Rockchip Developer Guide PCIE EP 标准卡

文件标识：RK-KF-YF-489

发布版本：V1.6.0

日　　期：2024-09-18

文件密级：□绝密　□秘密　□内部资料　■公开

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
RK3568	Linux 4.19

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

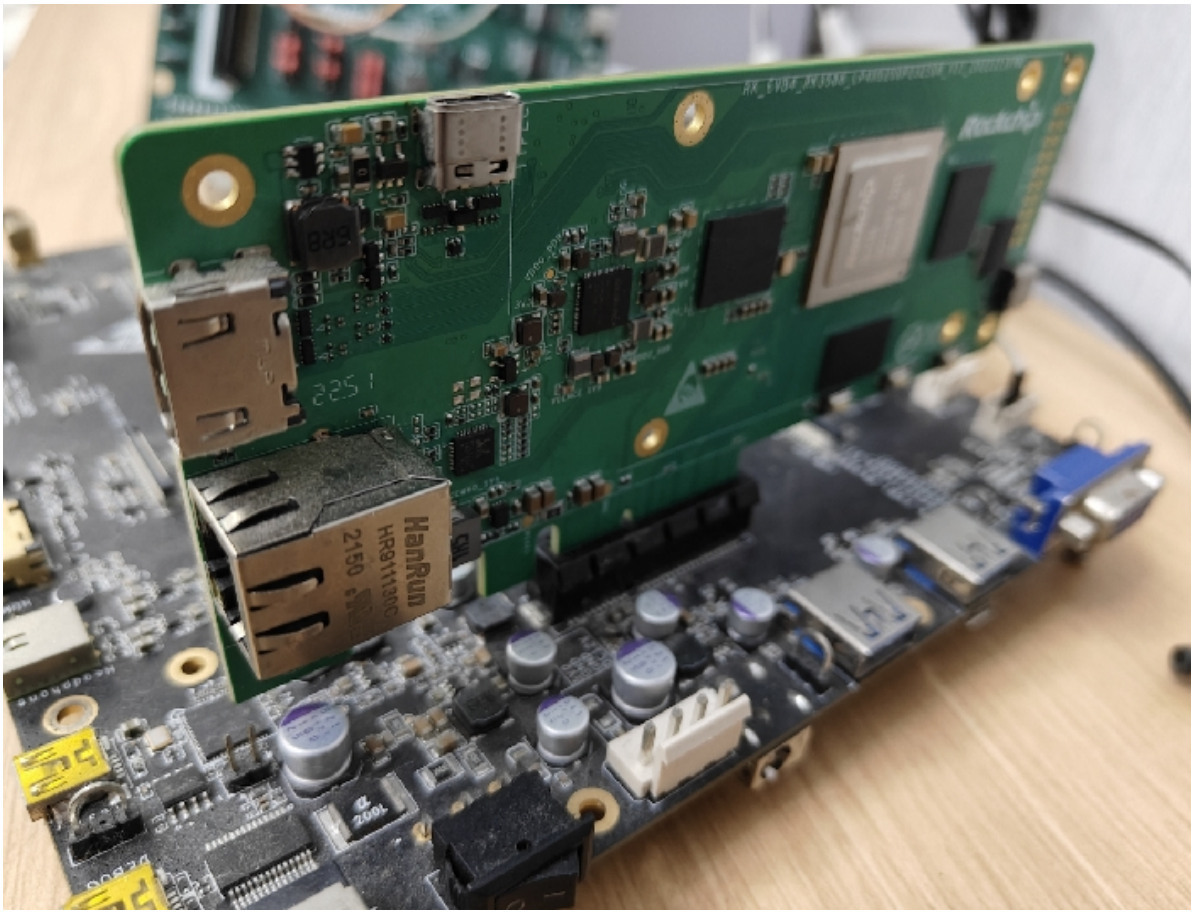
软件开发工程师

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
V1.1.0	yp.xiao	2023-05-22	更新SDK API以及部分配置说明
V1.2.0	yp.xiao	2023-06-15	添加EP卡固件编译、烧录、OTA升级说明 添加应用异常说明 更新 struct pcie_dev_st 结构体，支持超时配置、PCIE速率获取、RC DMA配置。
v1.3.0	Jon Lin	2023-10-07	增加pcie.bin说明
v1.4.0	yp.xiao, Jon Lin	2023-10-23	更新SDK API 添加EP验收标准章节 更新双存储说明 更新PCIE BIN说明
v1.5.0	Jon Lin	2024-05-23	更新PHY Mode设置
v1.6.0	Jon Lin	2024-09-18	添加用户态测试Demo - pcie_speed_test性能参考、增加BAR1/5支持

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
│   ├── common_dev_sdk          # EP卡业务基础，公共基础功能实现
│   └── include                 # EP卡业务基础，对外头文件
├── 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
│   └── pcie_auto_test          # EP卡功能自动化测试Demo
├── images                      # 功能验证固件
└── patch*                      # 补丁目录
```

开发包主要目录介绍：

common_dev_sdK目录对基础功能实现，包括dma、log、linked_list、rk_mem、chips等RC和EP通用组件。

- dma： 实现DMA传输，负责和芯片DMA配置交互。
- log：实现日志管理，对外提供统一日志接口。
- linked_list：实现基础链表功能。
- rk_mem：实现不同配置内存申请释放。
- chips：用于区分不同芯片。

ep_dev_sdk目录封装EP用户接口层实现，对外开放特定API方便用户开发，API头文件`include/rk_pcie_ep.h`。该目录下由三部分组成：

- 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`include/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_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。

include目录提供SDK对外接口头文件供用户调用开发。

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`。

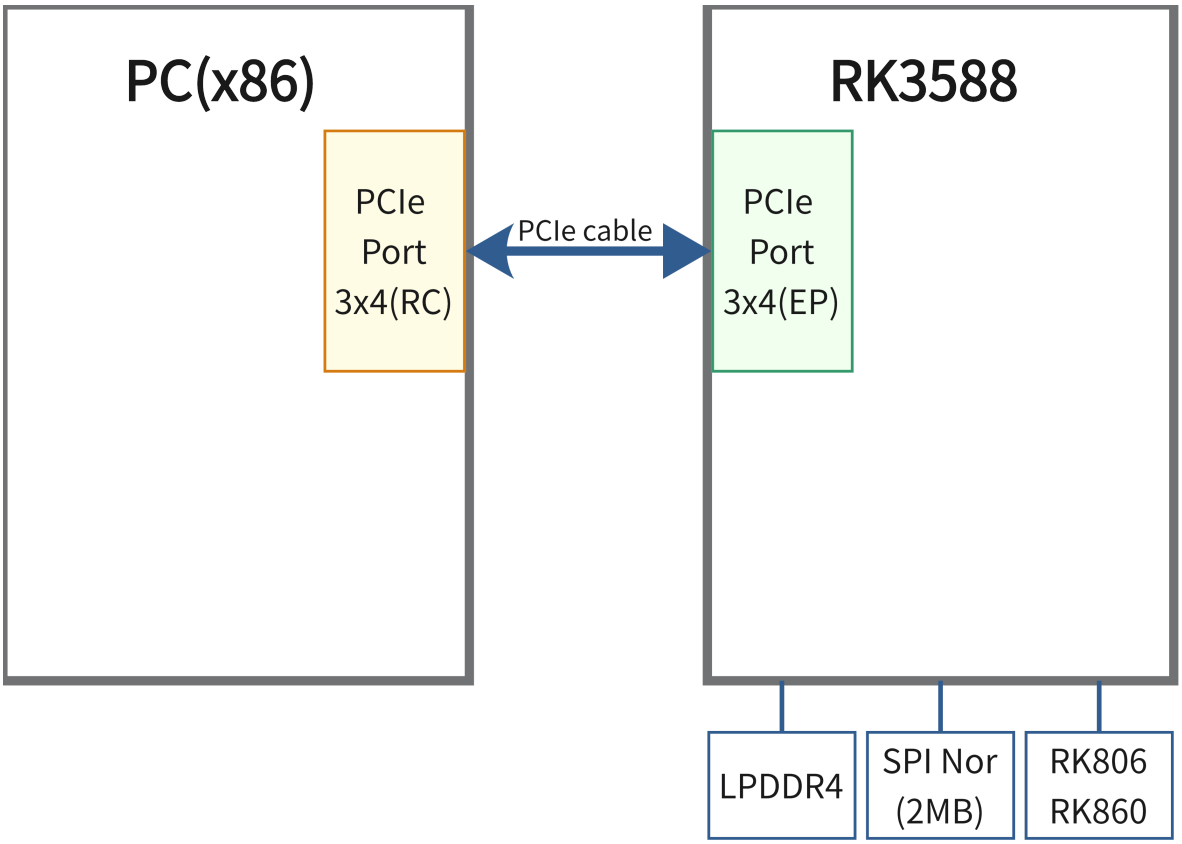
1.3 EP应用场景

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：

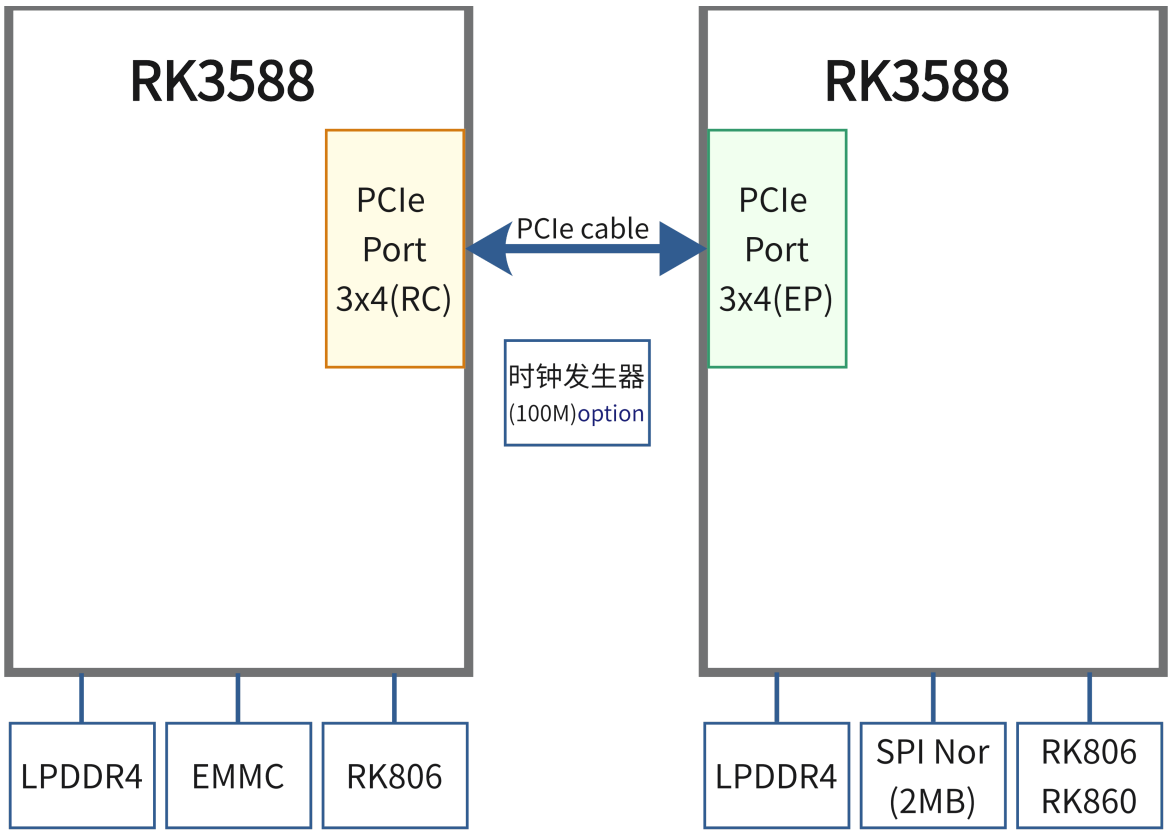


1.3.2 单EP卡对接RK RC

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：

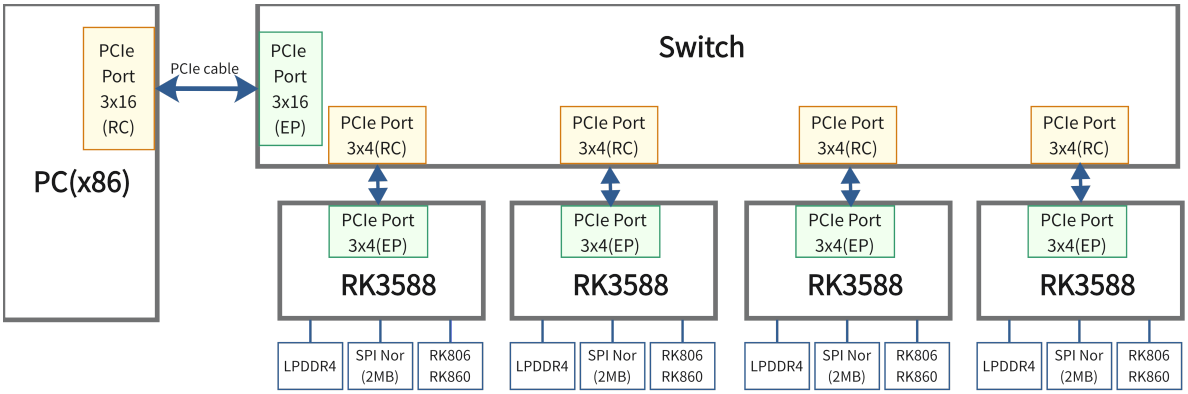


1.3.3 多EP卡对接通用RC

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

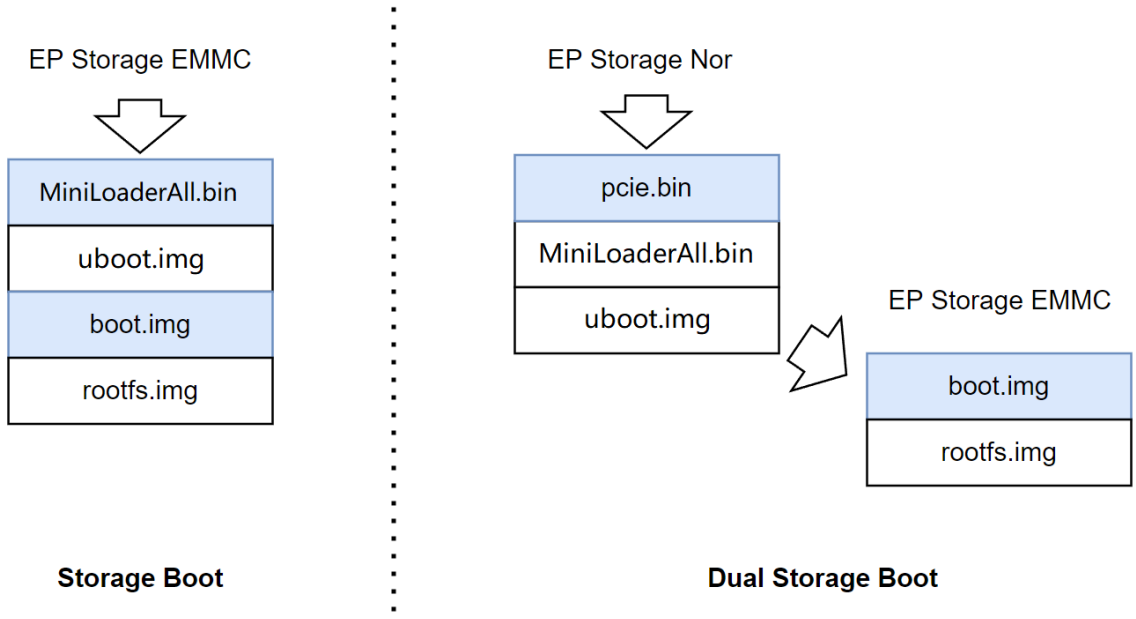
3.1 EP卡启动方案

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

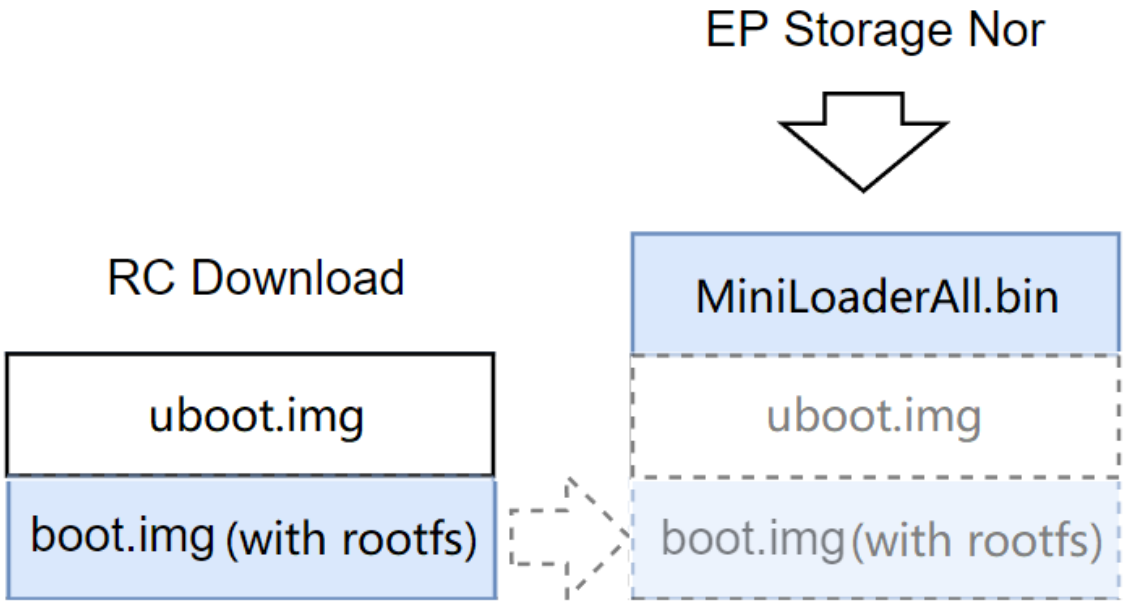
Nor快速启动 + EMMC大容量固件的双存储方案

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：

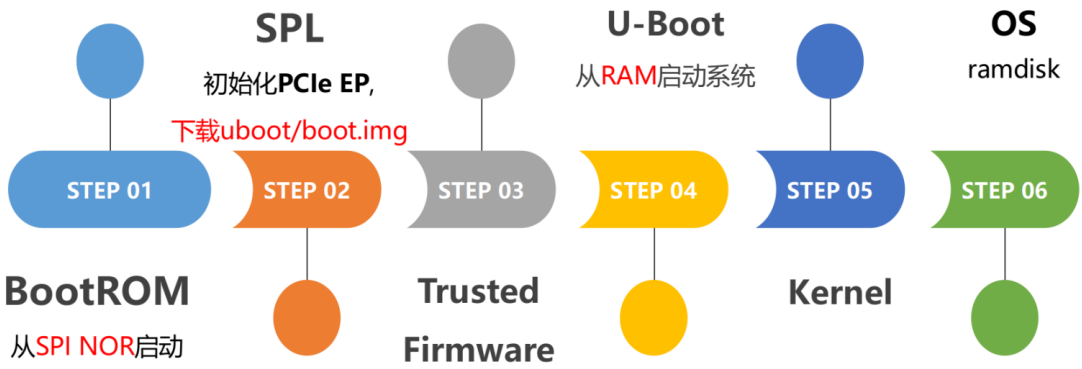
- 蓝色框图流程为运行PCIe驱动的流程，白色框图为无PCIe驱动流程
- 虚线框架代表由RC推送到EP内存的固件
- 主要流程说明：
  - MiniLoaderALL.bin 为 ddr.bin 和 spl.bin 的打包文件，其中：
    - ddr.bin：初始化ddr
    - spl.bin：初始化EP PCIe（可选），引导后级固件
  - boot.img：内核PCIe支持，涉及EP卡业务支持

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

PCIe bin仅支持完成PCIe控制器EP mode功能的初始化，且早于dram.bin初始化。

提供编译完成的pcie.bin以及源码供自定义修改，pcie.bin存放在SDK/rkbin/bin/rkxx/对应文件下，源码路径如下：`patch-*/bootloader`

建议客户直接修改pcie.bin配置，除非特殊修改再去通过源码自定义编译。

3.3.2.1 动态配置pcie.bin

pcie.bin支持直接修改bin文件中预留的配置空间从而实现常用的PCIe配置变换，目前已经实现以下配置修改：

- PCIe Generation
- PCIe lanes
- pcie.bin串口波特率
- PCIe vendor id
- PCIe device id

修改用脚本为 rkbin/tools/pcie_idb_config.sh文，命令介绍：

```
./tools/pcie_idb_config.sh <soc> <gen> <lanes> <uart_baundary> <vid_l> <vid_h> <did_l> <did_h>
<input>
like following:
./tools/pcie_idb_config.sh RK3588 3 4 1500000 87 1d 6a 35 bin/rk35/rk3588_pcie_v*.bin #vid=0x1d87
did=0x356a
./tools/pcie_idb_config.sh RK3588 3 2 1500000 87 1d 6a 35 bin/rk35/rk3588_pcie_v*.bin
./tools/pcie_idb_config.sh RK3588 3 1 1500000 87 1d 6a 35 bin/rk35/rk3588_pcie_v*.bin
./tools/pcie_idb_config.sh RK3588 3 4 115200  87 1d 6a 35 bin/rk35/rk3568_pcie_v*.bin
./tools/pcie_idb_config.sh RK3568 3 2 1500000 87 1d 6a 35 bin/rk35/rk3588_pcie_v*.bin
```

若以上开放配置不满足需求，客户可以修改源码自定义功能。

3.3.2.2 源码编译pcie.bin

源码依赖 rkbin 内的工具，要求源码解压缩在 rkbin/ 同级目录进行开发，或者直接修改 tools/mk_pcie_idb.sh 里 boot_merger 和 programmer_image_tool 工具路径

编译工具链

从src/platform/common/GCC/Cortex-A.mk中确认工具链：

```
################################################################################
# Cross compiler
################################################################################
ifneq ($(wildcard ${ROOT_PATH}/../prebuilts/gcc/linux-x86/aarch64/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu/bin),)
CROSS_COMPILE  ?= ${ROOT_PATH}/../prebuilts/gcc/linux-x86/aarch64/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-bin/aarch64-none-linux-gnu-
else ifeq ($(wildcard /opt/prebuilts/gcc/linux-x86/aarch64/arm-gnu-toolchain-11.3.rel1-x86_64-aarch64-none-linux-gnu/bin),)
CROSS_COMPILE  ?= /opt/prebuilts/gcc/linux-x86/aarch64/arm-gnu-toolchain-11.3.rel1-x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu-
else
CROSS_COMPILE  ?= aarch64-none-linux-gnu-
endif
```

编译命令

RK3588：

```
./tools/mk_pcie_idb.sh RK3588
```

RK3568：

```
./tools/mk_pcie_idb.sh RK3588
```

输出文件：

pcie_idb.img

3.3.2.2.1 BSP配置

以RK3588为例，源码src/platform/RK3588/bsp/platform.h

```
#define BOOT_USING_UART_BAUD 1500000            // 修改波特率，支持1500000或115200
```

3.3.2.2.2 BAR默认配置

源码src/driver/pcie/pcie_ep.c

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

3.3.2.2.3 BAR大小配置

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

配置BAR SIZE，(bar_id * 0x8)指定目标Bar ID，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)，以下为Bar0~5配置：

```
    writel(0x2c0, resbar_base + 0x8 + 0 * 0x8);
    writel(0x2c0, resbar_base + 0x8 + 1 * 0x8);
    writel(0x6c0, resbar_base + 0x8 + 2 * 0x8);
    writel(0x6c0, resbar_base + 0x8 + 3 * 0x8);
    writel(0x0c0, resbar_base + 0x8 + 4 * 0x8); /* wired register */
    writel(0x6c0, resbar_base + 0x8 + 5 * 0x8);
```

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：

```
    //writel(0x0, dbi_base + 0x100000 + 0x10 + 0 * 4);
    writel(0x0, dbi_base + 0x100000 + 0x10 + 1 * 4); /* BAR1 disabled */
    //writel(0x0, dbi_base + 0x100000 + 0x10 + 2 * 4);
    //writel(0x0, dbi_base + 0x100000 + 0x10 + 3 * 4);
    //writel(0x0, dbi_base + 0x100000 + 0x10 + 4 * 4);
    writel(0x0, dbi_base + 0x100000 + 0x10 + 5 * 4);  /* BAR5 disabled */
```

须知：

- 映射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

3.3.2.2.4 BAR映射配置

以 RK3588 为例，源码 src/platform/RK3588/bsp/platform.h

```
#define BOOT_USING_PCIE_EP_BAR0_CPU_ADDRESS 0x3C000000   // 调整Bar0映射地址
```

代码中实现：

```
HAL_PCIE_InboundConfig(pcie, 0, 0, BOOT_USING_PCIE_EP_BAR0_CPU_ADDRESS);
```

须知：

- PCIe.bin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

3.3.2.2.5 PHY模式配置

RK3588 默认 phy0/1 为合并使用以支持 Gen3x4，可支持拆分为 Gen3x2、Gen3x2、Gen3x1，配置如下：

RK3588 PCIe PHY 3x2实例：

```
diff --git a/src/hal/lib/bsp/RK3588/hal_bsp.c b/src/hal/lib/bsp/RK3588/hal_bsp.c
index 4a6c2e6..f03995b 100644
--- a/src/hal/lib/bsp/RK3588/hal_bsp.c
+++ b/src/hal/lib/bsp/RK3588/hal_bsp.c
@@ -123,7 +123,7 @@ const struct HAL_UART_DEV g_uart9Dev =
 #ifdef HAL_PCIE_MODULE_ENABLED
 const struct HAL_PHY_SNPS_PCIE3_DEV g_phy_pcie3Dev =
 {
-    .phyMode = PHY_MODE_PCIE_AGGREGATION,
+    .phyMode = PHY_MODE_PCIE_NANBNB,
 };

 struct HAL_PCIE_DEV g_pcieDev =
```

RK3588 PCIe PHY 3x1实例：

```
diff --git a/src/hal/lib/bsp/RK3588/hal_bsp.c b/src/hal/lib/bsp/RK3588/hal_bsp.c
index 4a6c2e6..f03995b 100644
--- a/src/hal/lib/bsp/RK3588/hal_bsp.c
+++ b/src/hal/lib/bsp/RK3588/hal_bsp.c
@@ -123,7 +123,7 @@ const struct HAL_UART_DEV g_uart9Dev =
 #ifdef HAL_PCIE_MODULE_ENABLED
 const struct HAL_PHY_SNPS_PCIE3_DEV g_phy_pcie3Dev =
 {
-    .phyMode = PHY_MODE_PCIE_AGGREGATION,
+    .phyMode = PHY_MODE_PCIE_NABIBI,
 };

 struct HAL_PCIE_DEV g_pcieDev =
```

3.3.2.3 PCIe Bin打包烧写

3.3.2.3.1 工具烧写

1. pcie.bin转pcie_idb.img

```
#生成 loader
./tools/boot_merger RKBOOT/RK3588MINIALL_PCIE_EP.ini
 rkbin git:(master)  ls rk3588_pcie_loader_v1.00.bin
rk3588_pcie_loader_v1.00.bin

#spinand 制作 idb，要求使用 spinand type 做对齐
./tools/programmer_image_tool -i rk3588_pcie_loader_v1.00.bin -b 128 -p 2 -t spinand -o ./
#emmc
./tools/programmer_image_tool -i rk3588_pcie_loader_v1.00.bin -t emmc -o ./

#制作双备份的 pcie_idb.img
cat idblock.img > pcie_idb.img
cat idblock.img >> pcie_idb.img
rm idblock.img
```

2. 分立烧写
   - 工具添加pcie_idb分区，emmc flash起始地址：0x40；spi flash起始地址：0x100;
   - 修改烧写工具配置文件，烧写工具目录下config.ini使能配置：`CLOSE_CHECK_IDB=TRUE`

- 板子进入Maskrom模式，工具点击烧写

3. 完整包烧写
   - 修改分区表添加pcie_idb分区，如下是emmc分区，spi nand只是地址不同

```
CMDLINE:
mtdparts=rk29xxnand:0x00000400@0x00000040(pcie_idb),0x00000800@0x00002000(security),0x00002000@0x00004000(uboot),0x00010000@0x00006000(boot),0x00064000@0x00016000(rootfs),-@0x0007a000(userdata:grow)
```

   - 修改package-file，tools/linux/Linux_Pack_Firmware/rockdev/rk3588-package-file-nvr-emmc / spi-nand

```
package-file    package-file
bootloader    Image/MiniLoaderAll.bin
pcie_idb      Image/pcie_idb.img   #添加pcie_idb文件
parameter     Image/parameter.txt
uboot         Image/uboot.img
boot          Image/boot.img
rootfs        Image/rootfs.img
userdata      Image/userdata.img
```

   - 执行build_emmc.sh 编译update.img烧写

3.3.2.3.2 烧录器烧写

参考EP卡固件烧录

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或其他：

RK3588 PCIe PHY 3x2实例：

```
diff --git a/arch/arm/mach-rockchip/spl_pcie_ep_boot.c b/arch/arm/mach-rockchip/spl_pcie_ep_boot.c
index 8b054d91558..54f9d22cde2 100644
--- a/arch/arm/mach-rockchip/spl_pcie_ep_boot.c
+++ b/arch/arm/mach-rockchip/spl_pcie_ep_boot.c
@@ -384,7 +384,7 @@ static void pcie_board_init(void)
 #define PHY_MODE_PCIE_NABINB  2     /* P1:PCIe3x1*2 + P0:PCIe3x2 */
 #define PHY_MODE_PCIE_NABIBI  3     /* P1:PCIe3x1*2 + P0:PCIe3x1*2 */

-#define PHY_MODE_PCIE           PHY_MODE_PCIE_AGGREGATION
+#define PHY_MODE_PCIE           PHY_MODE_PCIE_NANBNB

 #define CRU_BASE_ADDR           0xfd7c0000
 #define CRU_SOFTRST_CON32       (CRU_BASE_ADDR + 0x0a80)
```

RK3588 PCIe PHY 3x1实例：

```
diff --git a/arch/arm/mach-rockchip/spl_pcie_ep_boot.c b/arch/arm/mach-rockchip/spl_pcie_ep_boot.c
index 8b054d91558..54f9d22cde2 100644
--- a/arch/arm/mach-rockchip/spl_pcie_ep_boot.c
+++ b/arch/arm/mach-rockchip/spl_pcie_ep_boot.c
@@ -384,7 +384,7 @@ static void pcie_board_init(void)
 #define PHY_MODE_PCIE_NABINB  2     /* P1:PCIe3x1*2 + P0:PCIe3x2 */
 #define PHY_MODE_PCIE_NABIBI  3     /* P1:PCIe3x1*2 + P0:PCIe3x1*2 */

-#define PHY_MODE_PCIE           PHY_MODE_PCIE_AGGREGATION
+#define PHY_MODE_PCIE           PHY_MODE_PCIE_NABIBI

 #define CRU_BASE_ADDR           0xfd7c0000
 #define CRU_SOFTRST_CON32       (CRU_BASE_ADDR + 0x0a80)
```

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";
};
```

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 BAR大小配置

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

```
/* Resize BAR0 4M 32bits, BAR2 64M 64bits-pref, BAR4 1MB 32bits */
bar = BAR_0;
dw_pcie_writel_dbi(pci, resbar_base + 0x4 + bar * 0x8, 0xfffff0);
dw_pcie_writel_dbi(pci, resbar_base + 0x8 + bar * 0x8, 0x2c0);
rockchip_pcie_ep_set_bar_flag(rockchip, bar, PCI_BASE_ADDRESS_MEM_TYPE_32);

bar = BAR_2;
dw_pcie_writel_dbi(pci, resbar_base + 0x4 + bar * 0x8, 0xfffff0);
dw_pcie_writel_dbi(pci, resbar_base + 0x8 + bar * 0x8, 0x6c0);
rockchip_pcie_ep_set_bar_flag(rockchip, bar,PCI_BASE_ADDRESS_MEM_PREFETCH |
PCI_BASE_ADDRESS_MEM_TYPE_64);

bar = BAR_4;
dw_pcie_writel_dbi(pci, resbar_base + 0x4 + bar * 0x8, 0xfffff0);
dw_pcie_writel_dbi(pci, resbar_base + 0x8 + bar * 0x8, 0xc0);
```

```
rockchip_pcie_ep_set_bar_flag(rockchip, bar, PCI_BASE_ADDRESS_MEM_TYPE_32);
```

关掉不使用的BAR：

```
/* Disable BAR1 BAR5*/
bar = BAR_1;
dw_pcie_writel_dbi(pci, PCIE_TYPE0_HDR_DBI2_OFFSET + 0x10 + bar * 4, 0);
bar = BAR_5;
dw_pcie_writel_dbi(pci, PCIE_TYPE0_HDR_DBI2_OFFSET + 0x10 + bar * 4, 0);
```

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

3.3.4.4 BAR映射配置

每个BAR对应的空间，通过配置映射到内部的内存地址区域，可以修改dts来修改要映射的目标地址。

```
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>;
    };
};
```

说明：

- 支持新增bar1/bar5映射内存地址，同时自动使能bar1/5，如果使用PCIe Bin/SPL初始化PCIe，需参考“BAR大小配置”注释掉bar1/5 disabeld代码

3.3.4.5 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卡固件编译

参考patch目录说明打上对应补丁后根据场景选择对应的编译说明：

1. Storage Boot方案固件编译

- 单存储

  单存储固件和普通固件一样不需要其它特殊配置，直接编译出各个分区img以及update.img。

- 双存储

  双存储固件以Nor+ EMMC举例：

  - Nor固件配置

    Nor部分包含MiniLoader 、parameter.txt 、 uboot.img这三个分区。

    Miniloader：双存储固件不需要单独修改。

    parameter.txt：只需要保留uboot分区以及前面的分区信息，uboot后的分区可以删除（EMMC有自己的分区表）。

    uboot.img：需要配置启动模式到EMMC。

    - uboot依赖提交

      ```
      commit 5029b47418a9dc93760a3b707903d6305331c62c
      Author: Joseph Chen <chenjh@rock-chips.com>
      Date:   Mon Apr 18 10:10:54 2022 +0000

          rockchip: Introduce CONFIG_ROCKCHIP_BOOTDEV

          Usually we get boot device from preloader or scan list on the
          board with single storage. While for the multiple storage, we
          introduce this option to determine which we really want to be
          the boot device, which contains kernel, rootfs and etc.

          When this option is NULL string, we fall through to get boot
          device from preloader or scan list.

          Example: CONFIG_ROCKCHIP_BOOTDEV="nvme 0".

          Signed-off-by: Joseph Chen <chenjh@rock-chips.com>
          Change-Id: I734dc0ab2bb7104a1584cbddc15620c890970223
      ```

    - 启动配置

      ```
      CONFIG_ROCKCHIP_BOOTDEV="mmc 0"
      CONFIG_ROCKCHIP_EMMC_IOMUX=y
      ```

  - EMMC固件配置

    EMMC部分包含parameter.txt 、boot.img、rootfs.img、userdata.img这四个分区。

    parameter.txt：只需要保留从boot开始的分区信息，前面的loader以及uboot分区信息在Nor的分区表中。**注意：EMMC前面4M默认被idb以及分区表占用，用户分区需要从4M开始划分。**

    boot.img、rootfs.img、userdata.img：这三个分区不需要特殊配置，按照单存储方式编译。

SDK默认不支持把双存储固件打包成update.img，需要借助tools目录下的dual_storage工具打包，参考tools/dual_storage/README.txt说明把各个分区img固件拷贝到对应目录下，执行build.sh即可把分立img打包成update.img。

2. EP Boot方案固件编译

   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）。启动固件和单存储固件编译一致，不需要特殊配置。

3.5 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，参考各芯片发布文档刷机说明章节即可。

- 双存储

  详细参考 《Rockchip_Developer_Guide_Dual_Storage_CN.pdf》文档”RK PCIe EP 双存储方案固件打包“章节。

2. EP Boot方案固件烧写

   启动固件和正常固件烧写流程一致，参考各芯片发布文档刷机说明章节即可。

3.6 EP卡OTA

分区升级原理都是通过分区表信息让内核创建对应的设备节点，上层通过写入/dev/下的设备节点，来实现更新分区的功能。

OTA升级可以使用RK方案通过recovery模式进行分区更新，或者使用自定义方案，只需要写入/dev/下的设备节点。

1. Storage Boot方案OTA

- 单存储

  单存储方案OTA和正常固件方式一致，参考docs\Linux\Recovery\Rockchip_Developer_Guide_Linux_Recovery_CN.pdf章节更新即可。

- 双存储

  详细参考 《Rockchip_Developer_Guide_Dual_Storage_CN.pdf》文档”RK PCIe EP 双存储方案OTA 升级 “章节。

2. EP Boot方案固件OTA

   EP Boot方案只有loader分区，需要在分区表中添加loader分区信息。

   Nor：可以参考EP卡OTA->双存储章节->Nor配置 Uboot分区配置。

   EMMC：可以参考EP卡OTA->双存储章节->EMMC配置，直接在分区表添加loader信息。

   RK OTA方案不支持单独升级loader，需要自定义实现。

   不推荐对loader分区进行升级，风险较高。

注意：MiniLoaderAll.bin和parameter.txt 不能直接写到/dev/下的设备节点，需要转成二进制文件才能写入。

3.7 EP卡设备驱动

PCIe EP（Endpoint）设备驱动是运行在Root Complex（RC）侧的驱动程序，用于控制总线上的数据传输，并为应用层提供所需的内核接口。

3.7.1 Kernel阶段配置

RK RC Kconfig配置

```
CONFIG_PCIE_FUNC_RKEP=y
```

通用RC-FUNC驱动

参考`chips_connect/patch-kernelx.xx/通用RC-FUNC驱动/`编译。

4. EP卡业务基础

4.1 原理介绍

EP卡业务内容主要是将源数据和处理器处理后的目标数据通过PCIe链路完成上下游间的传输，主要资源包括：

RC端

- 通过开源标准库libpci操作PCIE EP设备。
- 通过pcie-rkep驱动完成：
  - BAR映射，推荐作为控制流传输方式
  - DMA传输，推荐作为数据块传输方式
  - 处理MSI中断，用户层主动poll等待以提高响应速率，但涉及多层握手，实时性一般
  - 触发ELBI中断
  - 申请DMA内存

EP端

- 通过pcie_ep驱动完成：
  - 默认 class id 0x1200（Processing accelerators）
  - BAR inbound物理地址空间映射
  - DMA传输
  - 发送MSI中断
  - 处理ELBI中断，用户层主动poll等待以提高响应速率，但涉及多层握手，实时性一般
- 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

SPL 及 Kernel 阶段：

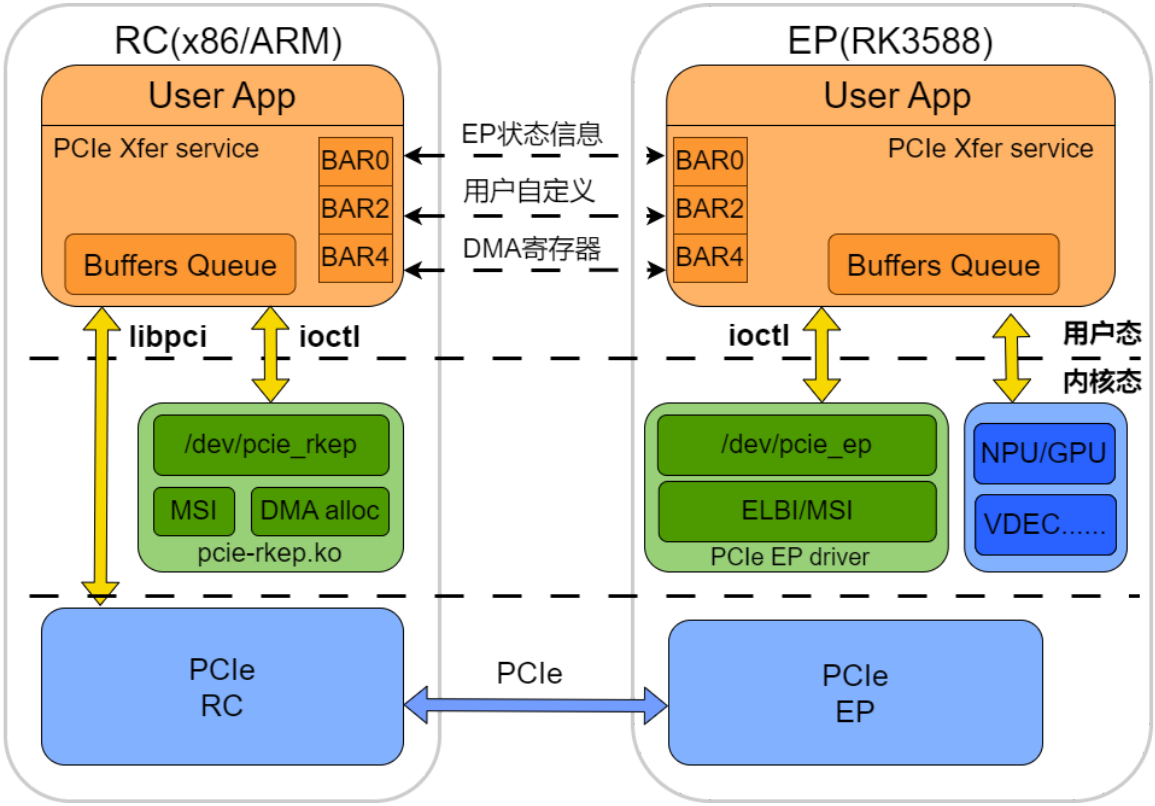
BAR空间	起始地址	结束地址	功能
BAR0	0	4KB	EP保留空间，驱动版本信息、中断状态等等。
	12KB	16KB	DEBUG功能相关配置信息。
	16KB	128KB	EP设备状态信息。
	128KB	1M	用户自定义。
	1M	4M	框架命令交互以及buff管理。
BAR1（Defaut Disabled）	0	1MB	用户自定义，如何使能参考“BAR大小配置”章节配置。
BAR2（BAR2/3 Bar Repair）	0	64MB	Using RC DMA使用。
BAR4	0	4MB	wired寄存器区，用于操作DMA。
BAR5（Defaut Disabled）	-	-	用户自定义，如何使能参考“BAR大小配置”章节配置。

说明：

- EP端BAR用户自定义空间可用于二次开发扩展，其余BAR空间为该业务代码需求空间，例如：
  - BAR1 默认关闭，全部空间为用户自定义空间，开启后用作控制流
  - BAR5 默认关闭，全部空间为用户自定义空间，开启后用作控制流
- EP端BAR inbound CPU address默认由“BAR映射配置”固定配置，支持RC端动态调整

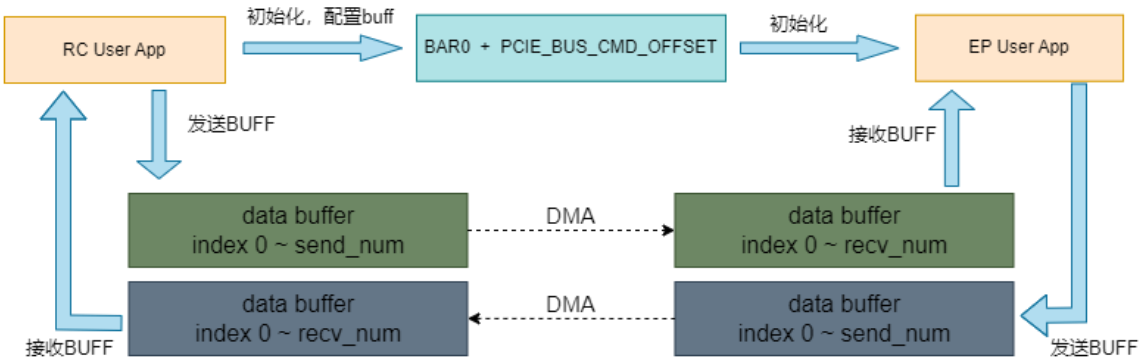
4.2 PCIe传输系统架构

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、DEVICE_ID以及总线地址初始化对应的PCIE设备。

   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_device_init` 接口初始化设备。

2. 调用 `rk_pcie_task_create` 创建任务对内存、BUFF状态进行初始化，可通过配置 `struct pcie_task_attr_st` 结构体来设置BUFF的数量和大小，配置信息会通过BAR0空间传递给EP设备，并且等待EP设备初始化完成。EP通过 `rk_pcie_get_task_msg` 获取创建任务消息然后调用task初始化，两边会进行初始化配置交互。

RC端应用根据编译配置信息，选择Hugepage/RKEP/RKDRM方式申请内存。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（需要打开对应配置）。

EP端收到配置信息后，初始化BUFF状态，并调用接口申请物理连续的内存，支持Hugepage/RKDRM方式申请内存。

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。

```
#define VENDOR_ID            0x1d87
#define DEVICE_ID            0x356a

struct pcie_dev_bus dev_bus;
int ret;
//获取系统匹配的PCI设备总线地址
ret = rk_pcie_get_pci_bus_addresses(VENDOR_ID, DEVICE_ID, &rc_ctx.dev_bus);
if (ret != 0) {
  printf("get pci bus address fail\n");
  return -1;
}

printf("ep dev max num : %d\n", rc_ctx.dev_bus.dev_max_num);
if (rc_ctx.dev_bus.dev_max_num == 0) {
  printf("no ep device, exit ... \n");
} else {
  rc_ctx.dev_attr.using_rc_dma = 0;
  rc_ctx.dev_attr.rc_dma_chn = 0;
  rc_ctx.dev_attr.enable_speed = 1;
  rc_ctx.dev_attr.vendor_id = VENDOR_ID;
  rc_ctx.dev_attr.device_id = DEVICE_ID;
  memcpy(rc_ctx.dev_attr.pci_bus_address, pci_bus_address, strlen(pci_bus_address));

  //选择第一个匹配设备初始化
  rc_ctx.rk_handle = rk_pcie_device_init(&rc_ctx.dev_attr);
  //获取EP设备模式
  rk_pcie_get_ep_mode(rc_ctx.rk_handle, &rc_ctx.devmode);
  printf("ep device init, dev mode: %d, submode: %d\n", rc_ctx.devmode.mode,
rc_ctx.devmode.submode);
}
```

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

```
struct pcie_buff_node *node = rk_pcie_get_buff(rk_handle, E_SEND);
if(node) {
  node->size = rk_pcie_get_buff_max_size(rk_handle, E_SEND);
  //操作buff 写入数据
  rk_pcie_send_buff(ctx->rk_handle, 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` 。

```
struct pcie_buff_node *node = rk_pcie_get_buff(rk_handle, E_RECV);
if(node) {
  //操作buff 读取数据
  rk_pcie_release_buff(rk_handle, 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均支持多种内存申请方式：

- 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)

3. 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;
};
```

- 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)

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

Android修改内存申请可以直接通过环境变量配置或者添加到 `build/make/envsetup.sh` 。

直接添加到环境变量：

```
export RK_RC_MEM_CONFIG=libdrm   //支持libdrm/rkep/hugepage，默认不配置就是hugepage
export RK_EP_MEM_CONFIG=libdrm   //支持libdrm/hugepage，默认不配置就是hugepage
```

添加到脚本build/make/envsetup.sh，通过执行 `source build/envsetup.sh` 添加到环境变量

```
+++ b/envsetup.sh
@@ -345,6 +345,8 @@ function setpaths()
   unset PRODUCT_KERNEL_VERSION
   export PRODUCT_KERNEL_VERSION=$(get_build_var PRODUCT_KERNEL_VERSION)

+  export RK_RC_MEM_CONFIG=rkdrm
+  export RK_EP_MEM_CONFIG=rkdrm
   # needed for building linux on MacOS
   # TODO: fix the path
   #export HOST_EXTRACFLAGS="-I "$T/system/kernel_headers/host_include
```

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`大小，最小不能小于`send_buff_size`。

使能模式

RC以及EP设备驱动不需要修改，只需要RC上层应用初始化阶段参数配置

```
dev_attr.using_rc_dma = 1; //使用RC DMA
dev_attr.rc_dma_chn = 1; //通道1，支持0/1
```

使能成功后可以在RC应用看到如下LOG:

```
rk pcie version: v2.0.0 - 202300927
using rc dma: 1, chn: 1   //使能RC DMA，通道1
open rkrmd: /dev/pcie-rkep-0000:01:00.0 Success.
```

EP应用创建Task会有如下LOG：

```
task create id: 1
using rc dma: 1
```

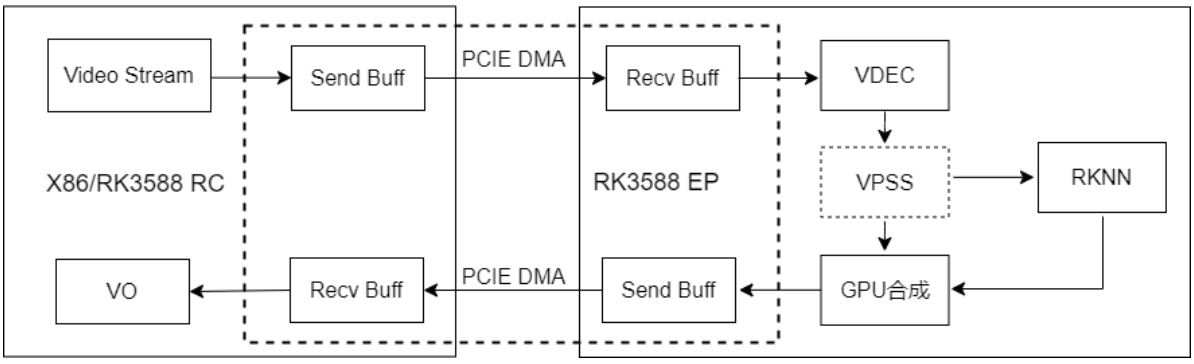
推荐配置

- 单EP场景：推荐使能using_rc_dma，并且配置为通道0。
- 多EP场景：推荐关闭using_rc_dma，使用EP各自的DMA搬运数据，可以让每个EP的传输速率保持大致相同。

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. EP验收标准

首次使用PCIE EP开发包建议按照如下流程确认软件、硬件、基础业务流程正常后，再开始自定义功能开发。

7.1 补丁更新 | 一键生效“更新包”源码

测试目标

标准卡驱动版本升级，支持一键添加“更新包”目录下的补丁。

测试方法

RK3588：

```
./ep_patch.sh release patch-kernel5.10/更新包 ./path-to-dst/
```

RK3568：

```
./ep_patch.sh release patch-kernel4.19/更新包 ./path-to-dst/
```

其他说明：

- 生成补丁包：

```
./ep_patch.sh release ./.. patch-kernel5.10/更新包
cd ./../kernel/
git checkout develop-4.19
cd -
./ep_patch.sh kernel ./.. patch-kernel4.19/更新包
```

7.2 功能 | 关键日志信息

7.2.1 RC端lspci关键信息

确认以下信息：

- bar使能且正常分配
- MSI使能且正常分配
- LnkCap，Gen3x4支持、ASPM not supported
- LnkSta，Gen3x4状态
- Kernel driver in use: pcie-rkep

log示例：

```
01:00.0 Class 1200: Device 1d87:356a (rev 01)
	Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx+
	Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
```

```
	Latency: 0
	Interrupt: pin A routed to IRQ 157
	Region 0: Memory at f0400000 (32-bit, non-prefetchable) [size=4M]
	Region 2: Memory at 900000000 (64-bit, prefetchable) [size=64M]
	Region 4: Memory at f0200000 (32-bit, non-prefetchable) [size=1M]
	Capabilities: [40] Power Management version 3
		Flags: PMEClk- DSI- D1+ D2+ AuxCurrent=375mA PME(D0+,D1+,D2-,D3hot+,D3cold-)
		Status: D0 NoSoftRst+ PME-Enable- DSel=0 DScale=0 PME-
	Capabilities: [50] MSI: Enable+ Count=4/32 Maskable+ 64bit+
		Address: 00000000fe670040  Data: 0000
		Masking: fffffff0  Pending: 00000000
	Capabilities: [70] Express (v2) Endpoint, MSI 08
		DevCap: MaxPayload 256 bytes, PhantFunc 0, Latency L0s unlimited, L1 unlimited
			ExtTag+ AttnBtn- AttnInd- PwrInd- RBE+ FLReset- SlotPowerLimit 0.000W
		DevCtl: Report errors: Correctable- Non-Fatal- Fatal- Unsupported-
			RlxdOrd+ ExtTag+ PhantFunc- AuxPwr- NoSnoop-
			MaxPayload 256 bytes, MaxReadReq 256 bytes
		DevSta: CorrErr- UncorrErr- FatalErr- UnsuppReq- AuxPwr- TransPend-
		LnkCap: Port #0, Speed 8GT/s, Width x4, ASPM not supported, Exit Latency L0s <4us, L1 <16us
			ClockPM- Surprise- LLActRep- BwNot- ASPMOptComp+
		LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- CommClk-
			ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
		LnkSta: Speed 8GT/s, Width x4, TrErr- Train- SlotClk+ DLActive- BWMgmt- ABWMgmt-
		DevCap2: Completion Timeout: Not Supported, TimeoutDis+, LTR+, OBFF Via message/WAKE#
		DevCtl2: Completion Timeout: 50us to 50ms, TimeoutDis-, LTR+, OBFF Disabled
		LnkCtl2: Target Link Speed: 8GT/s, EnterCompliance- SpeedDis-
			 Transmit Margin: Normal Operating Range, EnterModifiedCompliance- ComplianceSOS-
			 Compliance De-emphasis: -6dB
		LnkSta2: Current De-emphasis Level: -6dB, EqualizationComplete+, EqualizationPhase1+
			 EqualizationPhase2+, EqualizationPhase3+, LinkEqualizationRequest-
	Capabilities: [b0] MSI-X: Enable- Count=128 Masked-
		Vector table: BAR=4 offset=00020000
		PBA: BAR=4 offset=00028000
	Capabilities: [100 v2] Advanced Error Reporting
		UESta:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq-
ACSViol-
		UEMsk:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq-
ACSViol-
		UESvrt: DLP+ SDES+ TLP- FCP+ CmpltTO- CmpltAbrt- UnxCmplt- RxOF+ MalfTLP+ ECRC-
UnsupReq- ACSViol-
		CESta:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr-
		CEMsk:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
		AERCap: First Error Pointer: 00, GenCap+ CGenEn- ChkCap+ ChkEn-
	Capabilities: [148 v1] #19
	Capabilities: [168 v1] Address Translation Service (ATS)
		ATSCap: Invalidate Queue Depth: 00
		ATSCtl: Enable-, Smallest Translation Unit: 00
	Capabilities: [178 v1] Page Request Interface (PRI)
		PRICtl: Enable- Reset-
		PRISta: RF- UPRGI- Stopped+
		Page Request Capacity: 00000001, Page Request Allocation: 00000000
	Capabilities: [188 v1] Latency Tolerance Reporting
		Max snoop latency: 0ns
		Max no snoop latency: 0ns
	Capabilities: [190 v1] L1 PM Substates
		L1SubCap: PCI-PM_L1.2+ PCI-PM_L1.1+ ASPM_L1.2+ ASPM_L1.1+ L1_PM_Substates+
			 PortCommonModeRestoreTime=10us PortTPowerOnTime=10us
	Capabilities: [1a0 v1] #16
```

```
    Capabilities: [1d0 v1] Vendor Specific Information: ID=0002 Rev=4 Len=100 <?>
    Capabilities: [2d0 v1] Vendor Specific Information: ID=0006 Rev=0 Len=018 <?>
    Capabilities: [2e8 v1] #15
    Kernel driver in use: pcie-rkep
```

7.2.2 EP端spl.bin启动 log

确认以下信息：

- kenrel阶段打印“already linkup”

log示例：

```
... #spl关键打印

U-Boot SPL board init
RKEP: Init PCIe EP
RKEP: 185814 - PHY Mode 0x4
RKEP: 187065 - RKEP: GRF:904=7, a04=7...
RKEP: 187657 - RKEP: GRF:904=7, a04=f...
RKEP: 188009 - RKEP: GRF:904=f, a04=f...
RKEP: 188345 - RefClock in common clock_mode
RKEP: 198137 - BAR0: 0x3c000000
RKEP: 198409 - BAR2: 0x010000000
RKEP: 198794 - init PCIe fast Link up
RKEP: 217146 - Link up 230011
RKEP: 251464 - Done

U-Boot SPL 2017.09-ga8d570efd78-231012-dirty #ldq (Oct 20 2023 - 18:37:12)

... #内核关键打印

[00-06-34.831][    3.148198] rk-pcie-ep fe150000.pcie: bar0: assigned [0x3c000000-3c3fffff]
[00-06-34.834][    3.149015] rk-pcie-ep fe150000.pcie: bar2: assigned [0x40000000-43ffffff]
[00-06-34.834][    3.149027] rk-pcie-ep fe150000.pcie: no vpcie3v3 regulator found
[00-06-34.837][    3.149046] rk-pcie-ep fe150000.pcie: already linkup
[00-06-34.837][    3.149320] rk-pcie-ep fe150000.pcie: register misc device pcie_ep
```

7.2.3 EP端pcie.bin启动 log

确认以下信息：

- 记录pcie.bin运行时间

版本	耗时
V1.00	360ms

- spl阶段打印“already link up”
- kenrel阶段打印“already linkup”

log示例：

```
... #pcie.bin关键打印
[10-20-18:42:00:071]PCIe_EP Release Time: Sep 29 2023 V1.00
[10-20-18:42:00:073]Gen3x4
[10-20-18:42:00:076]phyMode4
[10-20-18:42:00:080]HW
[10-20-18:42:00:083]controlelr initialed
[10-20-18:42:00:085]GRF:904=0007, a04=0007...
[10-20-18:42:00:089]GRF:904=004F, a04=004F...
[10-20-18:42:00:091]GRF:904=004F, a04=000F...
[10-20-18:42:00:092]GRF:904=000F, a04=000F...
[10-20-18:42:00:092]phy locked
[10-20-18:42:00:094]phy initialed
[10-20-18:42:00:095]vid=0x1D87 did=0x356A
[10-20-18:42:00:097]Linking, ltssm:
[10-20-18:42:00:099]00000001
[10-20-18:42:00:100]00000002
[10-20-18:42:00:102]0001000F
[10-20-18:42:00:103]00210022
[10-20-18:42:00:104]00210023
[10-20-18:42:00:105]00230011
[10-20-18:42:00:106]Link up
[10-20-18:42:00:425]Link stable, ltssm
[10-20-18:42:00:428]00230011
[10-20-18:42:00:431]out

... #spl关键打印

[00-06-31.468]U-Boot SPL board init
[00-06-31.468]RKEP: Init PCIe EP
[00-06-31.468]RKEP: already link up

... #内核关键打印

[00-06-34.831][    3.148198] rk-pcie-ep fe150000.pcie: bar0: assigned [0x3c000000-3c3fffff]
[00-06-34.834][    3.149015] rk-pcie-ep fe150000.pcie: bar2: assigned [0x40000000-43ffffff]
[00-06-34.834][    3.149027] rk-pcie-ep fe150000.pcie: no vpcie3v3 regulator found
[00-06-34.837][    3.149046] rk-pcie-ep fe150000.pcie: already linkup
[00-06-34.837][    3.149320] rk-pcie-ep fe150000.pcie: register misc device pcie_ep
```

7.3 功能 | 固件烧录

请阅读”单EP卡对接通用RC“章节，确认是否为需要快速启动的产品形态，并确认所使用存储方案并做对应PC升级烧录验证，主要包括：

- 单存储
- 双存储
- EP Boot

详细参考“EP卡固件编译”章节。

7.4 性能 | 用户态测试Demo - pcie_speed_test

7.4.1 测试目标

验证业务基础，主要包括：

- RC端wired DMA to EP传输、DMA传输速率及双向DMA传输
- RC端local DMA to RC传输、DMA传输速率及双向DMA传输
- bar映射、访问正常
- ELBI 工作正常，支持RC 触发中断事件通知EP
- MSI 工作正常，支持EP 触发中断事件通知RC

根据需求验证：

- 如果选用支持DMA的RK RC，支持RC端Local DMA to EP传输，DMA传输速率
- firmware download

7.4.2 测试方法

详细参考examples/pcie_speed_test/README.md文档。

7.4.3 RK3588 Gen3x4测试结果

测试设备

RC：RK3588-EVB1-LP4X-V10 PCIe 3.0 4 lanes大核2.4G小核1.8G

EP：RK3588-EVB4-LP4X-V10 PCIe 3.0 4 lanes大核2.4G小核1.8G

标准 EP 版本

Rockchip_PCIE_EP_Stardard_Card_20231215.tar.gz

测试参数：

- 使用 EP DMA 引擎
- task_num = 1

测试结果

DMA大小	64B	256B	4K	64KB	1MB	4MB
写	0.16MB/S	0.65MB/S	10.4MB/S	165.8MB/S	1755.7MB/S	2707.3MB/s
读	0.17MB/S	0.73MB/S	11.0MB/S	171.1MB/S	1597.6MB/S	1671.6MB/s

说明：

- 如果是RC发起测试，且使用默认的EP DMA引擎，写测试实际为EP DMA读，读测试实际为EP DMA写
- 测试粒度越小，调度参与越多，性能越差，但可以通过开多线程提高整体性能

7.4.4 RK3588 Gen2x1测试结果

测试设备

RC：RK3588-EVB1-LP4X-V10 PCIe 3.0 4 lanes大核2.4G小核1.8G

EP：RK3588-EVB4-LP4X-V10 PCIe 3.0 4 lanes（降为Gen2 1 lanes） 大核2.4G小核1.8G

标准EP版本

Rockchip_PCIE_EP_Stardard_Card_20231215.tar.gz

测试参数：

- 使用EP DMA引擎
- task_num = 1

测试结果

DMA大小	64B	256B	4K	64KB	1MB	4MB
写	0.16MB/S	0.64MB/S	10.2MB/S	147.6MB/S	394.0MB/S	396.2MB/s
读	0.17MB/S	0.70MB/S	11.0MB/S	149.2MB/S	384.0MB/S	384.1MB/s

说明：

- 如果是RC发起测试，且使用默认的EP DMA引擎，写测试实际为EP DMA读，读测试实际为EP DMA写
- 测试粒度越小，调度参与越多，性能越差，但可以通过开多线程提高整体性能

7.5 兼容性 | 枚举

测试目标

验证EP板卡在不同RC上探测、识别、枚举的兼容性和健壮性。

测试方法

RK EP主要对接的RC：

EP设备	RC设备	冷启	休眠唤醒	热启	确认方法
RK3588 EVB4 V11	RK3588 EVB1 V10	20/20	20/20	20/20	Linux：lspci | grep 356a
RK3588 EVB4 V11	PC ubuntu	20/20	20/20	20/20	Linux：lspci | grep 356a
RK3588 EVB4 V11	PC windows	20/20	20/20	20/20	设备管理器确认是否有对应设备
RK3568 iotest 设备	RK3588 EVB1 V10	20/20	20/20	20/20	Linux：lspci | grep 356a
RK3568 iotest 设备	PC ubuntu	20/20	20/20	20/20	Linux：lspci | grep 356a

说明：

- 实际产品尽可能多的覆盖目标RC设备
- 冷启：PC断电、上电、按下POWER按键
- 休眠唤醒：
  - RK RC：POWER按键进出休眠唤醒
  - ubuntu：桌面版本通过可以通过界面按键进入，其他场景自行确认
  - Windows：CTRL+L组合按键进入，其他场景自行确认
- 热启：
  - RK3588：进入root权限，shell界面输入reboot指令
  - ubuntu：进入root权限，shell界面输入reboot指令
  - Windows：桌面下，ALT+F4组合键弹出复位选项，选择复位，部分PC支持直接按POWER按键复位

7.6 兼容性 | EP复位RC检测并重新枚举

测试目标

测试”EP单独复位“场景下的业务健壮性，主要测试以下内容：

- PCIe remove/rescan 功能
- 通讯是否正常恢复
- 业务是否茁壮

测试方法

以下测试需要上述几项功能测试都正常情况下进行，并且编译examples/pcie_auto_test可执行文件，参考examples/pcie_auto_test/README.md文档部署测试环境。

- PCIe remove/rescan 功能
  - RC端执行`./pcie_auto_test_rc 2 0`以及`./pcie_auto_test_rc 3 0`选择第0个设备进行Remove以及Rescan操作。
- 通讯是否正常恢复
- 业务是否茁壮

- RC端执行 `./pcie_auto_test_rc 0 0` 选择第0个设备进行API稳定性操作。

7.7 功能 | OTA

请阅读”单EP卡对接通用RC“章节，确认是否为需要快速启动的产品形态，并确认所使用存储方案并做对应OTA验证，主要包括：

- 单存储
- 双存储
- EP Boot

详细参考“EP卡OTA”章节。

8. 常见问题处理

8.1 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
```

8.2 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。

8.2.1 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。

8.2.2 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。

8.2.3 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。

8.2.4 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。

8.2.5 Cold Reset(Power On Reset)

上电复位对应设备的开机流程。

8.3 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开始增加linux 内核锁定功能，如果在bios使能安全启动功能，会打开内核锁定导致mmap失败。 需要进入bios关闭安全启动即可正常。

8.4 HugePage配置

RK平台Hugepage大小支持2M、32M、1GB，其它平台支持大小以实际系统为主。单个Hugepage大小配置需要大于SDK中单个发送buff以及单个接收buff的大小，Hugepage数量需要超过所需要的Buff数量。

比如：5个发送Buff，单个Buff是4M，5个接收Buff，单个Buff是8M。

这种情况下就不能配置Hugepage为2M，最小配置需要32M

发送Buff需要的Hugepage个数 = (int) (5 / (32M / 4M)) + 1 = 1

接收Buff需要的Hugepage个数 = (int) (5 / (32M / 8M)) + 1 = 2

则最终配置为 `default_hugepagesz=32M, hugepagesz=32M, hugepages=3`

8.4.1 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
```

8.4.2 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=32M hugepagesz=32M hugepages=16`,配置Hugepage为32M大小，并且可使用个数为16。

```
+++ 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
root=PARTUUID=614e0000-0000 rw rootwait default_hugepagesz=32M hugepagesz=32M hugepages=16 ";
        };

        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:/userdata]# cat /proc/meminfo | grep Huge
HugePages_Total:    16
HugePages_Free:     16
HugePages_Rsvd:      0
HugePages_Surp:      0
Hugepagesize:    32768 kB
Hugetlb:       1048576 kB
```

8.5 使用API接口获取user cmd区域或者ep info区域后，操作该区域导致内核报错或者报错Bus error

有两种原因会导致这个问题出现：

1. 使用memset或者memcpy等接口去操作这块内存。
2. 应用的编译优化超过了-O1。

第一种情况下检查代码去掉memset或者memcpy接口，使用for循环操作。
第二种情况可以降低优化等级，或者在编译参数中添加 `-fno-tree-loop-optimize` 关闭循环优化。

9. API参考

9.1 PCIE RC

9.1.1 rk_pcie_get_pci_bus_addresses

【描述】

获取系统中所有匹配的PCI设备总线地址。

【语法】

```
int rk_pcie_get_pci_bus_addresses(int vendor_id, int device_id, struct pcie_dev_bus *pci_bus)
```

【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
pci_bus	pci设备总线地址结构体	输出

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码。

【举例】

请参见rk_pcie_device_init的举例。

【相关主题】

rk_pcie_get_pci_bus_addresses

9.1.2 rk_pcie_device_init

【描述】

初始化pcie设备。

【语法】

```
RK_PCIE_HANDLES rk_pcie_device_init(struct pcie_dev_attr_st *dev)
```

【参数】

参数名称	描述	输入/输出
dev	PCIE设备配置结构体	输入

【返回值】

返回值	描述
非NULL	成功
NULL	失败

【举例】

```
#define VENDOR_ID           0x1d87
#define DEVICE_ID           0x356a

struct pcie_dev_bus dev_bus;
struct pcie_dev_attr_st dev_attr;
RK_PCIE_HANDLES rk_handle;
int ret;

ret = rk_pcie_get_pci_bus_addresses(VENDOR_ID, DEVICE_ID, &dev_bus);
if (ret != 0) {
   printf("get pci bus address fail\n");
   return -1;
}

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

```
if (dev_bus.dev_max_num == 0) {
    printf("no ep device, exit ... \n");
} else {
    //初始化PCIE设备
    dev_attr.using_rc_dma = 0; //关闭RC DMA，使用EP DMA传输
    dev_attr.rc_dma_chn = 0;
    dev_attr.vendor_id = VENDOR_ID;
    dev_attr.device_id = DEVICE_ID;
    dev_attr.use_dma_only_in_rc = 0;
    //选择第一个匹配设备
    memcpy(dev_attr.pci_bus_address, dev_bus.pci_bus_address[0],
strlen(dev_bus.pci_bus_address[0]));

    //初始化EP设备
    rk_handle = rk_pcie_device_init(&dev_attr);
    if (rk_handle == NULL) {
        printf("pcie device init fail:%s \n", dev_attr.pci_bus_address);
        return -1;
    }

    //反初始化pcie设备，释放资源
    ret = rk_pcie_device_deinit(rk_handle, 0);
    if (ret != 0) {
        printf("pcie deinit device fail....ret = %d\n", ret);
        return -1;
    }
}
```

【相关主题】

rk_pcie_device_deinit

9.1.3 rk_pcie_device_deinit

【描述】

反初始化pcie设备，释放资源。

【语法】

```
int rk_pcie_device_deinit(RK_PCIE_HANDLES handles, int en_remove)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
en_remove	释放资源并移除该设备	输入

【返回值】

返回值	描述
0	正常
非0	异常，请参见错误码

【举例】

请参见rk_pcie_device_init的举例。

【相关主题】

rk_pcie_device_init

9.1.4 rk_pcie_task_create

【描述】

根据属性创建task, 申请对应内存、BUFF管理等。该接口会阻塞等待EP设备调用task创建接口。

【语法】

```
int rk_pcie_task_create(RK_PCIE_HANDLES handles, struct pcie_task_attr_st *task)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
task	PCIE Task配置结构体	输入

【返回值】

返回值	描述
大于0	正常，task_id
小于0	异常，请参见错误码

【举例】

```
//选择第一个匹配设备
rc_ctx.rk_handle = rk_pcie_device_init(&rc_ctx.dev_attr);
if (rc_ctx.rk_handle == NULL) {
   printf("pcie device init fail:%s \n", rc_ctx.dev_attr.pci_bus_address);
   return -1;
}

rc_ctx.task_attr.send_buff_num = 5; //发送BUF个数
rc_ctx.task_attr.send_buff_size = 1048576;  //发送BUF大小
rc_ctx.task_attr.recv_buff_num = 5; //接收BUF个数
rc_ctx.task_attr.recv_buff_size = 8294408 ; //接收BUF大小
rc_ctx.task_attr.wait_timeout_ms = -1;  //阻塞等待
rc_ctx.task_id = rk_pcie_task_create(rc_ctx.rk_handle, &rc_ctx.task_attr);
```

```
if (rc_ctx.task_id < 0) {
    printf("pcie init create task fail....\n");
    return -1;
}

rk_pcie_get_ep_mode(rc_ctx.rk_handle, &rc_ctx.devmode);
printf("ep ap init, dev mode: %d, submode: %d\n", rc_ctx.devmode.mode, rc_ctx.devmode.submode);

ret = rk_pcie_task_destroy(rc_ctx.rk_handle, rc_ctx.task_id);
if (ret != 0) {
    printf("pcie destory task fail....ret = %d\n", ret);
    return -1;
}
```

【相关主题】

rk_pcie_task_destroy

【注意】

该接口依赖 `rk_pcie_dev_init`，需要先调用 `rk_pcie_dev_init` 初始化设备才能进行task初始化。

9.1.5 rk_pcie_task_destroy

【描述】

反初始化Task，释放Task对应的内存、BUFF管理等资源。

【语法】

```
int rk_pcie_task_destroy(RK_PCIE_HANDLES handles, int task_id)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
task_id	Task ID	输入

【返回值】

返回值	描述
0	正常
非0	失败，请参见错误码。

【举例】

请参见rk_pcie_ap_init的举例。

【相关主题】

rk_pcie_ap_init

9.1.6 rk_pcie_boot_init

【描述】

初始化PCIE BOOT。

【语法】

```
int rk_pcie_boot_init(RK_PCIE_HANDLES handles)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入

【返回值】

返回值	描述
0	正常
非0	失败，请参见错误码。

【举例】

```
//初始化PCIE设备
rc_ctx.rk_handle = rk_pcie_device_init(&rc_ctx.dev_attr);
if (rc_ctx.rk_handle == NULL) {
   printf("device init fail \n");
   return -1;
}

//初始化PCIE Boot
ret = rk_pcie_boot_init(rc_ctx.rk_handle);
if (ret != 0) {
   printf("pcie boot init ep fail....ret = %d\n", ret);
   return -1;
}

// download uboot
printf("download uboot addr: 0x%x....\n", RKEP_LOAD_UBOOT_ADDR);
ret = rk_ep_download_firmware(&rc_ctx, uboot_path, RKEP_LOAD_UBOOT_ADDR);
if (ret != 0) {
   printf("pcie download uboot to ep fail....ret = %d\n", ret);
   return -1;
}

// download boot
printf("download boot addr: 0x%x....\n", RKEP_LOAD_BOOT_ADDR);
ret = rk_ep_download_firmware(&rc_ctx, boot_path, RKEP_LOAD_BOOT_ADDR);
if (ret != 0) {
   printf("pcie download boot to ep fail....ret = %d\n", ret);
   return -1;
}
```

```
printf("run ep ....\n");
//启动EP固件
ret = rk_pcie_boot_run(rc_ctx.rk_handle);
if (ret != 0) {
   printf("pcie boot run ep fail....ret = %d\n", ret);
   return -1;
}

//释放PCIE Boot资源
ret = rk_pcie_boot_deinit(rc_ctx.rk_handle);
if (ret != 0) {
   printf("pcie boot deinit ep fail....ret = %d\n", ret);
   return -1;
}

printf("download ep end\n");
```

【相关主题】

rk_pcie_boot_deinit

【注意】

该接口依赖 `rk_pcie_dev_init`，需要先调用 `rk_pcie_dev_init` 初始化设备才能进行boot初始化。

9.1.7 rk_pcie_boot_deinit

【描述】

反初始化PCIE BOOT。

【语法】

```
int rk_pcie_boot_deinit(RK_PCIE_HANDLES handles)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入

【返回值】

返回值	描述
0	正常
非0	失败，请参见错误码。

【举例】

rk_pcie_boot_init

【相关主题】

rk_pcie_boot_init

9.1.8 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, unsigned char *firmware, size_t firmware_size, unsigned long remote_load_addr)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	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
非0	失败，请参见错误码。

【举例】

```
int rk_ep_download_firmware(struct rc_context_st *rc_ctx, char *firmware, unsigned long load_addr)
{
    int fd;
    unsigned long len;
    unsigned long size = 1024 * 1024;
    void *buffer = malloc(size);
    int ret;
    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) {
        ret = rk_pcie_boot_download_firmware(rc_ctx->rk_handle, buffer, len, load_addr);
        if (ret != 0) {
            return -1;
        }

        load_addr += len;
```

```
    }

    close(fd);
    return 0;
}
```

【相关主题】

rk_ep_download_firmware

9.1.9 rk_pcie_boot_run

【描述】

启动PCIE BOOT固件。

【语法】

```
int rk_pcie_boot_run(RK_PCIE_HANDLES handles)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入

【返回值】

返回值	描述
0	正常
非0	失败，请参见错误码。

【举例】

rk_pcie_boot_init

【相关主题】

rk_pcie_boot_run

9.1.10 rk_pcie_get_ep_info

【描述】

获取EP状态信息，最大支持96KB。

【语法】

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

【参数】

参数名称	描述	输入/输出
handles	设备句柄	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
非0	失败，请参见错误码。

【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, 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

9.1.11 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。最大支持256KB。

【语法】

```
void *rk_pcie_get_user_cmd(RK_PCIE_HANDLES handles)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入

【返回值】

返回值	描述
非NULL	正常，自定义命令区内存指针
NULL	差异

【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);
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

9.1.12 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, int task_id, enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
task_id	Task ID	输入
type	PCIE Buff类型	输入

【返回值】

返回值	描述
非NULL	可用Buff指针
NULL	无可用Buff指针

【举例】

```
node = rk_pcie_get_buff(ctx->rk_handle, ctx->task_id, E_RECV); //获取可用接收BUF
if (node) {
  if (check_info(node->vir_addr + node->size - CHECK_SIZE) == 0) { //校验BUF数据正确
    pthread_mutex_unlock(&ctx->mutex_stream_speed);
    ctx->stream_speed_index++;
    pthread_mutex_unlock(&ctx->mutex_stream_speed);
    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++) {
      unsigned char *p = node->vir_addr + node->size - CHECK_SIZE;
      printf("%d ", p[i]);
    }
    printf("\ncheck data ===ssss========node->size = %d======== error \n", (int)node->size);
  }
  rk_pcie_release_buff(ctx->rk_handle, ctx->task_id, node);  //释放BUF
} else {
  usleep(1000);
}
```

【相关主题】

rk_pcie_release_buff

9.1.13 rk_pcie_send_buff

【描述】

发送PCIE buff，会阻塞至DMA传输完成。

【语法】

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

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
task_id	Task ID	输入
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
0	发送成功，表示数据已经通过DMA传输完成
非0	失败，请参见错误码。

【举例】

```
node = rk_pcie_get_buff(ctx->rk_handle, ctx->task_id, E_SEND); //获取可用的发送BUF
if (node) {
   node->size = buffer_size;
   //操作BUF,写入数据
   memcpy(node->vir_addr, mptr, node->size);
   add_check_info(node->vir_addr + node->size - CHECK_SIZE);
   rk_pcie_send_buff(ctx->rk_handle, ctx->task_id, node); //发送到对端
   index++;
} else {
   usleep(1000);
}
```

【相关主题】

rk_pcie_send_buff

9.1.14 rk_pcie_get_bar1_user_buffer

【描述】

获取bar1虚拟映射地址。

【语法】

```
void *rk_pcie_get_bar1_user_buffer(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
void *	虚拟地址
NULL	失败，请参见错误码。

9.1.15 rk_pcie_get_bar5_user_buffer

【描述】

获取bar1虚拟映射地址。

【语法】

```
void *rk_pcie_get_bar5_user_buffer(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
void *	虚拟地址
NULL	失败，请参见错误码。

9.1.16 rk_pcie_set_bar_inbound_cpu_addr

【描述】

设置bar inbound映射。

【语法】

```
u_int64_t rk_pcie_set_bar_inbound_cpu_addr(RK_PCIE_HANDLES handles, u_int32_t atu_index,
u_int64_t cpu_address, u_int32_t 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
atu_index	atu index	输入
cpu_address	映射cpu地址	输入
size	映射大小限制	输入

【返回值】

返回值	描述
cpu_address	返回cpu地址

9.1.17 rk_pcie_release_buff

【描述】

释放PCIE buff。

【语法】

```
int rk_pcie_release_buff(RK_PCIE_HANDLES handles, int task_id, 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
task_id	Task ID	输入
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
0	释放成功
非0	失败，请参见错误码。

【举例】

rk_pcie_get_buff

【相关主题】

rk_pcie_release_buff

9.1.18 rk_pcie_get_buff_max_size

【描述】

获取buff支持的最大size。

【语法】

```
long rk_pcie_get_buff_max_size(RK_PCIE_HANDLES handles, int task_id, 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
task_id	Task ID	输入
type	PCIE Buff类型	输入

【返回值】

返回值	描述
大于等于0	buff size
小于0	失败，请参见错误码

【举例】

```
long buffer_size = rk_pcie_get_buff_max_size(ctx->rk_handle, ctx->task_id, E_SEND);
```

【相关主题】

rk_pcie_get_buff_max_size

9.1.19 rk_pcie_get_ep_mode

【描述】

获取EP设备当前运行模式，支持从loader、kernel、application等模式下状态获取。模式信息可以在rk_pcie_rc.h获取

【语法】

```
int rk_pcie_get_ep_mode(RK_PCIE_HANDLES handles, struct pcie_dev_mode *devmode)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
devmode	设备模式结构体	输出

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【举例】

```
struct rc_context_st {
```

```
    RK_PCIE_HANDLES rk_handle;
    struct pcie_task_attr_st dev_attr;
    struct pcie_dev_mode devmode;
}rc_ctx;

//选择第一个匹配设备初始化
rc_ctx.rk_handle = rk_pcie_device_init(&rc_ctx.dev_attr);
if (rc_ctx.rk_handle == NULL) {
    printf("device init fail \n");
    usleep(1000000);
    continue;
}

rk_pcie_get_ep_mode(rc_ctx.rk_handle, &rc_ctx.devmode);
printf("111 ep device init, dev mode: 0x%.2x, submode: 0x%.2x\n", rc_ctx.devmode.mode,
rc_ctx.devmode.submode);
```

【相关主题】

rk_pcie_get_ep_mode

9.1.20 rk_pcie_rescan_devices

【描述】

重新扫描系统PCI设备，EP设备异常重启后恢复需要remove设备再rescan设备。

【语法】

```
void rk_pcie_rescan_devices(void)
```

【参数】

无

【返回值】

无

【举例】

无

【相关主题】

rk_pcie_rescan_devices

9.1.21 rk_pcie_send_msg_to_ep_bus

【描述】

发送消息到EP端，单个消息最大支持1504Byte

【语法】

```
int rk_pcie_send_msg_to_ep_bus(RK_PCIE_HANDLES handles, int task_id, void *data, size_t size)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
task_id	Task ID	输入
data	消息指针	输入
size	消息长度	输入

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【举例】

```
rc_ctx.task_attr.send_buff_num = 5;
rc_ctx.task_attr.send_buff_size = 5 * 1024 * 1024;
rc_ctx.task_attr.recv_buff_num = 5;
rc_ctx.task_attr.recv_buff_size = 5 * 1024 * 1024;
rc_ctx.task_attr.wait_timeout_ms = -1;
rc_ctx.task_id = rk_pcie_task_create(rc_ctx.rk_handle, &rc_ctx.task_attr);
if (rc_ctx.task_id < 0) {
   printf("pcie init create task fail....\n");
   return -1;
}

rk_pcie_get_ep_mode(rc_ctx.rk_handle, &rc_ctx.devmode);
printf("ep ap init, dev mode: 0x%.2x, submode: 0x%.2x\n", rc_ctx.devmode.mode,
rc_ctx.devmode.submode);

rc_ctx.user_cmd.cmd = EP_TEST_STATUS_RUN_APP;
rk_pcie_send_msg_to_ep_bus(rc_ctx.rk_handle, rc_ctx.task_id, (void *)&rc_ctx.user_cmd, sizeof(struct
task_user_cmd_st));
```

【相关主题】

rk_pcie_send_msg_to_ep_bus

9.1.22 rk_pcie_get_msg_for_task_id

【描述】

通过Task id获取EP消息

【语法】

```
int rk_pcie_get_msg_for_task_id(RK_PCIE_HANDLES handles, struct pcie_task_msg_st *msg, int task_id,
int timeout_ms)
```

【参数】

参数名称	描述	输入/输出
handles	设备句柄	输入
msg	消息结构体指针	输出
task_id	任务ID	输入
timeout_ms	超时时间，-1为阻塞模式(不推荐，会一直在内核等待)，建议100ms，单位ms	输入

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【举例】

```
unsigned char cmd;
struct pcie_task_msg_st msg;
ret = rk_pcie_get_msg_for_task_id(rc_ctx.rk_handle, &msg, rc_ctx.task_id, 100);
if (ret == RK_PCIE_OK) {
  cmd = msg.data[0];
  printf("recv ep msg: %d \n", cmd);
  if (cmd == 10) {
    break;
  }
}
```

【相关主题】

rk_pcie_get_msg_for_task_id

9.1.23 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。
- ETask_Msg_Type：定义Task消息类型。
- struct pcie_buff_node：定义PCIE buff属性结构体。
- struct pcie_dev_bus：定义PCI设备总线信息。
- struct pcie_dev_mode：定义PCIE EP设备模式。
- struct pcie_dev_attr_st：定义PCIE EP设备属性结构体。
- struct pcie_task_attr_st：定义Task属性结构体。
- struct pcie_task_msg_st：定义Task消息结构体。

9.1.23.1 RK_PCIE_HANDLES

【说明】

定义设备的句柄。

【定义】

```
typedef void* RK_PCIE_HANDLES;
```

9.1.23.2 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。

9.1.23.3 ETask_Msg_Type

【说明】

定义PCIE Msg类型。

【定义】

```
enum ETask_Msg_Type {
  E_TASK_MSG_NONE = 0,
  E_TASK_MSG_CREATE,
  E_TASK_MSG_DESTORY,
  E_TASK_MSG_USER,
  E_TASK_MSG_MAX,
};
```

【成员】

成员名称	描述
E_TASK_MSG_NONE	默认值，无意义。
E_TASK_MSG_CREATE	创建任务。
E_TASK_MSG_DESTORY	销毁任务。
E_TASK_MSG_USER	用户自定义消息。
E_TASK_MSG_MAX	最大消息类型。

9.1.23.4 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。

9.1.23.5 pcie_dev_bus

【说明】

定义PCI设备总线信息。

【定义】

```
struct pcie_dev_bus {
   int dev_max_num;
   char pci_bus_address[PCI_DEV_MAX][PCI_BUS_MAX_NAME];
};
```

【成员】

成员名称	描述
dev_max_num	系统匹配的设备数量。
pci_bus_address	PCI设备的总线地址。

9.1.23.6 pcie_dev_mode

【说明】

定义PCIE EP设备模式。

【定义】

```
struct pcie_dev_mode {
  unsigned short mode;
  unsigned short submode;
};
```

【成员】

成员名称	描述
mode	PCIE EP设备当前模式，取值范围如下： #define RKEP_MODE_BOOTROM 1 #define RKEP_MODE_LOADER 2 #define RKEP_MODE_KERNEL 3 #define RKEP_MODE_FUN0 4 #define RKEP_MODE_ERR 0xffff //设备异常需要重新加载
submode	PCIE EP设备模式对应的状态，取值范围如下： #define RKEP_SMODE_INIT 0 #define RKEP_SMODE_LNKRDY 1 #define RKEP_SMODE_LNKUP 2 #define RKEP_SMODE_FWDLRDY 0x10 #define RKEP_SMODE_FWDLDONE 0x11 #define RKEP_SMODE_APPRDY 0x20 #define RKEP_SMODE_APPDEINIT 0x21 #define RKEP_SMODE_ERR 0xffff

9.1.23.7 pcie_dev_attr_st

【说明】

定义PCIE EP设备属性结构体。

【定义】

```
struct pcie_dev_st {
  int vendor_id;
  int device_id;
  char pci_bus_address[PCI_BUS_MAX_NAME];
  unsigned char using_rc_dma: 1;
  unsigned char rc_dma_chn: 1;
  unsigned char enable_speed: 1;
  unsigned char use_dma_only_in_rc: 1;// 只在RC端调用DMA收发数据
};
```

【成员】

成员名称	描述
vendor_id	EP设备Vendor ID。
device_id	EP设备Device ID。
pci_bus_address	EP设备的总线地址。
using_rc_dma	RK互联场景下，可使能该模式使用RC端DMA增加传输速率。多EP场景下推荐关闭配置，使用EP DMA。
rc_dma_chn	RC DMA支持2个通道，取值0/1。
enable_speed	使能速率统计功能。可通过cat /tmp/pcie_debug*查看当前EP设备总传输速率。
use_dma_only_in_rc	只在RC端调用DMA收发数据

9.1.23.8 pcie_task_attr_st

【说明】

定义PCIE Task属性结构体。

【定义】

```
struct pcie_task_attr_st {
  int wait_timeout_ms;
  unsigned int send_buff_num;
  size_t send_buff_size;
  unsigned int recv_buff_num;
  size_t recv_buff_size;
};
```

【成员】

成员名称	描述
wait_timeout_ms	设置接口超时时间，-1为阻塞模式。
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。

9.1.23.9 pcie_task_msg_st

【说明】

定义PCIE Task消息结构体。

【定义】

```
struct pcie_task_msg_st {
  enum ETask_Msg_Type type;
  int task_id;
  size_t size;
  char lock;
  unsigned char data[TASK_MSG_DATA_SIZE];
};
```

【成员】

成员名称	描述
type	消息类型。
task_id	Task ID。
size	消息长度。
lock	保留字。
data	消息内容。

9.2 PCIE EP

9.2.1 rk_pcie_device_init

【描述】

初始化PCIE设备

【语法】

```
int rk_pcie_device_init(void)
```

【参数】

无

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【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(void)
{
  int ret = 0;
  struct task_context_st task_ctx[PCIE_DMA_TASK_MAX] = {0};
  struct pcie_task_msg_st msg;
  struct task_user_cmd_st *user_cmd;

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

  while (1) {
    ret = rk_pcie_get_msg_for_bus(&msg, 100);
    if (ret) {
      printf("task_id:%d type:%d size:%ld \n", msg.task_id, msg.type, msg.size);
      switch (msg.type) {
      case E_TASK_MSG_CREATE:
        task_create(&task_ctx[msg.task_id], msg.task_id);
        break;
      case E_TASK_MSG_DESTORY:
        task_destory(&task_ctx[msg.task_id], msg.task_id);
        break;
      case E_TASK_MSG_USER: {
        user_cmd = (struct task_user_cmd_st *)msg.data;
        switch (user_cmd->cmd) {
        case TASK_USER_CMD_INIT_APP:
          app_create(&task_ctx[msg.task_id], msg.task_id);
          break;
        case TASK_USER_CMD_DEINIT_APP:
          app_destory(&task_ctx[msg.task_id], msg.task_id);
          break;
        }
      }
```

```
            break;
            default:
              printf("error msg type!!! \n");
              break;
            }
          }
        }

        rk_pcie_device_deinit();

        return 0;
}
```

【相关主题】

rk_pcie_device_deinit

9.2.2 rk_pcie_device_deinit

【描述】

反初始化pcie，释放资源

【语法】

```
int rk_pcie_device_deinit(void)
```

9.2.3 rk_pcie_task_create

【描述】

创建Task，并且初始化内存、BUFF管理等。

【语法】

```
int rk_pcie_task_create(int task_id)
```

【参数】

参数名称	描述	输入/输出
task_id	Task ID，RC端通过消息传递过来	输入

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【举例】

请参见rk_pcie_device_init的举例。

【相关主题】

rk_pcie_device_deinit

9.2.4 rk_pcie_task_destroy

【描述】

反初始化Task，释放内存、BUFF管理等资源。

【语法】

```
int rk_pcie_task_destroy(int task_id)
```

【参数】

参数名称	描述	输入/输出
task_id	Task ID	输入

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【举例】

请参见rk_pcie_device_init的举例。

【相关主题】

rk_pcie_device_init

9.2.5 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	成功
非0	失败，请参见错误码

【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

9.2.6 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
非NULL	正常，自定义命令区内存指针
NULL	失败

【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

9.2.7 rk_pcie_get_bar1_user_buffer

【描述】

获取bar1虚拟映射地址。

【语法】

```
void *rk_pcie_get_bar1_user_buffer(void)
```

【参数】

无。

【返回值】

返回值	描述
void *	虚拟地址
NULL	失败，请参见错误码。

9.2.8 rk_pcie_get_bar5_user_buffer

【描述】

获取bar1虚拟映射地址。

【语法】

```
void *rk_pcie_get_bar5_user_buffer(void)
```

【参数】

无。

【返回值】

返回值	描述
void *	虚拟地址
NULL	失败，请参见错误码。

9.2.9 rk_pcie_set_bar_inbound_cpu_addr

【描述】

设置bar inbound映射。

【语法】

```
u_int64_t rk_pcie_set_bar_inbound_cpu_addr(u_int32_t atu_index, u_int64_t cpu_address, u_int32_t size)
```

【参数】

参数名称	描述	输入/输出
atu_index	atu index	输入
cpu_address	映射cpu地址	输入
size	映射大小限制	输入

【返回值】

返回值	描述
cpu_address	返回cpu地址

9.2.10 rk_pcie_get_buff

【描述】

获取可用PCIE buff。

【语法】

```
struct pcie_buff_node *rk_pcie_get_buff(int task_id, enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
task_id	Task ID	输入
type	PCIE Buff类型	输入

【返回值】

返回值	描述
非NULL	可用Buff指针
NULL	无可用Buff

【举例】

```
node = rk_pcie_get_buff(ctx->task_id, E_SEND);
if (node) {
   node->size = buffer_size;
   memset(node->vir_addr, index, node->size);
   add_check_info(node->vir_addr + node->size - CHECK_SIZE);
   rk_pcie_send_buff(ctx->task_id, node);
   index++;
} else {
   usleep(1000);
}
```

【相关主题】

rk_pcie_send_buff

9.2.11 rk_pcie_send_buff

【描述】

发送PCIE buff。

【语法】

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

【参数】

参数名称	描述	输入/输出
task_id	Task ID	输入
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
0	发送成功
非0	失败，请参见错误码

【举例】

rk_pcie_get_buff

【相关主题】

rk_pcie_get_buff

9.2.12 rk_pcie_release_buff

【描述】

释放PCIE buff。

【语法】

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

【参数】

参数名称	描述	输入/输出
task_id	Task ID	输入
pbuff	PCIE Buff指针	输入

【返回值】

返回值	描述
0	释放成功
非0	失败，请参见错误码

【举例】

```
node = rk_pcie_get_buff(ctx->task_id, E_RECV);
if (node) {
  if (check_info(node->vir_addr + node->size - CHECK_SIZE) == 0) {
    pthread_mutex_lock(&ctx->mutex_stream_speed);
    ctx->stream_speed_index++;
```

```
        pthread_mutex_unlock(&ctx->mutex_stream_speed);
        // if (mptr != NULL) {
        //    memcpy(mptr, node->vir_addr, 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++) {
            unsigned char *p = node->vir_addr + node->size - CHECK_SIZE;
            printf("%d ", p[i]);
        }
        printf("\ncheck data ===ssss========node->size = %d======== error \n", (int)node->size);
    }
    rk_pcie_release_buff(ctx->task_id, node);
} else {
    usleep(1000);
}
```

【相关主题】

rk_pcie_release_buff

9.2.13 rk_pcie_get_buff_max_size

【描述】

获取buff支持的最大size。

【语法】

```
long rk_pcie_get_buff_max_size(int task_id, enum EBuff_Type type)
```

【参数】

参数名称	描述	输入/输出
task_id	Task ID	输入
type	PCIE Buff类型	输入

【返回值】

返回值	描述
大于等于0	buff size
小于0	失败，请参见错误码

【举例】

```
long buffer_size = rk_pcie_get_buff_max_size(ctx->task_id, E_SEND);
```

【相关主题】

rk_pcie_get_buff_max_size

9.2.14 rk_pcie_get_msg_for_bus

【描述】

等待RC端消息。

【语法】

```
int rk_pcie_get_msg_for_bus(struct pcie_task_msg_st *msg, int timeout_ms)
```

【参数】

参数名称	描述	输入/输出
timeout_ms	超时时间，-1为阻塞模式(不推荐，会一直在内核等待)，建议100ms，单位ms	输入

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【举例】

请参见rk_pcie_device_init的举例。

【相关主题】

rk_pcie_device_deinit

9.2.15 rk_pcie_send_msg_to_rc_task

【描述】

发送消息到RC端

【语法】

```
int rk_pcie_send_msg_to_rc_task(int task_id, void *data, size_t size)
```

【参数】

参数名称	描述	输入/输出
task_id	Task ID	输入
data	消息指针	输入
size	消息长度	输入

【返回值】

返回值	描述
0	成功
非0	失败，请参见错误码

【举例】

请参见rk_pcie_device_init的举例。

【相关主题】

rk_pcie_device_deinit

9.2.16 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。
- ETask_Msg_Type：定义Task消息类型。
- struct pcie_buff_node：定义PCIE buff属性结构体。
- struct pcie_task_msg_st：定义消息属性结构体。

9.2.16.1 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。

9.2.16.2 ETask_Msg_Type

【说明】

定义Task消息类型。

【定义】

```
enum ETask_Msg_Type {
  E_TASK_MSG_NONE = 0,
  E_TASK_MSG_CREATE,
  E_TASK_MSG_DESTORY,
  E_TASK_MSG_USER,
  E_TASK_MSG_MAX,
};
```

【成员】

成员名称	描述
E_TASK_MSG_NONE	默认值，无意义
E_TASK_MSG_CREATE	创建任务
E_TASK_MSG_DESTORY	销毁任务
E_TASK_MSG_USER	用户自定义消息
E_TASK_MSG_MAX	最大消息类型

9.2.16.3 pcie_buff_node

【说明】

定义PCIE buff属性结构体。该结构体修改需要同时改RC和EP。

【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。

9.2.16.4 pcie_task_msg_st

【说明】

定义消息属性结构体。该结构体修改需要同时改RC和EP。

【定义】

```
struct pcie_task_msg_st {
  enum ETask_Msg_Type type;
  int task_id;
  size_t size;
  char lock;
  unsigned char data[TASK_MSG_DATA_SIZE];
};
```

【成员】

成员名称	描述
type	消息类型。
task_id	Task ID。
size	消息长度。
lock	保留字。
data	消息内容。

9.3 错误码

错误代码	宏定义	描述
-1	RK_PCIE_ERR_BAD	PCIE链接异常
-2	RK_PCIE_ERR_UNKNOWN	系统异常
-3	RK_PCIE_ERR_NULL_PTR	空指针错误
-4	RK_PCIE_ERR_MALLOC	分配内存失败，如系统内存不足
-5	RK_PCIE_ERR_OPEN_FILE	打开文件失败
-6	RK_PCIE_ERR_VALUE	错误异常值
-8	RK_PCIE_ERR_TIMEOUT	超时返回
-10	RK_PCIE_ERR_UNSUPPORT	不支持的功能
-13	RK_PCIE_ERR_HW_UNSUPPORT	硬件不支持
-15	RK_PCIE_ERR_MMAP	mmap失败
-65	RK_PCIE_ERR_INIT	初始化失败
-66	RK_PCIE_ERR_NOINIT	系统没有初始化
-68	RK_PCIE_ERR_NOMEM	内存不足
-69	RK_PCIE_ERR_OUTOF_RANGE	参数超出范围