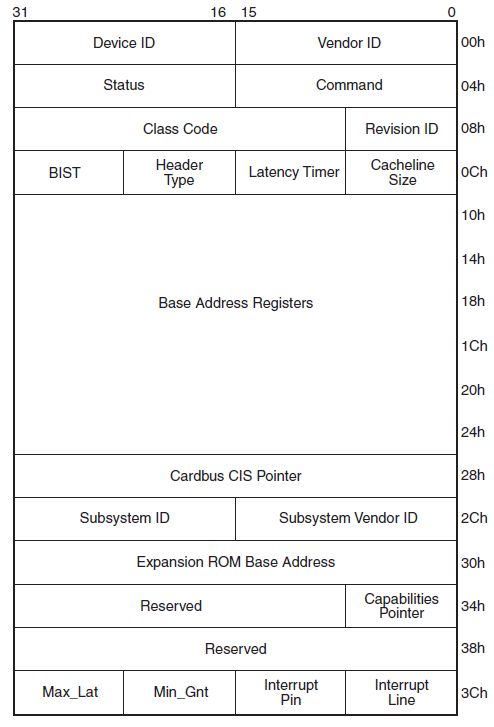
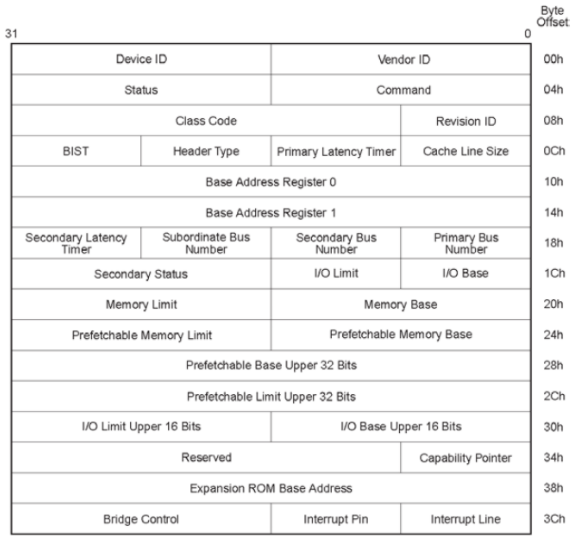
Hi3536 PCIE学习笔记2(linux3.10.y)

何良斌

# PCI总线简单介绍

## 1、PCI总线配置

两类配置请求：Type 00h配置请求，Type 01配置请求。Type 00h配置请求：访问与HOST主桥或者PCI桥直接相连的PCI Agent设备或这PCI桥；Type 01h配置请求：至少需要穿越一个PCI桥，访问没有与HOST主桥或PCI桥直接相连的PCI Agent设备或者PCI桥，只有PCI桥接收该类型。

PCIAgent type 00h 配置空间 PCI桥 Type 01h 配置空间

# 二、内核总概述

## 1、内核配置

（1）kernel添加对PCI总线的支持

（2）涉及驱动

底层驱动：hi35xx\_dev\_host.ko pcit\_dma\_host.ko

PCIE消息驱动：mcc\_drv\_host.ko mcc\_usrdev\_host.ko

PCIV模块驱动：hi3536\_pciv.ko hi3536\_pciv\_fmw.ko

加载uboot需要的驱动：boot\_device.ko

## 2、内核初始化过程

### （1）内核启动流程

start\_kernel() -> rest\_init() -> kernel\_init() -> kernel\_init\_freeable() -> do\_basic\_setup() -> do\_initcalls()

do\_initcalls()：

static void \_\_init do\_initcalls(void)

{

int level;

for (level = 0; level < ARRAY\_SIZE(initcall\_levels) - 1; level++)

do\_initcall\_level(level);

}

各个级别的定义如下：

#define pure\_initcall(fn) \_\_define\_initcall(fn, 0)

#define core\_initcall(fn) \_\_define\_initcall(fn, 1)

#define core\_initcall\_sync(fn) \_\_define\_initcall(fn, 1s)

#define postcore\_initcall(fn) \_\_define\_initcall(fn, 2)

#define postcore\_initcall\_sync(fn) \_\_define\_initcall(fn, 2s)

#define arch\_initcall(fn) \_\_define\_initcall(fn, 3)

#define arch\_initcall\_sync(fn) \_\_define\_initcall(fn, 3s)

#define subsys\_initcall(fn) \_\_define\_initcall(fn, 4)

#define subsys\_initcall\_sync(fn) \_\_define\_initcall(fn, 4s)

#define fs\_initcall(fn) \_\_define\_initcall(fn, 5)

#define fs\_initcall\_sync(fn) \_\_define\_initcall(fn, 5s)

#define rootfs\_initcall(fn) \_\_define\_initcall(fn, rootfs)

#define device\_initcall(fn) \_\_define\_initcall(fn, 6)

#define device\_initcall\_sync(fn) \_\_define\_initcall(fn, 6s)

#define late\_initcall(fn) \_\_define\_initcall(fn, 7)

#define late\_initcall\_sync(fn) \_\_define\_initcall(fn, 7s)

#define module\_init(x) device\_initcall(fn);

### （2）PCI系统根据initcall确定调用顺序

内核目录：drivers/pci

./probe.c: postcore\_initcall(pcibus\_class\_init);

./pci-driver.c: postcore\_initcall(pci\_driver\_init);

./slot.c: subsys\_initcall(pci\_slot\_init);

./hipcie/pcie.c: subsys\_initcall(hisi\_pcie\_init);

./quirks.c: fs\_initcall\_sync(pci\_apply\_final\_quirks);

./proc.c: device\_initcall(pci\_proc\_init);

./pci.c: late\_initcall(pci\_resource\_alignment\_sysfs\_init);

./pci-sysfs.c: late\_initcall(pci\_sysfs\_init);

./hipcie/pcie.c：对PCIE总线进行扫描

### （3）配置寄存器头

PCIe控制器有4K字节的配置寄存器空间，其中前256字节为PCI兼容的配置空间，有两种类型的PCI配置寄存器头PCIe Header Type0和PCIe Header Type1。当偏移0x0E地址段 head\_type为0时代表Header Type0：表示当前设备是一个EP设备即PCI设备；当head\_type为1：代表Header Type1：表示当前设备是一个RC设备即PCI桥or switch。

pci常用三种：pci i/o、pci memory、pci conf，三种都可以被CPU存储。设备驱动程序使用pci i/o pci mem，pci初始化使用pci conf。

# 三、PCI总线文件简述（内核）

## 1、probe.c

PCI detection and setup code

static struct class pcibus\_class = {

.name = "pci\_bus",

.dev\_release = &release\_pcibus\_dev,

.dev\_attrs = pcibus\_dev\_attrs,

};

static int \_\_init pcibus\_class\_init(void)

{

return class\_register(&pcibus\_class);

}

postcore\_initcall(pcibus\_class\_init);

主要作用是注册一个名为“pci\_bus”的class结构，执行完之后会在/sys/class目录下产生一个“pci\_bus”的目录。

## 2、pci-driver.c

该函数也与sysfs文件系统相关，执行完毕后，在/sys/bus目录下建立一个“pci”目录，之后当linux的pci设备使用device\_register函数注册一个新的pci设备时，将在/sys/bus/pci/drivers目录下创建这个设备使用的目录。

（1）总线初始化

struct bus\_type pci\_bus\_type = {

.name = "pci",

.match = pci\_bus\_match,

.uevent = pci\_uevent,

.probe = pci\_device\_probe,

.remove = pci\_device\_remove,

.shutdown = pci\_device\_shutdown,

.dev\_attrs = pci\_dev\_attrs,

.bus\_attrs = pci\_bus\_attrs,

.drv\_attrs = pci\_drv\_attrs,

.pm = PCI\_PM\_OPS\_PTR,

};

static int \_\_init pci\_driver\_init(void)

{

return bus\_register(&pci\_bus\_type);

}

postcore\_initcall(pci\_driver\_init);

（2）bus\_register函数

注册一个总线系统（如这里的pci），总线包含两个重要的成员devices\_kset和drivers\_kset，代表这个总线上的两个链表：设备链表和驱动链表。两个重要链表创建后同时会添加到sysfs系统里，就是通常看到的/sys/bus/pci/drivers和/sys/bus/pci/devices。

（3）pci\_bus\_match

判断一个当前PCI设备是否在驱动支持的列表中。

（4）pci\_uevent

将PCI的关键值存放到kobj\_uevent\_env的环境buffer里

（5）pci\_device\_probe

检查一个驱动想获取一个特定的PCI设备

## 3、slot.c

static int pci\_slot\_init(void)

{

struct kset \*pci\_bus\_kset;

pci\_bus\_kset = bus\_get\_kset(&pci\_bus\_type);

pci\_slots\_kset = kset\_create\_and\_add("slots", NULL,

&pci\_bus\_kset->kobj);

if (!pci\_slots\_kset) {

printk(KERN\_ERR "PCI: Slot initialization failure\n");

return -ENOMEM;

}

return 0;

}

subsys\_initcall(pci\_slot\_init);

初始化PCI总线的卡槽位内核对象，创建后会添加到sysfs系统里，就是通常看到的/sys/bus/pci/slots

**4、quirks.c**

## 5、proc.c

static int \_\_init pci\_proc\_init(void)

{

struct pci\_dev \*dev = NULL;

proc\_bus\_pci\_dir = proc\_mkdir("bus/pci", NULL);

proc\_create("devices", 0, proc\_bus\_pci\_dir,

&proc\_bus\_pci\_dev\_operations);

proc\_initialized = 1;

for\_each\_pci\_dev(dev)

pci\_proc\_attach\_device(dev);

return 0;

}

device\_initcall(pci\_proc\_init);

在proc下创建PCI相关的信息，对应的目录为proc/bus/pci

## 6、pci.c

BUS\_ATTR(resource\_alignment, 0644, pci\_resource\_alignment\_show,

pci\_resource\_alignment\_store);

static int \_\_init pci\_resource\_alignment\_sysfs\_init(void)

{

return bus\_create\_file(&pci\_bus\_type,

&bus\_attr\_resource\_alignment);

}

late\_initcall(pci\_resource\_alignment\_sysfs\_init);

在sys/bus/pci/resource\_alignment文件

## 7、pci-sysfs.c

static int \_\_init pci\_sysfs\_init(void)

{

struct pci\_dev \*pdev = NULL;

int retval;

sysfs\_initialized = 1;

for\_each\_pci\_dev(pdev) {

retval = pci\_create\_sysfs\_dev\_files(pdev);

if (retval) {

pci\_dev\_put(pdev);

return retval;

}

}

return 0;

}

late\_initcall(pci\_sysfs\_init);

将每一个设备添加到sysfs文件系统相应目录下。

## 8、hipcie/pcie.c（控制器驱动）

### （1）注册hipcie设备和驱动

const struct dev\_pm\_ops hisi\_pcie\_pm\_ops = {

.suspend = NULL,

.suspend\_noirq = hisi\_pcie\_plat\_driver\_suspend,

.resume = NULL,

.resume\_noirq = hisi\_pcie\_plat\_driver\_resume

};

#define HISI\_PCIE\_PM\_OPS (&hisi\_pcie\_pm\_ops)

#define PCIE\_RC\_DRV\_NAME "hisi pcie root complex"

static struct platform\_driver hisi\_pcie\_platform\_driver = {

.probe = hisi\_pcie\_plat\_driver\_probe,

.remove = hisi\_pcie\_plat\_driver\_remove,

.driver = {

.owner = THIS\_MODULE,

.name = PCIE\_RC\_DRV\_NAME,

.bus = &platform\_bus\_type,

.pm = HISI\_PCIE\_PM\_OPS

},

};

//PCIE register: 0x1f000000 ~0x1f000fff

static struct resource hisi\_pcie\_resources[] = {

[0] = {

.start = PCIE\_DBI\_BASE, //0x1f000000

.end = PCIE\_DBI\_BASE + \_\_4KB\_\_ - 1,

.flags = IORESOURCE\_REG, //register offset

}

};

static u64 hipcie\_dmamask = DMA\_BIT\_MASK(32);

static struct platform\_device hisi\_pcie\_platform\_device = {

.name = PCIE\_RC\_DRV\_NAME,

.id = 0,

.dev = {

.platform\_data = NULL,

.dma\_mask = &hipcie\_dmamask,

.coherent\_dma\_mask = DMA\_BIT\_MASK(32),

.release = hisi\_pcie\_platform\_device\_release,

},

.num\_resources = ARRAY\_SIZE(hisi\_pcie\_resources),

.resource = hisi\_pcie\_resources,

};

static int \_\_init hisi\_pcie\_init(void)

{

int ret;

ret = platform\_device\_register(&hisi\_pcie\_platform\_device);

if (ret)

goto err\_device;

ret = platform\_driver\_register(&hisi\_pcie\_platform\_driver);

if (ret)

goto err\_driver;

if (pcie\_init()) {

pcie\_error("pcie sys init failed!");

goto err\_init;

}

}

subsys\_initcall(hisi\_pcie\_init);

hipcie通过虚拟地址总线机制platform的方式向内核注册设备和驱动。platform\_driver\_register() ->driver\_register() -> bus\_add\_driver() -> driver\_attach() ->bus\_for\_each\_dev(),对在每个挂在虚拟的platform bus的设备作\_\_driver\_attach() ->driver\_probe\_device()判断drv->bus->match()是否执行成功，此时通过指针执行platform\_match-> strncmp(pdev->name , drv->name , BUS\_ID\_SIZE),如果相符就调用really\_probe(实际就是执行相应设备的platform\_driver->probe(platform\_device),这里probe空实现)

platform\_match:是通过判断设备和驱动name的一致性。

### （2）hipcie初始化pcie\_init()

A．static struct hw\_pci hipcie \_\_initdata = {

.nr\_controllers = 1,

.preinit = pcie\_preinit, //空实现

.swizzle = pci\_common\_swizzle, //中断引脚转换

.setup = pcie\_setup, //获取资源及配置PCIE iATU转换表

.scan = pcie\_scan\_bus, //扫描PCIE物理总线上的设备

.map\_irq = pcie\_map\_irq, //映射中断引脚

};

B．static int \_\_init pcie\_init(void)

{

….

/\*

\* Scene: PCIe host(RC)<--->SWITCH<--->PCIe device(\*)

\* |

\* |------->NULL SLOT

\* PCIe will generate a DataAbort to ARM, when scaning NULL SLOT.

\* Register hook to capture this exception and handle it.

\*/

hook\_fault\_code(22, pcie\_fault, 7, BUS\_OBJERR,

"external abort on non-linefetch");

if (\_\_arch\_pcie\_info\_setup(pcie\_info, &pcie\_controllers\_nr))

return -EIO;

if (\_\_arch\_pcie\_sys\_init(pcie\_info))

goto pcie\_init\_err;

hipcie.nr\_controllers = pcie\_controllers\_nr;

pr\_err("Number of PCIe controllers: %d\n",

hipcie.nr\_controllers);

pci\_common\_init(&hipcie);

return 0;

pcie\_init\_err:

\_\_arch\_pcie\_info\_release(pcie\_info);

return -EIO;

}

\_\_arch\_pcie\_info\_setup完成控制地址转换：

/\* RC configuration space \*/

info->conf\_base\_addr = (unsigned int)ioremap\_nocache(PCIE\_DBI\_BASE,\_\_4KB\_\_); //0x1f000000

/\* Configuration space for all EPs \*/

info->base\_addr = (unsigned int)ioremap\_nocache(PCIE\_EP\_CONF\_BASE, cfg\_size); //0x20000000, 8M, 在pcie\_setup中进行了地址转换设置

misc\_ctrl\_virt = ioremap\_nocache(MISC\_CTRL\_BASE, \_\_4KB\_\_); //0x12120000

\_\_arch\_pcie\_sys\_init：配置PCIE控制器相关内容（开启，时钟，RC模式，配置部分PCI配置空间等）

pci\_common\_init(&hipcie)：调用pcie\_scan\_bus接口实现设备扫描，资源分配，及添加到pci\_bus总线上。

C．pci\_common\_init函数解析

pcibios\_init\_hw(hw, &head);配置资源，扫描设备。

hw->setup(nr, sys)（pcie\_setup）

pcibios\_init\_resources(nr, sys);

sys->bus = hw->scan(nr, sys);（pcie\_scan\_bus）

pci\_fixup\_irqs(pcibios\_swizzle, pcibios\_map\_irq);将pcie虚拟引脚1 2 3 4和CPU的INTA B C D的中断号映射好。

D．逐一解析函数

pcie\_setup：request\_pcie\_res->\_\_arch\_get\_pcie\_resh 和 \_\_arch\_config\_iatu\_tbl:

request\_pcie\_res->\_\_arch\_get\_pcie\_resh: //memory/io存储域在pcie\_scan\_bus时给各个EP进行分配

static struct resource pcie\_mem; //PCIE io/memory空间

static struct resource pcie\_io; //PCIE ioport空间

static void \_\_arch\_get\_pcie\_res(int controller,

struct resource \*\*pmem,

struct resource \*\*pio)

{

\*pmem = &pcie\_mem;

(\*pmem)->start = PCIE\_MEM\_BASE; //0x30000000

(\*pmem)->end = PCIE\_MEM\_BASE + 0x10000000 - 1; //256M

(\*pmem)->flags = IORESOURCE\_MEM;

(\*pmem)->name = "memory";

\*pio = &pcie\_io;

(\*pio)->start = 0x0; //没有对应空间

(\*pio)->end = 0x0;

(\*pio)->flags = IORESOURCE\_IO;

(\*pio)->name = "io";

}

pci\_add\_resource\_offset(&sys->resources, io, sys->io\_offset)

pci\_add\_resource\_offset(&sys->resources, mem, sys->mem\_offset) //mem\_offset=0

\_\_arch\_config\_iatu\_tbl: //所有EP段配置空间转换

struct pcie\_iatu iatu\_table[] = {

{

.viewport = 0, //atu\_viewport

.region\_ctrl\_1 = 0x00000004,

.region\_ctrl\_2 = 0x90000000,

.lbar = PCIE\_EP\_CONF\_BASE + (1<<20)//0x20100000 atu\_base\_low

.ubar = 0x0, //atu\_base\_high

.lar = PCIE\_EP\_CONF\_BASE + (2<<20) - 1, //atu\_limit

.ltar = 0x01000000, //atu\_target\_low

.utar = 0x00000000, //atu\_target\_high

},配置事务0，本地总线地址0x20100000 – 0x201fffff

{

.viewport = 1,

.region\_ctrl\_1 = 0x00000005,

.region\_ctrl\_2 = 0x90000000,

.lbar = PCIE\_EP\_CONF\_BASE + (2<<20), //0x20200000 atu\_base\_low

.ubar = 0x0, //atu\_base\_high

.lar = PCIE\_EP\_CONF\_BASE + (\_\_256MB\_\_ - 1), //atu\_limit

.ltar = 0x02000000, //atu\_target\_low

.utar = 0x00000000, //atu\_target\_high

},配置事务1，本地总线地址0x20200000 – 0x2fffffff

};

只有执行此config后，PCIE才能实现CFG\_TYPE0 和CFG\_TYPE1的配置事务访问（寄存器的具体配置请结合datasheet ATU地址转换，各个数值说明见PDF1886页注意事项）

pcibios\_init\_resources:

pci\_add\_resource\_offset(&sys->resources, &sys->io\_res, sys->io\_offset);

pcie\_scan\_bus:

为总线提供读写操作接口：

static struct pci\_ops pcie\_ops = {

.read = pcie\_read\_conf,

.write = pcie\_write\_conf,

};

pci\_scan\_root\_bus(NULL, sys->busnr, &pcie\_ops, sys, &sys->resources)

-->pci\_create\_root\_bus()建立相应的数据结构pci\_bus, pci\_host\_bridge等

-->pci\_scan\_child\_bus(b) 枚举整个pci总线上的设备

-->pci\_bus\_size\_bridges(buf)和pci\_bus\_assign\_resources(bus) 分配总线资源

Hi3536的驱动pci\_config\_read和pci\_config\_write最终就是调用上面的接口。

pci\_scan\_child\_bus -> pci\_scan\_slot -> pci\_scan\_single\_device -> pci\_scan\_device -> pci\_setup\_device 读取配置空间获取设备相应信息（配置空间信息见1.1节细节）

pci\_bus\_assign\_resources -> pbus\_assign\_resources\_sorted -> \_\_assign\_resources\_sorted -> assign\_requested\_resources\_sorted -> pci\_assign\_resource -> \_pci\_assign\_resource 和 pci\_update\_resource，重新分配设备资源和更新设备的BAR起始基地址。

## 9、内核启动打印

Number of PCIe controllers: 1

PCI host bridge to bus 0000:00

pci\_bus 0000:00: root bus resource [io 0x0000]

pci\_bus 0000:00: root bus resource [mem 0x30000000-0x3fffffff]

pci\_bus 0000:00: No busn resource found for root bus, will use [bus 00-ff]

\*\*\*\*\*\*\*\*\*\* hdr\_type:0x1

\*\*\*\*\*\*\*\*\*\* class:0x6040001

\*\*\*\*\*\*\*\*\*\* cfg\_size:0x1000

\*\*\*\*\*\*\*\*\*\* region[0x10] start:0x0, end:0x7fffff, size:0x7fffff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0x7fffff

\*\*\*\*\*\*\*\*\*\* region[0x14] start:0x0, end:0xffff, size:0xffff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0xffff

PCI: bus0: Fast back to back transfers disabled

pci 0000:00:00.0: bridge configuration invalid ([bus 00-00]), reconfiguring

\*\*\*\*\*\*\*\*\*\* hdr\_type:0x0

\*\*\*\*\*\*\*\*\*\* class:0x4800001

\*\*\*\*\*\*\*\*\*\* cfg\_size:0x1000

\*\*\*\*\*\*\*\*\*\* PCI\_HEADER\_TYPE\_NORMAL

\*\*\*\*\*\*\*\*\*\* region[0x10] start:0x0, end:0xfff, size:0xfff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0xfff

\*\*\*\*\*\*\*\*\*\* region[0x14] start:0x0, end:0x1ffffff, size:0x1ffffff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0x1ffffff

\*\*\*\*\*\*\*\*\*\* region[0x18] start:0x0, end:0x1ffffff, size:0x1ffffff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0x1ffffff

\*\*\*\*\*\*\*\*\*\* region[0x1c] start:0x0, end:0x1ffffff, size:0x1ffffff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0x1ffffff

\*\*\*\*\*\*\*\*\*\* region[0x20] start:0x0, end:0x1ffffff, size:0x1ffffff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0x1ffffff

\*\*\*\*\*\*\*\*\*\* region[0x24] start:0x0, end:0x1ffffff, size:0x1ffffff

\*\*\*\*\*\*\*\*\*\* bus\_region start:0x30000000, end:0x3fffffff

\*\*\*\*\*\*\*\*\*\*offset:0x0, res start:0x0, end0x1ffffff

PCI: bus1: Fast back to back transfers disabled

\*\*\*\*\*pin:1 映射到INT\_B中断号

\*\*\*\*\*pin:1

pci 0000:00:00.0: \*\*\*\*BAR 9: assigned [mem 0x02000000-0x0bffffff pref]

pci 0000:00:00.0: BAR 9: assigned [mem 0x30000000-0x39ffffff pref]

pci 0000:00:00.0: \*\*\*\*BAR 0: assigned [mem 0x00000000-0x007fffff pref]

pci 0000:00:00.0: BAR 0: assigned [mem 0x3a000000-0x3a7fffff pref]

pci 0000:00:00.0: \*\*\*\*BAR 8: assigned [mem 0x00100000-0x001fffff]

pci 0000:00:00.0: BAR 8: assigned [mem 0x3a800000-0x3a8fffff]

pci 0000:00:00.0: \*\*\*\*BAR 1: assigned [mem 0x00000000-0x0000ffff]

pci 0000:00:00.0: BAR 1: assigned [mem 0x3a900000-0x3a90ffff]

pci 0000:01:00.0: \*\*\*\*BAR 1: assigned [mem 0x00000000-0x01ffffff pref]

pci 0000:01:00.0: BAR 1: assigned [mem 0x30000000-0x31ffffff pref]

pci 0000:01:00.0: \*\*\*\*BAR 2: assigned [mem 0x00000000-0x01ffffff pref]

pci 0000:01:00.0: BAR 2: assigned [mem 0x32000000-0x33ffffff pref]

pci 0000:01:00.0: \*\*\*\*BAR 3: assigned [mem 0x00000000-0x01ffffff pref]

pci 0000:01:00.0: BAR 3: assigned [mem 0x34000000-0x35ffffff pref]

pci 0000:01:00.0: \*\*\*\*BAR 4: assigned [mem 0x00000000-0x01ffffff pref]

pci 0000:01:00.0: BAR 4: assigned [mem 0x36000000-0x37ffffff pref]

pci 0000:01:00.0: \*\*\*\*BAR 5: assigned [mem 0x00000000-0x01ffffff pref]

pci 0000:01:00.0: BAR 5: assigned [mem 0x38000000-0x39ffffff pref]

pci 0000:01:00.0: \*\*\*\*BAR 0: assigned [mem 0x00000000-0x00000fff]

pci 0000:01:00.0: BAR 0: assigned [mem 0x3a800000-0x3a800fff]

pci 0000:00:00.0: PCI bridge to [bus 01]

pci 0000:00:00.0: bridge window [mem 0x3a800000-0x3a8fffff]

pci 0000:00:00.0: bridge window [mem 0x30000000-0x39ffffff pref]

# 四、hi3536 PCIE驱动

## 1、涉及驱动

底层驱动：hi35xx\_dev\_host.ko pcit\_dma\_host.ko

PCIE消息驱动：mcc\_drv\_host.ko mcc\_usrdev\_host.ko

PCIV模块驱动：hi3536\_pciv.ko hi3536\_pciv\_fmw.ko（业务相关驱动，无源码不分析）

加载uboot需要的驱动：boot\_device.ko

## 2、boot\_device.ko

涉及文件：hios\_boot\_usrdev.c和ddr\_reg\_init.c。加在boot kernel fs等到所有的EP设备。

以miscdevice的设备方式注册驱动，ioctl支持获取所有设备、传输数据、启动设备、复位设备。

由booter应用程序控制加载。

## 3、hi35xx\_dev\_host.ko

涉及文件：hi35xx\_dev\_host.c和mcc\_proc\_msg.c。注册PCI设备驱动，初始化设备，使能设备，做好映射。

### （1）驱动初始化

static int \_\_init hi35xx\_pcie\_module\_init(void)

{

….

g\_local\_handler = kmalloc(sizeof(struct pci\_operation), GFP\_KERNEL); //所有pcie操作相关的结构体。

ret = hidev\_host\_init(g\_local\_handler); //初始化读写配置结构体

…

ret = pci\_arch\_init(g\_local\_handler); //初始化结构体剩余部分，大部分是由该出初始化

….

ret = pci\_register\_driver(&pci\_hidev\_driver); //注册pcie驱动

…

#ifdef MCC\_PROC\_ENABLE

ret = mcc\_proc\_init();

…

mcc\_create\_proc("mcc\_info", NULL, NULL); //创建proc下面相关目录和文件，pci\_mcc/mcc\_info

ret = mcc\_init\_sysctl()—> register\_sysctl\_table()

//创建sys/dev/pci/proc\_message\_enable

…

#endif

return 0;

}

### （2）注册pcie驱动pci\_register\_driver(&pci\_hidev\_driver)

pci\_register\_driver -> \_\_pci\_register\_driver(drv->driver.bus = &pci\_bus\_type 总线为pci\_bus) -> driver\_register —> bus\_add\_driver —> driver\_attach —> \_\_driver\_attach —>driver\_match\_device(说明如下) -> driver\_probe\_device -> really\_probe —> dev->bus->probe（pci\_bus\_type.pci\_device\_probe）

pcie 的driver\_match\_device调用pci\_bus\_type的pci\_bus\_match —> pci\_match\_device —> pci\_match\_one\_device(&dynid->id,dev)使用pci\_device\_id进行匹配这里为hi35xx\_pci\_tbl。

pci总线pci\_device\_probe调用 \_\_pci\_device\_probe —> pci\_call\_probe —> local\_pci\_probe —> ddi->drv->probe（pci\_hidev\_driver. pci\_hidev\_probe）

### （3）pci\_hidev\_probe

pci\_enable\_device：激活PCI设备，在驱动程序可以访问PCI设备的任何设备资源之前(I/O区域或者中断)，驱动程序必须调用该函数；主要就是把PCI配置空间的Command域的0位和1 位置成了1，从而达到了开启设备的目的。设置PCI设备使用的irq号。

pci\_request\_regions：通知内核该设备对应的IO端口和内存资源已经使用，其他的PCI设备不要再使用这个区域

hi\_dev->bar0 = pci\_resource\_start(pdev, 0); //0x3a800000 在扫描设备pci\_read\_bases中读取pci设备bar返回的是存储器域的物理地址，更新之后PCIe设备的BARx基地址空间属于PCI总线域。

hi\_dev->bar1 = pci\_resource\_start(pdev, 1); //0x30000000

hi\_dev->bar2 = pci\_resource\_start(pdev, 2); //0x32000000

hi\_dev->pci\_bar0\_virt = (unsigned int)ioremap\_nocache(hi\_dev->bar0, 0x800000); //8M

hi\_dev->pci\_bar1\_virt = (unsigned int)ioremap\_nocache(hi\_dev->bar1, 0x10000); //64K

/\* [pci\_hidev\_probe, 177]: hidev# [slot:0x1] [controller:0x0] [bar0:0x3a800000] [bar1:0x30000000] [bar2:0x32000000]. \*/

g\_local\_handler->init\_hidev(hi\_dev):

static int init\_hidev(struct hi35xx\_dev \*hi\_dev)

{

s\_pcie\_opt->move\_pf\_window\_in(hi\_dev,

s\_pcie\_opt->sysctl\_reg\_phys,

0x1000,

1);

s\_pcie\_opt->move\_pf\_window\_in(hi\_dev,

s\_pcie\_opt->misc\_reg\_phys,

0x1000,

3);

s\_pcie\_opt->pci\_config\_read(hi\_dev,

CFG\_BAR3\_REG,

&hi\_dev->bar3); //0x34000008

hi\_dev->pci\_bar3\_virt = (unsigned int)ioremap(hi\_dev->bar3, 0x2000); //8K

return 0;

}

move\_pf\_window\_in和move\_pf\_window\_out配置PCI设备（EP端）的inbound和outbound.

例子：

g\_local\_handler->move\_pf\_window\_in(hi\_dev, target\_addr, 0xffff, 0);

copy\_from\_user((void \*)hi\_dev->pci\_bar0\_virt, (void \*)source\_addr, MAX\_BAR0\_SPACE\_SIZE)

设置EP端的inbound将要拷贝的目的地址映射到BAR0，RC端将源数据拷贝到pci\_bar0\_virt地址就会拷贝到EP目的地址。



## 4、pcit\_dma\_host.ko

涉及文件：dma\_trans.c

Hi3536 PCIE控制器内含DMA控制器，DMA控制器包含两个DMA通道（一个读和一个写），用于大数据量存储器读写事务，实现全双工运行。

DMA写：将一块数据从本地内存空间搬运至对端设备的内存空间

DMA读：将一块数据从对端设备的内存空间搬移至本地内存空间。

### （1）驱动初始化

static int \_\_init pci\_dma\_transfer\_init(void)

{

…

if (dma\_sys\_init()) { //初始化dma.write和dma.read两个通道，每个都有消息和数据两个任务队列，支持128个

PCIT\_PRINT(PCIT\_ERR, "PCI DMA sys init failed!");

return -1;

}

ret = dma\_arch\_init(); //初始化dma中断并开启中断，中断号为86,处理函数host\_dma\_irq\_handler

…

ret = misc\_register(&pci\_dma\_miscdev); //注册为miscdevice类型

…

}

### （2）中断处理函数host\_dma\_irq\_handler

irqreturn\_t host\_dma\_irq\_handler(int irq, void \*dev\_id)

{

\_\_do\_pcit\_dma\_trans(&s\_pcit\_dma.write, PCI\_DMA\_WRITE);

\_\_do\_pcit\_dma\_trans(&s\_pcit\_dma.read, PCI\_DMA\_READ);

return IRQ\_HANDLED;

}

同时处理读写通道。

\_\_do\_pcit\_dma\_trans：先检查是否中止（如果中止，重试3次）；正常传输完成，调用任务钩子函数finish（任务创建者自己定义），从队列中删除该任务；开启新的任务（消息任务高于数据）。

### （3）misc设备pci\_dma\_miscdev

其他函数为空，只有dma\_trans\_ioctl实现读写任务创建。pcit\_create\_task —> \_\_pcit\_create\_task -> \_\_do\_create\_task: 将任务添加到队列（如果队列空则马上启动dma，如果有任务则添加到任务末尾），DMA任务完成或者中止会产生中断。

## 5、mcc\_drv\_host.ko

涉及文件：pci\_proto\_init\_v2.c和pci\_vdd\_ops.c

### （1）pci\_proto\_init\_v2.c

初始化对int\_x中断请求，中断处理函数message\_irq\_handler：

static int \_\_init msg\_com\_init(void)

{

…

handler = message\_irq\_handler;

if (g\_local\_handler->request\_msg\_irq(handler)) {

mcc\_trace(MCC\_ERR, "Request msg irq failed!");

return -1;

}

init\_timer(&msg\_trigger\_timer);

…

}

message\_irq\_handler：处理recv\_irq\_mem和recv\_thread\_mem的消息，两个函数处理机制差不多，循环处理消息buf的消息，直到消息buf为空为止。

pos = head->src << MSG\_PORT\_NEEDBITS | head->port

handle = hisi\_handle\_table[pos]

根据target\_id 和port得到相应消息类型的队列，将消息放入该队列（各种消息类型队列创建在hios\_mcc\_usrdev通过ioctl），调用有mcc\_usrdev\_host.o定义的消息处理钩子函数vdd\_notifier\_recvfrom->hios\_mcc\_notifier\_recv\_pci。

g\_local\_handler->request\_msg\_irq -> request\_message\_irq：通过INT\_A INT\_B INIT\_C INIT\_D PCIE标准中断（无实际硬件端口），EP通过设置这几个中断使RC产生中断读取消息。（注：目前DSP上，RC无法通过设置这几个来使EP产生中断）。

### （2）pci\_vdd\_ops.c

封装一层drv消息操作接口（open, sendto, setopt等）,供mcc\_usrdev\_host.ko驱动模块使用。调用流程：hios\_mcc\_usrdev.c -> hios\_mcc.c -> pci\_vdd\_ops.c。

## 6、mcc\_usrdev\_host.ko

涉及文件：hios\_mcc\_usrdev.c和hios\_mcc.c

为用户态提供消息相关操作的驱动模块。

### （1）hios\_mcc\_usrdev.c

注册一个miscdevice设备，该驱动与用户态程序进行消息交互。/dev/mcc\_userdev该设备每个消息类型都会打开一次，根据target\_id和port创建消息句柄。

主要接口说明如下：

hios\_mcc\_notifier\_recv\_pci：

消息结构钩子函数（vdd\_notifier\_recvfrom在中断中处理），将消息放入队列供用户读取,不同消息类型都有一个自己的队列（如通用命令消息，码流读消息，码流写消息等）

usrdev\_setopt\_recv\_pci:

获取处理句柄、设置接收消息属性，通过ioctl控制

hi\_mcc\_userdev\_release:

释放句柄，清空消息列表

hi\_mcc\_userdev\_read：

读取各自队列中的消息（中断接收见pci\_init\_proto\_v2.c）

hi\_mcc\_userdev\_write：

发送消息，放入共享内存，调用trigger\_msg\_irq触发pci从设备中断（通过bar0向DSP寄存器设置产生中断）。

trigger\_msg\_irq:INT\_A/B/C/D中断方式如果是RC端则通过设置EP端的寄存器触发中断通知EP端，如果是EP端则通过触发本地PCIE中断通知RC

hi\_mcc\_userdev\_ioctl：

HI\_MCC\_IOC\_ATTR\_INIT：初始化消息属性

HI\_MCC\_IOC\_CHECK：根据target\_id同从设备进行握手（host有step0, step1, step2; slave有step0, step1），获取从设备的共享内存地址（主要用于消息收发），并进行初始化（hisi\_shared\_mem\_init）。

HI\_MCC\_IOC\_CONNECT：hios\_mcc\_open根据target\_id和消息port创建句柄并存放在hisi\_handle\_table[pos] = handle这个全局表格（中断接到消息时根据target\_id和port计算出pos获取到对应消息类型的句柄）；file->private\_data = (void \*)handle将消息句柄存放在打开该文件的私有数据，方便读写等地方使用。

hisi\_shared\_mem\_init：对共享内存进行划分

假定总范围：0x60000000 ~ 0x600C0000 (768 \* 1024)

send\_msg:

0x60000000+4\*32(first\_half\_base) ~ 0x60060000-6\*32(first\_half\_end) (384\*1024)

recv\_msg:

0x60060000+4\*32(second\_half\_base) ~ 0x600c0000-10\*32(second\_half\_end) (384\*1024)

以下以send为类（recv类似）：

\_init\_mem send\_irqmem:

0x60000000+4\*32 ~ 0x60020000-4\*32 (128\*1024)

\_init\_mem send\_threadmem:

0x60020000+4\*32 ~ 0x6006000-6\*32

以下以send\_irqmem为例（其他类似）：

\_init\_meminfo：

p->base\_addr = 0x60000000+4\*32

p->end\_addr = 0x60020000-4\*32

p->rp\_addr=0x60000000+4\*32 - 64

p->wp\_addr=0x60000000+4\*32 - 32



### （2）hios\_mcc.c

对pci\_vdd\_ops.c接口进行一次封装，供hios\_mcc\_usrdev.c使用。接口有：

hios\_mcc\_close（）

hios\_mcc\_open（）

hios\_mcc\_sendto（）

hios\_mcc\_sendto\_user（）

hios\_mcc\_getopt（）

hios\_mcc\_setopt（）

hios\_mcc\_getlocalid（）

hios\_mcc\_getremoteids（）

hios\_mcc\_check\_remote（）