

文档目录

MPP 开发参考	1
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 解码器接口	17
3.1.1 <i>decode_put_packet</i>	17
3.1.2 <i>decode_get_frame</i>	18
3.1.3 <i>decode</i>	19
3.1.4 <i>control</i>	19
3.2 解码器使用要点	20
3.2.1 解码器单/多线程使用方式	20
3.2.2 图像内存分配以及交互模式	20
3.3 编码器接口	23
3.3.1 <i>control</i> 编码配置命令	23
3.3.2 <i>encode_put_frame</i>	27
3.3.3 <i>encode_get_packet</i>	28
3.3.4 <i>encode</i>	28
3.3.5 <i>control</i> 其他命令	28
第四章 MPP DEMO 说明	30
4.1 解码器 DEMO	30
4.2 编码器 DEMO	31
4.3 实用工具	32
第五章 MPP 库编译与使用	33

5.1 下载源代码	33
5.2 编译.....	33
5.2.1 <i>Android</i> 平台交叉编译.....	33
5.2.2 <i>Unix/Linux</i> 平台编译.....	33
第六章 常见问题 FAQ.....	34

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.....	20
图表 13 解码器图像内存纯内部分配模式示意图.....	21
图表 14 解码器图像内存纯内部分配模式代码流程.....	21
图表 15 解码器图像内存半内部分配模式代码流程.....	22
图表 16 解码器图像内存纯外部分配模式示意图.....	22
图表 17 解码器图像内存纯外部分配模式代码流程.....	22
图表 18 编码器基本代码流程.....	23

第一章 MPP 介绍

1.1 概述

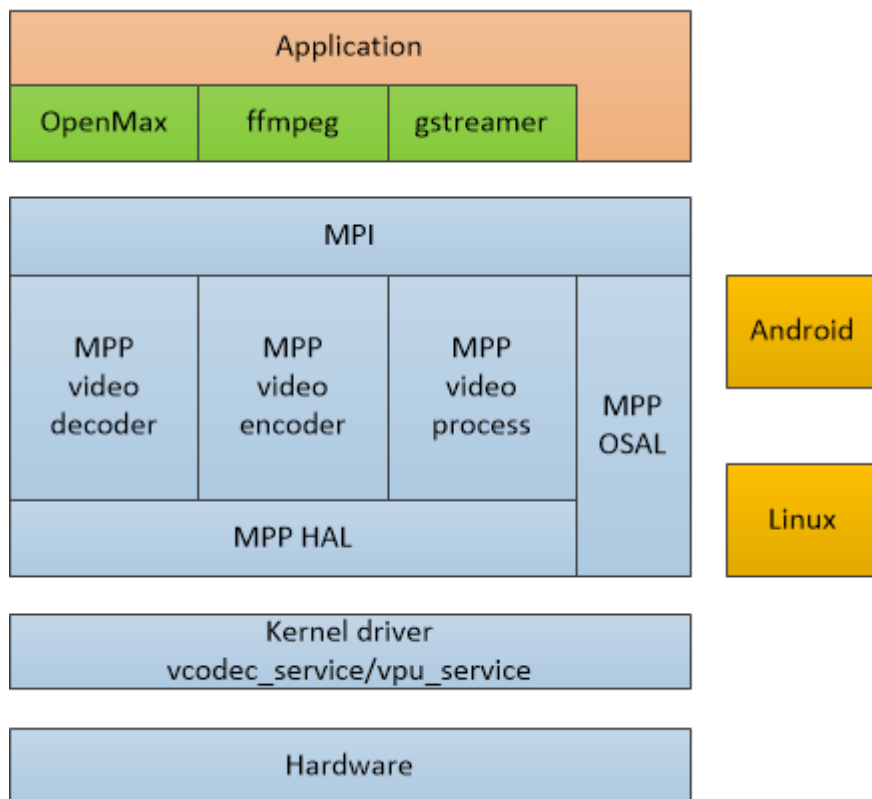
瑞芯微提供的媒体处理软件平台（Media Process Platform，简称 MPP）是适用于瑞芯微芯片系列的通用媒体处理软件平台。该平台对应用软件屏蔽了芯片相关的复杂底层处理，其目的是为了屏蔽不同芯片的差异，为用户提供统一的视频媒体处理接口（Media Process Interface，缩写 MPI）。MPP 提供的功能包括：

- 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：

RK3188 系列，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

RV1108 / RV1107 系列

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。

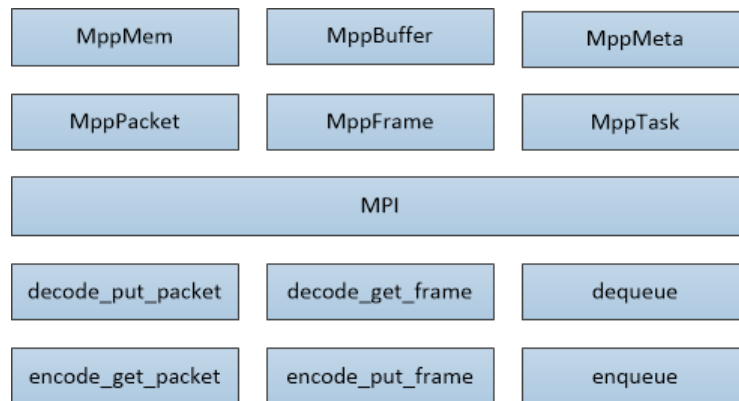
第二章接口设计说明

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。

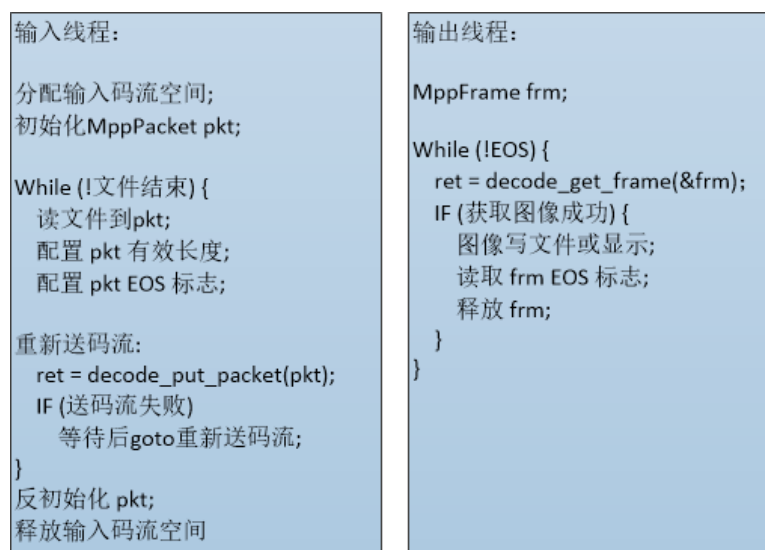
MppBuffer 为硬件用的 dmabuf 内存的封装。

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。

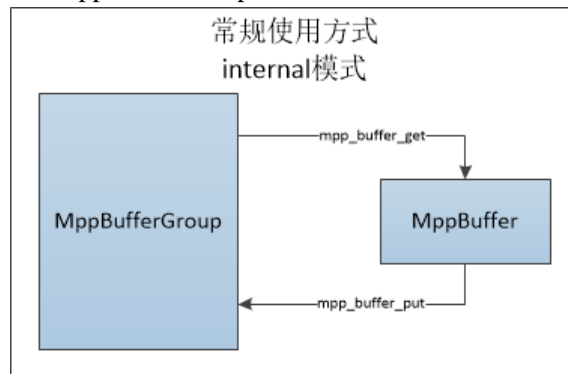
注意：以上这些接口数据结构都是使用 void*句柄来引用使用，其目的是为了更方便扩展和前向兼容。本段落中提到的成员都是通过形如 mpp_xxx_set/get_xxx 的接口来访问。

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：

```

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);

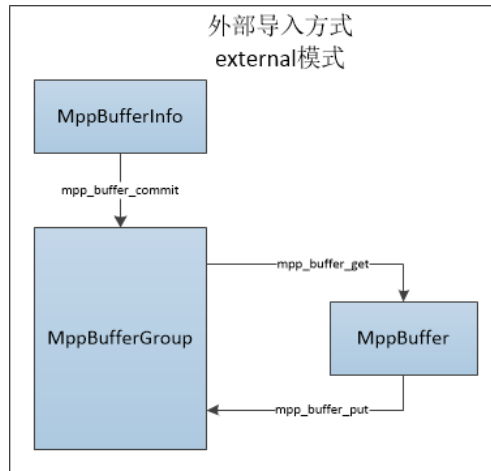
// 把缓存池配置给解码器，让解码器从缓存池中取缓存解码
mpi->control(dec, MPP_DEC_SET_EXT_BUF_GROUP, group);

// 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, &buffer_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：

```
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, &buffer_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);
```

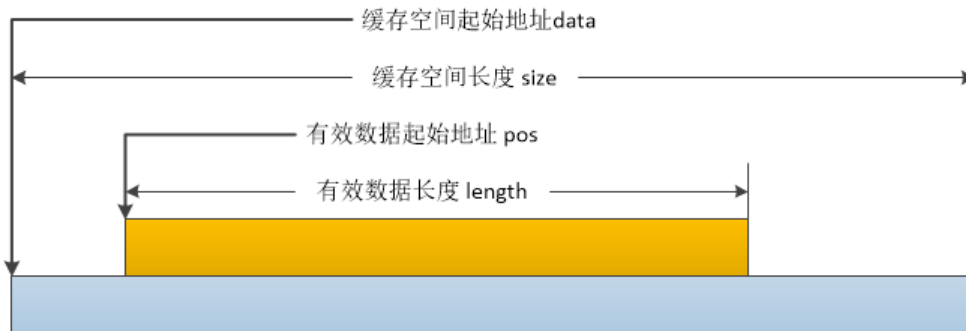
这种方式可以使得解码器使用外部的缓存，可以对接 OpenMax/ffmpeg/gstreamer 这样的中间件，也方便对接用户已有的上层代码，便于实现零拷贝显示。

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： #define MPP_PACKET_FLAG_EOS (0x00000001) #define MPP_PACKET_FLAG_EXTRA_DATA (0x00000002) #define MPP_PACKET_FLAG_INTERNAL (0x00000004) #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_packet_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_packet_deinit(&pkt); // <<-- 不会释放 data
mpp_packet_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

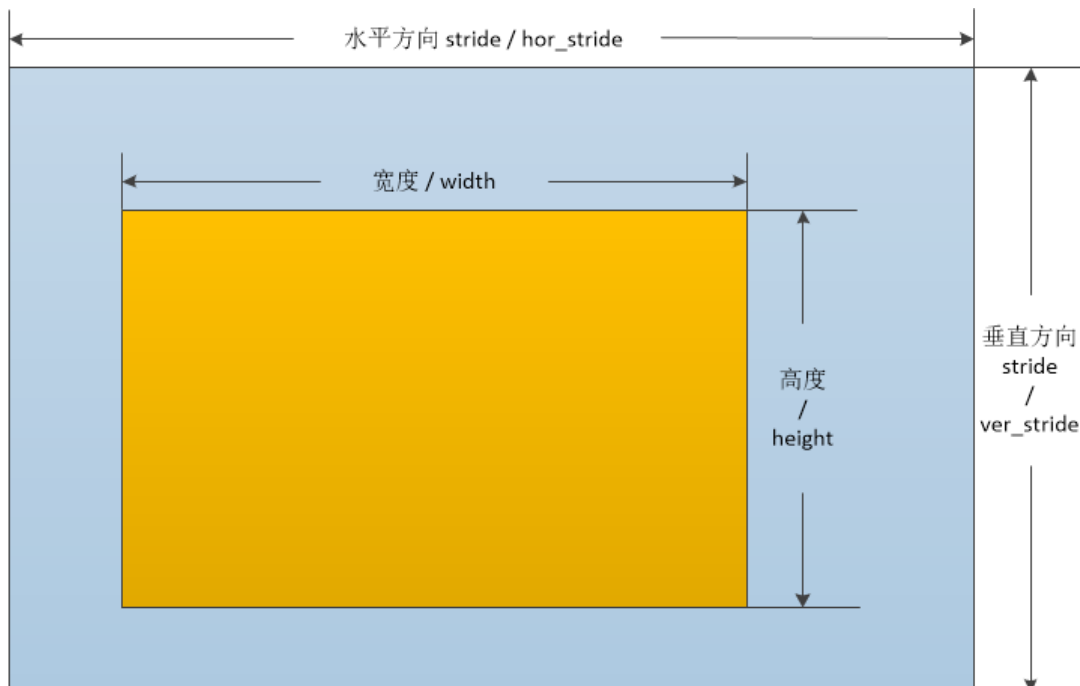
mpp_packet_deinit(&pkt); // <<-- 自动减引用

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:

成员名称	成员类型	描述说明
mode	RK_U32	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_PAIRIED_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>
pts	RK_U64	表示图像的显示时间戳 (Present Time Stamp)。
dts	RK_U64	表示图像的解码时间戳 (Decoding Time Stamp)。
eos	RK_U32	表示图像的结束标志 (End Of Stream)。
errinfo	RK_U32	表示图像的错误标志, 是否图像内有解码错误。
discard	RK_U32	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。
buf_size	size_t	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。
info_change	RK_U32	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。 可能的 info_change 原因有: 1. 图像序列宽高变化。 2. 图像序列格式变化, 如 8bit 变为 10bit。 一旦 info_change 产生, 需要重新分析解码器使用的内存池。
fmt	MppFrame Format	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, /* YUYVUYVY... (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_824 = 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>
color_range	MppFrame ColorRange	表示图像数据彩色空间范围: YUV full range: 0 ~ 255 (8bit)

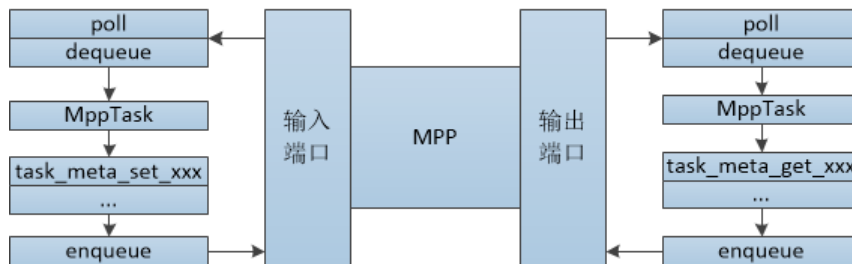
		<p>YUV limit range: 16 ~ 235 (8bit)</p> <pre> /* * MPEG vs JPEG YUV range. */ typedef enum { MPP_FRAME_RANGE_UNSPECIFIED = 0, MPP_FRAME_RANGE_MPEG = 1, ///< the normal 219*2^(n-8) "MPEG" YUV ranges MPP_FRAME_RANGE_JPEG = 2, ///< the normal 2^n-1 "JPEG" YUV ranges MPP_FRAME_RANGE_NB, ///< Not part of ABI } MppFrameColorRange; </pre>
buffer	MppBuffer	表示 MppFrame 对应的 MppBuffer。

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。

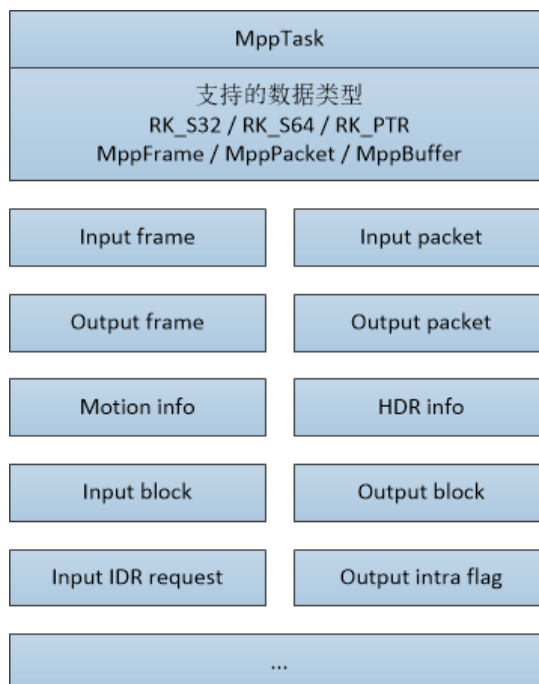
2.4 高级任务封装 MppTask

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/dequeue/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

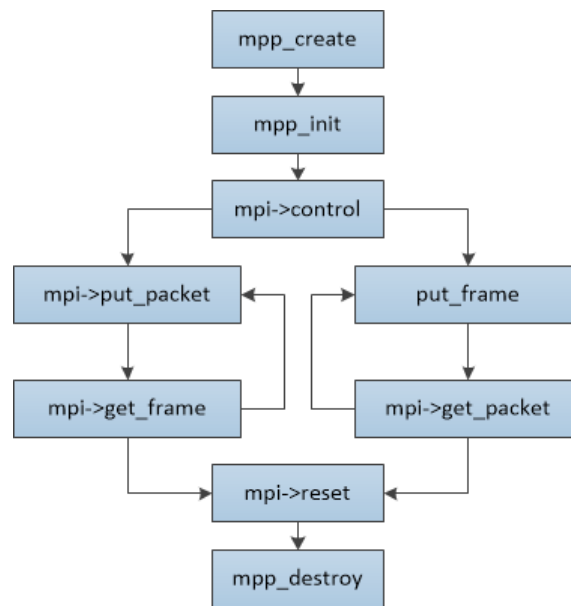
在实际使用中，需要从 MPP 的输入端口通过 dequeue 接口获取 MppTask，把需要处理的数据通过 mpp_task_meta_set_xxx 系列接口配置到 MppTask 里，之后 enqueue 输出到 MPP 实例进行处理。MPP 的输出端口流程类似，需要把 mpp_task_meta_set_xxx 系列接口换成 mpp_task_meta_get_xxx 系列接口来从 MppTask 里获取数据。

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/dequeue/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) 视频解码接口，同时进行输入与输出，单独使用。 ctx : MPP 实例上下文。 packet : 输入码流。 frame : 输出图像。 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) 视频解码输入接口，与 decode_get_frame 配合使用。 ctx : MPP 实例上下文。 packet : 输入码流。

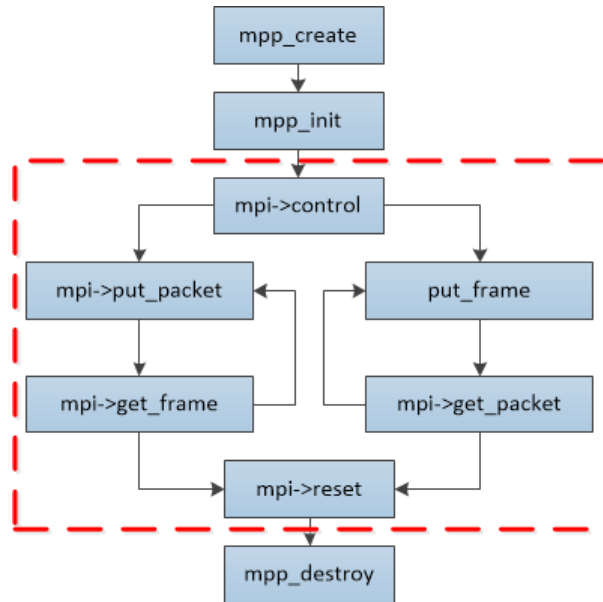
		返回值 : 0 为正常, 表示码流已被 MPP 处理; 非零为出现错误, 码流未被处理, 需要把码流重新输入。
decode_get_frame	函数指针	MPP_RET (*decode_get_frame)(MppCtx ctx, MppFrame *frame) 视频解码输出接口, 与 decode_put_packet 配合使用。 ctx : MPP 实例上下文。 frame : 输出图像。 返回值 : 0 为正常, 表示获取输出过程正常, 需要判断是否有得到有值的 frame 指针; 非零为出现错误。
encode	函数指针	MPP_RET (*encode)(MppCtx ctx, MppFrame frame, MppPacket *packet) 视频编码接口, 同时进行输入与输出, 单独使用。 ctx : MPP 实例上下文。 frame : 输入图像。 packet : 输出码流。 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) 视频编码输入接口, 与 encode_get_packet 配合使用。 ctx : MPP 实例上下文。 frame : 输入图像。 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) 视频编码输出接口, 与 encode_put_frame 配合使用。 ctx : MPP 实例上下文。 packet : 输出码流。 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) 端口查询接口, 用于查询端口是否有数据可供 dequeue。 ctx : MPP 实例上下文。 type : 端口类型, 分为输入端口与输出端口。 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。 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) 端口出队列接口, 用于从端口中取出 MppTask 结构。 ctx : MPP 实例上下文。 type : 端口类型, 分为输入端口与输出端口。 task : MppTask。 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) 端口入队列接口, 用于往端口送入 MppTask 结构。 ctx : MPP 实例上下文。 type : 端口类型, 分为输入端口与输出端口。 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	<p>MPP_RET (*reset)(MppCtx ctx)</p> <p>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。</p> <p>ctx : MPP 实例上下文。</p> <p>返回值 : 0 为正常, 有数据可取出, 非零为错误码。</p>
control	函数指针	<p>MPP_RET (*control)(MppCtx ctx, MpiCmd cmd, MppParam param)</p> <p>控制接口, 用于向 MPP 实例进行额外控制操作的接口。</p> <p>ctx : MPP 实例上下文。</p> <p>cmd : Mpi 命令类型, 表示控制命令的不同类型的。</p> <p>task : Mpi 命令参数, 表示控制控制命令的附加参数。</p> <p>返回值 : 0 为正常, 有数据可取出, 非零为错误码。</p>

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 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。 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 3.2.2 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。

3.1.4 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_VCI_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

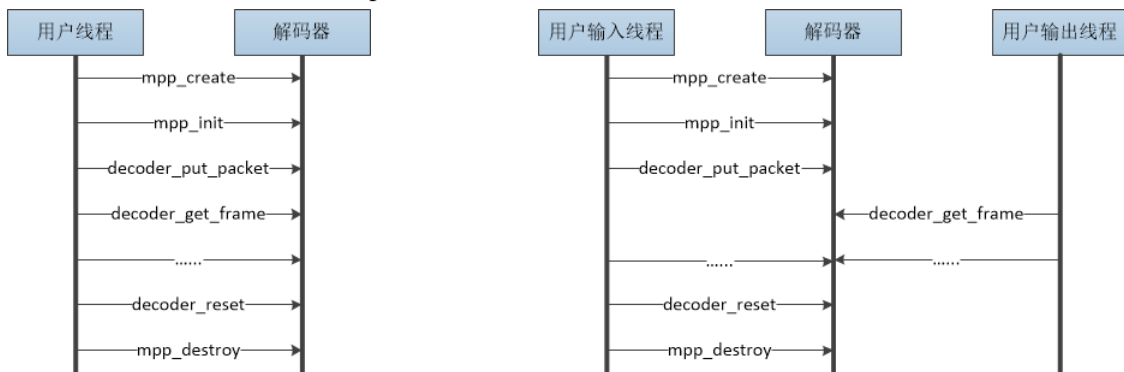
命令参数为 RK_U32*，用于使能 H.264 解码器的立即输出模式。一旦使能，H.264 解码器会忽略丢帧导致的帧序不连续情况，立即输出解码的图像。命令调用时机在 decode_put_packet 之前。

3.2 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：

3.2.1 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，可以在多线程环境下使用。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

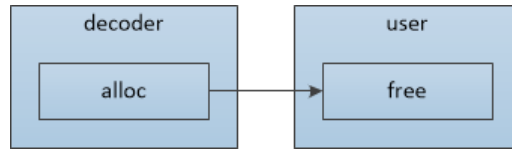
3.2.2 图像内存分配以及交互模式

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

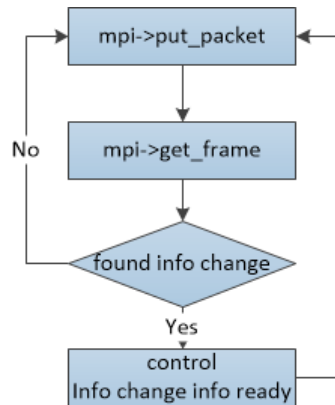
图像内存直接从 MPP 解码器内部分配，内存由解码器直接分配，用户得到解码器输出图像，在使用

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：

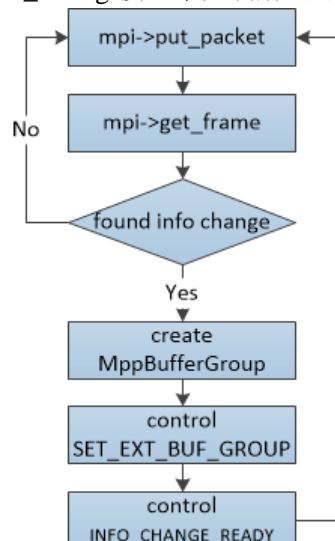
过程简单，可以快速完成一个可用的 demo，评估解码器的性能。

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:

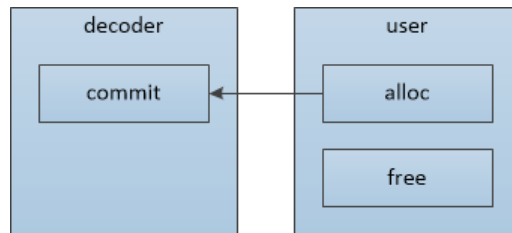
过程简单, 易上手, 可以一定程度限制内存的使用。

缺点:

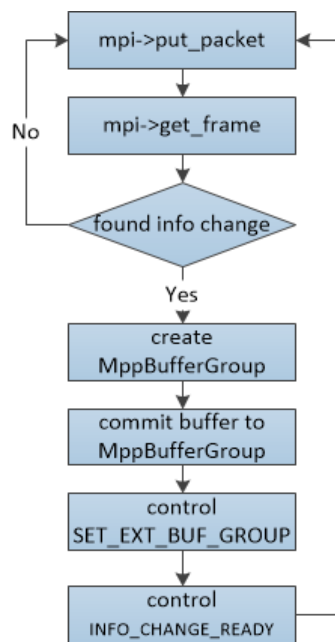
1. 内存空间的限制并不准确, 内存的使用量不是 100% 固定的, 会有波动。
2. 同样难于实现零拷贝的显示。

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: $hor_stride * ver_stride / 2$

- 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。
- 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。

3.3 编码器接口

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，同时 control 接口提供了编码器的配置能力。

3.3.1 control 编码配置命令

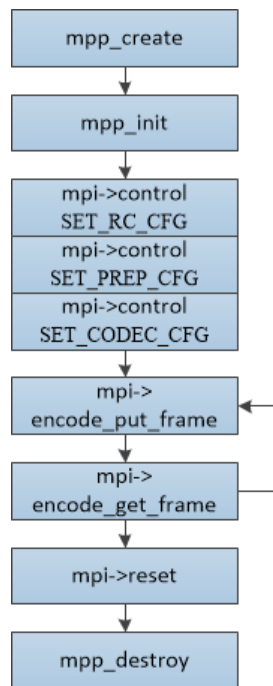
编码器与解码器不同，需要用户进行一定的参数配置。编码器需要用户通过 control 接口配置三类信息之后才可以进行编码工作。这三类信息分别为：

码率控制配置 (MppEncRcCfg)，通过 MPP_ENC_SET_RC_CFG 命令进行配置；

输入控制配置 (MppEncPrepCfg)，通过 MPP_ENC_SET_PREP_CFG 命令进行配置；

协议控制配置 (MppEncCodecCfg)，通过 MPP_ENC_SET_CODEEC_CFG 命令进行配置；

基本的流程如下图：



图表 18 编码器基本代码流程

三个配置中，码率控制配置和输入控制配置是必需的，协议控制配置是可选的高级配置。

码率控制配置 MppEncRcCfg 的重要参数成员如下：

成员名称	成员类型	描述说明
change	RK_U32	每比特表示是否更新特定的参数。 目的是为了标记出需要更新的参数，减少不必要的参数配置和参数比较。

		<pre>typedef enum MppEncRcCfgChange_e { MPP_ENC_RC_CFG_CHANGE_RC_MODE = (1 << 0), MPP_ENC_RC_CFG_CHANGE_QUALITY = (1 << 1), MPP_ENC_RC_CFG_CHANGE_BPS = (1 << 2), MPP_ENC_RC_CFG_CHANGE_FPS_IN = (1 << 5), MPP_ENC_RC_CFG_CHANGE_FPS_OUT = (1 << 6), MPP_ENC_RC_CFG_CHANGE_GOP = (1 << 7), MPP_ENC_RC_CFG_CHANGE_SKIP_CNT = (1 << 8), MPP_ENC_RC_CFG_CHANGE_ALL = (0xFFFFFFFF), } MppEncRcCfgChange;</pre>
rc_mode	MppEncRcMode	<p>表示码率控制模式，目前支持 CBR 和 VBR 两种： CBR 为 Constant Bit Rate，固定码率模式。 在固定码率模式下，目标码率起决定性作用。 VBR 为 Variable Bit Rate，可变码率模式。 在可变码率模式下，最大最小码率起决定性作用。</p> <pre>typedef enum MppEncRcMode_t { MPP_ENC_RC_MODE_VBR, MPP_ENC_RC_MODE_CBR, MPP_ENC_RC_MODE_BUTT } MppEncRcMode;</pre>
quality	MppEncRcQuality	<p>表示编码图像质量模式，共分为七档，主要在 VBR 模式下有生效，因为 CBR 模式为码率优先，图像质量模式不起作用。</p> <pre>typedef enum MppEncRcQuality_t { MPP_ENC_RC_QUALITY_WORST, MPP_ENC_RC_QUALITY_WORSE, MPP_ENC_RC_QUALITY_MEDIUM, MPP_ENC_RC_QUALITY_BETTER, MPP_ENC_RC_QUALITY_BEST, MPP_ENC_RC_QUALITY_CQP, MPP_ENC_RC_QUALITY_AQ_ONLY, MPP_ENC_RC_QUALITY_BUTT } MppEncRcQuality;</pre> <p>图像质量水平从 worst 到 best 逐档变好。 CQP 模式为固定 QP 模式，供 demo 使用。 AQ_ONLY 模式为自适应量化模式，在高码率下使用。</p>
bps_target	RK_S32	表示 CBR 模式下的目标码率。
bps_max	RK_S32	表示 VBR 模式下的最高码率。
bps_min	RK_S32	表示 VBR 模式下的最低码率。
fps_in_flex	RK_S32	<p>表示输入帧率是否可变的标志位，默认为 0。 为 0 表示输入帧率固定，帧率计算方式为 fps_in_num/fps_in_denorm，可以表示分数帧率。 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。</p>
fps_in_num	RK_S32	表示输入帧率分数值的分子部分，如为 0 表示默认 30fps。
fps_in_denorm	RK_S32	表示输入帧率分数值的分母部分。如为 0 表示为 1
fps_out_flex	RK_S32	<p>表示输出帧率是否可变的标志位，默认为 0。 为 0 表示输出帧率固定，帧率计算方式为 fps_out_num/fps_out_denorm，可以表示分数帧率。 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。</p>
fps_out_num	RK_S32	表示输出帧率分数值的分子部分，如为 0 表示默认 30fps。

fps_out_denorm	RK_S32	表示输出帧率分数值的分母部分。如为 0 表示为 1
gop	RK_S32	表示 Group Of Picture，即两个 I 帧之间的间隔，含义如下。 0 – 表示只有一个 I 帧，其他帧均为 P 帧 1 – 表示全为 I 帧 2 – 表示序列为 IPIPIP... 3 – 表示序列为 IPPIPPIPP... 一般情况下，gop 选择为输入帧率的整数倍。

输入控制配置 MppEncPrepCfg 的重要参数成员如下：

成员名称	成员类型	描述说明
change	RK_U32	每比特表示是否更新特定的参数。 目的是为了标记出需要更新的参数，减少不必要的参数配置和参数比较。 <pre>typedef enum MppEncPrepCfgChange_e { MPP_ENC_PREP_CFG_CHANGE_INPUT = (1 << 0), MPP_ENC_PREP_CFG_CHANGE_FORMAT = (1 << 2), /* transform parameter */ MPP_ENC_PREP_CFG_CHANGE_ROTATION = (1 << 4), MPP_ENC_PREP_CFG_CHANGE_MIRRORING = (1 << 5), /* enhancement parameter */ MPP_ENC_PREP_CFG_CHANGE_DENOISE = (1 << 8), MPP_ENC_PREP_CFG_CHANGE_SHARPEN = (1 << 9), MPP_ENC_PREP_CFG_CHANGE_ALL = (0xFFFFFFFF), } MppEncPrepCfgChange;</pre>
width	RK_S32	表示输入图像水平方向像素数，单位为像素个数。
height	RK_S32	表示输入图像垂直方向像素数，单位为像素个数。
hor_stride	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。
ver_stride	RK_S32	表示输入图像分量之间的以行数间隔数，单位为 1。
format	MppFrameFormat	表示输入图像色彩空间格式以及内存排布方式。
color	MppFrameColorSpace	表示输入图像数据彩色空间范围。
rotation	MppEncRotationCfg	表示输入图像旋转属性，默认为 0，不旋转。 <pre>/* * input frame rotation parameter * 0 - disable rotation * 1 - 90 degree * 2 - 180 degree * 3 - 270 degree */ typedef enum MppEncRotationCfg_t { MPP_ENC_ROT_0, MPP_ENC_ROT_90, MPP_ENC_ROT_180, MPP_ENC_ROT_270, MPP_ENC_ROT_BUTT } MppEncRotationCfg;</pre>
mirroring	RK_S32	表示输入图像镜像属性，默认为 0，不做镜像。 <pre>/* * input frame mirroring parameter * 0 - disable mirroring * 1 - horizontal mirroring * 2 - vertical mirroring */</pre>

协议控制配置 MppEncCodecCfg 的重要参数成员如下：

成员名称	成员类型	描述说明
------	------	------

coding	MppCodingType	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。
change	RK_U32	每比特表示是否更新特定的参数。
h264	MppEncH264Cfg	表示 H.264 编码器的协议参数配置。
h265	MppEncH265Cfg	表示 H.265 编码器的协议参数配置。
jpeg	MppEncJpegCfg	表示 JPEG 编码器的协议参数配置。
vp8	MppEncVp8Cfg	表示 VP8 编码器的协议参数配置。

在目前的 MPP 代码版本中，只支持 H.264 和 JPEG 编码器的协议参数配置，H.265 和 VP8 编码器的暂不支持。在正常情况下，MppEncCodecCfg 不需要进行配置，使用默认参数即可，如果有特殊需求或者测试评估，可以使用协议相关的配置对编码器的输出码流语法细节进行控制。

H.264 编码器协议配置 MppEncH264Cfg 的重要参数成员如下：

成员名称	成员类型	描述说明
change	RK_U32	每比特表示是否更新特定的参数，具体参见代码中 MppEncH264CfgChange 的说明。
stream_type	RK_S32	表示输入 H.264 的码流格式类型，默认为 0。 0 – 表示 Annex B 格式，即加入 00 00 00 01 的起始码。 1 – 表示没有起始码的格式。 目前内部固定为带 00 00 00 01 起始码的格式。
svc	RK_S32	表示是否编码为 SVC 格式，暂不支持。
profile	RK_S32	表示 SPS 中的 profile_idc 参数： 66 – 表示 Baseline profile。 77 – 表示 Main profile。 100 – 表示 High profile。
level	RK_S32	表示 SPS 中的 level_idc 参数，其中 10 表示 level 1.0： 10/11/12/13 – qcif@15fps / cif@7.5fps / cif@15fps / cif@30fps 20/21/22 – cif@30fps / half-D1@25fps / D1@12.5fps 30/31/32 – D1@25fps / 720p@30fps / 720p@60fps 40/41/42 – 1080p@30fps / 1080p@30fps / 1080p@60fps 50/51/52 – 4K@30fps / 4K@30fps / 4K@60fps 一般配置为 level 4.1 即可满足要求。
entropy_coding_mode	RK_S32	表示编码器使用的熵编码格式： 0 – CAVLC，自适应变长编码。 1 – CABAC，自适应算术编码。
cabac_init_idc	RK_S32	表示协议语法中的 cabac_init_idc，在 entropy_coding_mode 为 1 时有效，有效值为 0~2。
transform8x8_mode	RK_S32	表示协议语法中的 8x8 变换使能标志。 0 – 为关闭，在 Baseline/Main profile 时固定关闭。 1 – 为开启，在 High profile 时可选可启。
constrained_intra_pred_mode	RK_S32	表示协议语法中 constrained_intra_pred_mode 模式使能标志。 0 – 为关闭，1 – 为开启。
chroma_cb_qp_offset	RK_S32	表示协议语法中 chroma_cb_qp_offset 值。 有效范围为[-12, 12]。
chroma_cr_qp_offset	RK_S32	表示协议语法中 chroma_cr_qp_offset 值。 有效范围为[-12, 12]。

deblock_disable	RK_S32	表示协议语法中 deblock_disable 标志，有效范围为[0, 2]。 0 – deblocking 使能。 1 – deblocking 关闭。 2 – 在 slice 边界关闭 deblocking。
deblock_offset_alpha	RK_S32	表示协议语法中 deblock_offset_alpha 值。 有效范围为[-6, 6]。
deblock_offset_beta	RK_S32	表示协议语法中 deblock_offset_beta 值。 有效范围为[-6, 6]。
use_longterm	RK_S32	表示长期参考帧模式标志位，暂不使用。
qp_init	RK_S32	表示初始 QP 值，一般情况请勿配置。
qp_max	RK_S32	表示最大 QP 值，一般情况请勿配置。
qp_min	RK_S32	表示最大 QP 值，一般情况请勿配置。
qp_max_step	RK_S32	表示相临两帧之间的帧级 QP 变化幅度。
intra_refresh_mode	RK_S32	表示 Intra 块刷新模式，暂不使用。
intra_refresh_arg	RK_S32	表示 Intra 块刷新参数，暂不使用。
slice_mode	RK_S32	表示 slice 编码的划分模式，暂不使用。
slice_arg	RK_S32	表示 slice 编码的划分参数，暂不使用。

需要注意的是，H.264 编码器协议配置参数还在更新中，还不是最终版本。

JPEG 编码器协议配置 MppEncJpegCfg 的重要参数成员如下：

成员名称	成员类型	描述说明
change	RK_U32	每比特表示是否更新特定的参数。
quant	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。

3.3.2 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 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。

3.3.3 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 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_CODEEC_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。

3.3.4 encode

暂未实现

3.3.5 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_CODEEC | CMD_CTX_ID_ENC,
/* basic encoder setup control */
MPP_ENC_SET_ALL_CFG, /* set MppEncCfgSet structure */
MPP_ENC_GET_ALL_CFG, /* get MppEncCfgSet structure */
MPP_ENC_SET_PREP_CFG, /* set MppEncPrepCfg structure */
MPP_ENC_GET_PREP_CFG, /* get MppEncPrepCfg structure */
MPP_ENC_SET_RC_CFG, /* set MppEncRcCfg structure */
MPP_ENC_GET_RC_CFG, /* get MppEncRcCfg structure */
MPP_ENC_SET_CODEEC_CFG, /* set MppEncCodeecCfg structure */
MPP_ENC_GET_CODEEC_CFG, /* get MppEncCodeecCfg structure */
/* runtime encoder setup control */
MPP_ENC_SET_IDR_FRAME, /* next frame will be encoded as intra frame */
MPP_ENC_SET_OSD_PLT_CFG, /* set OSD palette, parameter should be pointer to MppEncOSDPlt */
MPP_ENC_SET_OSD_DATA_CFG, /* set OSD data with at most 8 regions, parameter should be pointer to
MPP_ENC_GET_OSD_CFG,
MPP_ENC_SET_EXTRA_INFO,
MPP_ENC_GET_EXTRA_INFO, /* get vps / sps / pps from hal */
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_CMD_END,

```

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_ALL_CFG ， MPP_ENC_SET/GET_PREP_CFG ，

MPP_ENC_SET/GET_RC_CFG 和 MPP_ENC_SET/GET_CODEEC_CFG 命令已经做为基本的配置命令在 3.3.1 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 rkvcnc 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。rkvcnc 系列只支持 H.264 编码，目前只配备于 rv110x 系统芯片，其支持的编码功能相对于 vepu 系统会更多更强。

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_PLT_CFG

命令参数为 MppEncOSDPlt，用于配置 rkvcnc 系列硬件的 OSD 调色板。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 rkvcnc 系列支持。

MPP_ENC_SET_OSD_DATA_CFG

命令参数为 MppEncOSDData，用于配置 rkvcnc 系列硬件的 OSD 数据。需要每帧进行配置，每编码一帧之后需要重新配置。

MPP_ENC_GET_OSD_CFG

预留，暂不支持。

MPP_ENC_SET_EXTRA_INFO

预留，暂不支持。

MPP_ENC_GET_EXTRA_INFO

命令参数为 MppPacket*，用户可以通过获取到的 MppPacket 结构得到 H.264 编码器输出的 SPS/PPS 码流数据。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。

MPP_ENC_SET_SEI_CFG

命令参数为 MppEncSeiMode*，用户可以通过本命令对 H.264 编码器的 SEI 包模式进行配置。

```
typedef enum MppEncSeiMode_t {
    MPP_ENC_SEI_MODE_DISABLE,           /* default mode, SEI writing is disabled */
    MPP_ENC_SEI_MODE_ONE_SEQ,          /* one sequence has only one SEI */
    MPP_ENC_SEI_MODE_ONE_FRAME,       /* one frame may have one SEI, if SEI info has changed */
} MppEncSeiMode;
```

默认关闭。

MPP_ENC_GET_SEI_DATA

命令参数为 MppPacket*，用户可以通过本命令获取 H.264 编码器的 SEI 包数据。

MPP_ENC_PRE_ALLOC_BUFF

无命令参数，用于让编码器提前进行内部内存分配，节省第一帧图像的编码时间。编码器由于在宽高信息配置完成之后，不一定会得到相同大小的图像数据做为输入，所以会在收到第一帧图像数据时才进行参考帧和重构帧的内存分配，这样会导致第一帧编码的时间过长。本命令是用于优化编码器第一帧延迟，在宽高信息配置完成之后，第一帧图像输入数据之前，通过调用本命令让编码器提前完成内存分配。

MPP_ENC_SET_QP_RANGE

内部调试使用，暂不支持。

MPP_ENC_SET_ROI_CFG

命令参数为 MppEncROICfg*，用户可以通过接口对 H.264 编码器的 ROI 进行配置。

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：

```

i1shell@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/utills ( 1230): -i input_file          input bitstream file
01-01 12:13:07.460 I/utills ( 1230): -o output_file        output bitstream file,
01-01 12:13:07.460 I/utills ( 1230): -w width            the width of input bitstream
01-01 12:13:07.460 I/utills ( 1230): -h height          the height of input bitstream
01-01 12:13:07.460 I/utills ( 1230): -t type            input stream coding type
01-01 12:13:07.460 I/utills ( 1230): -d debug          debug flag
01-01 12:13:07.460 I/utills ( 1230): -x timeout        output timeout interval
01-01 12:13:07.460 I/utills ( 1230): -n frame_num     max decode frame number
i1shell@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。

下载命令: `git clone -b release 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/arm` 目录和 `build/android/aarch64` 目录下 `make-Android.bash` 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:

A: