Rockchip Developer Guide PCIE EP 标准卡

文件标识：RK-KF-YF-489

发布版本：V1.0.0

日　　期：2023-04-13

文件密级：□绝密　□秘密　□内部资料　■公开

免责声明

本文档按“现状”提供，瑞芯微电子股份有限公司（“本公司”，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址：　福建省福州市铜盘路软件园A区18号

网址：　www.rock-chips.com

客户服务电话：+86-4007-700-590

客户服务传真：+86-591-83951833

客户服务邮箱：fae@rock-chips.com

前言

概述

本文档对RK3588 PCIE 标准EP开发包使用进行说明，以及对相关API做介绍。

产品版本

芯片名称	内核版本
RK3588	Linux 5.10

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

版本号	作者	修改日期	修改说明
V1.0.0	yp.xiao，Kever Yang，Jon Lin	2023-03-07	初始版本

Rockchip Developer Guide PCIE EP 标准卡

1. EP卡概述

RK PCIe EP标准卡开发包是RK”PCIe EP 卡形态产品“的软硬件综合开发包，开发基于RK EP卡Demo板（以下简称EP卡），开发包主要包含：

- EP卡的快速操作指南
- EP卡启动方案、PCIe相关软硬件设计开发以及升级OTA相关内容
- EP卡业务基础
- EP卡业务基础框架下的业务实例

1.1 须知

为了后续开发顺利，以下为须知内容：

- RK SOC BootRom不支持PCIe的EP Boot，所以如需尽早初始化PCIe EP功能，要求外挂SPI Nor Flash
- 驱动层面是把PCIE相关的内核驱动封装成用户层API，方便用户开发
- EP卡兼容Linux操作系统的RC设备，并已在特定RC上完成验证
- EP卡理论上兼容Windows操作系统的RC设备，但未做相应验证，且无Windows驱动，需客户自行参考Linux驱动进行开发
- EP卡开发包实现PCIe主要EP需求功能：Bar访问、DMA传输、MSI中断、EP卡中断事件ELBI中断
- EP卡开发包本质为PCIe产品功能展示开发包，非一站式解决方案，需客户基于次做PCIe及上层应用的二次开发

1.2 开发包清单

目录结构介绍

```
.
├── core
│   ├── ep_dev_sdk                      # EP卡业务基础，EP用户层驱动
│   └── rc_dev_sdk                      # EP卡业务基础，RC用户层设备驱动
├── docs                                # PCIE EP SDK文档介绍
├── examples
│   ├── pcie_camera_test                # EP卡典型业务，抓取EP Camera数据数据处理，仅
android端使用
│   ├── pcie_speed_test                 # EP卡典型业务，PCIE传输速率测试Demo
│   ├── pcie_video_test                 # EP卡典型业务，视频卡Demo
│   └── pcie_download_test              # EP卡EP Boot启动方案Demo
└── images                              # 功能验证固件
└── patch                               # 补丁目录
```

开发包主要目录介绍：

ep_dev_sdk目录下封装了EP用户接口层的实现，对外开发特定API方便用户开发，API头文件`ep_dev_sdk/rk_pcie_ep.h`。该目录下由三部分组成：

- rk_mem：负责EP端DMA内存管理。
- rk_pcie： 负责与内核驱动通信，以及DMA传输配置。

- rk_pcie_ep.c： 负责对EP操作接口封装，对外提供API。

rc_dev_sdk目录下封装了RC用户接口层的实现，对外开发特定API方便用户开发，API头文件`rc_dev_sdk/rk_pcie_rc.h`。该目录下由四部分组成：

- libpci：libpci标准库V3.8.0版本源码，会编译生成静态库，用于RC端操作pci设备。
- rk_mem：负责RC端DMA内存管理，支持rkep以及hugepage两种方式申请内存。
- rk_pcie：通过libpci接口与内核驱动通信，以及DMA传输配置。
- rk_pcie_rc.c： 负责对RC操作接口封装，对外提供API。

docs：标准EP板卡说明文档

examples：提供不同测试Sample，Sample介绍以及操作说明查看对应目录下的README。

images：目录下存放可直接用于基础功能测试的固件以及Sample。

patch：SDK包对应的补丁，支持linux和android系统。补丁说明请查看`patch/README`。

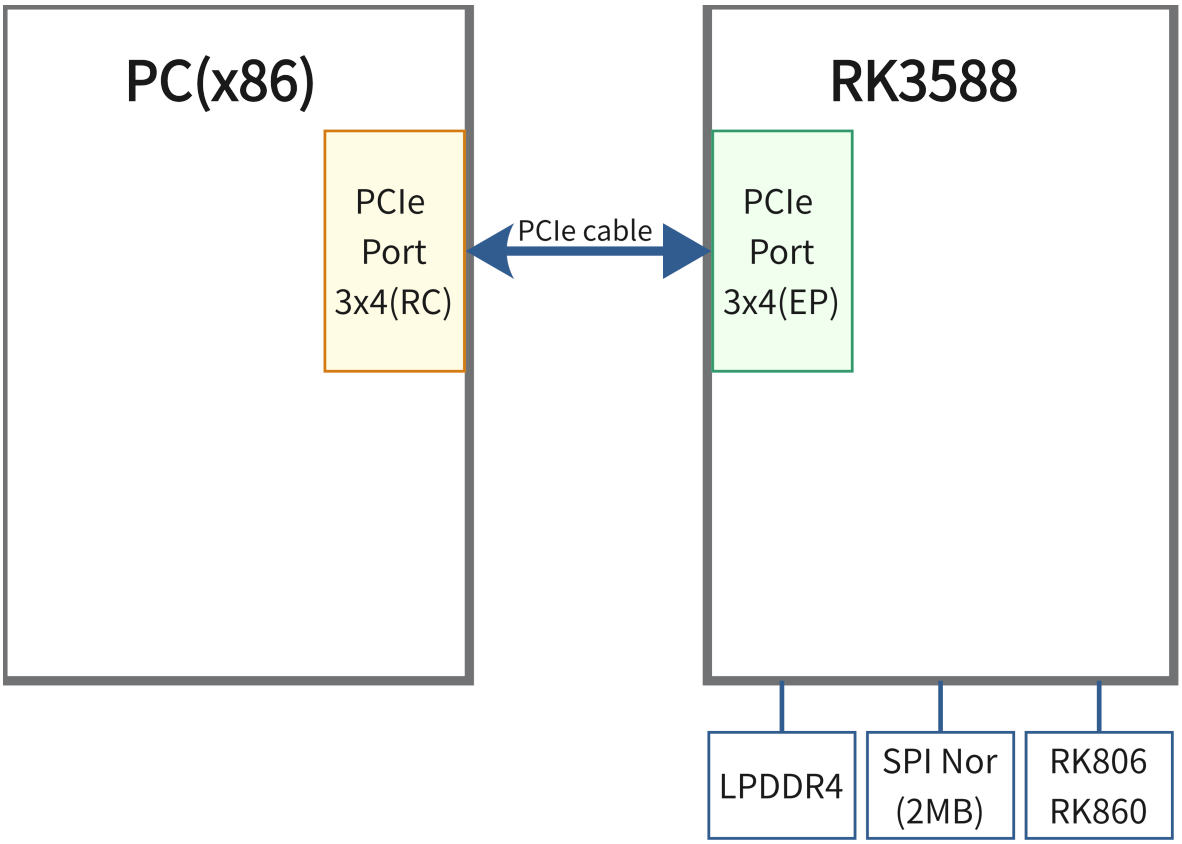
1.3 EP应用场景

1.3.1 单EP卡对接通用RC

EP卡接入标准RC的PCIe Slot中，采用所有PCIe Slot的标准信号，EP卡的硬件实现请参考RK提供的硬件参考图，其中需要说明的有：

- 使用common clock模式， Refclock使用RC端提供的时钟；
- EP端可以使用PCIe EP Boot，使用SPI NOR Flash存放以及loader，后续所有需要的固件由RC下载，不需要EMMC。

以下为RK3588 EP对接通用RC示例：

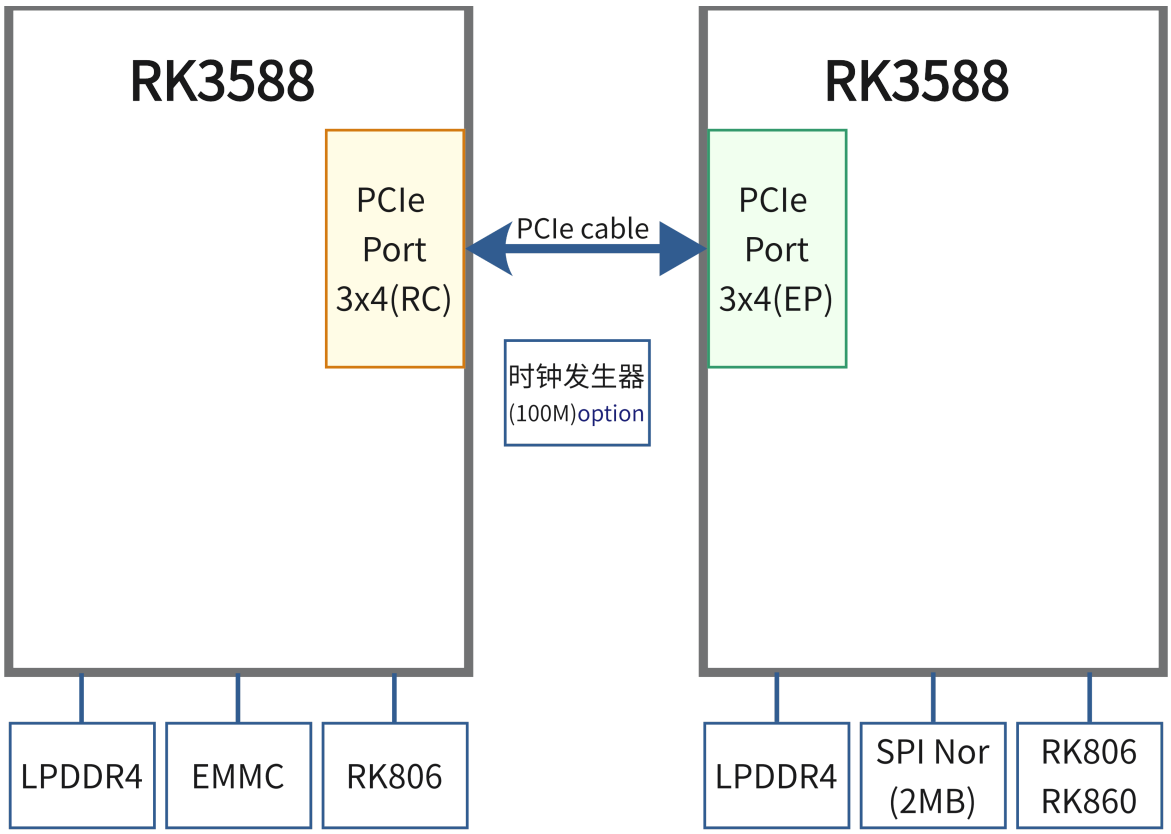


1.3.2 单EP卡对接RK RC

这个模型是前一个模型的特例，在两边多采用RK3588芯片的情况下：

- 可以采用SRNS模式，Refclock使用各自的内部时钟，布板上可以节省时钟发生器和时钟buffer芯片；
- 需要采用PCIe EP Boot下载固件；

以下为RK3588 EP对接RK3588 RC示例：

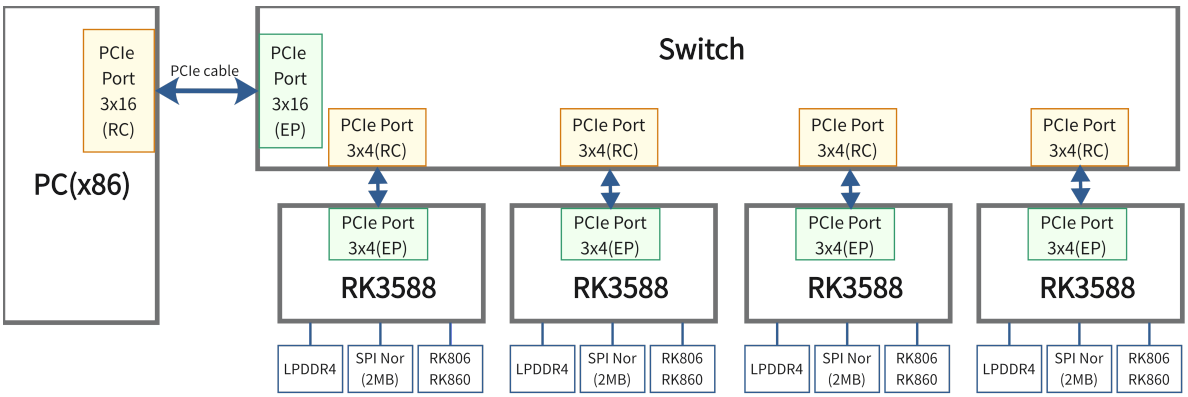


1.3.3 多EP卡对接通用RC

这个模型可以使用4口或者8口的switch芯片，连接4~8个EP卡，每一个EP卡都是一个独立的模块。

- 使用common clock模式， Refclock使用RC端提供的时钟；
- 需要采用PCIe EP Boot下载固件；

以下为RK3588 EP示例：



2. EP卡操作指南

2.1 环境准备

EP卡支持对接通用RC以及RK RC，这里对两种场景下基础功能验证做简单说明。

2.1.1 硬件环境

1. EP卡对接RK RC

   准备2块硬件板卡，RC使用RK3588-EVB1-V10、EP使用RK3588-EVB4-V11（型号可以查看板卡丝印来确认）。需要为RC板卡连接电源线、串口线、USB转type-c固件下载线、HDMI输出线 (连接HDMI_OUT0)，EP板卡通过RC直接供电，不需要单独电源线。

   EP卡不支持热拔插，需要先接上再对RC上电。

2. EP卡对接通用RC

   EP卡使用RK3588-EVB4-V11。EP卡不支持热拔插，需要先接上再对RC上电，不需要其它额外配置。

2.1.2 软件环境

芯片需要的loader、uboot、 boot、以及文件系统可在发布包的 images 目录下获得这些镜像文件。

EP卡对接RK RC或者通用RC，均支持2种固件启动方式:

- EP Flash 启动方式
- EP DDR 启动方式

1. EP采用Storage Boot启动方案，启动文件 如下表所示

项目	文件名称	描述
RC(对接通用RC不需要)	update-rc-evb1-v10.img	烧写到RC Flash
EP	update-ep-evb4-v11.img	烧写到EP Flash

   说明：update.img包含了MiniLoaderAll.bin、uboot.img、boot.img、rootfs.img等文件。

2. EP采用EP Boot 启动方案，启动文件 如下表所示。

项目	文件名称	描述
RC(对接通用RC不需要)	update-rc-evb1-v10.img	烧写到RC Flash
EP	PCIE-BOOT_MiniLoaderAll.bin	烧写到EP Flash
EP	uboot.img	由RC通过PCIE导入EP DDR 内存
EP	boot.img （包含内核+rootfs）	由RC通过PCIE导入EP DDR 内存

对接通用RC场景只需要烧写EP卡固件，RC不需要修改。

RC/EP Flash固件烧写请参考《Rockchip_RK3588_Linux_NVR_SDK_Release*.pdf》中的刷机说明章节。

RC通过PCIE对EP下载固件请参考下一章节功能验证说明

2.2 功能验证

验证PCIE EP功能的样例代码位于 SDK 发布包的examples目录下，包括EP和RC分别使用的Sample代码、封装的消息通讯代码和封装的数据通讯代码。

通用RC功能验证需要的bin和资源存放在images目录下需要手动拷贝到RC平台(仅X86平台)，RK RC的资源直接打包到固件root目录中不需要额外操作。

在硬件、软件环境准备好后，执行完整的 PCIE 功能验证的步骤如下。不需要对EP卡下载固件可以直接跳过第一点。

1. RC通过PCIE对EP下载固件

   步骤1. EP卡串口波特率是1500000（RC波特率也是1500000），连接串口后会一直打印如下log，说明EP和RC的PCIE通路Link成功，正在等待RC传输固件。

```
U-Boot SPL 2017.09-g614fe864bf-dirty #xyp (Apr 20 2023 - 11:19:22)
RKEP: 191277 - Link ready! Waiting RC to download Firmware:
RKEP: 191723 - Download uboot.img  to BAR2+0
RKEP: 192068 - Download boot.img   to BAR2+0x400000
RKEP: 192466 - Send CMD_LOADER_RUN to BAR0+0x400
RKEP: 1092850 - Waiting for FW, CMD: 0
RKEP: 2093146 - Waiting for FW, CMD: 0
RKEP: 3093443 - Waiting for FW, CMD: 0
RKEP: 4093740 - Waiting for FW, CMD: 0
RKEP: 5094036 - Waiting for FW, CMD: 0
```

   说明:

   EP卡直接进到系统说明flash中是完整固件，请进入maskrom下重新烧写`PCIE-BOOT_MiniLoaderAll.bin`。

   步骤2.RC端执行`./pcie_download_test_rc 0 ./uboot.img ./boot.img`，启动RC与EP之间的数据传输，通过DMA把固件搬运到EP卡指定内存地址，然后发送启动指令让EP通过DDR启动，EP卡此时会打印正常的开机log进入系统。

   说明:

   RC端执行应用后异常报错请查看`examples/pcie_download_test/README`排查问题。

2. 视频解码预览Sample

步骤1. EP串口波特率是1500000（RC波特率也是1500000），连接串口后，RC执行`lspci -vvv |grep 356a`以及`ls /dev/pcie-rkep*`可以看到如下设备说明识别到EP卡并且RC驱动加载正常。

```
[root@RK3588:/userdata]# lspci -vvv |grep 356a
01:00.0 Class 1200: Device 1d87:356a (rev 01)
[root@RK3588:/userdata]# ls /dev/pcie-rkep*
/dev/pcie-rkep0000:01:00.0
```

步骤2. EP卡正常进入系统后，此时终端上将打印如下log，并被阻塞，等待RC应用启动。

```
Loading model ...
rknn_init ...
model input num: 1, output num: 2
input tensors:
output tensors:
rk pcie version: v1.1.0 - 20230403 - using rc dma
 - id=356a1d87
 - magic=524b4550, ver=1
 - bar2 cpu_addr=0x40000000
PCIE_BUS_CMD_OFFSET: 0x8000
wait rc init....
```

步骤3. RC执行如下指令，完成与EP卡的握手然后启动与EP之间的数据传输。

- 通用RC：`./pcie_video_test_rc ./output1.h264 36 704 576 0`
- RK RC：`/root/pcie_video_test_rc /root/output1.h264 36 704 576 0`

此时RC的显示设备会输出1080P60分辨率，可以看到36路视频解码以及单路NN人型检测功能。
说明：
RC端是通过DRM显示框架来输出显示数据，RK RC默认支持只需要把HDMI OUT0接上显示设备即可，通用RC需要安装`libdrm-dev`组件并且通过ctrl+F3切换到其它无GUI终端再执行测试指令。

步骤4. 如果需要停止Sample程序，可以在RC应用输入回车键，RC会发消息给EP卡销毁相关 业务，并退出程序；EP卡接收消息后销毁相关业务并退出程序 。需要正常退出业务流程，否则EP卡业务不会退出导致下次运行异常。

说明：
EP端应用默认开机自启并且循环启动，若需要运行其它测试程序需要先在开机脚本把当前测试注释然后重启RC和EP。

修改/etc/init.d/S98_lunch_init，把下面的代码注释。

```
cd /root/
chmod 777 pcie_video_test_ep
while true;
do
./pcie_video_test_ep
done &
```

2.3 实测性能

PCIE EP传输性能

EP	RC	硬件配置	场景	最大传输数率
RK3588	RK3588	PCIE 3.0x4Lane	EP写数据到RC	2.95GB/s
			EP从RC读数据	1.8GB/s
			EP同时读写RC	各自1.6GB/s
RK3588	RK3588 (开启Using RC DMA)	PCIE 3.0x4Lane	EP写数据到RC	2.95GB/s
			EP从RC读数据	2.95GB/s
			EP同时读写RC	各自2.85GB/s
	X86	PCIE 3.0x4Lane	EP写数据到RC	2.9GB/s
			EP从RC读数据	2.5GB/s
			EP同时读写RC	各自2.4GB/s

后续性能会继续优化

2.4 Samples编译

各sample编译以及使用说明请参看`examples/*/README`。

3. EP卡启动方案及软硬件配置

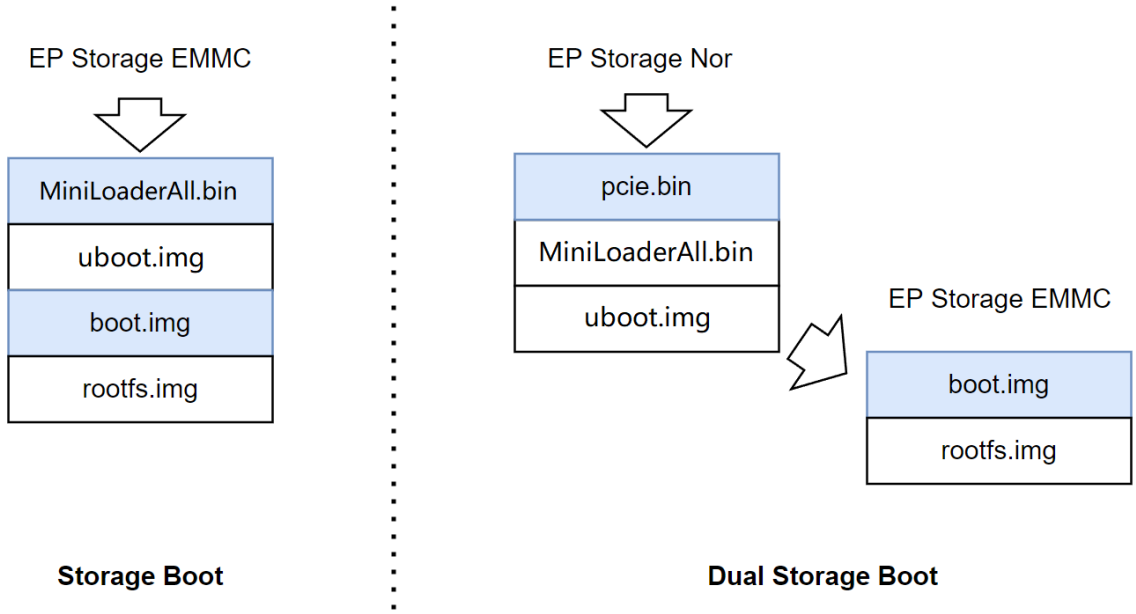
3.1 EP卡启动方案

RK EP Demo卡默认选用Storage Boot方案，集成EMMC存储器件存放完整固件，RC在EP SPL阶段完成枚举功能，除此之外，EP卡开发还支持双存储的Storage Boot方案和EP Boot方案。

3.1.1 Storage Boot

EP集成的存储器件存放完整EP卡固件。

基础流程



说明：

- 蓝色框图流程为运行PCIe驱动的流程，白色框图为无PCIe驱动流程
- 图示左侧为单存储EP Demo卡默认硬件设计，图示右侧为双存储方案
- 主要流程说明：
  - pcie.bin（可选）：初始化EP PCIe
  - MiniLoaderALL.bin 为 ddr.bin 和 spl.bin 的打包文件，其中：
    - ddr.bin：初始化ddr
    - spl.bin：初始化EP PCIe（可选），引导后级固件
  - boot.img：内核PCIe支持，涉及EP卡业务支持

单存储方案

对接特定的RC产品，如特定工控PC，设置可优化Bios PCIe流程

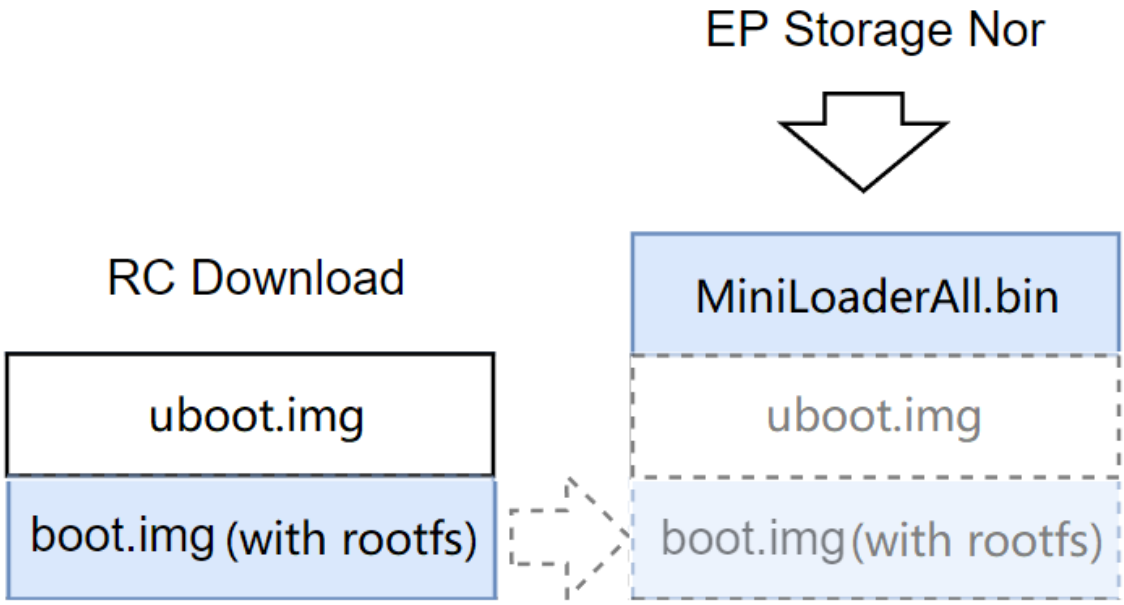
Nor快速启动 + EMMC大容量固件的双存储方案

对接通用的RC产品，对PCIe枚举时间有严苛要求

3.1.2 EP Boot

EP集成的存储器件存放EP卡部分启动固件，后级固件由RC通过PCIe推送。

基础流程



EP Boot

说明：

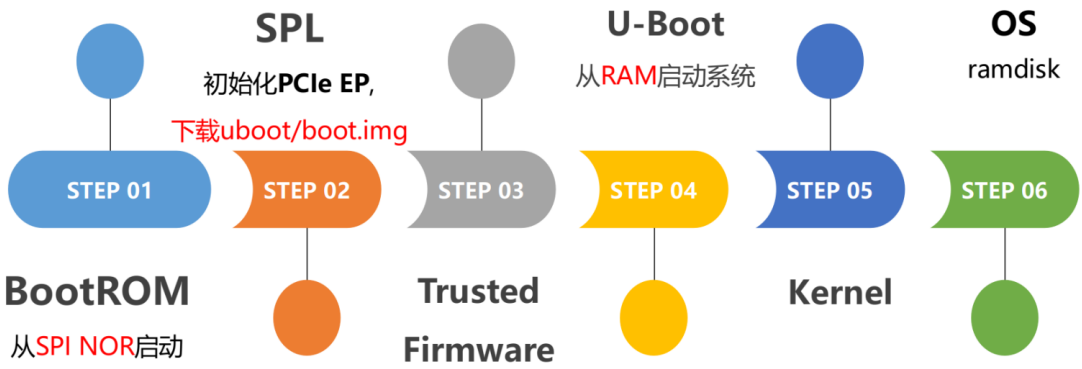
- 蓝色框图流程为运行PCIe驱动的流程，白色框图为无PCIe驱动流程
- 虚线框架代表由RC推送到EP内存的固件
- 主要流程说明：
  - MiniLoaderALL.bin 为 ddr.bin 和 spl.bin 的打包文件，其中：
    - ddr.bin：初始化ddr
    - spl.bin：初始化EP PCIe（可选），引导后级固件
  - boot.img：内核PCIe支持，涉及EP卡业务支持

详细介绍

RK3588的BootRom并未支持PCIe EP Boot，所以方案上需要外挂一片SPI NOR Flash来存放固件初始化PCIe口为EP模式同时提供EP Boot功能。SPI NOR Flash的大小仅需要能够放下一级loader即可，后续的uboot.img、boot.img等固件都是通过PCIe下载到RAM来启动的。

EP端视角的启动步骤为：

- BootRom从SPI NOR Flash读取固件启动运行；
- 位于SPI NOR Flash的固件包含DDR初始化和SPL，SPL完成PCIe EP初始化，然后等待固件下载；
- RC检测到EP后，下载所需固件，一般为FIT格式的uboot.img(含ATF和U-Boot)和boot.img(含kernel, dtb和ramdisk)；
- EP收到固件，生成RAM partition存放boot.img，并通过ATAGS配置为从RAM启动传递给后级U-Boot，开始解析；
- ATF(Trusted Firmware ARM)启动跟正常启动一致，没有差别，完成后跳转到U-Boot；
- 进入U-Boot后直接从RAM启动，解析已有的boot.img启动系统；



NOTE:从x86的RC端来看，UEFI初始化后PCIe信息传递给kernel使用，后续除非发生异常，否则不会有重新扫描的流程，所以在EP端需要在第一次就完成设备信息的配置，也就是SPL阶段就完成配置，且保持SPL与kernel驱动中EP配置的一致性。

3.2 EP卡硬件配置

PCIe EP产品，除了PCIe spec对物理信号的要求，和选定实际产品需要使用的互联模型进行布板以外，在RK平台需要考虑内容主要有：

1. 基于互联模型确定产品使用的参考时钟模型
    - 对于RC和EP均采用RK芯片的方案，可以使用SRNS模式，EP采用内部100M时钟，可以省掉外置的时钟发生器和时钟buffer芯片；
    - 对于完全标准的EP产品，只能采用common mode的参考时钟，EP使用RC端提供的100M参考时钟。
2. 基于产品要求确定异常处理方法
    - PERST信号作为RC端物理复位EP的最后手段，EP必须保证这个信号能够恢复任何异常，作为EP端建议该信号直接控制主控芯片和PMIC的NPOR信号来完整重启设备；

      NOTE: 由于RC的PERST是PP强驱，EP上PMIC是OD脚，所以不管RC的PERST是3.3V还是1.8V，都需要通过电路转换来驱动EP端的RESET_L。
    - Spec规定RC在PERST信号释放后100ms，开始扫描设备，作为EP端需要保证PERST信号释放100ms内完成EP的初始化，鉴于RK3588芯片默认状态PCIe并未使能，需要以最快的方式加载固件完成PCIe EP的初始化；这一要求导致我们必须使用SPI Nor Flash作为存储达到要求，因为eMMC的初始化包含协议初始化和设备的ready时间，其中设备ready时间的SPEC要求是1S内，尽管大部分设备一般是几十ms，但明显无法解决概率性超过100ms未完成初始化的情况。

详细的硬件设计可以参考瑞芯微提供的原理图参考设计<RK_PCIeEP_DEMO_RK3588_LP4XD200P232SD8_V10_20230323RZF.pdf>。

3.3 EP卡驱动软件开发

3.3.1 软件开发须知

EP卡部分软件配置为关联配置，如做修改应修改所有关联处，如关于Bar大小配置：

- PCIe Bin阶段-Bar大小配置（双存储方案）
- SPL阶段-Bar大小配置

3.3.2 PCIe Bin阶段配置

原理同SPL阶段配置，目前相关源码未开放，后续提供基于u-boot源码工程下编译工程。

须知：

- PCIe Bin阶段-Bar大小配置 为关联配置，应同时修改：
  - PCIe Bin阶段-Bar大小配置（双存储方案）
  - SPL阶段-Bar大小配置
  - Kernel阶段-Bar大小配置

3.3.3 SPL阶段配置

3.3.3.1 Kconfig配置

本功能在SPL中有一下三个Kconfig项，对于需要PCIe EP Boot功能的产品需要全部选上，对于不需要EP Boot功能的产品只需要选上SPL_PCIE_EP_SUPPOR即可。

```
CONFIG_SPL_PCIE_EP_SUPPORT=y
CONFIG_SPL_RAM_SUPPORT=y
CONFIG_SPL_RAM_DEVICE=y
```

3.3.3.2 BAR默认配置

Demo使用如下默认配置，除了BAR4，项目可以根据实际需求进行修改：

- BAR0： 32bits 4MB，命令流；
- BAR2： 64bits 64MB，数据流；
- BAR4：32bits 1MB，EP端PCIe寄存器，不可修改；

其中BAR4的offset映射如下，可以通过RC端直接操作对应寄存器：

- 0: DMA
- 0x2000: iATU
- 0x20000: MSI-X table

3.3.3.3 BAR大小配置

在pcie_bar_init()完成BAR的size配置，通过控制器的resize BAR功能实现。

配置BAR SIZE，offset+0x8寄存器，bit[13:8]用于配置BAR size，结果为2^n，所以支持1MB(2^0)到8EB(2^20)：

```
        /* Resize BAR0 to support 4M 32bits */
        resbar_base = dbi_base + 0x2e8;
        writel(0xfffff0, resbar_base + 0x4);
        writel(0x2c0, resbar_base + 0x8);
        /* BAR2: 64M 64bits */
        writel(0xfffff0, resbar_base + 0x14);
        writel(0x6c0, resbar_base + 0x18);
        /* BAR4: Fixed for EP wired register, 2M 32bits */
        writel(0xfffff0, resbar_base + 0x24);
        writel(0xc0, resbar_base + 0x28);
```

配置BAR属性，选择已定义的宏进行配置，支持32bit和64bit，是否PREFETCH：

```
        rockchip_pcie_ep_set_bar_flag(dbi_base, 0, PCI_BASE_ADDRESS_MEM_TYPE_32);
        rockchip_pcie_ep_set_bar_flag(dbi_base, 2, PCI_BASE_ADDRESS_MEM_PREFETCH
| PCI_BASE_ADDRESS_MEM_TYPE_64);
        rockchip_pcie_ep_set_bar_flag(dbi_base, 4, PCI_BASE_ADDRESS_MEM_TYPE_32);
```

关掉不使用的BAR：

```
        /* Close bar1 bar3 bar5 */
        writel(0x0, dbi_base + 0x100000 + 0x14);
        //writel(0x0, dbi_base + 0x100000 + 0x18);
        writel(0x0, dbi_base + 0x100000 + 0x1c);
        //writel(0x0, dbi_base + 0x100000 + 0x20);
        writel(0x0, dbi_base + 0x100000 + 0x24);
        /* Close ROM BAR */
        writel(0x0, dbi_base + 0x100000 + 0x30);
```

须知：

- SPL 阶段-Bar大小配置为关联配置，应同时修改：
  - PCIe Bin阶段-Bar大小配置（双存储方案）
  - SPL阶段-Bar大小配置
  - Kernel阶段-Bar大小配置

3.3.3.4 BAR映射配置

每个BAR对应的空间，通过pcie_inbound_config()配置映射到内部的内存地址区域，可以修改RKEP_BAR0_ADDR， RKEP_BAR2_ADDR的宏定义来修改要映射的目标地址。

```
        /* BAR0: RKEP_BAR0_ADDR */
        writel(RKEP_BAR0_ADDR, base + PCIE_ATU_CPU_ADDR_LOW);
        writel(0, base + PCIE_ATU_CPU_ADDR_HIGH);
        writel(0, base + PCIE_ATU_UNR_REGION_CTRL1);
        /* PCIE_ATU_UNR_REGION_CTRL2 */
        writel(PCIE_ATU_ENABLE | PCIE_ATU_BAR_MODE_ENABLE | (0 << 8),
               base + PCIE_ATU_UNR_REGION_CTRL2);
```

须知：

- SPL阶段-Bar映射配置与Kernel阶段配置独立，可根据需求映射不同空间

3.3.3.5 PHY模式配置

Demo默认使用的是0xfe150000控制器PCIe3x4模式，也就是PHY也是4lane模式，由于PHY是支持拆分的，如果方案需要修改为其他模式，可以修改pcie_cru_init()的对应配置, 如PCIe工作于2lane+2lane模式可以将PHY_MODE_PCIE_AGGREGATION修改为PHY_MODE_PCIE_NANBNB：

```
        /* Phy mode: Aggregation NBNB */
        writel((0x7 << 16) | PHY_MODE_PCIE_AGGREGATION,
               0xfd5b8000 + RK3588_PCIE3PHY_GRF_CMN_CON0);
```

3.3.3.6 EP Boot配置

原理介绍中提到EP会等待固件并运行固件，涉及跟RC间的固件下载协议和BootLoader前后级的固件位置约定。

固件的下载函数pcie_wait_for_fw()和后级固件位置约定函数pcie_update_atags(), 可以通过下文提到的地址来做对应修改，请务必保证RC与EP约定的命令和地址一致，BootLoader前后级约定的固件位置一致，才能顺利完成整个固件的下载和启动过程。如果不理解全过程，请勿修改。

跟RC间的固件下载协议内容是，先按要求把固件下载到指定位置，然后发送运行固件命令。

1. 固件下载位置

   约定下载两个固件

固件名	打包格式	内容	下载地址
uboot.img	FIT	ATF和U-Boot	BAR2+0
boot.img	FIT	Kernel、DTB、ramdisk	BAR2+0x400000

2. 运行固件命令

命令名称	命令位置	命令值	操作
CMD_LOADER_RUN	BAR0 + 0x400	0x524b4501	运行下载的固件

从RC下载过来的两个固件，其中uboot.img会被SPL(运行EP boot的环境)直接解析并运行，然后通过一个RK自定义的RAM_PARTITION来传递boot.img的位置给U-Boot。pcie_update_atags()函数定义了boot分区在RAM的其实地址和SIZE，由于默认使用的BAR2是64M，其中起始4M分给了uboot.img，后续的60M分给了boot.img，这是为了方便RC端直接通过CPU从BAR2把固件写过来，如果size不够，可以在RC端调用EP的dma来下载固件，这样size不受BAR2大小限制，需要注意同步更新RAM_PARTITION中的size配置。

3.3.3.7 SRNS配置

对于可以使用SRNS模式参考时钟的方案，需要打开相关配置，打开配置后EP端会使用内部的100MHz时钟作为参考时钟，不依赖外部提供的时钟。在代码中直接打开如下宏配置即可：

```
/* SRNS: Use Seperate refclk(internal clock) instead of from RC */
// #define PCIE_ENABLE_SRNS_PLL_REFCLK
```

须知：

- SPL 阶段-SRNS配置为关联配置，应同时修改：
  - PCIe Bin阶段-SRNS配置（双存储方案）
  - SPL阶段-SRNS大小配置
  - Kernel阶段-SRNS配置

3.3.4 Kernel阶段配置

3.3.4.1 DTS配置

```
/{
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;
        bar0_region: bar0-region@3c000000 {
            reg = <0x0 0x3c000000 0x0 0x00400000>;
        };
        bar2_region: bar2-region@40000000 {
            reg = <0x0 0x40000000 0x0 0x04000000>; # Bar大小配置：0x04000000Bytes
        };
    };
}
&pcie3x4 {
        compatible = "rockchip,rk3588-pcie-std-ep";
        memory-region = <&bar0_region>, <&bar2_region>;
        memory-region-names = "bar0", "bar2";
        status = "okay";
};
```

须知：

- Kernel阶段-Bar大小配置为关联配置，应同时修改：
  - PCIe Bin阶段-Bar大小配置（双存储方案）
  - SPL阶段-Bar大小配置
  - Kernel阶段-Bar大小配置
- Kernel阶段-Bar映射配置与SPL阶段配置独立，可根据需求映射不同空间

3.3.4.2 Kconfig配置

```
CONFIG_PCIE_DW_ROCKCHIP_EP=y
# CONFIG_STRICT_DEVMEM is not set   #建议开发阶段关闭STRICT_DEVMEM限制，方便使用io命令调试
```

3.3.4.3 SRNS配置

SRNS配置为补丁形式添加，请从开发包中获取补丁文件 0001-TEST-kernel5.10-rk3588-SRNS.patch

Kernel 阶段-SRNS配置为关联配置，应同时修改：

- PCIe Bin阶段-SRNS配置（双存储方案）

- SPL阶段-SRNS大小配置
- Kernel阶段-SRNS配置

3.4 EP卡固件烧录

1. Storage Boot方案固件烧写

- 单存储

  单存储方案固件烧写和正常固件烧写方式一致，不需要特殊配置，参考各芯片发布文档刷机说明章节即可。

- 双存储

  待完善。

2. EP Boot方案固件烧写

   EP Boot方案只需要在集成的存储器件存放EP卡部分启动固件（MiniLoaderALL.bin）。启动固件和正常固件烧写流程一致，不需要特殊配置，参考各芯片发布文档刷机说明章节即可。

3.5 EP卡OTA

待完善。

4. EP卡业务基础

4.1 原理介绍

标准EP开发包的原理是把PCIE BAR资源通过mmap映射给用户层使用，并且把DMA功能默认映射到BAR4空间，可以直接在用户层操作DMA读写。

RC端

- 通过开源标准库libpci操作PCIE EP设备。
- 通过pcie-rkep驱动申请DMA内存以及处理MSI中断。

EP端

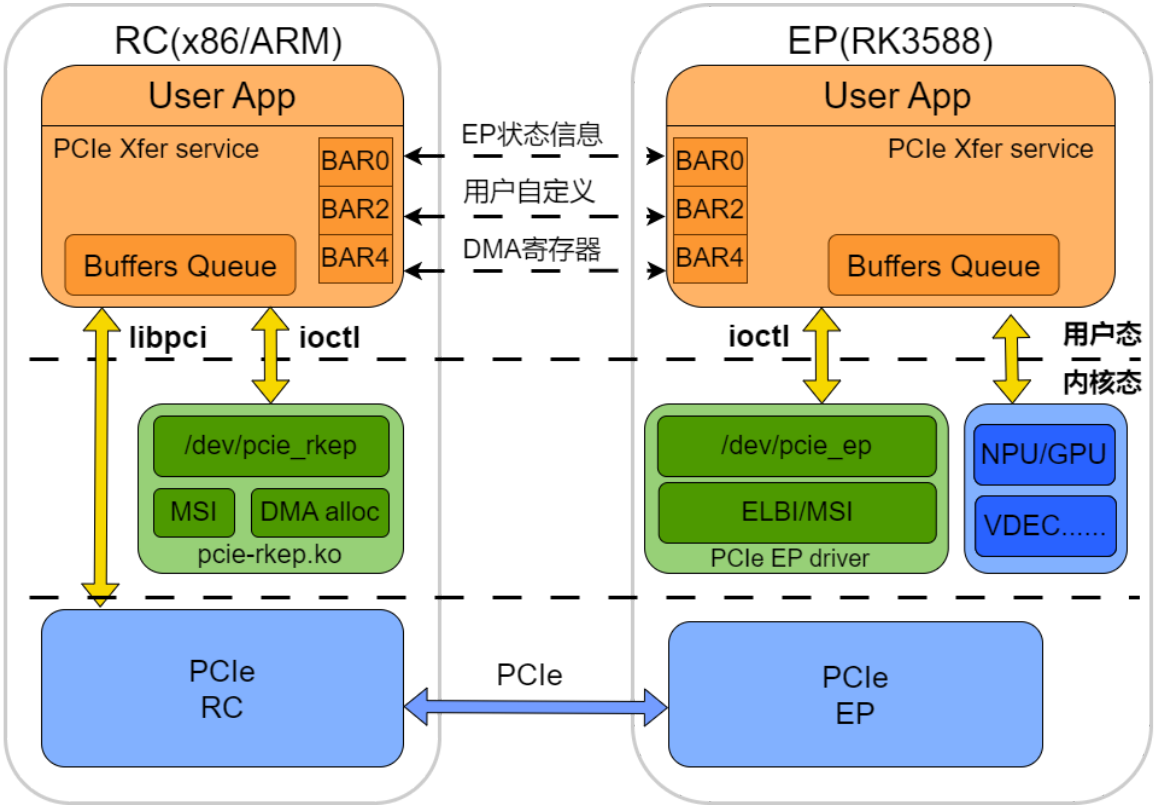
- 通过pcie_ep驱动获取BAR空间映射以及处理MSI/ELBI中断。
- VDEC、VENC、RGA、GPU、NPU等硬件资源可以通过Rockit 或者RKNN在用户层调用。

BAR空间分配

BAR空间	起始地址	结束地址	功能
BAR0	0	4KB	EP保留空间，驱动版本信息、中断状态等等。
	16KB	32KB	EP设备状态信息。
	32KB	64KB	命令交互以及Buff管理。
	128KB	192KB	用户自定义命令。
BAR2	0	64MB	Using RC DMA使用。
BAR4	0	4MB	wired寄存器区，用于操作DMA。

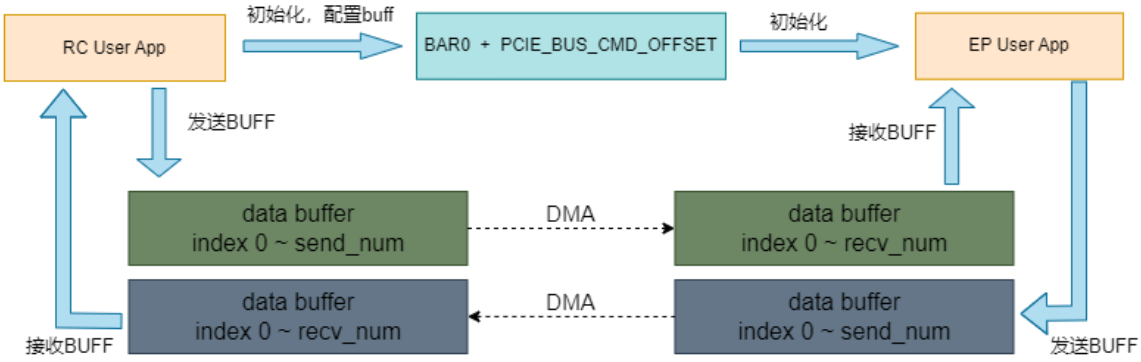
4.2 系统架构

PCIE RC与EP系统框图



4.3 业务BUFF管理结构及传输

BUFF管理



这里从RC的角度简单说明下从初始化到一次完整的读写流程：

4.3.1 初始化阶段

主要是做两件事：

1. 根据VENDOR_ID和DEVICE_ID获取当前系统所有PCIE设备信息。

   内部主要是通过libpci接口遍历系统PCIE设备并映射BAR空间到应用层。

   支持传递不同的设备ID重复调用`rk_pcie_register_all_dev`接口获取设备信息。

2. 调用`rk_pcie_init`初始化EP，可通过配置`struct pcie_dev_st`结构体来选择不同EP设备并且初始化，配置信息会通过BAR0空间传递给EP设备。

   RC端会根据配置的recv buff和send buff信息通过`/dev/pcie-rkep*`节点mmap一块物理连续的大内存，在应用层会根据配置信息分割成不同buff，以供后面DMA传输使用。每个EP卡设备都会在RC端生成对应的`/dev/pcie-rkep*`节点，并且申请单独的连续大内存。

   EP端收到配置信息后，初始化BUFF状态，并调用DRM接口申请物理连续的内存。

   RC和EP的BUFF信息最终会赋值到BAR0特定区域，这块区域用来对BUFF状态管理。

```
struct pcie_dev_st dev;
RK_PCIE_HANDLES rk_handle;
//注册系统中所有匹配的PCIE设备
rk_handle = rk_pcie_register_all_dev(VENDOR_ID, DEVICE_ID, &dev_max_num);
//选择第一个匹配设备
dev.send_buff_num = 5;    //发送buff个数
dev.send_buff_size = send_buff_size; //单个buff大小
dev.recv_buff_num = 5;
dev.recv_buff_size = recv_buff_size;
dev.dev_num = 0; //第1个设备
//初始化EP设备
rk_pcie_init(rk_handle, &dev);
```

4.3.2 发送数据

发送数据可以分为3个步骤

1. 查询是否有可用BUFF。
2. 获取可用BUFF，并写入数据。
3. 发送BUFF到对端。

以发送BUFF举例，初始化配置`send_buff_num = 5`说明系统中最多只有5个发送BUFF，发送端调用接口获取可用BUFF后，可以通过`struct pcie_buff_node`的属性获取BUFF物理地址和虚拟地址以及BUFF大小，然后进行业务操作。最后调用发送接口把BUFF数据传输到对端，发送接口内部会先对BUFF做flush cache，保证物理内存是最新的数据，然后调用DMA接口搬运数据到对端。DMA传输的数据大小就是`node->size`，不能超过BUFF的大小。

注意：发送到对端的BUFF，需要对端释放才会重新变成可用BUFF，对端一直不释放整个流程就会被阻塞住。

`rk_pcie_is_avaiable_buff`暂不支持阻塞模式，所以需要通过调用sleep来循环等待。

```
ret = rk_pcie_is_avaiable_buff(rk_handle, &dev, E_SEND);
if (ret) {
    struct pcie_buff_node *node = rk_pcie_get_buff(rk_handle, &dev, E_SEND);
    if(node) {
        node->size = rk_pcie_get_buff_max_size(rk_handle, &dev, E_SEND);
        //操作buff 写入数据
        rk_pcie_send_buff(ctx->rk_handle, &ctx->dev, node);
    }
} else {
    usleep(1000);
}
```

4.3.3 接收数据

接收数据可以分为3个步骤

1. 查询是否有可用BUFF。
2. 获取可用BUFF，并读取数据。
3. 释放BUFF。

接收BUFF和发送BUFF内部实现类似，接收BUFF是根据初始化`recv_buff_num`来配置系统中BUFF个数。具体行为可以参考发送数据章节。BUFF可用数据的大小就是`node->size`。

```
ret = rk_pcie_is_avaiable_buff(rk_handle, &dev, E_SEND);
if (ret) {
    struct pcie_buff_node *node = rk_pcie_get_buff(rk_handle, &dev, E_RECV);
    if(node) {
        //操作buff 读取数据
        rk_pcie_release_buff(rk_handle, &dev, node);
    }
} else {
    usleep(1000);
}
```

4.3.4 BUFF配置

1. RC端发送到EP的BUFF配置

   解码卡场景下，RC端是发送裸码流到EP，单路的数据量较少。可以申请`4KB*video chn`的BUFF大小。每路视频流单次传输4KB的大小，这样可以保证每次传输的视频流都可以更新数据，并且每次数据量不会过小导致浪费PCIE带宽。

BUFF数据建议配置3-5个，方便轮转使用。最少必须配置三个，如果有对BUFF做其它处理需要增加BUFF数量，不然会导致轮转卡主影响传输数率。

多线程传输可以适当加大BUFF数量。

2. EP发送到RC的BUFF配置。

EP发送到RC的数据，正常是解码后的数据一般比较大，比如1080p解成YUV422的单帧数据大概是3M的大小。单个BUFF可以配置为3MB-6MB大小，每次传输1路或者2路解码后的数据。因为需要多个BUFF轮转，单个BUFF不易配置太大会占用太大的内存空间。

4.4 用户层内存配置

RC/EP均支持2种内存申请方式：

- RC端：

1. pcie-rkep驱动申请内存，SDK通过mmap映射获取。

使能配置：CONFIG_PCIE_FUNC_RKEP_USERPAGES=y （默认不使能）

修改内存大小：修改驱动pcie-rkep.c RKEP_USER_MEM_SIZE宏 （默认是64M）

2. 通过HugePages 申请内存。(默认方式)

- EP端：

1. 通过DRM驱动申请内存，需要在dts中对CMA预先分配。

DMA内存大小修改设备树：arch/arm64/boot/dts/rockchip/rk3588-linux.dtsi

```
@@ -71,7 +71,7 @@ reserved-memory {
cma {
    compatible = "shared-dma-pool";
    reusable;
    reg = <0x0 0x50000000 0x0 0x10000000>; //从地址0x50000000开始256MB大小
    linux,cma-default;
};
```

2. 通过HugePages 申请内存。(默认方式)

不同内存申请方式可以在SDK中通过编译参数`MEM_CONFIG`来指定

- RC端：rkep / hugepage
- EP端：rkdrm / hugepage

说明：

使用HugePages方式申请需要对系统进行配置，可以查看HugePage章节进行配置。

4.5 用户层驱动优点

把PCIe的数据传输服务驱动提到用户层主要是方便使用和性能优化两方面考虑。

使用方面：

- 基于libpci的驱动，方便不同平台的移植；
- 方便直接跟应用对接，可以直接调用RK提供的rockit接口，或者第三方gstreamer接口等；

对比kernel接口驱动，应用层驱动在性能方面优化主要有：

- 使用轮询替换中断，减少频繁中断导致的CPU模式切换开销；
- 减少kernel和userspace切换导致的CPU模式切换开销；
- 建议绑定CPU，减少多核调度开销;
- 去掉kernel space和userspace之间内存拷贝开销，可以做到memory zero-copy；
- 使用系统Huge Page memory，减少TLB cache miss;

5. EP卡业务基础配置

5.1 EP卡用户层设备驱动

EP卡用户层设备驱动指RC端运行的EP function驱动，包括内核模块和用户层APP。

5.1.1 Using-RC-DMA模式

Using-RC-DMA模式只能工作在EP卡对接RK RC设备情况下，该模式下可以让PCIE速率达到最大化。

默认配置下只会使用EP端的DMA，Using-RC-DMA模式的原理就是把RC端的DMA使用起来。EP写数据到RC，则调用EP的DMA，RC写数据到EP，则调用RC的DMA，互不干扰。

RC端DMA有限制只能搬运到BAR有做 memory inbound 映射的物理空间，所以这里把BAR2给RC DMA使用。默认大小是64M，可以在EP端dts中修改配置大小。这个内存只是RC写数据到EP的时候使用，也就是初始化阶段配置的`send_buff_size * send_buff_num`大小。

使能模式

RC以及EP设备驱动不需要修改，只需要上层应用编译阶段添加编译宏即可`-DPCIE_DMA_RC_USING_RC_DMA`

使能成功后可以在RC应用看到如下LOG:

```
- id=356a1d87
- magic=524b4550, ver=1
- bar2 bus_addr=0x900000000, cpu_addr=0x40000000
- using rc udma    //表示运行在Using-RC-DMA模式
```

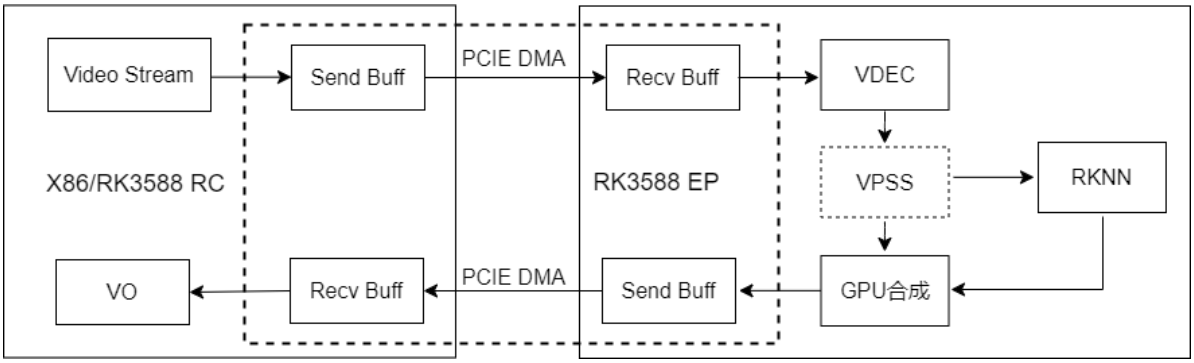
EP应用LOG:

```
wait rc init....
rc init success, start ep init!!
using rc dma: 1     //该值为1，表示运行在Using-RC-DMA模式
```

6. EP卡典型业务场景

6.1 EP视频加速卡

视频加速卡是将RC的视频流通过PCIE总线传输给EP解码并处理后再通过PCIE总线传给主片显示 。基本的数据流处理如下图所示:



码流发送端首先从视频流中获取码流数据，将其拷贝至准备好的 send stream buffer 中，然后通过 PCIE 的 DMA 将码流数据发送到 PCIE 对端的 recv stream buffer 中，对端再将码流取出通过VDEC解码送入VPSS，VPSS输出2路数据，一路送入GPU合成，一路数据送入RKNN进行处理。把合成后的数据拷贝至准备好的send stream buffer 中，然后通过 PCIE 的 DMA 将数据发送到 PCIE 对端的recv stream buffer 中，对端将数据取出送入VO模块显示。

SDK已经实现发送端和接收端的 stream buffer ， 初始化阶段配置参数即可使用，用户只需要关注业务流程。

7. 常见问题处理

7.1 EP卡启动异常排查

EP卡SPL阶段打印

```
U-Boot SPL board init
RKEP: Init PCIe EP
RKEP: 175301 - PHY Mode 0x4
RKEP: 176542 - RefClock in common clock_mode
RKEP: 187578 - BAR0: 0x3c000000
RKEP: 187848 - BAR2: 0x010000000
RKEP: 188227 - init PCIe fast Link up
RKEP: 208576 - Link up 230011
RKEP: 243683 - Done
```

问题简单分析:

- 如设备仅运行到"RKEP: Init PCIe EP"，通常为枚举fail，请确认EP卡的参考时钟、12V供电是否正常，PERST#是否释放

EP卡Kernel阶段打印

```
[root@RK3588:/]# dmesg | grep pci
[    3.371073] rk-pcie-ep fe150000.pcie: bar0: assigned [0x3c000000-3c3fffff]
[    3.371890] rk-pcie-ep fe150000.pcie: bar2: assigned [0x40000000-43ffffff]
[    3.371901] rk-pcie-ep fe150000.pcie: no vpcie3v3 regulator found
[    3.371918] rk-pcie-ep fe150000.pcie: already linkup
[    3.372001] rk-pcie-ep fe150000.pcie: register misc device pcie_ep
[    3.725930] ehci-pci: EHCI PCI platform driver
[    3.869983] pcie30_avdd1v8: supplied by avcc_1v8_s0
[    3.870541] pcie30_avdd0v75: supplied by avdd_0v75_s0
```

EP卡设备驱动打印

内核模块:

```
rk3588_s:/ # dmesg | grep rkep
[    2.278964] pcie-rkep 0000:01:00.0: success to request msi irq
               # 成功申请MSI 中断 log
[    2.280601] pcie-rkep 0000:01:00.0: successfully allocate continuouse buffer
for userspace   # 成功预留用户内存，默认不开
[    2.280619] pcie-rkep 0000:01:00.0: vid=1d87
               # 正常访问EP卡config空间
[    2.280625] pcie-rkep 0000:01:00.0: did=356a
[    2.280629] pcie-rkep 0000:01:00.0: obj_info magic=524b4550, ver=100
               # 正常访问EP卡bar空间
```

7.2 异常处理

实际产品在使用过程中，会因为系统稳定性或者外部环境干扰等各种原因，导致一些异常，PCIe的物理设计机制已经能够进行一定程度的抗干扰，比如每一层协议有增加CRC，在出错时有重传机制，保证控制器输出的数据本身一定是没有错误的数据，如果发生仅通过PCIe控制器无法恢复的问题，则需要更多手段来保障设备的运行。

7.2.1 EP状态信息

在软件上，建议在EP运行一个小程序用来表征EP设备当前的健康状态，可以通过tick进行简单更新，这样在RC端可以通过读取EP的健康状态来确定设备是否正常工作。我们在Demo的BAR0中预留了一段空间用来配合实现EP健康状态信息的更新和查询。

7.2.2 Watchdog

RK3588带有watchdog，可以在系统异常的时候自动复位。watchdog可以考虑加在健康状态信息程序中进行喂狗，系统异常时复位系统。这种情况比较可能的是EP端的PCIe模块本身工作异常，但是系统其它模块发生异常导致系统卡住，无法正常运行健康状态信息更新程序。

如果EP端因系统异常主动进行复位，那么RC的软件层是不知道的，在EP完成复位后PCIe物理层会自动进行重连，但是RC扫描过程配置到EP的寄存器如BAR地址等，需要EP自行恢复，才能跟RC继续通讯。

7.2.3 Hot Reset

Host Reset是PCIe定义的信号，通过普通数据线传输实现，EP端控制器可以检测到该信号彬触发中断，需要注意的是HotReset在控制器硬件会自动进行响应，复位掉BAR相关的寄存器，所以在收到Hot Reset的中断处理中需要做一次EP的初始化，保证BAR和ATU配置正确。

7.2.4 Warm Reset(PERST)

PERST是PCIe模块的独立复位信号，观察发现在x86的重启过程中会拉PERST信号，但是不会对PCIe Slot的供电进行重新上下电，所以EP端需要保证PERST时能够进行完整复位，在RK平台的具体做法就是按我们建议之间接到RK3588芯片和PMIC的NPOR信号，目标是达到跟上电复位一样的效果。

如果部分产品对于RC端可控，对上电复位要求没有那么高，PERST可以作为一个可中断的GPIO，在中断触发时通过软件配置RK3588芯片的global reset进行全局复位。

7.2.5 Cold Reset(Power On Reset)

上电复位对应设备的开机流程。

7.3 RC端x86平台执行初始化PCIE报如下错误mmap failed, Operation not permitted

```
root@rk:/home/rc-sample# ./pcie_test 10240
resource path = /sys/bus/pci/devices/0000:01:00.0/enable
resource path = /sys/bus/pci/devices/0000:01:00.0/resource0
mmap failed, ret = 0, Operation not permitted
ep dev num : 1
Segmentation fault (core dumped)
```

答：这个是因为ubuntu从20.04开始有个linux 内核锁定的功能，如果在bios启动了安全启动功能，就会打开内核锁定导致去mmap的时候失败。 进入bios下关闭安全启动就不会报错了。

7.4 HugePage配置

7.4.1 X86平台

查看Hugepagesize大小

```
rk@debian:~$ cat /proc/meminfo | grep Huge
HugePages_Total:       2
HugePages_Free:        2
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    2048 kB
Hugetlb:         4096 kB
```

修改大小为1GB

- 修改cmdline参数，编解/etc/default/grub, 添加default_hugepagesz=1G

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet default_hugepagesz=1G hugepagesz=1G
hugepages=1"
GRUB_CMDLINE_LINUX=""
```

- 生成配置

```
/usr/sbin/grub-mkconfig -o /boot/grub/grub.cfg
```

- 重启设备 reboot 或者 systemctl poweroff
- 查看size和count，cat /proc/meminfo | grep Hugepagesize

```
rk@debian:~$ cat /proc/meminfo | grep Huge
HugePages_Total:       1
HugePages_Free:        1
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB
Hugetlb:         1048576 kB
```

7.4.2 RK平台

内核开启Hugepage配置

```
+++ b/arch/arm64/configs/rockchip_linux_defconfig
@@ -1,5 +1,10 @@
 CONFIG_DEFAULT_HOSTNAME="localhost"
 CONFIG_SYSVIPC=y
+CONFIG_HUGETLBFS=y
 CONFIG_NO_HZ=y
 CONFIG_HIGH_RES_TIMERS=y
 CONFIG_PREEMPT_VOLUNTARY=y
```

配置Hugepage默认大小以及个数:

cmdline增加三个属性`default_hugepagesz=1G hugepagesz=1G hugepages=1`,配置Hugepage为1G大小，并且可使用个数为1。

```
+++ b/arch/arm64/boot/dts/rockchip/rk3588-linux.dtsi
@@ -6,7 +6,7 @@

 / {
        chosen: chosen {
-               bootargs = "earlycon=uart8250,mmio32,0xfeb50000 console=ttyFIQ0
irqchip.gicv3_pseudo_nmi=0 clk_gate.always_on=1 pm_domains.always_on=1
root=PARTUUID=614e0000-0000 rw rootwait";
+               bootargs = "earlycon=uart8250,mmio32,0xfeb50000 console=ttyFIQ0
irqchip.gicv3_pseudo_nmi=0 clk_gate.always_on=1 pm_domains.always_on=1
default_hugepagesz=1G hugepagesz=1G hugepages=1 root=PARTUUID=614e0000-0000 rw
rootwait";
        };

        cspmu: cspmu@fd10c000 {
```

重新编译内核烧写，开机后查看/proc/meminfo确认是否添加成功

```
[root@RK3588:~]# cat /proc/meminfo | grep Huge
HugePages_Total:       1
HugePages_Free:        1
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB
Hugetlb:         1048576 kB
```

8. API参考

8.1 PCIE RC

8.1.1 rk_pcie_register_all_dev

【描述】

注册系统中所有匹配的PCIE设备，返回数量以及所有匹配设备的句柄。

【语法】

```
RK_PCIE_HANDLES rk_pcie_register_all_dev(int vendor_id, int device_id, int
*dev_max_num)
```

【参数】

参数名称	描述	输入/输出
vendor_id	PCIE 厂商标识	输入
device_id	PCIE 产品标识	输入
dev_max_num	匹配的设备数量	输出

【返回值】

返回值	描述
0	失败。
不为0	成功

【举例】

```
struct rc_context_st
{
    RK_PCIE_HANDLES rk_handle;
    struct pcie_dev_st dev;
    int dev_max_num;
    pthread_t thr_recv_ep;
    pthread_t thr_send_ep;
    long buff_size;
    char recv_ep_exit;
    char send_ep_exit;
};

struct rc_context_st rc_ctx;
//注册系统中所有匹配的PCIE设备
rc_ctx.rk_handle = rk_pcie_register_all_dev(VENDOR_ID, DEVICE_ID,
&rc_ctx.dev_max_num);

printf("ep dev num : %d\n", rc_ctx.dev_max_num);

if(rc_ctx.dev_max_num == 0) {
    printf("no ep device, exit ... \n");
} else {
    //选择第一个匹配设备
    rc_ctx.dev.send_buff_num = 5;
    rc_ctx.dev.send_buff_size = rc_ctx.buff_size;
    rc_ctx.dev.recv_buff_num = 5;
    rc_ctx.dev.recv_buff_size = rc_ctx.buff_size;
    rc_ctx.dev.dev_num = 0; //第1个设备

    //初始化EP设备
    rk_pcie_init(rc_ctx.rk_handle, &rc_ctx.dev);

    rc_ctx.recv_ep_exit = 0;
    rc_ctx.send_ep_exit = 0;

    pthread_create(&rc_ctx.thr_recv_ep, 0, recv_ep_straem_thread, &rc_ctx);
    pthread_create(&rc_ctx.thr_send_ep, 0, send_ep_straem_thread, &rc_ctx);

    getchar();

    rc_ctx.recv_ep_exit = 1;
    rc_ctx.send_ep_exit = 1;

    pthread_join(rc_ctx.thr_recv_ep, NULL);
    pthread_join(rc_ctx.thr_send_ep, NULL);

    //反初始化EP设备
```

```
    rk_pcie_deinit(rc_ctx.rk_handle, &rc_ctx.dev);
}
//注销系统中所有匹配的PCIE设备
rk_pcie_unregister_all_dev(rc_ctx.rk_handle);
```

【相关主题】

rk_pcie_unregister_all_dev

8.1.2 rk_pcie_unregister_all_dev

【描述】

注销系统中所有匹配的PCIE设备。

【语法】

```
int rk_pcie_unregister_all_dev(RK_PCIE_HANDLES handles)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入

【返回值】

返回值	描述
0	正常
-1	参数为NULL

【举例】

请参见rk_pcie_register_all_dev的举例。

【相关主题】

rk_pcie_register_all_dev

8.1.3 rk_pcie_init

【描述】

初始化EP设备

【语法】

```
int rk_pcie_init(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

请参见rk_pcie_register_all_dev的举例。

【相关主题】

rk_pcie_deinit

8.1.4 rk_pcie_deinit

【描述】

反初始化pcie，释放资源

【语法】

```
int rk_pcie_deinit(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

请参见rk_pcie_register_all_dev的举例。

【相关主题】

rk_pcie_init

8.1.5 rk_pcie_boot_init

【描述】

初始化PCIE BOOT

【语法】

```
int rk_pcie_boot_init(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

```
//选择第一个匹配设备
rc_ctx.dev.send_buff_num = 5;
rc_ctx.dev.send_buff_size = rc_ctx.buff_size;
rc_ctx.dev.recv_buff_num = 5;
rc_ctx.dev.recv_buff_size = rc_ctx.buff_size;
rc_ctx.dev.dev_num = 0; //第1个设备

rk_pcie_boot_init(rc_ctx.rk_handle, &rc_ctx.dev);

// download
printf("download uboot....\n");
rk_ep_download_firmware(&rc_ctx, uboot_path, RKEP_LOAD_UBOOT_ADDR);
printf("download boot....\n");
rk_ep_download_firmware(&rc_ctx, boot_path, RKEP_LOAD_BOOT_ADDR);

rk_pcie_boot_run(rc_ctx.rk_handle, &rc_ctx.dev);

rk_pcie_boot_deinit(rc_ctx.rk_handle, &rc_ctx.dev);
```

【相关主题】

rk_pcie_boot_deinit

8.1.6 rk_pcie_boot_deinit

【描述】

反初始化PCIE BOOT

【语法】

```
int rk_pcie_boot_deinit(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

rk_pcie_boot_init

【相关主题】

rk_pcie_boot_init

8.1.7 rk_pcie_boot_download_firmware

【描述】

下载PCIE BOOT固件

【语法】

```
int rk_pcie_boot_download_firmware(RK_PCIE_HANDLES handles, struct pcie_dev_st
*dev, void *firmware, size_t firmware_size, unsigned long remote_load_addr)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入
firmware	固件数据	输入
firmware_size	固件大小	输入
remote_load_addr	远程固件加载地址	输入

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

```
int fd;
unsigned long len;
unsigned long size = 1024 * 1024;
void *buffer = malloc(size);

fd = open(firmware, O_RDONLY);
if(fd == -1) {
printf(" open %s file error, ret = %d \n", firmware, fd);
return -1;
}

while ((len = read(fd, buffer, size)) > 0) {
    rk_pcie_boot_download_firmware(rc_ctx->rk_handle, &rc_ctx->dev, buffer, len,
load_addr);
    load_addr += len;
}

close(fd);
```

【相关主题】

rk_ep_download_firmware

8.1.8 rk_pcie_boot_run

【描述】

启动PCIE BOOT固件

【语法】

```
int rk_pcie_boot_run(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

rk_pcie_boot_init

【相关主题】

rk_pcie_boot_run

8.1.9 rk_pcie_get_ep_info

【描述】

获取EP状态信息

【语法】

```
int rk_pcie_get_ep_info(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev, void
*status, size_t status_size)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入
status	void指针，状态结构体由用户自定义	输出
status_size	状态结构体size	输入

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

```
struct ep_info_st {
    unsigned int link_status;
    unsigned int temp;
    long long timestamp;
    unsigned char cpu_load;
    unsigned char npu_load;
    unsigned char gpu_load;
};

long long timestamp;
while(1) {
    ret = rk_pcie_get_ep_info(rc_ctx.rk_handle, &rc_ctx.dev, ep_info,
sizeof(struct ep_info_st));
    if(timestamp == ep_info->timestamp) {
        printf("EP device status is abnormal \n");
        break;
    }
    timestamp = ep_info->timestamp;
}
```

【相关主题】

rk_pcie_get_ep_info

8.1.10 rk_pcie_get_user_cmd

【描述】

获取BAR0特定位置一块内存指针，用于用户自定义命令区

【语法】

```
void *rk_pcie_get_user_cmd(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入

【返回值】

返回值	描述
NULL	异常
非NULL	正常

【举例】

```
struct user_cmd_st {
    unsigned int max_chn;
    unsigned int width;
    unsigned int height;
    unsigned int codecid;
    unsigned int output_id;
    unsigned int init;
};

user_cmd = (struct user_cmd_st *)rk_pcie_get_user_cmd(rc_ctx.rk_handle,
&rc_ctx.dev);
user_cmd->init = 0;

user_cmd->max_chn = rc_ctx.max_chn;
user_cmd->width = rc_ctx.width;
user_cmd->height = rc_ctx.height;
user_cmd->codecid = rc_ctx.codecid;
user_cmd->output_id = 1; //0:RGB888 1:BGRA8888
user_cmd->init = 1;
```

【相关主题】

rk_pcie_get_user_cmd

8.1.11 rk_pcie_is_avaiable_buff

【描述】

查询是否有可用PCIE buff

【语法】

```
int rk_pcie_is_avaiable_buff(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev,
enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入
type	PCIE Buff类型	输入

【返回值】

返回值	描述
-1	参数错误
0	无可用Buff
1	有可用Buff

【举例】

```
ret = rk_pcie_is_avaiable_buff(ctx->rk_handle, &ctx->dev, E_SEND);

if (ret) {
    struct pcie_buff_node *node = rk_pcie_get_buff(ctx->rk_handle, &ctx->dev,
E_SEND);
    if(node) {
        node->size = rk_pcie_get_buff_max_size(ctx->rk_handle, &ctx->dev,
E_SEND);

        add_check_info(node->vir_addr + node->size - CHECK_SIZE);

        if(check_info(node->vir_addr + node->size - CHECK_SIZE) != 0) {
            printf("===== add check info src error !!!!\n");
        }

        rk_pcie_send_buff(ctx->rk_handle, &ctx->dev, node);
    }
} else {
    usleep(1000);
}
```

【相关主题】

rk_pcie_is_avaiable_buff

8.1.12 rk_pcie_get_buff

【描述】

获取可用PCIE buff

【语法】

```
struct pcie_buff_node *rk_pcie_get_buff(RK_PCIE_HANDLES handles, struct
pcie_dev_st *dev, enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入
type	PCIE Buff类型	输入

【返回值】

返回值	描述
NULL	无可用Buff
非0	可用Buff指针

【举例】

rk_pcie_is_avaiable_buff

【相关主题】

rk_pcie_is_avaiable_buff

8.1.13 rk_pcie_send_buff

【描述】

发送PCIE buff

【语法】

```
int rk_pcie_send_buff(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev, struct
pcie_buff_node *pbuff)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
-1	参数错误
0	发送成功

【举例】

rk_pcie_is_avaiable_buff

【相关主题】

rk_pcie_is_avaiable_buff

8.1.14 rk_pcie_release_buff

【描述】

释放PCIE buff

【语法】

```
int rk_pcie_release_buff(RK_PCIE_HANDLES handles, struct pcie_dev_st *dev, struct
pcie_buff_node *pbuff)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
-1	参数错误
0	释放成功

【举例】

```
struct pcie_buff_node *node = rk_pcie_get_buff(ctx->rk_handle, &ctx->dev,
E_RECV);
if(node) {
    if(check_info(node->vir_addr + node->size - CHECK_SIZE) == 0) {
        total_size += node->size;
        memset(node->vir_addr + node->size - CHECK_SIZE, 0, CHECK_SIZE);
    } else {
        printf("check data ==================== error \n");
        for(int i=0;i<CHECK_SIZE;i++) {
            char *p = node->vir_addr + node->size - CHECK_SIZE;
            printf("%d ",p[i]);
        }
        printf("\ncheck data =======size = %d======== error \n",node->size);
    }

    rk_pcie_release_buff(ctx->rk_handle, &ctx->dev, node);
} else {
    usleep(1000);
}
```

【相关主题】

rk_pcie_release_buff

8.1.15 rk_pcie_get_buff_max_size

【描述】

获取buff支持的最大size

【语法】

```
size_t rk_pcie_get_buff_max_size(RK_PCIE_HANDLES handles, struct pcie_dev_st
*dev, enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
handles	所有匹配设备的句柄	输入
dev	PCIE设备配置结构体	输入
type	PCIE Buff类型	输入

【返回值】

返回值	描述
-1	参数错误
非0	buff size

【举例】

rk_pcie_is_avaiable_buff

【相关主题】

rk_pcie_get_buff_max_size

8.1.16 数据类型

PCIE RC相关数据类型、数据结构定义如下：

- RK_PCIE_HANDLES：定义设备的句柄。
- EBuff_Type：定义PCIE Buff类型。
- struct pcie_buff_node：定义PCIE buff属性结构体。
- struct pcie_dev_st：定义PCIE EP设备结构体。

8.1.16.1 RK_PCIE_HANDLES

【说明】

定义设备的句柄。

【定义】

```
typedef void* RK_PCIE_HANDLES;
```

8.1.16.2 EBuff_Type

【说明】

定义PCIE Buff类型。

【定义】

```
enum EBuff_Type {
    E_SEND = 0,
    E_RECV,
};
```

【成员】

成员名称	描述
E_SEND	发送buff。
E_RECV	接收buff。

8.1.16.3 pcie_buff_node

【说明】

定义PCIE buff属性结构体。

【定义】

```
struct pcie_buff_node{
    void *vir_addr;
    size_t phy_addr;
    size_t size;
};
```

【成员】

成员名称	描述
vir_addr	虚拟地址。
phy_addr	物理地址。
size	内存大小。 发送buff：需要用户配置数据大小，不能超过总的buff内存大小。 接收buff：对端发送过来的数据大小。

8.1.16.4 pcie_dev_st

【说明】

定义PCIE EP设备结构体。

【定义】

```
struct pcie_dev_st {
    unsigned int send_buff_num;
    size_t send_buff_size;
    unsigned int recv_buff_num;
    size_t recv_buff_size;
    unsigned int dev_num;
};
```

【成员】

成员名称	描述
send_buff_num	PCIE发送buff的数量，用于buff轮询。
send_buff_size	发送buff的内存大小。
recv_buff_num	PCIE接收buff的数量，用于buff轮询。
recv_buff_size	接收buff的内存大小。
dev_num	PCIE EP设备ID，不同handle下ID可以相同。

8.2 PCIE EP

8.2.1 rk_pcie_init

【描述】

初始化PCIE设备

【语法】

```
int rk_pcie_init(void)
```

【参数】

无

【返回值】

返回值	描述
0	正常
-1	参数异常

【举例】

```
typedef struct
{
    pthread_t thr_recv_rc;
    pthread_t thr_send_rc;
    char recv_rc_exit;
    char send_rc_exit;
```

```
} ep_context_st;

int main(int argc, char **argv)
{
    int ret = 0;
    ep_context_st ep_ctx;

    ret = rk_pcie_init();

    if(ret != 0) {
        printf("pcie init ep error....\n");
        return -1;
    }

    ep_ctx.recv_rc_exit = 0;
    ep_ctx.send_rc_exit = 0;
    pthread_create(&ep_ctx.thr_recv_rc, 0, recv_rc_straem_thread, &ep_ctx);
    pthread_create(&ep_ctx.thr_send_rc, 0, send_rc_straem_thread, &ep_ctx);

    getchar();

    ep_ctx.recv_rc_exit = 1;
    ep_ctx.send_rc_exit = 1;

    pthread_join(ep_ctx.thr_recv_rc, NULL);
    pthread_join(ep_ctx.thr_send_rc, NULL);

    rk_pcie_deinit();

    return 0;
}
```

【相关主题】

rk_pcie_deinit

8.2.2 rk_pcie_deinit

【描述】

反初始化pcie，释放资源

【语法】

```
int rk_pcie_deinit(void)
```

【参数】

无

【返回值】

返回值	描述
0	正常
-1	失败

【举例】

请参见rk_pcie_init的举例。

【相关主题】

rk_pcie_init

8.2.3 rk_pcie_set_ep_info

【描述】

设置EP状态信息

【语法】

```
int rk_pcie_set_ep_info(void *status, size_t status_size)
```

【参数】

参数名称	描述	输入/输出
status	void指针，状态结构体由用户自定义	输入
status_size	状态结构体size	输入

【返回值】

返回值	描述
0	正常
-1	失败

【举例】

```
struct ep_info_st {
    unsigned int link_status;
    unsigned int temp;
    long long timestamp;
    unsigned char cpu_load;
    unsigned char npu_load;
    unsigned char gpu_load;
};

ep_info_st ep_info;
while(1) {
    ep_info.timestamp = time(NULL);
    rk_pcie_set_ep_info((void *)&ep_info, sizeof(struct ep_info_st));
```

```
}
```

【相关主题】

rk_pcie_get_ep_info

8.2.4 rk_pcie_get_user_cmd

【描述】

获取BAR0特定位置一块内存指针，用于用户自定义命令区

【语法】

```
void *rk_pcie_get_user_cmd(void)
```

【参数】

无

【返回值】

返回值	描述
NULL	异常
非NULL	正常

【举例】

```
struct user_cmd_st {
    unsigned int max_chn;
    unsigned int width;
    unsigned int height;
    unsigned int codecid;
    unsigned int output_id;
    unsigned int init;
};

ep_ctx.user_cmd = rk_pcie_get_user_cmd();

while(ep_ctx.user_cmd->init != 1) {
    usleep(1000);
}

ep_ctx.video_obj.width = ep_ctx.user_cmd->width;
ep_ctx.video_obj.height = ep_ctx.user_cmd->height;
ep_ctx.video_obj.input_mode = ep_ctx.user_cmd->codecid;
ep_ctx.video_obj.output_mode = ep_ctx.user_cmd->output_id;
ep_ctx.video_obj.max_chn = ep_ctx.user_cmd->max_chn;
ep_ctx.max_chn = ep_ctx.video_obj.max_chn;
```

【相关主题】

rk_pcie_get_user_cmd

8.2.5 rk_pcie_is_avaiable_buff

【描述】

查询是否有可用PCIE buff

【语法】

```
int rk_pcie_is_avaiable_buff(enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
type	PCIE Buff类型	输入

【返回值】

返回值	描述
0	无可用Buff
1	有可用Buff

【举例】

```
ret = rk_pcie_is_avaiable_buff(E_SEND);

if (ret) {
    struct pcie_buff_node *node = rk_pcie_get_buff(E_SEND);
    if (node) {
        node->size = rk_pcie_get_buff_max_size(E_SEND);

        add_check_info(node->vir_addr + node->size - CHECK_SIZE);

        if(check_info(node->vir_addr + node->size - CHECK_SIZE) != 0) {
            printf("============================ add check info src error
!!!!\n");
        }

        rk_pcie_send_buff(node);
    }
} else {
    usleep(1000);
}
```

【相关主题】

rk_pcie_is_avaiable_buff

8.2.6 rk_pcie_get_buff

【描述】

获取可用PCIE buff

【语法】

```
struct pcie_buff_node *rk_pcie_get_buff(enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
type	PCIE Buff类型	输入

【返回值】

返回值	描述
NULL	无可用Buff
非0	可用Buff指针

【举例】

rk_pcie_is_avaiable_buff

【相关主题】

rk_pcie_is_avaiable_buff

8.2.7 rk_pcie_send_buff

【描述】

发送PCIE buff

【语法】

```
int rk_pcie_send_buff(struct pcie_buff_node *pbuff)
```

【参数】

参数名称	描述	输入/输出
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
-1	参数错误
0	发送成功

【举例】

rk_pcie_is_avaiable_buff

【相关主题】

rk_pcie_is_avaiable_buff

8.2.8 rk_pcie_release_buff

【描述】

释放PCIE buff

【语法】

```
int rk_pcie_release_buff(struct pcie_buff_node *pbuff)
```

【参数】

参数名称	描述	输入/输出
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
-1	参数错误
0	释放成功

【举例】

```
struct pcie_buff_node *node = rk_pcie_get_buff(E_RECV);
if(node) {
     if(check_info(node->vir_addr + node->size - CHECK_SIZE) == 0) {
        total_size += node->size;
        memset(node->vir_addr + node->size - CHECK_SIZE, 0, CHECK_SIZE);
    } else {
        printf("check data ==================== error \n");
        for(int i=0;i<CHECK_SIZE;i++) {
            char *p = node->vir_addr + node->size - CHECK_SIZE;
            printf("%d ",p[i]);
        }
        printf("\ncheck data =======size = %d======== error \n",node->size);
    }

    rk_pcie_release_buff(node);
```

```
} else {
    usleep(1000);
}
```

【相关主题】

rk_pcie_release_buff

8.2.9 rk_pcie_get_buff_max_size

【描述】

获取buff支持的最大size

【语法】

```
size_t rk_pcie_get_buff_max_size(enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
type	PCIE Buff类型	输入

【返回值】

返回值	描述
-1	参数错误
非0	buff size

【举例】

rk_pcie_is_avaiable_buff

【相关主题】

rk_pcie_get_buff_max_size

8.2.10 数据类型

PCIE RC相关数据类型、数据结构定义如下：

- EBuff_Type：定义PCIE Buff类型。
- struct pcie_buff_node：定义PCIE buff属性结构体。

8.2.10.1 EBuff_Type

【说明】

定义PCIE Buff类型。

【定义】

```
enum EBuff_Type {
    E_SEND = 0,
    E_RECV,
};
```

【成员】

成员名称	描述
E_SEND	发送buff。
E_RECV	接收buff。

8.2.10.2 pcie_buff_node

【说明】

定义PCIE buff属性结构体。

【定义】

```
struct pcie_buff_node{
    void *vir_addr;
    size_t phy_addr;
    size_t size;
};
```

【成员】

成员名称	描述
vir_addr	虚拟地址。
phy_addr	物理地址。
size	内存大小。 发送buff：需要用户配置数据大小，不能超过总的buff内存大小。 接收buff：对端发送过来的数据大小。