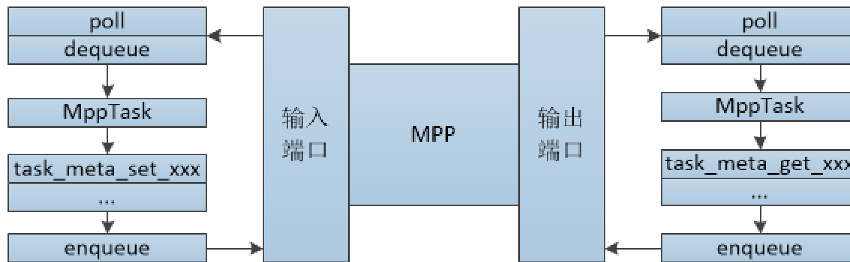
```
/*
 * MPEG vs JPEG YUV range.
 */
typedef enum {
    MPP_FRAME_RANGE_UNSPECIFIED = 0,
    MPP_FRAME_RANGE_MPEG        = 1,    ///< the normal 219*2^(n-8) "MPEG" YUV ranges
    MPP_FRAME_RANGE_JPEG        = 2,    ///< the normal     2^n-1   "JPEG" YUV ranges
    MPP_FRAME_RANGE_NB,                 ///< Not part of ABI
} MppFrameColorRange;
```
</td></tr>
<tr><td>buffer</td><td>MppBuffer</td><td>表示 MppFrame 对应的 MppBuffer。</td></tr>
</table>

对于解码器来说，MppFrame 是其输出的信息结构体，码流解码后的信息（包括像素数据与 pts，错误信息等相关信息）都需要带在 MppFrame 结构体给调用者。MppFrame 中的 pts/dts，以及 eos 标志，就是继承自对应的输入 MppPacket。

同时，一旦发现码流分辨率改变，MppFrame 中的 info_change 标志就会对应置位，向用户通知发生了 info_change 事件，需要用户进行缓存池修改处理。

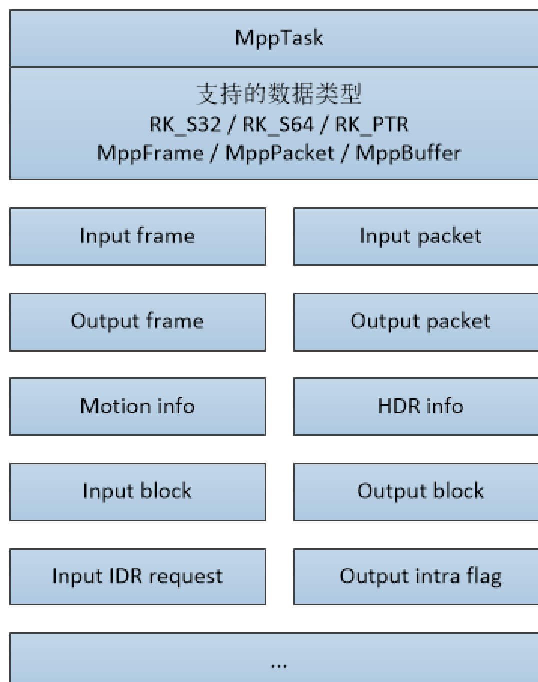
## 2.4 高级任务封装 MppTask

当 MppPacket 与 MppFrame 组成的接口无法满足需求时，需要使用 MppTask 做为一个数据容器，来满足复杂的输入输出需求。MppTask 需要与 poll/dequeuer/enqueue 接口来配合使用，对比 put_packet/get_frame 等简单流程接口，MppTask 的使用流程复杂，效率低，是为了满足复杂需求的代价。




图表 8 使用 MppTask 来进行输入输出

MppTask 是一个通过关键字 key 值（MppMetaKey）来进行扩展的结构，可以通过扩展支持的数据类型来支持复杂的高级需求。可以使用通过 mpp_task_meta_set/get_xxx 系列接口来对 MppTask 里的不同关键字数据进行访问。



图表 9 MppTask 支持的数据类型与关键字类型

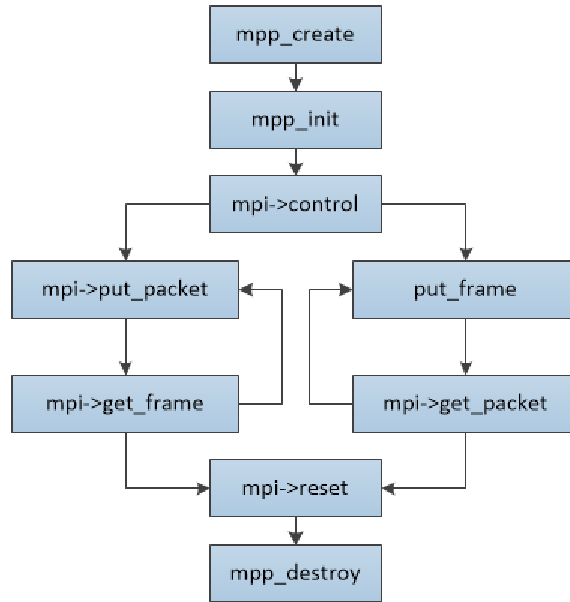
在实际使用中，需要从 MPP 的输入端口通过 dequeue 接口获取 MppTask，把需要处理的数据通过 mpp_task_meta_set_xxx 系列接口配置到 MppTask 里，之后 enqueue 输出到 MPP 实例进行处理。MPP 的输出端口流程类似，需要把 mpp_task_meta_set_xxx 系列接口换成 mpp_task_meta_get_xxx 系列接口来从 MppTask 里获取数据。

目前实用的编码器接口，以及 MJPEG 解码接口有使用 MppTask 进行实现。

## 2.5 实例上下文封装 MppCtx

MppCtx 是提供给用户使用的 MPP 实例上下文句柄，用于指代解码器或编码器实例。

用户可以通过 mpp_create 接口获取 MppCtx 实例以及 MppApi 结构体，再通过 mpp_init 来初始化 MppCtx 的编解码类型与格式，之后通过 decode_xxx/encode_xxx 以及 poll/dequeuer/enqueue 接口来进行访问，使用结束时通过 mpp_destroy 接口进行销毁。




图表 10 MppCtx 使用过程

## 2.6 API 封装 MppApi（MPI）

MppApi 结构体封装了 MPP 的对外 API 接口，用户通过使用 MppApi 结构中提供的函数指针实现视频编解码功能，其结构如下：

| 成员名称 | 成员类型 | 描述 |
|---|---|---|
| size | RK_U32 | MppApi 结构体大小。 |
| version | RK_U32 | MppApi 结构体版本。 |
| decode | 函数指针 | MPP_RET (*decode)(MppCtx ctx, MppPacket packet, MppFrame *frame)<br>视频解码接口，同时进行输入与输出，单独使用。<br>ctx ： MPP 实例上下文。<br>packet ： 输入码流。<br>frame ： 输出图像。<br>返回值 ： 0 为正常，非零为错误码。 |
| decode_put_packet | 函数指针 | MPP_RET (*decode_put_packet)(MppCtx ctx, MppPacket packet)<br>视频解码输入接口，与 decode_get_frame 配合使用。<br>ctx ： MPP 实例上下文。<br>packet ： 输入码流。 |

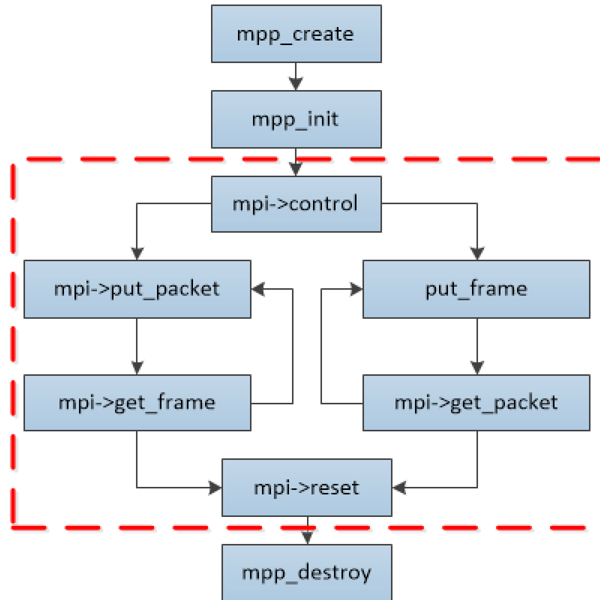
| | | |
|---|---|---|
| | | 返回值 ：0 为正常，表示码流已被 MPP 处理；非零为出现错误，码流未被处理，需要把码流重新输入。 |
| decode_get_frame | 函数指针 | MPP_RET (*decode_get_frame)(MppCtx ctx, MppFrame *frame)<br>视频解码输出接口，与 decode_put_packet 配合使用。<br>ctx ：MPP 实例上下文。<br>frame ：输出图像。<br>返回值 ：0 为正常，表示获取输出过程正常，需要判断是否有得到有值的 frame 指针；非零为出现错误。 |
| encode | 函数指针 | MPP_RET (*encode)(MppCtx ctx, MppFrame frame, MppPacket *packet)<br>视频编码接口，同时进行输入与输出，单独使用。<br>ctx ：MPP 实例上下文。<br>frame ：输入图像。<br>packet ：输出码流。<br>返回值 ：0 为正常，非零为错误码。 |
| encode_put_frame | 函数指针 | MPP_RET (*encode_put_frame)(MppCtx ctx, MppFrame frame)<br>视频编码输入接口，与 encode_get_packet 配合使用。<br>ctx ：MPP 实例上下文。<br>frame ：输入图像。<br>返回值 ：0 为正常，非零为错误码。 |
| encode_get_packet | 函数指针 | MPP_RET (*encode_get_packet)(MppCtx ctx, MppPacket *packet)<br>视频编码输出接口，与 encode_put_frame 配合使用。<br>ctx ：MPP 实例上下文。<br>packet ：输出码流。<br>返回值 ：0 为正常，非零为错误码。 |
| poll | 函数指针 | MPP_RET (*poll)(MppCtx ctx, MppPortType type, MppPollType timeout)<br>端口查询接口，用于查询端口是否有数据可供 dequeue。<br>ctx ：MPP 实例上下文。<br>type ：端口类型，分为输入端口与输出端口。<br>timeout ：查询超时参数，-1 为阻塞查询，0 为非阻塞查询，正值为超时毫秒数。<br>返回值 ：0 为正常，有数据可取出，非零为错误码。 |
| dequeue | 函数指针 | MPP_RET (*dequeue)(MppCtx ctx, MppPortType type, MppTask *task)<br>端口出队列接口，用于从端口中取出 MppTask 结构。<br>ctx ：MPP 实例上下文。<br>type ：端口类型，分为输入端口与输出端口。<br>task ：MppTask。<br>返回值 ：0 为正常，有数据可取出，非零为错误码。 |
| enqueue | 函数指针 | MPP_RET (*enqueue)(MppCtx ctx, MppPortType type, MppTask task)<br>端口入队列接口，用于往端口送入 MppTask 结构。<br>ctx ：MPP 实例上下文。<br>type ：端口类型，分为输入端口与输出端口。<br>task ：MppTask。 |

| | | |
|---|---|---|
| | | 返回值 ：0 为正常，有数据可取出，非零为错误码。 |
| reset | 函数指针 | MPP_RET (*reset)(MppCtx ctx)<br>复位接口，用于对 MppCtx 的内部状态进行复位，回到可用的初始化状态。需要注意的是，reset 接口是阻塞的同步接口。<br>ctx ：MPP 实例上下文。<br>返回值 ：0 为正常，有数据可取出，非零为错误码。 |
| control | 函数指针 | MPP_RET (*control)(MppCtx ctx, MpiCmd cmd, MppParam param)<br>控制接口，用于向 MPP 实例进行额外控制操作的接口。<br>ctx ：MPP 实例上下文。<br>cmd ：Mpi 命令类型，表示控制命令的不同类型的。<br>task ：Mpi 命令参数，表示控制控制命令的附加参数。<br>返回值 ：0 为正常，有数据可取出，非零为错误码。 |

# 第三章 MPI 接口使用说明

本章节描述了用户使用 MPI 接口的具体过程，以及过程是的一些注意事项。

MPI（Media Process Interface）是 MPP 提供给用户的接口，用于提供硬件编解码功能，以及一些必要的相关功能。MPI 是通过 C 结构里的函数指针方式提供给用户，用户可以通过 MPP 上下文结构 MppCtx 与 MPI 接口结构 MppApi 组合使用来实现解码器与编码器的功能。




图表 11 MPI 接口范围

如上图所示，mpp_create，mpp_init，mpp_destroy 是操作 MppCtx 接口的过程，其中 mpp_create 接口也获取到了 MPI 接口结构体 MppApi，真正的编码与解码过程是通过调用 MppApi 结构体里内的函数指针来实现，也就是上图中红框内的部分。红框内的函数调用分为编解码流程接口 put/get_packet/frame 和相关的 control 和 reset 接口。下文先描述编解码器接口，再对编解码器工作中的一些要点进行说明。

## 3.1 解码器数据流接口

解码器接口为用户提供了输入码流，输出图像的功能，接口函数为 MppApi 结构体里的 decode_put_packet 函数，decode_get_frame 函数和 decode 函数。这组函数提供了最简洁的解码功能支持。

### 3.1.1 decode_put_packet

| 接口定义 | MPP_RET decode_put_packet(MppCtx ctx, MppPacket packet) |
|---|---|
| 输入参数 | ctx ： MPP 解码器实例<br>packet ： 待输入的码流数据 |
| 返回参数 | 运行错误码 |
| 功能 | 输入 packet 码流包数据给 ctx 指定的 MPP 解码器实例 |

输入码流的形式：分帧与不分帧

MPP 的输入都是没有封装信息的裸码流，裸码流输入有两种形式：

一种是已经按帧分段的数据，即每一包输入给 decode_put_packet 函数的 MppPacket 数据都已经包含完整的一帧，不多也不少。在这种情况下，MPP 可以直接按包处理码流，是 MPP 的默认运行情况。

另一种是按长度读取的数据，这样的数据无法判断一包 MppPacket 数据是否是完整的一帧，需要 MPP 内部进行分帧处理。MPP 也可以支持这种形式的输入，但需要在 mpp_init 之前，通过 control 接口的 MPP_DEC_SET_PARSER_SPLIT_MODE 命令，MPP 内的 need_split 标志打开。

```
// NOTE: decoder split mode need to be set before init
RK_U32 need_split = 1;
mpi_cmd = MPP_DEC_SET_PARSER_SPLIT_MODE;
param = &need_split;
ret = mpi->control(ctx, mpi_cmd, param);
if (MPP_OK != ret) {
    mpp_err("mpi->control failed\n");
    goto MPP_TEST_OUT;
}
```

这样，调用 decode_put_packet 输入的 MppPacket 就会被 MPP 重新分帧，进入到情况一的处理。
如果这两种情况出现了混用，会出现码流解码出错的问题。
分帧方式处理效率高，但需要输入码流之前先进行解析与分帧；
不分帧方式使用简单，但效率会受影响。
在 mpi_dec_test 的测试用例中，使用的是方式不分帧的方式。在瑞芯微的 Android SDK 中，使用的是分帧的方式。用户可以根据自己的应用场景和平台条件进行选择。

输入码流的消耗
输入 MppPacket 的有效数据长度为 length，在送入 decode_put_packet 之后，如果输入码流被成功地消耗，函数返回值为零（MPP_OK），同时 MppPacket 的 length 被清为 0。如果输入码流还没有被处理，会返回非零错误码，MppPacket 的 length 保持不变。

函数的工作模式
decode_put_packet 函数的功能是输入待解码码流给 MPP 实例，但在一些情况下，MPP 实例无法接收更多的数据，这时，工作于非阻塞模式的 decode_put_packet 会报出错误信息并直接返回。用户得到 decode_put_packet 返回的错误码之后，需要进行一定时间的等待，再重新送入码流数据，避免额外的频繁 cpu 开销。

最大缓冲数据包数量
MPP 实例默认可以接收 4 个输入码流包在处理队列中，如果码流送得太快，就会报出错误码要求用户等待后再送。

### 3.1.2 decode_get_frame

| 接口定义 | MPP_RET decode_get_frame(MppCtx ctx, MppFrame *frame) |
|---|---|
| 输入参数 | ctx ： MPP 解码器实例。<br>frame ： 用于获取 MppFrame 实例的指针。 |
| 返回参数 | 运行错误码 |
| 功能 | 从 ctx 指定的 MPP 解码器实例里获取完成解码的 frame 描述信息。 |

MPP 解码输出的图像是通过 MppFrame 结构来描述的，同时 MppFrame 结构也是 MPP 实例输出信息的管道，图像的错误信息，以及变宽高信息（info change）也是带在 MppFrame 结构进行输出的。

输出图像的错误信息
图像的错误信息为 errinfo，表示图像内容是否有错误，errinfo 不为零则表示码流解码时发生了错误，图像内容是有问题的，可以做丢弃处理。

解码器图像空间需求的确认

解码器在解码时，需要为输出图像获取保存像素数据的内存空间，用户需要给解码器提供足够大小，这个空间大小的需求，会在 MPP 解码器内部根据不同的芯片平台以及不同的视频格式需求进行计算，计算后的内存空间需求会通过 MppFrame 的成员变量 buf_size 提供给用户。用户需要按 buf_size 的大小进行内存分配，即可满足解码器的要求。

输出图像的变宽高信息（Info change）
当码流的宽高，格式，像素位深等信息发生变化时，需要反馈给用户，用户需要更新解码器使用的内存池，把新的内存更新给解码器。这里涉及到解码内存分配与使用模式，会在 3.3.2 图像内存分配以及交互模式进行说明。

### 3.1.3 decode

decode 函数是 decode_put_packet 与 decode_get_frame 数据的结合，为用户提供了两个函数的复合调用。其内部逻辑为：

1. 获取输出图像；
2. 如果输出图像获取成功即返回；
3. 判断码流已送入成功则返回；
4. 送入输入码流；
5. 标记码流送入是否成功并循环第一步；

在用户看来，decode 函数首先是获取解码图像，有解码图像优先返回解码图像，没有可输出的解码图像的情况下送入码流，最后再尝试一次获取解码图像并退出。

## 3.2 解码器控制接口

### 3.2.1 control

在定义于 rk_mpi_cmd.h 文件的 MpiCmd 枚举类型定义了 control 接口命令字，其中与解码器和解码过程相关的命令如下：

```
MPP_DEC_CMD_BASE                = CMD_MODULE_CODEC | CMD_CTX_ID_DEC,
MPP_DEC_SET_FRAME_INFO,         /* vpu api legacy control for buffer slot dimension init */
MPP_DEC_SET_EXT_BUF_GROUP,      /* IMPORTANT: set external buffer group to mpp decoder */
MPP_DEC_SET_INFO_CHANGE_READY,
MPP_DEC_SET_PRESENT_TIME_ORDER, /* use input time order for output */
MPP_DEC_SET_PARSER_SPLIT_MODE,  /* Need to setup before init */
MPP_DEC_SET_PARSER_FAST_MODE,   /* Need to setup before init */
MPP_DEC_GET_STREAM_COUNT,
MPP_DEC_GET_VPUMEM_USED_COUNT,
MPP_DEC_SET_VC1_EXTRA_DATA,
MPP_DEC_SET_OUTPUT_FORMAT,
MPP_DEC_SET_DISABLE_ERROR,      /* When set it will disable sw/hw error (H.264 / H.265) */
MPP_DEC_SET_IMMEDIATE_OUT,
MPP_DEC_CMD_END,
```

从 MPP_DEC_CMD_BASE 到 MPP_DEC_CMD_END 之间的命令为解码器的 control 接口命令，分别介绍这些命令的功能如下：

MPP_DEC_SET_FRAME_INFO
命令参数为 MppFrame，用于配置解码器的默认宽高信息，返回的 MppFrame 结构会从解码器中带出需要分配的图像缓存大小。命令调用时机一般在 mpp_init 之后，mpi->decode_put_packet 之前。

MPP_DEC_SET_EXT_BUF_GROUP
命令参数为 MppBufferGroup，用于把解码器图像解码所需的 MppBufferGroup 配置给解码器。命令调用时机视图像内存分配模式有不同。

MPP_DEC_SET_INFO_CHANGE_READY
无命令参数，用于标记解码器使用的 MppBufferGroup 已经完成 Info Change 操作的 reset 处理，可以继续解码。命令调用时机视图像内存分配模式有不同。

MPP_DEC_SET_PRESENT_TIME_ORDER
命令参数为 RK_U32*，用于处理异常的码流时间戳。

MPP_DEC_SET_PARSER_SPLIT_MODE
命令参数为 RK_U32*，用于使能 MPP 内的协议解析器使用内部分帧处理，默认为码流按帧输入，不开启。命令调用时机是在 mpp_init 之前。

MPP_DEC_SET_PARSER_FAST_MODE
命令参数为 RK_U32*，用于使能 MPP 内的快速帧解析，提升解码的软硬件并行度，但副作用是会对错误码流的标志有影响，默认关闭。命令调用时机是在 mpp_init 之前。

MPP_DEC_GET_STREAM_COUT
命令参数为 RK_U32*，用于外部应用获取还未处理的码流包数量，历史遗留接口。

MPP_DEC_GET_VPUMEM_USED_COUT
命令参数为 RK_U32*，用于外部应用获取 MPP 使用的 MppBuffer 数量，历史遗留接口。

MPP_DEC_SET_VC1_EXTRA_DATA
暂未实现，历史遗留接口。

MPP_DEC_SET_OUTPUT_FORMAT
命令参数为 MppFrameFormat，用于外部应用配置 JPEG 解码器的输出格式，默认不使用。

MPP_DEC_SET_DISABLE_ERROR
命令参数为 RK_U32*，用于关闭 MPP 解码器的错误处理。一旦使能，MPP 解码会无视码流的错误情况，输出全部的可解码图像，同时不对输出的 MppFrame 结构里的 errinfo 进行标记。命令调用时机在 decode_put_packet 之前。

MPP_DEC_SET_IMMEDIATE_OUT
命令参数为 RK_U32*，用于使能 H.264 解码器的立即输出模式。一旦使能，H.264 解码器会忽略丢

帧导致的帧序不连续情况，立即输出解码的图像。命令调用时机在 decode_put_packet 之前。

### 3.2.2 reset

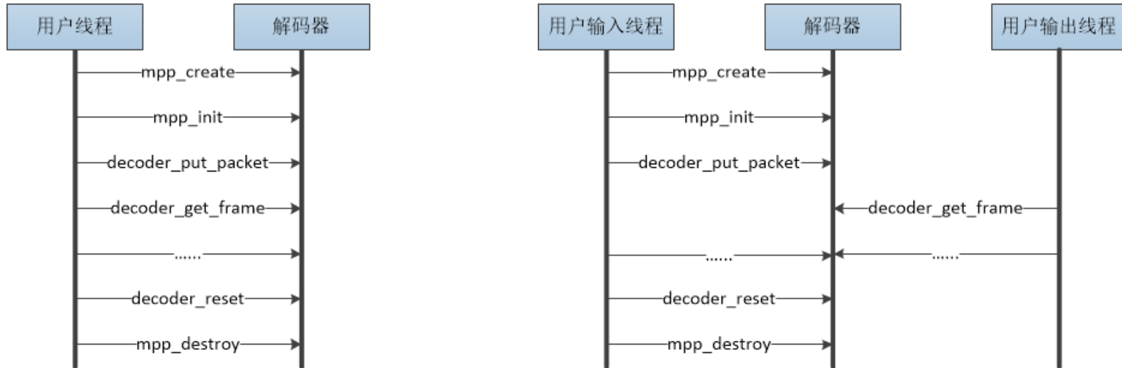
reset 接口用于把解码器恢复为正常初始化后的状态。
当用户发送最后一包 MppPacket 码流，并置上 EOS 标记送入解码器，解码器在处理完这最后一包数据之后会进入 EOS 的状态，不再接收和处理码流，需要 reset 之后再才能再继续接收新的码流。

## 3.3 解码器使用要点

解码器在使用过程中，需要注意的一些重要事项：

### 3.3.1 解码器单/多线程使用方式

MPP 解码器的 MPI 接口是线程安全的，可以在多线环境下使用。单线程工作模式如 mpi_dec_test 的 demo 所示，多线程工作模式如 mpi_dec_mt_test 的 demo 所示。

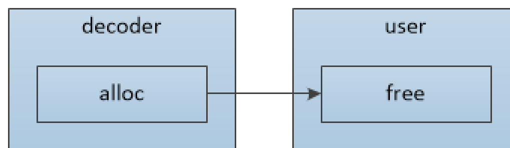



图表 12 解码器单线程与多线程使用方式

### 3.3.2 图像内存分配以及交互模式

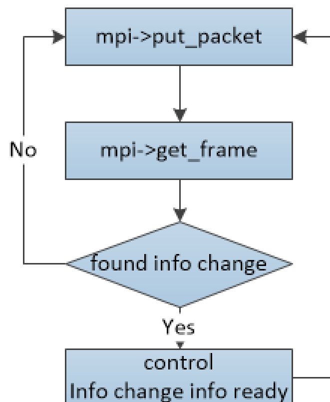
解码器在解码图像时，需要获取内存空间以写入数据，在解码完成之后，这块内存空间需要交给用户使用，在用户使用完成之后要释放给解码器，在关闭解码器时要释放全部内存空间。在这种工作方式下，解码器与用户之间才可以形成零拷贝的数据交互。MPP 解码器支持三种内存分配以及与用户交互图像数据的模式：

模式一：纯内部分配模式
图像内存直接从 MPP 解码器内部分配，内存由解码器直接分配，用户得到解码器输出图像，在使用完成之后直接释放。




图表 13 解码器图像内存纯内部分配模式示意图

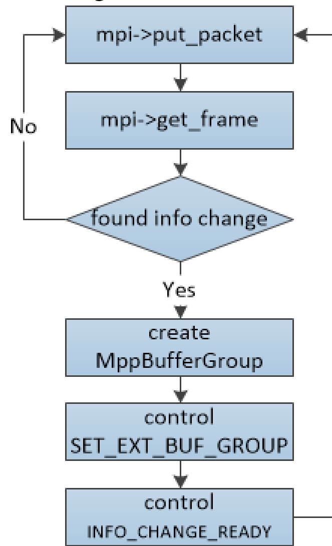
在这种方式下，用户不需要调用解码器 control 接口的 MPP_DEC_SET_EXT_BUF_GROUP 命令，只需要在解码器上报 info change 时直接调用 control 接口的 MPP_DEC_SET_INFO_CHANGE_READY 命令即可。解码器会自动在内部进行内存分配，用户需要把获取到的每帧数据直接释放。




图表 14 解码器图像内存纯内部分配模式代码流程

优点：
过程简单，可以快速完成一个可用的 demo，评估解码器的性能。
缺点：
1. 内存都是从解码器内部分配的，如果内存在解码器被销毁时还没有被释放，有可能出现内存泄漏或崩溃问题。
2. 无法控制解码器的内存使用量。解码器可以不受限制地使用内存，如果码流输入的速度很快，用户又没有及时释放内存，解码器会很快消耗掉全部的可用内存。
3. 实现零拷贝的显示比较困难，因为内存是从解码器内部分配的，不一定和用户的显示系统兼容。

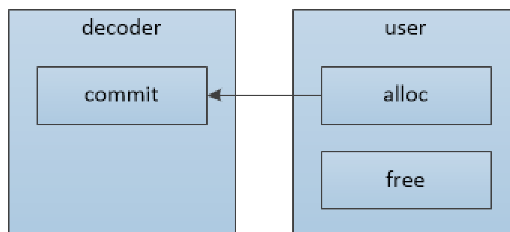
模式二：半内部分配模式
这种模式是 mpi_dec_test demo 使用的默认模式。用户需要根据 get_frame 返回的 MppFrame 的 buf_size 来创建 MppBufferGroup，并通过 control 接口的 MPP_DEC_SET_EXT_BUF_GROUP 配置给解码器。用户可以通过 mpp_buffer_group_limit_config 接口来限制解码器的内存使用量。



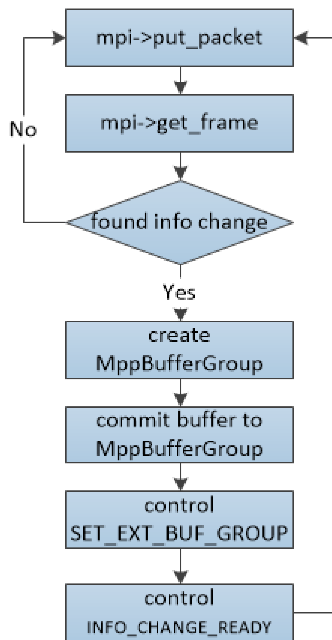
图表 15 解码器图像内存半内部分配模式代码流程

优点：
过程简单，易上手，可以一定程度限制内存的使用。
缺点：
1. 内存空间的限制并不准确，内存的使用量不是 100%固定的，会有波动。
2. 同样难于实现零拷贝的显示。

模式三：纯外部分配模式
这种模式通过创建空的 external 模式的 MppBufferGroup，从用户那里导入外部分配器分析的内存块文件句柄（一般是 dmabuf/ion/drm）。在 Android 平台上，Mediaserver 通过 gralloc 从 SurfaceFlinger 获取显示用内存，把 gralloc 得到的文件句柄提交（commit）到 MppBufferGroup 里，再把 MppBufferGroup 通过 control 接口 MPP_DEC_SET_EXT_BUF_GROUP 命令配置给解码器，然后 MPP 解码器将循环使用 gralloc 得到的内存空间。



图表 16 解码器图像内存纯外部分配模式示意图




图表 17 解码器图像内存纯外部分配模式代码流程

优点：

直接使用外部显示用的内存，容易实现零拷贝。

缺点：

1. 理解与使用较困难。
2. 需要修改用户程序，一些用户程序的调用方式限制了纯外部分配方式的使用。

纯外部分配模式使用时的注意事项：

1. 如果图像内存分配是在解码器创建之前，需要有额外的方式来得到图像内存的大小。

一般的 YUV420 图像内存空间计算方法：

图像像素数据：hor_stride * ver_stride * 3 / 2

额外附加信息：hor_stride * ver_stride / 2

2. 内存块数需要考虑解码和显示的需求，如果内存块数分配得太少，可能会卡住解码器。

H.264/H.265 这类参考帧较多的协议需要 20+内存块能保证一定能解码。其它协议需要 10+内存块能保证一定能解码。

3. 如果在码流解码过程中发生了 info change，需要把已有的 MppBufferGroup 进行 reset，再 commit 进新的图像缓存，同时外部的显示也需要相应调整。

## 3.4 编码器数据流接口

编码器接口为用户提供了输入图像，输出码流的功能，接口函数为 MppApi 结构体里的 encode_put_frame 函数， encode_get_packet 函数和 encode 函数。这组函数提供了简洁的编码功能支持。

### 3.4.1 encode_put_frame

| 接口定义 | MPP_RET encode_put_frame(MppCtx ctx, MppFrame frame) |
|---|---|
| 输入参数 | ctx ： MPP 解码器实例<br>frame ： 待输入的图像数据 |
| 返回参数 | 运行错误码 |
| 功能 | 输入 frame 图像数据给 ctx 指定的 MPP 编码器实例 |

函数的工作模式
由于编码器的输入图像一般都比较大，如果进行拷贝，效率会大幅度下降，所以编码器的输入函数需要等待编码器硬件完成对输入图像内存的使用，才能把输入函数返回，把使用后的图像归还给调用者。基于以上的考虑，encode_put_frame 是阻塞式函数，会把调用者阻塞住，直到输入图像使用完成，会一定程度上导致软硬件运行无法并行，效率下降。

拷贝与零拷贝输入
编码器的输入不支持 CPU 分配的空间，如果需要支持编码 CPU 分配的地址，需要分配 MppBuffer 并把数据拷贝进去，这样做会很大程度影响效率。编码器更喜欢 dmabuf/ion/drm 内存形式的输入，这样可以实现零拷贝的编码，额外的系统开销最小。

### 3.4.2 encode_get_packet

| 接口定义 | MPP_RET encode_get_packet(MppCtx ctx, MppPacket *packet) |
|---|---|
| 输入参数 | ctx ： MPP 解码器实例<br>packet ： 用于获取 MppPacket 实例的指针。 |
| 返回参数 | 运行错误码 |
| 功能 | 从 ctx 指定的 MPP 编码器实例里获取完成编码的 packet 描述信息。 |

取头信息与图像数据
以 H.264 编码器为例，编码器的输出数据分为头信息（sps/pps）和图像数据（I/P slice）两部分，头信息需要通过 control 接口的 MPP_ENC_GET_EXTRA_INFO 命令获取，图像数据则是通过 encode_get_packet 接口来获取。获取的时机是在 control 接口的 SET_RC_CFG/SET_PREP_CFG/SET_CODEC_CFG 参数配置命令完成之后。在参数配置命令调用时，编码器会进行各个参数的更新，在更新全部完成之后，调用 MPP_ENC_GET_EXTRA_INFO 获取到的头信息才是最新的。

H.264 编码器输出码流的格式
目前硬件固定输出带 00 00 00 01 起始码的码流，所以 encode_get_packet 函数获取到码流都是带有 00 00 00 01 起始码。如果需要去掉起始码，可以从起始码之后的地址进行拷贝。

码流数据的零拷贝
由于使用 encode_put_frame 和 encode_get_packet 接口时没有提供配置输出缓存的方法，所以使用 encode_get_packet 时一定会进行一次拷贝。一般来说，编码器的输出码流相对于输入图像不算大，

数据拷贝可以接受。如果需要使用零拷贝的接口，需要使用 enqueue/dequeue 接口以及 MppTask 结构。

### 3.4.3 encode

暂未实现

## 3.5 编码器控制接口

编码器与解码器不同，需要用户进行一定的参数配置。编码器需要用户通过 control 接口配置编码器配置信息之后才可以进行编码工作。

### 3.5.1 control 与 MppEncCfg

MPP 推荐使用封装后的 MppEncCfg 结构通过 control 接口的 MPP_ENC_SET_CFG/MPP_ENC_GET_CFG 命令来进行编码器信息配置。

由于编码器可配置的选项与参数繁多，使用固定结构体容易出现接口结构体频繁变化，导致接口二进制兼容性无法得到保证，版本管理复杂，极大增加维护量。

为了缓解这个问题 MppEncCfg 使用(void *)作为类型，使用<字符串-值>进行 key map 式的配置，函数接口分为 s32/u32/s64/u64/ptr，对应的接口函数分为 set 与 get 两组，如下：

```
MPP_RET mpp_enc_cfg_set_s32(MppEncCfg cfg, const char *name, RK_S32 val);

MPP_RET mpp_enc_cfg_set_s64(MppEncCfg cfg, const char *name, RK_S64 val);
MPP_RET mpp_enc_cfg_set_u64(MppEncCfg cfg, const char *name, RK_U64 val);
MPP_RET mpp_enc_cfg_set_ptr(MppEncCfg cfg, const char *name, void *val);

MPP_RET mpp_enc_cfg_get_s32(MppEncCfg cfg, const char *name, RK_S32 *val);
MPP_RET mpp_enc_cfg_get_u32(MppEncCfg cfg, const char *name, RK_U32 *val);
MPP_RET mpp_enc_cfg_get_s64(MppEncCfg cfg, const char *name, RK_S64 *val);
MPP_RET mpp_enc_cfg_get_u64(MppEncCfg cfg, const char *name, RK_U64 *val);
MPP_RET mpp_enc_cfg_get_ptr(MppEncCfg cfg, const char *name, void **val);
```

字符串一般用[类型:参数]的方式进行定义，可支持的字符串与参数类型如下：

| 参数字串 | 接口 | 实际类型 | 描述说明 |
|---|---|---|---|
| rc:mode | S32 | MppEncRcMode | 表示码率控制模式，目前支持 CBR 和 VBR 两种：<br>CBR 为 Constant Bit Rate，固定码率模式。<br>在固定码率模式下，目标码率起决定性作用。<br>VBR 为 Variable Bit Rate，可变码率模式。<br>在可变码率模式下，最大最小码率起决定性作用。<br>FIX_QP 为固定 QP 模式，用于调试和性能评估。 |

| | | | |
|---|---|---|---|
| | | | typedef enum MppEncRcMode_t {<br>    MPP_ENC_RC_MODE_VBR,<br>    MPP_ENC_RC_MODE_CBR,<br>    MPP_ENC_RC_MODE_FIXQP,<br>    MPP_ENC_RC_MODE_BUTT<br>} MppEncRcMode; |
| rc:bps_target | S32 | RK_S32 | 表示 CBR 模式下的目标码率。 |
| rc:bps_max | S32 | RK_S32 | 表示 VBR 模式下的最高码率。 |
| rc:bps_min | S32 | RK_S32 | 表示 VBR 模式下的最低码率。 |
| rc:fps_in_flex | S32 | RK_S32 | 表示输入帧率是否可变的标志位，默认为 0。<br>为 0 表示输入帧率固定，帧率计算方式为<br>fps_in_num/fps_in_denorm，可以表示分数帧率。<br>为 1 表示输入帧率可变，可变帧率的情况下，帧率不固定，对应的码率计算与分配的规则变为按实际时间进行计算。 |
| rc:fps_in_flex | S32 | RK_S32 | 表示输入帧率是否可变的标志位，默认为 0。<br>为 0 表示输入帧率固定，帧率计算方式为<br>fps_in_num/fps_in_denorm，可以表示分数帧率。<br>为 1 表示输入帧率可变，可变帧率的情况下，帧率不固定，对应的码率计算与分配的规则变为按实际时间进行计算。 |
| rc:fps_in_num | S32 | RK_S32 | 表示输入帧率分数值的分子部分,如为 0 表示默认 30fps。 |
| rc:fps_in_denorm | S32 | RK_S32 | 表示输入帧率分数值的分母部分。如为 0 表示为 1 |
| rc:fps_out_flex | S32 | RK_S32 | 表示输出帧率是否可变的标志位，默认为 0。<br>为 0 表示输出帧率固定，帧率计算方式为<br>fps_out_num/fps_out_denorm，可以表示分数帧率。<br>为 1 表示输出帧率可变，可变帧率的情况下，帧率不固定，对应的码流输出时间按实际时间进行计算。 |
| rc:fps_out_num | S32 | RK_S32 | 表示输出帧率分数值的分子部分,如为 0 表示默认 30fps。 |
| rc:fps_out_denorm | S32 | RK_S32 | 表示输出帧率分数值的分母部分。如为 0 表示为 1 |
| rc:gop | | RK_S32 | 表示 Group Of Picture，即两个 I 帧之间的间隔，含义如下。<br>0 – 表示只有一个 I 帧，其他帧均为 P 帧<br>1 – 表示全为 I 帧<br>2 – 表示序列为 I P I P I P…<br>3 – 表示序列为 I P P I P P I P P…<br>一般情况下，gop 选择为输入帧率的整数倍。 |
| rc:max_reenc_times | U32 | RK_U32 | 一帧图像最大重编码次数。 |
| prep:width | S32 | RK_S32 | 表示输入图像水平方向像素数，单位为像素个数。 |
| prep:height | S32 | RK_S32 | 表示输入图像垂直方向像素数，单位为像素个数。 |
| prep:hor_stride | S32 | RK_S32 | 表示输入图像垂直方向相邻两行之间的距离，单位为 byte 数。 |
| prep:ver_stride | S32 | RK_S32 | 表示输入图像分量之间的以行数间隔数，单位为 1。 |
| prep:format | S32 | MppFrameFormat | 表示输入图像色彩空间格式以及内存排布方式。 |
| prep:color | S32 | MppFrameColorSpace | 表示输入图像数据彩色空间范围。 |
| prep:range | S32 | MppFrameColorRange | 表示输入图像是 full range 还是 limit range |

<table>
<tr><td></td><td></td><td></td><td>/*<br> * MPEG vs JPEG YUV range.<br> */<br>typedef enum {<br>    MPP_FRAME_RANGE_UNSPECIFIED = 0,<br>    MPP_FRAME_RANGE_MPEG        = 1,<br>    MPP_FRAME_RANGE_JPEG        = 2,<br>    MPP_FRAME_RANGE_NB,<br>} MppFrameColorRange;<br><br>///< the normal 219*2^(n-8) "MPEG" YUV ranges<br>///< the normal     2^n-1   "JPEG" YUV ranges<br>///< Not part of ABI</td></tr>
<tr><td>prep:rotation</td><td>S32</td><td>MppEncRotationCfg</td><td>表示输入图像旋转属性，默认为 0，不旋转。<br>/*<br> * input frame rotation parameter<br> * 0 - disable rotation<br> * 1 - 90 degree<br> * 2 - 180 degree<br> * 3 - 270 degree<br> */<br>typedef enum MppEncRotationCfg_t {<br>    MPP_ENC_ROT_0,<br>    MPP_ENC_ROT_90,<br>    MPP_ENC_ROT_180,<br>    MPP_ENC_ROT_270,<br>    MPP_ENC_ROT_BUTT<br>} MppEncRotationCfg;</td></tr>
<tr><td>prep:mirroring</td><td>S32</td><td>RK_S32</td><td>表示输入图像镜像属性，默认为 0，不做镜像。<br>/*<br> * input frame mirroring parameter<br> * 0 - disable mirroring<br> * 1 - horizontal mirroring<br> * 2 - vertical mirroring<br> */</td></tr>
<tr><td>codec:type</td><td>S32</td><td>MppCodingType</td><td>表示 MppEncCodecCfg 对应的协议类型，需要与 MppCtx 初始化函数 mpp_init 的参数一致。</td></tr>
<tr><td>h264:stream_type</td><td>S32</td><td>RK_S32</td><td>表示输入 H.264 的码流格式类型，默认为 0。<br>0 – 表示 Annex B 格式，即加入 00 00 00 01 的起始码。<br>1 – 表示没有起始码的格式。<br>目前内部固定为带 00 00 00 01 起始码的格式。</td></tr>
<tr><td>h264:profile</td><td>S32</td><td>RK_S32</td><td>表示 SPS 中的 profile_idc 参数：<br>66 – 表示 Baseline profile。<br>77 – 表示 Main profile。<br>100 – 表示 High profile。</td></tr>
<tr><td>h264:level</td><td>S32</td><td>RK_S32</td><td>表示 SPS 中的 level_idc 参数，其中 10 表示 level 1.0：<br>10/11/12/13 – qcif@15fps / cif@7.5fps / cif@15fps / cif@30fps<br>20/21/22 – cif@30fps / half-D1@25fps / D1@12.5fps<br>30/31/32 – D1@25fps / 720p@30fps / 720p@60fps<br>40/41/42 – 1080p@30fps / 1080p@30fps / 1080p@60fps<br>50/51/52 – 4K@30fps / 4K@30fps / 4K@60fps<br>一般配置为 level 4.1 即可满足要求。</td></tr>
<tr><td>h264:cabac_en</td><td>S32</td><td>RK_S32</td><td>表示编码器使用的熵编码格式：</td></tr>
</table>

| | | | |
|---|---|---|---|
| | | | 0 – CAVLC，自适应变长编码。<br>1 – CABAC，自适应算术编码。 |
| h264:cabac_idc | S32 | RK_S32 | 表示协议语法中的 cabac_init_idc，在 cabac_en 为 1 时有效，有效值为 0~2。 |
| h264:trans8x8 | S32 | RK_S32 | 表示协议语法中的 8x8 变换使能标志。<br>0 – 为关闭，在 Baseline/Main profile 时固定关闭。<br>1 – 为开启，在 High profile 时可选可启。 |
| h264:const_intra | S32 | RK_S32 | 表示协议语法中 constrained_intra_pred_mode 模式使能标志。<br>0 – 为关闭，1 – 为开启。 |
| h264:scaling_list | S32 | RK_S32 | 表示协议语法中 scaling_list_matrix 模式<br>0 – 为 flat matrix，1 – 默认 matrix。 |
| h264:cb_qp_offset | S32 | RK_S32 | 表示协议语法中 chroma_cb_qp_offset 值。<br>有效范围为[-12, 12]。 |
| h264:cr_qp_offset | S32 | RK_S32 | 表示协议语法中 chroma_cr_qp_offset 值。<br>有效范围为[-12, 12]。 |
| h264:dblk_disable | S32 | RK_S32 | 表示协议语法中 deblock_disable 标志，有效范围为[0, 2]。<br>0 – deblocking 使能。<br>1 – deblocking 关闭。<br>2 – 在 slice 边界关闭 deblocking。 |
| h264:dblk_alpha | S32 | RK_S32 | 表示协议语法中 deblock_offset_alpha 值。<br>有效范围为[-6, 6]。 |
| h264:dblk_beta | S32 | RK_S32 | 表示协议语法中 deblock_offset_beta 值。<br>有效范围为[-6, 6]。 |
| h264:qp_init | S32 | RK_S32 | 表示初始 QP 值，一般情况请勿配置。 |
| h264:qp_max | S32 | RK_S32 | 表示最大 QP 值，一般情况请勿配置。 |
| h264:qp_min | S32 | RK_S32 | 表示最小 QP 值，一般情况请勿配置。 |
| h264:qp_max_i | S32 | RK_S32 | 表示最大 I 帧 QP 值，一般情况请勿配置。 |
| h264:qp_min_i | S32 | RK_S32 | 表示最小 I 帧 QP 值，一般情况请勿配置。 |
| h264:qp_step | S32 | RK_S32 | 表示相临两帧之间的帧级 QP 变化幅度。 |
| h265:profile | S32 | RK_S32 | 表示 VPS 中的 profile_idc 参数：<br>固定为 1，Main profile |
| h265:level | S32 | RK_S32 | 表示 VPS 中的 level_idc 参数 |
| h265:scaling_list | S32 | RK_S32 | 表示协议语法中 scaling_list_matrix 模式<br>0 – 为 flat matrix，1 – 默认 matrix。 |
| h265:cb_qp_offset | S32 | RK_S32 | 表示协议语法中 chroma_cb_qp_offset 值。<br>有效范围为[-12, 12]。 |
| h265:cr_qp_offset | S32 | RK_S32 | 表示协议语法中 chroma_cr_qp_offset 值。<br>有效范围为[-12, 12]。 |
| h265:dblk_disable | S32 | RK_S32 | 表示协议语法中 deblock_disable 标志，有效范围为[0, 2]。<br>0 – deblocking 使能。<br>1 – deblocking 关闭。<br>2 – 在 slice 边界关闭 deblocking。 |
| h265:dblk_alpha | S32 | RK_S32 | 表示协议语法中 deblock_offset_alpha 值。<br>有效范围为[-6, 6]。 |

| | | | |
|---|---|---|---|
| h265:dblk_beta | S32 | RK_S32 | 表示协议语法中 deblock_offset_beta 值。<br>有效范围为[-6, 6]。 |
| h265:qp_init | S32 | RK_S32 | 表示初始 QP 值，一般情况请勿配置。 |
| h265:qp_max | S32 | RK_S32 | 表示最大 QP 值，一般情况请勿配置。 |
| h265:qp_min | S32 | RK_S32 | 表示最小 QP 值，一般情况请勿配置。 |
| h265:qp_max_i | S32 | RK_S32 | 表示最大 I 帧 QP 值，一般情况请勿配置。 |
| h265:qp_min_i | S32 | RK_S32 | 表示最小 I 帧 QP 值，一般情况请勿配置。 |
| h265:qp_step | S32 | RK_S32 | 表示相临两帧之间的帧级 QP 变化幅度。 |
| h265:qp_delta_ip | S32 | RK_S32 | 表示 I 帧与前一个 P 帧的 QP 差值。 |
| jpeg: quant | S32 | RK_S32 | 表示 JPEG 编码器使用的量化参数等级，编码器一共内置了 11 级量化系数表格，从 0 到 10，图像质量从差到好。 |
| split:mode | U32 | MppEncSplitMode | 表示 H.264/H.265 协议的 slice 切分模式<br>`typedef enum MppEncSplitMode_e {`<br>`MPP_ENC_SPLIT_NONE,`<br>`MPP_ENC_SPLIT_BY_BYTE,`<br>`MPP_ENC_SPLIT_BY_CTU,`<br>`} MppEncSplitMode;`<br>0– 不切分。<br>1– BY_BYTE 切分 slice 根据 slice 大小。<br>2– BY_CTU 切分 slice 根据宏块或 CTU 个数。 |
| split:arg | U32 | RK_U32 | Slice 切分参数：<br>在 BY_BYTE 模式下，参数表示每个 slice 的最大大小。<br>在 BY_CTU 模式下，参数表示每个 slice 包含的宏块或 CTU 个数。 |

其他的字符串与参数会进行后续扩展。

### 3.5.2 control 其他命令

在定义于 rk_mpi_cmd.h 文件的 MpiCmd 枚举类型定义了 control 接口命令字，其中与编码器和编码过程相关的命令如下：

```
MPP_ENC_CMD_BASE                = CMD_MODULE_CODEC | CMD_CTX_ID_ENC,
/* basic encoder setup control */
MPP_ENC_SET_CFG,                /* set MppEncCfg structure */
MPP_ENC_GET_CFG,                /* get MppEncCfg structure */
MPP_ENC_SET_PREP_CFG,           /* deprecated set MppEncPrepCfg structure, use MPP_ENC_SET_CFG instead */
MPP_ENC_GET_PREP_CFG,           /* deprecated get MppEncPrepCfg structure, use MPP_ENC_GET_CFG instead */
MPP_ENC_SET_RC_CFG,             /* deprecated set MppEncRcCfg structure, use MPP_ENC_SET_CFG instead */
MPP_ENC_GET_RC_CFG,             /* deprecated get MppEncRcCfg structure, use MPP_ENC_GET_CFG instead */
MPP_ENC_SET_CODEC_CFG,          /* deprecated set MppEncCodecCfg structure, use MPP_ENC_SET_CFG instead */
MPP_ENC_GET_CODEC_CFG,          /* deprecated get MppEncCodecCfg structure, use MPP_ENC_GET_CFG instead */
/* runtime encoder setup control */
MPP_ENC_SET_IDR_FRAME,          /* next frame will be encoded as intra frame */
MPP_ENC_SET_OSD_LEGACY_0,       /* deprecated */
MPP_ENC_SET_OSD_LEGACY_1,       /* deprecated */
MPP_ENC_SET_OSD_LEGACY_2,       /* deprecated */
MPP_ENC_GET_HDR_SYNC,           /* get vps / sps / pps which has better sync behavior parameter is MppPacket */
MPP_ENC_GET_EXTRA_INFO,         /* deprecated */
MPP_ENC_SET_SEI_CFG,            /* SEI: Supplement Enhancemant Information, parameter is MppSeiMode */
MPP_ENC_GET_SEI_DATA,           /* SEI: Supplement Enhancemant Information, parameter is MppPacket */
MPP_ENC_PRE_ALLOC_BUFF,         /* allocate buffers before encoding */
MPP_ENC_SET_QP_RANGE,           /* used for adjusting qp range, the parameter can be 1 or 2 */
MPP_ENC_SET_ROI_CFG,            /* set MppEncROICfg structure */
MPP_ENC_SET_CTU_QP,             /* for H265 Encoder,set CTU's size and QP */
```

从 MPP_ENC_CMD_BASE 到 MPP_ENC_CMD_END 之间的命令为编码器的 control 接口命令，其中配置命令的 MPP_ENC_SET/GET_ CFG 已经做为基本的配置命令在 3.5.1 进行了介绍。剩下的命令在下面进行简要的介绍，其中的命令与编码器硬件相关，只有部分硬件支持。

目前 MPP 支持的编码器硬件分为 vepu 系列和 rkvenc 系列，vepu 系列支持 H.264 编码，vp8 编码和 jpeg 编码，配备于绝大多数 RK 芯片中。rkvenc 系列只支持 H.264 编码，目前只配备于 RV1109/RV1126

系统芯片，其支持的编码功能相对于 vepu 系统会更多更强。

部分 CMD 命令简要说明：
~~MPP_ENC_SET_PREP_CFG/ MPP_ENC_GET_PREP_CFG~~
~~MPP_ENC_SET_RC_CFG/ MPP_ENC_GET_RC_CFG~~
~~MPP_ENC_SET_CODEC_CFG/ MPP_ENC_GET_CODEC_CFG~~
废弃命令，为了前向兼容保留，不要使用

MPP_ENC_SET_IDR_FRAME
无命令参数，用于向编码器请求 I 帖，编码器收到请求之后，会将待编码的下一帧编码为 IDR 帧。全部硬件都支持。

~~MPP_ENC_SET_OSD_LEGACY_0~~
~~MPP_ENC_SET_OSD_LEGACY_1~~
~~MPP_ENC_SET_OSD_LEGACY_2~~
废弃命令，前向兼容用保留，不要使用

MPP_ENC_GET_HDR_SYNC/ ~~MPP_ENC_GET_EXTRA_INFO~~
用于单独获取码流头数据的命令，其中 MPP_ENC_GET_EXTRA_INFO 为旧命令，不推荐使用。
MPP_ENC_GET_HDR_SYNC 输入参数为 MppPacket，需要外部用户分配好空间并封装为 MppPacket 再 control 到编码器，control 接口调用返回时就完成了数据拷贝，线程安全。调用时机在编码器基本配置完成之后。需要用户手动释放之前分配的 MppPacket。
MPP_ENC_GET_EXTRA_INFO 输入参数为 MppPacket*，会获取编码器的内部 MppPacket 来进行访问。调用时机在编码器基本配置完成之后。需要注意的是，这里得到的 MppPacket 是 MPP 的内部空间，不需要用户释放。
由于在多线程情况下，MPP_ENC_GET_EXTRA_INFO 命令获取的 MppPacket 有可能在读取时被其他 control 修改，所以这个命令并不是线程安全的，仅做为旧 vpu_api 的兼容用，不要再使用。

~~MPP_ENC_SET_SEI_CFG/MPP_ENC_GET_SEI_DATA~~
废弃命令，前向兼容用保留，不要使用

~~MPP_ENC_PRE_ALLOC_BUFF/ MPP_ENC_SET_QP_RANGE/ MPP_ENC_SET_ROI_CFG/ MPP_ENC_SET_CTU_QP~~
废弃命令，前向兼容用保留，不要使用

MPP_ENC_GET_RC_API_ALL
获取 MPP 目前支持的码率控制策略 API 信息的接口，输入 RcApiQueryAll*指针，在返回时填充好结构体内容

MPP_ENC_GET_RC_API_BY_TYPE
获取指定 MppCodingType 类型的所有码率控制策略 API 信息，输入 RcApiQueryType*指针并指定 MppCodingType，在返回时会填充好结构体内容。

MPP_ENC_SET_RC_API_CFG
注册外部码率控制策略 API，输入 RcImplApi*指针，该结构中的函数指针定义了码率控制策略插件的行为，注册之后的码率控制策略才可以被查询和激活使用。

MPP_ENC_GET_RC_API_CURRENT
返回当前使用的码率控制策略 API 信息，输入 RcApiBrief*指针，在返回时会填充好结构体内容。

MPP_ENC_SET_RC_API_CURRENT
激活指定名字的码率控制策略 API，输入 RcApiBrief*指针，编码器会搜索到 RcApiBrief 中指定字符串名字的码率控制策略 API 并激活为当前码率控制策略。

~~MPP_ENC_SET_HEADER_MODE/MPP_ENC_GET_HEADER_MODE~~
配置和获取 H.264/H.265 编码器的 SEI 调试信息输出方式，调试用开关，以后会被环境变量取代，不要使用。

~~MPP_ENC_SET_SPLIT/ MPP_ENC_GET_SPLIT~~
配置和获取 H.264/H265 编码器的 slice 切分配置信息，已被 MppEncCfg 中的 split:mode 和 split:arg 取代，不要使用

MPP_ENC_SET_REF_CFG
配置编码器高级参考帧模式，默认不需要配置，在需要配置长期参考帧，短期参考帧参考关系模式时使用，用于配置特殊的参考关系模式。高级接口，文档待完善。

MPP_ENC_SET_OSD_PLT_CFG
用于配置 rkvenc 系列硬件的 OSD 调色板，命令参数为 MppEncOSDPlt。一般只在编码开始时配置一次，全编码过程使用统一的调色板。仅 RV1109/RV1126 系列支持。

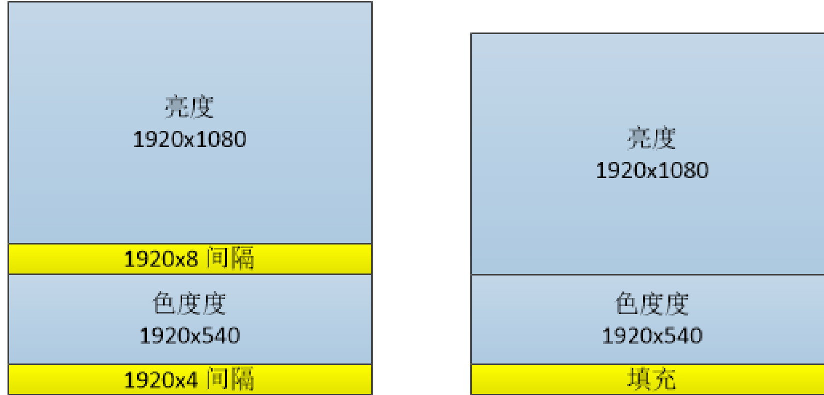
MPP_ENC_GET_OSD_PLT_CFG
用于获取 rkvenc 系列硬件的 OSD 调色板，命令参数为 MppEncOSDPlt*。一般不使用

~~MPP_ENC_SET_OSD_DATA_CFG~~
用于配置 rkvenc 系列硬件的 OSD 数据，命令参数为 MppEncOSDData。需要每帧进行配置，每编码一帧之后需要重新配置。本命令被 MppFrame 带的 MppMeta 中的 KEY_OSD_DATA 进行替代，不再使用。

## 3.6 编码器使用要点

### 3.6.1 输入图像的宽高与 stride

编码器的输入图像宽高配置需要与图像数据在内存中的排布一致。以 1920x1080 大小的 YUV420 图像编码为例，参考图表 7 MppFrame 重要参数说明的内容，假设有两种情况如下：




图表 18 编码器输入帧内存排布

左图情况：亮度分量的宽度为 1920，高度为 1080，亮度数据与色度数据不直接相接，中间有 8 行的空行。
这种情况下，水平 stride 为 1920，垂直 stride 为 1088，应用需要以 1920*1088*3/2 的大小分配空间并写入数据，使用宽 1920，高 1080，水平 stride 1920，垂直 stride 1088 的配置即可以正常进行编码。

右图情况：亮度分量的宽度为 1920，高度为 1080，亮度数据与色度数据直接相接，中间没有空行。
这种情况下，水平 stride 为 1920，垂直 stride 为 1080，但由于编码器对数据的访问是 16 对齐的，在读取亮度下边缘数据时会读取到色度部分，读取色度下边缘数据时会读取到色度数据之外的部分，需要用户开出额外的空间，这里的空间为 1920*1080*3/2+1920*4 的填充，才能保证编码器不出现访问未分配空间的情况。

### 3.6.2 编码器控制信息输入方式以及扩展

编码器控制信息的输入方式分为两种：
一种是全局性控制信息，如码率配置，宽高配置等，作用于整个编码器和编码过程；另一种是临时性控制信息，如每帧的 OSD 配置信息，用户数据信息等，只作用于单帧编码过程。
第一类控制信息主要通过 control 接口来进行配置，第二类控制信息主要是通过 MppFrame 所带的 MppMeta 接口来进行配置。
今后对控制信息的扩展也会遵循这两种规则来进行扩展。

### 3.6.3 编码器输入输出流程

目前编码器默认输入接口仅支持阻塞式调用，输出接口支持非阻塞式和阻塞式调用，默认为非阻塞式调用，有可能出现获取数据失败的情况，需要在使用中注意。

### 3.6.4 插件式自定义码率控制策略机制

MPP 支持用户自己定义码率控制策略，码率控制策略接口 RcImplApi 定义了几个编码处理流程上的钩子函数，用于在指定环节插入用户自定义的处理方法。具体使用方法可以参考默认的 H.264/H.265 码控策略实现（default_h264e/default_h265e 结构）。
码控插件机制在 MPP 中有预留，接口与流程都不算稳定，可以预见将来还会有不少调整，只建议有能力阅读理解代码，以及持续维护更新的用户使用这个机制，一般用户不建议使用。

# 第四章 MPP demo 说明

MPP 的 demo 程序变化比较快，以下说明仅供参考，具体情况以实际运行结果为准。Demo 的运行环境均以 Android 32bit 平台为准。

## 4.1 解码器 demo

解码器 demo 为 mpi_dec_test 系列程序，包括使用 decode_put_packet 和 decode_get_frame 接口的单线程 mpi_dec_test，多线程的 mpi_dec_mt_test 以及多实例的 mpi_dec_multi_test。

以下以 Android 平台上的 mpi_dec_test 为例进行使用说明。首先直接运行 mpi_dec_test，输入输出如下图：

```
1|shell@rk3228:/ # mpi_dec_test
01-01 12:13:07.460 I/mpi_dec_test( 1230): usage: mpi_dec_test [options]
01-01 12:13:07.460 I/utils    ( 1230): -i  input_file           input bitstream file
01-01 12:13:07.460 I/utils    ( 1230): -o  output_file          output bitstream file,
01-01 12:13:07.460 I/utils    ( 1230): -w  width                the width of input bitstream
01-01 12:13:07.460 I/utils    ( 1230): -h  height               the height of input bitstream
01-01 12:13:07.460 I/utils    ( 1230): -t  type                 input stream coding type
01-01 12:13:07.460 I/utils    ( 1230): -d  debug                debug flag
01-01 12:13:07.460 I/utils    ( 1230): -x  timeout              output timeout interval
01-01 12:13:07.460 I/utils    ( 1230): -n  frame_num            max decode frame number
1|shell@rk3228:/ # 01-01 12:13:07.460 I/mpi     ( 1230): mpp coding type support list:
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: mpeg2        id 2
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: mpeg4        id 4
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: h.263        id 3
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: h.264/AVC    id 7
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: h.265/HEVC   id 16777220
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: vp8          id 9
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: VP9          id 10
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: avs+         id 16777221
01-01 12:13:07.460 I/mpi      ( 1230): type: dec id 0 coding: jpeg         id 8
01-01 12:13:07.460 I/mpi      ( 1230): type: enc id 1 coding: h.264/AVC    id 7
01-01 12:13:07.460 I/mpi      ( 1230): type: enc id 1 coding: jpeg         id 8
```

mpi_dec_test 的命令参数中，输入文件（i），码流类型（t）为强制要求的参数，其他参数如输出文件（o），图像宽度（w）图像高度（h），解码帧数（n）等为可选参数，影响不大。
后面的打印显示了 MPP 库支持的 coding 格式，支持 MPEG2/4，H.263/4/5，VP8/9 的解码，id 后的数字为格式对应的-t 项后的参数值。参数值来源于 OMX 的定义，HEVC 和 AVS 的格式参数值与其他的格式参数值区别较大，需要留意。

然后以/sdcard/下的 tennis200.h264 解码 10 帧为例，对 demo 和输出进行说明。运行的命令为：
mpi_dec_test -t 7 -i /sdcard/tennis200.h264 -n 10
-t 7 表示是 H.264 码流，-i 表示输入文件，-n 10 表示解码 10 帧，如果一切正常，会得到如下的结果：

```
shell@rk3228:/ # mpi_dec_test -t 7 -i /sdcard/tennis200.h264 -n 10
01-01 12:18:29.320 I/mpi_dec_test( 1249): cmd parse result:
01-01 12:18:29.320 I/mpi_dec_test( 1249): input  file name: /sdcard/tennis200.h264
01-01 12:18:29.320 I/mpi_dec_test( 1249): output file name:
01-01 12:18:29.320 I/mpi_dec_test( 1249): width      :    0
01-01 12:18:29.320 I/mpi_dec_test( 1249): height     :    0
01-01 12:18:29.320 I/mpi_dec_test( 1249): type       : 7
01-01 12:18:29.320 I/mpi_dec_test( 1249): debug flag : 0
01-01 12:18:29.320 I/mpi_dec_test( 1249): mpi_dec_test start
01-01 12:18:29.320 I/mpi_dec_test( 1249): input file size 6854936
01-01 12:18:29.320 I/mpi_dec_test( 1249): mpi_dec_test decoder test start w 0 h 0 type 7
01-01 12:18:29.320 I/mpi     ( 1249): mpp version: aeb361f author: Herman Chen [cmake]: Remove static library VISIBILITY setting
01-01 12:18:29.330 I/hal_h264d_api( 1249): hal_h264d_init mpp_buffer_group_get_internal used ion In
01-01 12:18:29.330 I/mpp_rt  ( 1249): found ion allocator
01-01 12:18:29.330 I/mpp_rt  ( 1249): NOT found drm allocator
01-01 12:18:29.330 I/mpp_ion ( 1249): vpu_service iommu_enabled 1
01-01 12:18:29.330 I/mpp_ion ( 1249): using ion heap ION_HEAP_TYPE_SYSTEM
01-01 12:18:29.350 I/mpi_dec_test( 1249): decode_get_frame get info changed found
01-01 12:18:29.350 I/mpi_dec_test( 1249): decoder require buffer w:h [1920:1080] stride [1920:1088]
01-01 12:18:29.360 I/mpi_dec_test( 1249): decode_get_frame get frame 0
01-01 12:18:29.360 I/mpi_dec_test( 1249): decode_get_frame get frame 1
01-01 12:18:29.370 I/mpi_dec_test( 1249): decode_get_frame get frame 2
01-01 12:18:29.390 I/mpi_dec_test( 1249): decode_get_frame get frame 3
01-01 12:18:29.400 I/mpi_dec_test( 1249): decode_get_frame get frame 4
01-01 12:18:29.410 I/mpi_dec_test( 1249): decode_get_frame get frame 5
01-01 12:18:29.420 I/mpi_dec_test( 1249): decode_get_frame get frame 6
01-01 12:18:29.430 I/mpi_dec_test( 1249): decode_get_frame get frame 7
01-01 12:18:29.440 I/mpi_dec_test( 1249): decode_get_frame get frame 8
01-01 12:18:29.450 I/mpi_dec_test( 1249): decode_get_frame get frame 9
01-01 12:18:29.450 I/mpi_dec_test( 1249): test success
```

打印的信息里包含了 MPP 库的版本信息：
mpp version: aeb361f author: Herman Chen [cmake]: Remove static library VISIBILITY setting

mpp_rt 的内核分配器检测信息：
I/mpp_rt ( 1249): found ion allocator
I/mpp_rt ( 1249): NOT found drm allocator
I/mpp_ion ( 1249): vpu_service iommu_enabled 1
I/mpp_ion ( 1249): using ion heap ION_HEAP_TYPE_SYSTEM
表示找到了 ion 分配器，没有找到 drm 分配器，内核设备的 iommu 使能，使用 ion 的系统 heap。

I/mpi_dec_test( 1249): decode_get_frame get info changed found
为 mpi_dec_test 本身的打印，表示发现 MPP 解码器上报了 info change 事件。

I/mpi_dec_test( 1249): decoder require buffer w:h [1920:1080] stride [1920:1088]
为 mpi_dec_test 本身的打印，表示 MPP 解码器请求的图像内存情况。

I/mpi_dec_test( 1249): decode_get_frame get frame 0
为 mpi_dec_test 本身打印，表示解码器在正常解码和输出图像。

I/mpi_dec_test( 1249): test success
为 mpi_dec_test 本身打印，表示解码器完成了解码 10 帧的功能。

解码器的 demo 具体代码参见 test/mpi_dec_test.c。

## 4.2 编码器 demo

编码器 demo 为 mpi_enc_test 系列程序，包括单线程的 mpi_enc_test，及多实例的 mpi_enc_multi_test。

以下以 Android 平台上的 mpi_enc_test 为例进行使用说明。首先直接运行 mpi_enc_test，输出如下图：

```
130|shell@rk3228:/ # mpi_enc_test
01-01 12:25:20.840 I/mpi_enc_test( 1255): usage: mpi_enc_test [options]
01-01 12:25:20.840 I/utils    ( 1255): -i  input_file            input bitstream file
01-01 12:25:20.840 I/utils    ( 1255): -o  output_file           output bitstream file,
01-01 12:25:20.840 I/utils    ( 1255): -w  width                 the width of input picture
01-01 12:25:20.840 I/utils    ( 1255): -h  height                the height of input picture
01-01 12:25:20.840 I/utils    ( 1255): -f  format                the format of input picture
01-01 12:25:20.840 I/utils    ( 1255): -t  type                  output stream coding type
01-01 12:25:20.840 I/utils    ( 1255): -n  max frame number      max encoding frame number
01-01 12:25:20.840 I/utils    ( 1255): -d  debug                 debug flag
1|shell@rk3228:/ # 01-01 12:25:20.840 I/mpi     ( 1255): mpp coding type support list:
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: mpeg2       id 2
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: mpeg4       id 4
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: h.263       id 3
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: h.264/AVC   id 7
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: h.265/HEVC  id 16777220
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: vp8         id 9
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: VP9         id 10
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: avs+        id 16777221
01-01 12:25:20.840 I/mpi      ( 1255): type: dec id 0 coding: jpeg        id 8
01-01 12:25:20.840 I/mpi      ( 1255): type: enc id 1 coding: h.264/AVC   id 7
01-01 12:25:20.840 I/mpi      ( 1255): type: enc id 1 coding: jpeg        id 8
```

mpi_enc_test 的命令参数中，图像宽度（w）图像高度（h），码流类型（t）为强制要求参数，其他参数如输入文件（i），输出文件（o），编码帧数（n）等为可选参数。如果没有指定输入文件，mpi_enc_test 会生成默认彩条图像进行编码。

以编码/sdcard 下的 soccer_720x480_30fps.yuv 文件 10 帧为例，对 demo 和输出进行说明。运行的命令为：
mpi_enc_test -w 720 -h 480 -t 7 -i /sdcard/soccer_720x480_30fps.yuv -o /sdcard/out.h264 -n 10

然后使用 ls –l 查看输出的码流文件。

```
shell@rk3228:/ # mpi_enc_test -w 720 -h 480 -t 7 -i /sdcard/soccer_720x480_30f>
01-01 12:57:39.980 I/mpi_enc_test( 1284): cmd parse result:
01-01 12:57:39.980 I/mpi_enc_test( 1284): input  file name: /sdcard/soccer_720x480_30fps.yuv
01-01 12:57:39.980 I/mpi_enc_test( 1284): output file name: /sdcard/out.h264
01-01 12:57:39.980 I/mpi_enc_test( 1284): width      : 720
01-01 12:57:39.980 I/mpi_enc_test( 1284): height     : 480
01-01 12:57:39.980 I/mpi_enc_test( 1284): type       : 7
01-01 12:57:39.980 I/mpi_enc_test( 1284): debug flag : 0
01-01 12:57:39.980 I/mpi_enc_test( 1284): mpi_enc_test start
01-01 12:57:39.990 I/mpp_rt  ( 1284): found ion allocator
01-01 12:57:39.990 I/mpp_rt  ( 1284): NOT found drm allocator
01-01 12:57:39.990 I/mpp_ion ( 1284): vpu_service iommu_enabled 1
01-01 12:57:39.990 I/mpp_ion ( 1284): using ion heap ION_HEAP_TYPE_SYSTEM
01-01 12:57:39.990 I/mpi_enc_test( 1284): mpi_enc_test encoder test start w 720 h 480 type 7
01-01 12:57:39.990 I/mpi     ( 1284): mpp version: aeb361f author: Herman Chen [cmake]: Remove static library VISIBILITY setting
01-01 12:57:39.990 I/mpi_enc_test( 1284): mpi_enc_test bps 1296000 fps 30 gop 60
01-01 12:57:39.990 I/h264e_api( 1284): h264e_config MPP_ENC_SET_RC_CFG bps 1296000 [1215000 : 1377000]
01-01 12:57:40.010 I/mpi_enc_test( 1284): test_mpp_run encoded frame 0 size 63711
01-01 12:57:40.020 I/mpi_enc_test( 1284): test_mpp_run encoded frame 1 size 38005
01-01 12:57:40.020 I/mpi_enc_test( 1284): test_mpp_run encoded frame 2 size 3395
01-01 12:57:40.030 I/mpi_enc_test( 1284): test_mpp_run encoded frame 3 size 2558
01-01 12:57:40.040 I/mpi_enc_test( 1284): test_mpp_run encoded frame 4 size 2496
01-01 12:57:40.050 I/mpi_enc_test( 1284): test_mpp_run encoded frame 5 size 2235
01-01 12:57:40.050 I/mpi_enc_test( 1284): test_mpp_run encoded frame 6 size 2456
01-01 12:57:40.060 I/mpi_enc_test( 1284): test_mpp_run encoded frame 7 size 2593
01-01 12:57:40.070 I/mpi_enc_test( 1284): test_mpp_run encoded frame 8 size 2524
01-01 12:57:40.080 I/mpi_enc_test( 1284): test_mpp_run encoded frame 9 size 2577
01-01 12:57:40.080 I/mpi_enc_test( 1284): test_mpp_run encode max 10 frames
01-01 12:57:40.080 I/mpi_enc_test( 1284): mpi_enc_test success total frame 10 bps 2941200
shell@rk3228:/ # ls -l /sdcard/out.h264
-rwxrwxr-x system   sdcard_rw   122850 2011-01-01 12:57 out.h264
```

库以及环境相关的 log 同解码器 demo。

I/h264e_api( 1284): h264e_config MPP_ENC_SET_RC_CFG bps 1296000 [1215000 : 1377000]
编码器的码率控制参数配置，目标比特率为 1.3Mbps。

I/mpi_enc_test( 1284): test_mpp_run encoded frame 0 size 63711
编码器运行编码一帧，以及输出的一帧码流大小情况。

I/mpi_enc_test( 1284): mpi_enc_test success total frame 10 bps 2941200
编码器完成了 10 帧的编码，这 10 帧的码率为 2.9Mbps。注，这里编码不足 30 帧，码率有偏差，如果编码 30 帧，实际码率为 1.3Mbps。

编码器的 demo 具体代码参见 test/mpi_enc_test.c，但目前编码器 demo 使用的是 enqueue/dequeue 接口模式，后续会进行修改。

## 4.3 实用工具

MPP 提供了一些单元测试用的工具程序，这种程序可以对软硬件平台以及 MPP 库本身进行测试

mpp_info_test

用于读取和打印 MPP 库的版本信息，在反馈问题时，可以把打印出来信息附上。

mpp_buffer_test

用于测试内核的内存分配器是否正常。

mpp_mem_test

用于测试 C 库的内存分配器是否正常。

mpp_runtime_test

用于测试一些软硬件运行时环境是否正常。

mpp_platform_test

用于读取和测试芯片平台信息是否正常。

# 第五章 MPP 库编译与使用

## 5.1 下载源代码

MPP 源代码发布官方地址：https://github.com/rockchip-linux/mpp
发布分支为 release 分支，开发分支为 develop 分支，默认为开发分支。
下载命令：git clone https://github.com/rockchip-linux/mpp.git

## 5.2 编译

MPP 源代码编译脚本为 cmake，需要依赖 2.8.12 以上的版本，建议使用 2.8.12 版，使用高版本的 cmake 工具可能会产生较多的 warning。

### 5.2.1 Android 平台交叉编译

编译 Android 库需要使用 ndk 环境，默认脚本使用 android-ndk-r10d 进行编译。
r10d ndk 的下载路径可以在源代码目录下的 build/android/ndk_links.md 文件里查找。

把下载好的 ndk 解压到/home/pub/ndk/android-ndk-r10d，或者手动修改 build/android/目录下 env_setup.sh 脚本的 ANDROID_NDK 变量路径。

进入 build/android/arm/目录，运行 make-Android.bash 脚本生成编译用 Makefile，运行 make –j16 进行编译。

### 5.2.2 Unix/Linux 平台编译

先配置 build/linux/arm/目录下 arm.linux.cross.cmake 文件里的工具链，再运行 make-Makefiles.bash 脚本通过 cmake 生成 Makefile，最后运行 make –j16 进行编译。
MPP 也支持直接在开发板运行的 Debian 上编译。

# 第六章 常见问题 FAQ

Q：aarch64 编译出错，报错为 undefined reference to `__system_property_get'。
A：这是 google 64bit ndk 的问题，其 libc.so 中缺少一些符号定义，问题情况参见：
http://stackoverflow.com/questions/28413530/api-to-get-android-system-properties-is-removed-in-arm64-platforms
解决方法：MPP 中已经把对应的 libc.so 放到 build/android/aarch64/fix/目录下，把库拷贝到 path_to_ndk/platforms/android-21/arch-arm64/usr/lib/路径下，重新编译即正常。

Q：运行时会打印如下这样的内核 log，是不是有问题？
vpu_service_ioctl:1844: error: unknow vpu service ioctl cmd 40086c01
A：没有问题，mpp 对内核驱动有一些依赖，内核驱动有不同版本的接口，mpp 会进行多次尝试。如果遇到尝试失败会进行另外一种接口的尝试。这个打印是尝试失败时打印出来的，只会打印一次，可以忽略这个打印。

Q：MPP 运行异常如何分析？
A：首先先分析错误日志，如果出现打开内核设备失败的 log，需要先分析内核平台的视频编解码器硬件设备配置文件是否有使能，然后提交问题到 redmine 上，分析运行环境问题之后，再来分析 MPP 运行的内部问题。