Linux 学习笔记

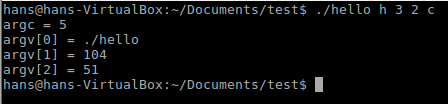
Created by Hans Hu at 2017/07/17

## 1. 系统函数使用说明

### 1.1 main函数原型：

int main(int *argc*, char \**argv[]*)

其中，argc为参数个数，argv为参数，首个字符串argv[0]标识函数名本身，参数个数不指定。示例如下图：



程序编写的只显示到参数argv[2]。参数个数为总的输入参数个数。从编译过程来看，除了第一个标识函数名本身的参数argv[0]为字符串之外，其余参数都是int类型参数，并将输入的参数按char型字符处理。如果某参数输入为字符串，只取第一个字符作为输入。

另外，输入的参数不是必须使用，但是如果有使用输入参数，则对应参数必须有输入，否则执行时将报错。

### 1.2 open函数原型：

int open(const char \**prthname*, int *flags*,…)

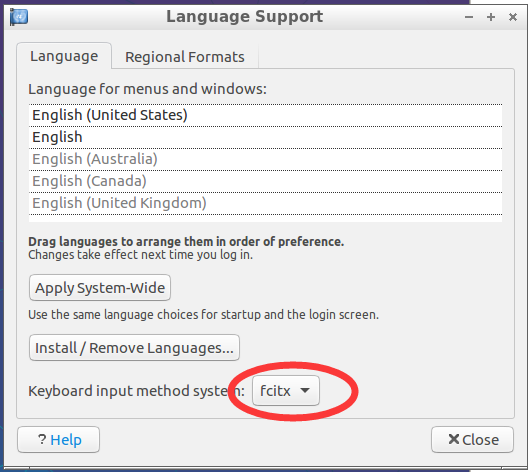
pathname为文件路径及文件名，flags为打开模式，打开模式一般为只读、只写、读写三者之一。返回值为int类型的值，文件打开成功返回一个正值，代表操作编号，类似句柄。文件打开错误则返回-1。

特别注意的是返回值fd，fd的值与文件的状态无关。在同一个进程下，fd的值按照顺序依次递增。不用进程下的fd之间无关，即使是打开同一个文件。可以理解成进程中的一个局部变量。

文件能否被打开多次，以及能否进行双工操作，由驱动函数完成，与系统函数无关。

## 2. Linux发行版lubuntu环境下中文输入法设置

1、首先设置系统preference下的language support下的keyboard input method system为fcitx如下图所示：



2、链接及内容如下

<http://www.lirui.name/post/251.html>

Lubuntu下面这个问题也算是老生常谈了。一直在更新，一直未修正。

简单说，如果运行了下面这条命令输入法就正常了的话：

killall fcitx-qimpanel

那么：

sudo apt-get remove fcitx-ui-qimpanel

问题解决！！

方法二[推荐]：

在输入法图标上右键，配置当前输入法。把Kimpanel前面的勾去掉。



3、链接及内容如下

安装第三方源并安装compton

sudo apt-add-repository ppa:richardgv/compton

sudo apt-get update && sudo apt-get install compton

配置compton

编辑文件 ~/.compton.conf，输入如下内容

backend = "glx";

paint-on-overlay = true;

glx-no-stencil = true;

glx-no-rebind-pixmap = true;

vsync = "opengl-swc";

# These are important. The first one enables the opengl backend. The last one

# is the vsync method. Depending on the driver you might need to use a

# different method.

# The other options are smaller performance tweaks that work well in most

# cases.

# You can find the rest of the options here:

# https://github.com/chjj/compton/wiki/perf-guide, and here:

# https://github.com/chjj/compton/wiki/vsync-guide

# Shadow

shadow = false;          # Enabled client-side shadows on windows.

no-dock-shadow = true;      # Avoid drawing shadows on dock/panel windows.

no-dnd-shadow = true;       # Don't draw shadows on DND windows.

clear-shadow = true;        # Zero the part of the shadow's mask behind the window (experimental).

shadow-radius = 7;      # The blur radius for shadows. (default 12)

shadow-offset-x = -7;       # The left offset for shadows. (default -15)

shadow-offset-y = -7;       # The top offset for shadows. (default -15)

shadow-exclude = [

    "! name~=''",

    "n:e:Notification",

    "n:e:Plank",

    "n:e:Docky",

    "g:e:Synapse",

    "g:e:Kupfer",

    "g:e:Conky",

    "n:w:\*Firefox\*",

    "n:w:\*Chrome\*",

    "n:w:\*Chromium\*",

    "class\_g ?= 'Notify-osd'",

    "class\_g ?= 'Cairo-dock'",

    "class\_g ?= 'Xfce4-notifyd'",

    "class\_g ?= 'Xfce4-power-manager'"

];

# The shadow exclude options are helpful if you have shadows enabled. Due to

# the way compton draws its shadows, certain applications will have visual

# glitches

# (most applications are fine, only apps that do weird things with xshapes or

# argb are affected).

# This list includes all the affected apps I found in my testing. The "!

# name~=''" part excludes shadows on any "Unknown" windows, this prevents a

# visual glitch with the XFWM alt tab switcher.

# Fading

fading = true; # Fade windows during opacity changes.

fade-delta = 4; # The time between steps in a fade in milliseconds. (default 10).

fade-in-step = 0.03; # Opacity change between steps while fading in. (default 0.028).

fade-out-step = 0.03; # Opacity change between steps while fading out. (default 0.03).

#no-fading-openclose = true; # Fade windows in/out when opening/closing

detect-client-opacity = true; # This prevents opacity being ignored for some apps. For example without this enabled my xfce4-notifyd is 100% opacity no matter what.

# Window type settings

wintypes:

{

  tooltip = { fade = true; shadow = false; };

};

重点是其中 shadow 一项的值设置为 false 。

自启动

最后，修改i3配置文件 ~/.i3/config，在最后加入下面的代码，实现自动启动compton和搜狗输入法

# auto start commands

exec --no-startup-id fcitx -r

exec --no-startup-id fcitx-qimpanel

exec --no-startup-id compton -b

安装完成了，系统也顺便带了半透明和淡入淡出效果。

## 3. 字符设备驱动

### 3.1 基本驱动

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/slab.h>

#include <linux/uaccess.h>

#include <linux/device.h>

#include "demo.h"

#ifndef MEMDEV\_NR\_DEVS

#define MEMDEV\_NR\_DEVS 2 /\*设备数\*/

#endif

#ifndef MEMDEV\_SIZE

#define MEMDEV\_SIZE 4096

#endif

//static struct class \*drv\_class;

//static struct device \*drv\_device;

/\*mem设备描述结构体\*/

struct mem\_dev

{

char \*data;

unsigned long size;

};

/\* seek文件定位函数 \*/

static loff\_t mem\_llseek(struct file \*filp, loff\_t offset, int orig)

{

loff\_t newpos;

switch(orig) {

case 0: /\* SEEK\_SET \*/

newpos = offset;

break;

case 1: /\* SEEK\_CUR \*/

newpos = filp->f\_pos + offset;

break;

case 2: /\* SEEK\_END \*/

newpos = MEMDEV\_SIZE -1 + offset;

break;

default: /\* can't happen \*/

return -EINVAL;

}

if ((newpos<0) || (newpos>MEMDEV\_SIZE))

return -EINVAL;

filp->f\_pos = newpos;

return newpos;

}

struct mem\_dev \*mem\_devp; /\*设备结构体指针\*/

static struct cdev cdev[2];

static dev\_t ndev;

/\*写函数\*/

static ssize\_t mem\_write(struct file \*filp, const char \_\_user \*buf, size\_t size, loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct mem\_dev \*dev = filp->private\_data; /\*获得设备结构体指针\*/

/\*分析和获取有效的写长度\*/

if (p >= MEMDEV\_SIZE)

return 0;

if (count > MEMDEV\_SIZE - p)

count = MEMDEV\_SIZE - p;

/\*从用户空间写入数据\*/

if (copy\_from\_user(dev->data + p, buf, count))

ret = - EFAULT;

else

{

\*ppos += count;

ret = count;

printk("written %d bytes(s) from %zu \n",count,p);

}

return ret;

}

static int mem\_open(struct inode\* nd, struct file\* filp)

{

int major;

int minor;

major = MAJOR(nd->i\_rdev);

minor = MINOR(nd->i\_rdev);

printk("chr\_open, major = %d, minor = %d\n", major, minor);

filp->private\_data = &mem\_devp[minor];

return 0;

}

/\*读函数\*/

static ssize\_t mem\_read(struct file \*filp, char \_\_user \*buf, size\_t size,loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct mem\_dev \*dev = filp->private\_data; /\*获得设备结构体指针\*/

/\*分析获取有效的读长度\*/

if (p >= MEMDEV\_SIZE)/\*当前位置不再文件范围内，报错\*/

return count ? - ENXIO: 0;

if (count > MEMDEV\_SIZE - p)/\*数量超出文件范围，修改读取的个数\*/

count = MEMDEV\_SIZE - p;

/\*复制数据\*/

if (copy\_to\_user(buf, (void\*)(dev->data + p), count))

ret = - EFAULT;/\*发生拷贝错误\*/

else

{

\*ppos += count;/\*修改当前位置指针\*/

ret = count;/\*返回读取的数量\*/

printk(KERN\_INFO "read %d bytes(s) from %d\n", count, p);

}

return ret;

}

/\*文件释放函数\*/

static int mem\_release(struct inode \*inode, struct file \*filp)

{

return 0;

}

struct file\_operations mem\_fops = {

.owner = THIS\_MODULE,

.llseek = mem\_llseek,

.read = mem\_read,

.write = mem\_write,

.open = mem\_open,

.release = mem\_release

};

/\*init func\*/

static int demo\_init(void)

{

int ret,i,devno;

ret = alloc\_chrdev\_region(&ndev, 0, MEMDEV\_NR\_DEVS, "cdev");/\* 动态分配设备号 \*/

if(ret < 0 )

{

return ret;

}

//printk("demo\_init(): major = %d, minor = %d\n", MAJOR(ndev), MINOR(ndev));

/\* 为设备描述结构分配内存\*/

mem\_devp = kmalloc(MEMDEV\_NR\_DEVS \* sizeof(struct mem\_dev), GFP\_KERNEL);//目前为止我们始终用GFP\_KERNEL

if (!mem\_devp) /\*申请失败\*/

{

ret = - ENOMEM;

goto fail\_malloc;

}

memset(mem\_devp, 0, MEMDEV\_NR\_DEVS \*sizeof(struct mem\_dev));

for (i=0; i < MEMDEV\_NR\_DEVS; i++)

{

/\*为设备分配内存\*/

mem\_devp[i].size = MEMDEV\_SIZE;

mem\_devp[i].data = kmalloc(MEMDEV\_SIZE, GFP\_KERNEL);//分配出来的地址存在此

memset(mem\_devp[i].data, 0, MEMDEV\_SIZE);

/\*初始化并注册cdev\*/

devno = MKDEV(MAJOR(ndev),i);

cdev\_init(&cdev[i], &mem\_fops);/\* 使cdev与char\_fops联系起l来 \*/

cdev[i].owner = THIS\_MODULE;

//cdev.ops = &mem\_fops;

ret = cdev\_add(&cdev[i], devno, 1); /\* 注册字符设备 \*/

if(ret < 0)

{

printk(KERN\_NOTICE "Error %d adding mem %d", ret, i);

return ret;

}

printk("demo\_init(): major = %d, minor = %d\n", MAJOR(ndev), MINOR(ndev));

}

// drv\_class = class\_create(THIS\_MODULE, "class\_create"); //创建类

// drv\_device = device\_create(drv\_class, NULL, ndev, NULL, "mem\_dev"); //在类下创建设备，供用户空间的应用程序使用

return 0;

fail\_malloc:

unregister\_chrdev\_region(ndev, 1);

return ret;

}

/\*exit func\*/

static void demo\_exit(void)

{

printk("demo\_exit process!\n");

cdev\_del(&cdev); /\*注销设备\*/

kfree(mem\_devp); /\*释放设备结构体内存\*/

unregister\_chrdev\_region(ndev, MEMDEV\_NR\_DEVS); /\*释放设备号\*/

// device\_unregister(drv\_device); //卸载类下的设备

// class\_destroy(drv\_class); //卸载类

}

module\_init(demo\_init);

module\_exit(demo\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("xxiomg@163.com");

MODULE\_DESCRIPTION("A simple char device example!");

### 3.2 main函数

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

int main()

{

int fd = NULL;

char Buf[45];

/\*初始化Buf\*/

strcpy(Buf,"Mem is char dev0!");

printf("BUF: %s\n",Buf);

/\*打开设备文件\*/

fd = open("/dev/char\_dev0",O\_RDWR);

if (fd == NULL)

{

printf("Open char\_dev0 Error!\n");

return -1;

}

printf("fd number is = %d\n",fd);

/\*写入设备\*/

write(fd, Buf, 40);

/\*重新定位文件位置（思考没有该指令，会有何后果)\*/

lseek(fd,0,SEEK\_SET);

/\*清除Buf\*/

strcpy(Buf,"Buf is NULL!");

printf("BUF: %s\n",Buf);

while(1)

{

/\*读出设备\*/

read(fd,Buf, 40);

/\*检测结果\*/

printf("BUF: %s\n",Buf);

lseek(fd,0,SEEK\_SET);

sleep(1);

}

close(fd);

return 0;

}

### 3.3 Makefile文件

ifneq ($(KERNELRELEASE),)

obj-m := demo.o

else

KVERS:= $(shell uname -r)

KDIR := /lib/modules/$(KVERS)/build

default:

$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:

rm -f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers

endif

### 3.4 相关shell指令

1）生成模块：make

2）装在模块：insmod demo.ko

3）卸载模块：rmmod demo

4）生成节点：mknod /dev/char\_dev0 c 246 0

5）生成可执行文件：gcc main.c –o app

6）执行文件 ：./app

### 3.5 相关说明

1）目前节点的添加方式是手动添加，主要是对应相同主设备的设备有两个，目前还未掌握自动添加两个设备节点的方法。

2）识别打开节点的方法是在文件打开函数中读取节点的次设备号，再将操作指针切换到对应的数据区。

3）引申问题，由于不同的设备采用相同的驱动，那么需要对设备资源进行保护访问时也应具有针对性。因此，对于读写等驱动程序，也要根据不同设备设置保护，也就是都要获取到此设备号才行，对此，可以将在打开时获取到的此设备号保存下来，放到设备结构中。

4）Linux中使用较少的全局变量，而是采用全局指针，好处是对于要使用的数据结构，动态进行申请即可，使用完毕之后释放。这样驱动独占的内存资源就比较少，有利于资源的优化利用。

## 4、ALSA驱动实例分析

本文选取的是内核版本为4.4.x的Linux自带的ALSA源码进行分析

### 4.1 驱动入口分析

#### 4.1.1 Driver基本入口

Alsa针对具体的设备，驱动模块的入口为module\_###\_driver(\*\*\*\_driver);其中###一般为I2C、USB、PCI、Platform等，\*\*\*为具体设备相关名。

1）module\_###\_driver来自系统路径\include\linux\###.h对应文件。以### = PCI为例，对应定义在linux/pci.h文件中:

#define module\_pci\_driver(\_\_pci\_driver) \

module\_driver(\_\_pci\_driver, pci\_register\_driver, \

pci\_unregister\_driver)

而module\_driver的定义来自\include\linux\device.h，定义内容为

#define module\_driver(\_\_driver, \_\_register, \_\_unregister, ...) \

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

{ \

return \_\_register(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \

} \

module\_init(\_\_driver##\_init); \

static void \_\_exit \_\_driver##\_exit(void) \

{ \

\_\_unregister(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \

} \

module\_exit(\_\_driver##\_exit);

由以上可以看出，调用module\_###\_driver同样完成了的设备模块的构造。

2）\*\*\*\_driver为具体设备的驱动函数入口结构体。类型为###\_driver,以azx\_driver为例，其定义为：

/\* pci\_driver definition \*/

static struct pci\_driver azx\_driver = {

.name = KBUILD\_MODNAME,

.id\_table = azx\_ids,

.probe = azx\_probe,

.remove = azx\_remove,

.shutdown = azx\_shutdown,

.driver = {

.pm = AZX\_PM\_OPS,

},

};

该结构体中分别定义了注册函数azx\_probe、azx\_remove、azx\_shutdown和pm，其中的pm指向和power manage 相关的resume和suspend相关的驱动函数。

#### 4.1.2 HDAC的两个入口文件

HDAC的驱动结构分为controller和codec两个主要部分，controller驱动的入口是hda\_intel.c文件，codec驱动的入口是hda\_generic.c或者patch\_\*.c文件。hda\_generic.c是codec驱动的核心部分，系统首先从patch\_\*.c进行codec匹配，匹配到的话就选择对应的入口，如果匹配不到就选择hda\_generic.c中的通用入口。

struct device\_driver {

const char \*name;

struct bus\_type \*bus;

struct module \*owner;

const char \*mod\_name; /\* used for built-in modules \*/

bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/

enum probe\_type probe\_type;

const struct of\_device\_id \*of\_match\_table;

const struct acpi\_device\_id \*acpi\_match\_table;

int (\*probe) (struct device \*dev);

int (\*remove) (struct device \*dev);

void (\*shutdown) (struct device \*dev);

int (\*suspend) (struct device \*dev, pm\_message\_t state);

int (\*resume) (struct device \*dev);

const struct attribute\_group \*\*groups;

const struct dev\_pm\_ops \*pm;

struct driver\_private \*p;

};

1. hda\_intel.c的入口

module\_pci\_driver(azx\_driver);

入口的注册过程见4.1.1节

module\_pci\_driver是系统定义的标准入口之一，其入口部分的结构体类型为pci\_driver，成员如下：

struct pci\_driver {

struct list\_head node;

const char \*name;

const struct pci\_device\_id \*id\_table; /\* must be non-NULL for probe to be called \*/

int (\*probe) (struct pci\_dev \*dev, const struct pci\_device\_id \*id); /\* New device inserted \*/

void (\*remove) (struct pci\_dev \*dev); /\* Device removed (NULL if not a hot-plug capable driver) \*/

int (\*suspend) (struct pci\_dev \*dev, pm\_message\_t state); /\* Device suspended \*/

int (\*suspend\_late) (struct pci\_dev \*dev, pm\_message\_t state);

int (\*resume\_early) (struct pci\_dev \*dev);

int (\*resume) (struct pci\_dev \*dev); /\* Device woken up \*/

void (\*shutdown) (struct pci\_dev \*dev);

int (\*sriov\_configure) (struct pci\_dev \*dev, int num\_vfs); /\* PF pdev \*/

const struct pci\_error\_handlers \*err\_handler;

struct device\_driver driver;

struct pci\_dynids dynids;

};

pci设备的插入和移除等函数是绑定在结构体的.probe.remove .shutdown 等成员上。以这些成员中进行入口分析即可。

1. patch\_via.c的入口

module\_hda\_codec\_driver(via\_driver);

其中：

#define module\_hda\_codec\_driver(drv) \

module\_driver(drv, hda\_codec\_driver\_register, \

hda\_codec\_driver\_unregister)

这样最终还是调用了module\_driver函数实现了模块加载。

不同于module\_pci\_driver，module\_hda\_codec\_driver是声卡专用的注册函数，其入口部分的结构体类型为hda\_codec\_driver，成员如下：

struct hdac\_driver {

struct device\_driver driver;

int type;

const struct hda\_device\_id \*id\_table;

int (\*match)(struct hdac\_device \*dev, struct hdac\_driver \*drv);

void (\*unsol\_event)(struct hdac\_device \*dev, unsigned int event);

};

下面分别以这两个驱动的结构进行分析：

### 4.2 HDA Controller驱动

#### 4.2.1 probe函数

该函数为驱动的入口函数，下面选取hda\_intel.c为例进行分析。probe函数原型如下：

static int azx\_probe(struct pci\_dev \*pci, const struct pci\_device\_id \*pci\_id)

{

static int dev;

struct snd\_card \*card;

struct hda\_intel \*hda;

struct azx \*chip;

bool schedule\_probe;

int err;

if (dev >= SNDRV\_CARDS)

return -ENODEV;

if (!enable[dev]) {

dev++;

return -ENOENT;

}

err = snd\_card\_new(&pci->dev, index[dev], id[dev], THIS\_MODULE, 0, &card);

if (err < 0) {

dev\_err(&pci->dev, "Error creating card!\n");

return err;

}

err = azx\_create(card, pci, dev, pci\_id->driver\_data, &chip);

if (err < 0)

goto out\_free;

card->private\_data = chip;

hda = container\_of(chip, struct hda\_intel, chip);

pci\_set\_drvdata(pci, card);//card整个结构作为pci设备的驱动数据

err = register\_vga\_switcheroo(chip);//貌似注册显卡相关的，VIA未用到，跳过

if (err < 0) {

dev\_err(card->dev, "Error registering VGA-switcheroo client\n");

goto out\_free;

}

if (check\_hdmi\_disabled(pci)) {//如果编译了hdmi\_codec模块，但目前并未编译，跳过

dev\_info(card->dev, "VGA controller is disabled\n");

dev\_info(card->dev, "Delaying initialization\n");

chip->disabled = true;

}

schedule\_probe = !chip->disabled; // schedule\_probe = 1

#ifdef CONFIG\_SND\_HDA\_PATCH\_LOADER //编译内核时配置使能的。允许加载补丁来改写BIOS的设置，patch内容作为外部参数传入

if (patch[dev] && \*patch[dev]) {//目前未导入补丁，暂不分析，跳过

dev\_info(card->dev, "Applying patch firmware '%s'\n",

patch[dev]);

err = request\_firmware\_nowait(THIS\_MODULE, true, patch[dev], &pci->dev, GFP\_KERNEL, card, azx\_firmware\_cb);

if (err < 0)

goto out\_free;

schedule\_probe = false; /\* continued in azx\_firmware\_cb() \*/

}

#endif /\* CONFIG\_SND\_HDA\_PATCH\_LOADER \*/

/\*

是否应用补丁，这里应该就是针对第三方，例如via的补丁文件，，

\*/

#ifndef CONFIG\_SND\_HDA\_I915

if (chip->driver\_caps & AZX\_DCAPS\_I915\_POWERWELL)

dev\_err(card->dev, "Haswell must build in CONFIG\_SND\_HDA\_I915\n"); //该定义未定义时，给个报错提示

#endif

if (schedule\_probe) schedule\_work(&hda->probe\_work);

/\*

启用工作队列中的hda->probe\_work，执行azx\_probe\_continue函数，完成进一步初始化。若CONFIG\_SND\_HDA\_PATCH\_LOADER被定义，同样会执行。

\*/

dev++;

if (chip->disabled) complete\_all(&hda->probe\_wait);

/\*

完成量，用于同步

\*/

return 0;

out\_free:

snd\_card\_free(card);

return err;

}

#### 4.2.2 snd\_card\_new()函数

函数源码如下，函数的主要作用是在系统中建立声卡结构，可以分为如图中所示的各区块：

/\*\*

\* snd\_card\_new - create and initialize a soundcard structure

\* @parent: the parent device object

\* @idx: card index (address) [0 ... (SNDRV\_CARDS-1)]

\* @xid: card identification (ASCII string)

\* @module: top level module for locking

\* @extra\_size: allocate this extra size after the main soundcard structure

\* @card\_ret: the pointer to store the created card instance

\*

\* Creates and initializes a soundcard structure.

\*

\* The function allocates snd\_card instance via kzalloc with the given

\* space for the driver to use freely. The allocated struct is stored

\* in the given card\_ret pointer.

\*

\* Return: Zero if successful or a negative error code.

\*/

int snd\_card\_new(struct device \*parent, int idx, const char \*xid,

struct module \*module, int extra\_size,

struct snd\_card \*\*card\_ret)

{

struct snd\_card \*card;

int err;

if (snd\_BUG\_ON(!card\_ret))

return -EINVAL;

\*card\_ret = NULL;

if (extra\_size < 0)

extra\_size = 0;

card = kzalloc(sizeof(\*card) + extra\_size, GFP\_KERNEL);

if (!card)

return -ENOMEM;

if (extra\_size > 0)

card->private\_data = (char \*)card + sizeof(struct snd\_card);

/\*\*

该部分为声卡结构分配空间，包括card 和其私有数据extra\_size。其中私有数据的分配方式有两种一种是在该函数中通过extra\_size进行分配，另一种是在该函数传入的extra\_size=0，然后在该函数之外的其他某个地方进行分配。

\*\*/

if (xid)

strlcpy(card->id, xid, sizeof(card->id));

/\*\*

该部分为声卡ID字符串的拷贝,如果需要的话

\*\*/

err = 0;

mutex\_lock(&snd\_card\_mutex);

if (idx < 0) /\* first check the matching module-name slot \*/

idx = get\_slot\_from\_bitmask(idx, module\_slot\_match, module);

if (idx < 0) /\* if not matched, assign an empty slot \*/

idx = get\_slot\_from\_bitmask(idx, check\_empty\_slot, module);

if (idx < 0)

err = -ENODEV;

else if (idx < snd\_ecards\_limit) {

if (test\_bit(idx, snd\_cards\_lock))

err = -EBUSY; /\* invalid \*/

} else if (idx >= SNDRV\_CARDS)

err = -ENODEV;

if (err < 0) {

mutex\_unlock(&snd\_card\_mutex);

dev\_err(parent, "cannot find the slot for index %d (range 0-%i), error: %d\n",

idx, snd\_ecards\_limit - 1, err);

kfree(card);

return err;

}

set\_bit(idx, snd\_cards\_lock); /\* lock it \*/

if (idx >= snd\_ecards\_limit)

snd\_ecards\_limit = idx + 1; /\* increase the limit \*/

mutex\_unlock(&snd\_card\_mutex);

/\*\*

该部分为声卡索引号的分配，首先检查是否有相匹配的，没有的话就分配一个。最多支持8个声卡

\*\*/

card->dev = parent;

card->number = idx;

card->module = module;

INIT\_LIST\_HEAD(&card->devices);

init\_rwsem(&card->controls\_rwsem);

rwlock\_init(&card->ctl\_files\_rwlock);

mutex\_init(&card->user\_ctl\_lock);

INIT\_LIST\_HEAD(&card->controls);

INIT\_LIST\_HEAD(&card->ctl\_files);

spin\_lock\_init(&card->files\_lock);

INIT\_LIST\_HEAD(&card->files\_list);

#ifdef CONFIG\_PM

mutex\_init(&card->power\_lock);

init\_waitqueue\_head(&card->power\_sleep);

#endif

/\*\*

该部分为声卡的一些必要初始化，包括父设备的绑定，索引号的绑定，内部结构的初始化等等。

\*\*/

device\_initialize(&card->card\_dev);

card->card\_dev.parent = parent;

card->card\_dev.class = sound\_class;

card->card\_dev.release = release\_card\_device;

card->card\_dev.groups = card->dev\_groups;

card->dev\_groups[0] = &card\_dev\_attr\_group;

err = kobject\_set\_name(&card->card\_dev.kobj, "card%d", idx);

if (err < 0)

goto \_\_error;

/\*\*

该部分为创建sysfs下的设备。重点一在于sound\_class，该结构是在/sound/sound\_core.c中的init\_soundcore函数中创建，该函数调用了sound\_devnode。函数内容表明声卡的class将会出现在文件系统的/sys/class/sound/下面，并且，sound\_devnode()也决定了相应的设备节点也将会出现在/dev/snd/下面。

该部分内容出现在旧版的snd\_card\_register()函数中。

\*\*/

/\* the control interface cannot be accessed from the user space until \*/

/\* snd\_cards\_bitmask and snd\_cards are set with snd\_card\_register \*/

err = snd\_ctl\_create(card);

if (err < 0) {

dev\_err(parent, "unable to register control minors\n");

goto \_\_error;

}

/\*\*

该部分为建立控制设备节点，通常是 /dev/snd/controlC0。

根据注释control接口在snd\_card\_register之后，用户空间才可以访问。

\*\*/

err = snd\_info\_card\_create(card);

if (err < 0) {

dev\_err(parent, "unable to create card info\n");

goto \_\_error\_ctl;

}

/\*\*

该部分为建立proc文件中的info节点，通常就是/proc/asound/card0。

\*\*/

\*card\_ret = card;

return 0;

\_\_error\_ctl:

snd\_device\_free\_all(card);

\_\_error:

put\_device(&card->card\_dev);

return err;

}

#### 4.2.3 azx\_create()函数

函数的调用原型：err = azx\_create(card, pci, dev, pci\_id->driver\_data, &chip); 其中：

card为刚刚创建，pci为总线设备，dev设备编号pci\_id->driver\_data特定声卡私有数据，chip空结构指针。chip为目标创建对象。函数的主要作用是建立所有的非codec相关的其余主要数据结构。源代码如下：

static int azx\_create(struct snd\_card \*card, struct pci\_dev \*pci, int dev, unsigned int driver\_caps, struct azx \*\*rchip)

{

static struct snd\_device\_ops ops = {

.dev\_disconnect = azx\_dev\_disconnect,

.dev\_free = azx\_dev\_free,

};

/\*\*

该部分实现snd\_device设备连接的断开和释放操作函数绑定。断开是设置标志位，等待后续执行；释放则是立即执行

\*\*/

struct hda\_intel \*hda;

struct azx \*chip;

int err;

\*rchip = NULL;

err = pci\_enable\_device(pci);

if (err < 0)

return err;

/\*\*

该部分使能该pci设备，在一个pci设备可以被使用之前，必须调用pci\_enable\_device进行激活，该函数会调用底层代码激活PCI设备上的I/O和内存，使之可用。

而pci\_disable\_device所做的事情刚好相反，告诉系统该PCI设备不再使用，同时，禁用相关的一些资源。

\*\*/

hda = kzalloc(sizeof(\*hda), GFP\_KERNEL);

if (!hda) {

pci\_disable\_device(pci);

return -ENOMEM;

}

/\*\*

该部分为声卡控制器描述结构hda\_intel分配空间，分配失败则失能pci设备。同时为具体声卡chip创建了描述结构azx。下面的代码chip = &hda->chip指定。

\*\*/

chip = &hda->chip;

mutex\_init(&chip->open\_mutex);

chip->card = card;//绑定声卡

chip->pci = pci;//绑定pci设备

chip->ops = &pci\_hda\_ops;//绑定控制器操作函数，全局变量

chip->driver\_caps = driver\_caps;//驱动的能力，由参数传入特定声卡的私有数据

chip->driver\_type = driver\_caps & 0xff; ;//驱动的类型，由参数传入特定声卡的私有数据

check\_msi(chip);//检查pci的信息信号中断标志

chip->dev\_index = dev;//声卡索引，由参数传入，一般就一个声卡 0

chip->jackpoll\_ms = jackpoll\_ms;//插孔轮询检测的时间

INIT\_LIST\_HEAD(&chip->pcm\_list);//初始化pcm链表

INIT\_WORK(&hda->irq\_pending\_work, azx\_irq\_pending\_work);//创建中断阻塞事件的工作队列

INIT\_LIST\_HEAD(&hda->list);//声卡链表，用于电源节能管理触发

init\_vga\_switcheroo(chip);//不是太理解，初始化，大概是与显示器链接时,用来判断HDMI是否被VGA接口失能掉，并以此来决定声卡的配置。判断方式通过PCI设备实现，与声卡本身无关。难道说hdmi连接的显示器上面自带codec芯片？？或者其它什么？这个需要了解下硬件结构吗？

init\_completion(&hda->probe\_wait);//初始化用于同步的完成量

/\*\*

该部分为声卡chip内部分成员的初始化

\*\*/

assign\_position\_fix(chip, check\_position\_fix(chip, position\_fix[dev]));

check\_probe\_mask(chip, dev);

/\*\*

尚不清楚……position\_fix：DMA指针的读取方法，外部参数传入

\*\*/

chip->single\_cmd = single\_cmd;//使用外部传输的参数single\_cmd与codec通信，用于debug，不同参数通信方法不一样

azx\_check\_snoop\_available(chip);//检查窥探(snoop)的可用性

if (bdl\_pos\_adj[dev] < 0) {//外部传入BDL position adjustment offset.

switch (chip->driver\_type) {

case AZX\_DRIVER\_ICH:

case AZX\_DRIVER\_PCH:

bdl\_pos\_adj[dev] = 1;

break;

default:

bdl\_pos\_adj[dev] = 32;

break;

}

}

chip->bdl\_pos\_adj = bdl\_pos\_adj;

/\*\*

见注释，具体意义上不清楚，BDL buffer description list，做了一个位置调整

\*\*/

err = azx\_bus\_init(chip, model[dev], &pci\_hda\_io\_ops);

if (err < 0) {

kfree(hda);

pci\_disable\_device(pci);

return err;

}

/\*\*

该部分初始化codec bus的数据结构hda\_bus以及HD-audio bus base driver结构体。主要是在几个数据结构之间建立联系，一些底层功能函数的绑定，例如cmd和response命令，读写字、字节、位命令的实际函数；根据卡的私有数据进行卡的能力配置等；

\*\*/

err = snd\_device\_new(card, SNDRV\_DEV\_LOWLEVEL, chip, &ops);

if (err < 0) {

dev\_err(card->dev, "Error creating device [card]!\n");

azx\_free(chip);

return err;

}

/\*\*

该部分为创建一个SNDRV\_DEV\_LOWLEVEL类型的声卡组件设备，传入的ops参数为设备的连接和释放两个底层函数，所以这个逻辑设备的功能应该就是管理连接的组件。然后绑定card、chip，并将设备链接到card->devices的链表头上

\*\*/

/\* continue probing in work context as may trigger request module \*/

INIT\_WORK(&hda->probe\_work, azx\_probe\_work);

/\*\*

该部分将azx\_probe\_work加入工作队列，等待执行

\*\*/

\*rchip = chip;

/\*返回chip指针\*/

return 0;

}

#### 4.2.4 probe\_continue函数

才会执行。

static int azx\_probe\_continue(struct azx \*chip)

{

struct hda\_intel \*hda = container\_of(chip, struct hda\_intel, chip);

struct hdac\_bus \*bus = azx\_bus(chip);

struct pci\_dev \*pci = chip->pci;

int dev = chip->dev\_index;

int err;

hda->probe\_continued = 1;

/\* Request display power well for the HDA controller or codec. For

\* Haswell/Broadwell, both the display HDA controller and codec need

\* this power. For other platforms, like Baytrail/Braswell, only the

\* display codec needs the power and it can be released after probe.

\*/

if (chip->driver\_caps & AZX\_DCAPS\_I915\_POWERWELL) {

/\* HSW/BDW controllers need this power \*/

if (CONTROLLER\_IN\_GPU(pci))

hda->need\_i915\_power = 1;

err = snd\_hdac\_i915\_init(bus);

if (err < 0) {

/\* if the controller is bound only with HDMI/DP

\* (for HSW and BDW), we need to abort the probe;

\* for other chips, still continue probing as other

\* codecs can be on the same link.

\*/

if (CONTROLLER\_IN\_GPU(pci))

goto out\_free;

else

goto skip\_i915;

}

err = snd\_hdac\_display\_power(bus, true);

if (err < 0) {

dev\_err(chip->card->dev,

"Cannot turn on display power on i915\n");

goto i915\_power\_fail;

}

}

skip\_i915:

err = azx\_first\_init(chip);

if (err < 0)

goto out\_free;

#ifdef CONFIG\_SND\_HDA\_INPUT\_BEEP

chip->beep\_mode = beep\_mode[dev];

#endif

/\* create codec instances \*/

err = azx\_probe\_codecs(chip, azx\_max\_codecs[chip->driver\_type]);

if (err < 0)

goto out\_free;

#ifdef CONFIG\_SND\_HDA\_PATCH\_LOADER

if (chip->fw) {

err = snd\_hda\_load\_patch(&chip->bus, chip->fw->size, chip->fw->data);

/\*加载并解析补丁文件\*/

if (err < 0)

goto out\_free;

#ifndef CONFIG\_PM

release\_firmware(chip->fw); /\* no longer needed \*/

chip->fw = NULL;

#endif

}

#endif

if ((probe\_only[dev] & 1) == 0) {

err = azx\_codec\_configure(chip);

//该函数调用了位于hda\_bind.c中的snd\_hda\_codec\_configure函数，其功能为解析给出的codec树，初始化补丁中的codec实例，注册codec设备到系统

if (err < 0)

goto out\_free;

}

err = snd\_card\_register(chip->card);

//将所有与声卡有关的设备中还未注册的都进行注册，该函数之后就可以通过外部访问调用声卡驱动进行操作了。因此该函数一般放在全部初始化完成之后。

if (err < 0)

goto out\_free;

chip->running = 1;

azx\_add\_card\_list(chip);

snd\_hda\_set\_power\_save(&chip->bus, power\_save \* 1000);

if (azx\_has\_pm\_runtime(chip) || hda->use\_vga\_switcheroo)

pm\_runtime\_put\_noidle(&pci->dev);

out\_free:

if (chip->driver\_caps & AZX\_DCAPS\_I915\_POWERWELL

&& !hda->need\_i915\_power)

snd\_hdac\_display\_power(bus, false);

i915\_power\_fail:

if (err < 0)

hda->init\_failed = 1;

complete\_all(&hda->probe\_wait);

return err;

}

#### 4.2.5 azx\_first\_init函数

static int azx\_first\_init(struct azx \*chip)

{

int dev = chip->dev\_index;

struct pci\_dev \*pci = chip->pci;

struct snd\_card \*card = chip->card;

struct hdac\_bus \*bus = azx\_bus(chip);

int err;

unsigned short gcap;

unsigned int dma\_bits = 64;

…………………………………………………………………………………………………………………………………………

……………………………………………………………省略号~~…………………………………………………………………

…………………………………………………………………………………………………………………………………………

//获得数据流的个数

/\* read number of streams from GCAP register instead of using

\* hardcoded value

\*/

case AZX\_DRIVER\_GENERIC:

default:

chip->playback\_streams = ICH6\_NUM\_PLAYBACK;

chip->capture\_streams = ICH6\_NUM\_CAPTURE;

break;

chip->capture\_index\_offset = 0;

chip->playback\_index\_offset = chip->capture\_streams;

chip->num\_streams = chip->playback\_streams + chip->capture\_streams;

//初始化数据流。为每个流创建一个azx\_dev结构，其中包含hdac\_stream结构体；对该结构进行了部分初始化并将该结构连接到hdac\_bus上的stream\_list链表

/\* initialize streams \*/

err = azx\_init\_streams(chip);

if (err < 0)

return err;

//分配包括数据流在内的相关数据区。参见x.2.1，申请了stream存储及相关dma存储区，并将之初始化到hdac\_bus上

err = azx\_alloc\_stream\_pages(chip);

if (err < 0)

return err;

//初始化芯片，这部分就是一个是否snoop的操作，表示什么意义还不懂

/\* initialize chip \*/

azx\_init\_pci(chip);

if (chip->driver\_caps & AZX\_DCAPS\_I915\_POWERWELL) {

struct hda\_intel \*hda;

hda = container\_of(chip, struct hda\_intel, chip);

haswell\_set\_bclk(hda);

}

//初始化CORB/RIRB寄存器及相关

hda\_intel\_init\_chip(chip, (probe\_only[dev] & 2) == 0);

//判断codec是否存在

/\* codec detection \*/

if (!azx\_bus(chip)->codec\_mask) {

dev\_err(card->dev, "no codecs found!\n");

return -ENODEV;

}

strcpy(card->driver, "HDA-Intel");

strlcpy(card->shortname, driver\_short\_names[chip->driver\_type],

sizeof(card->shortname));

snprintf(card->longname, sizeof(card->longname),

"%s at 0x%lx irq %i",

card->shortname, bus->addr, bus->irq);

return 0;

}

4.3 HDA Codec驱动

4.3.1 具体设备型号入口

仍以intel为例，其生产的多个设备中的控制器型号等可能会不一样，但是都会采用相同的驱动架构，也就是说，同一个驱动设备能够支持不同型号的设备，支持设备的设置是采用下面宏来实现：

MODULE\_DEVICE\_TABLE(pci, azx\_ids);

MODULE\_DEVICE\_TABLE的第一个参数是设备的类型，例如pci。后面一个参数是pci\_device\_id类型的设备表，这个设备表的最后一个元素是空的，用于标识结束。举例设备表其中一个元素如下：

/\* VIA VT8251/VT8237A \*/

{ PCI\_DEVICE(0x1106, 0x3288),

.driver\_data = AZX\_DRIVER\_VIA | AZX\_DCAPS\_POSFIX\_VIA },

其中0x1106是VIA厂家号，0x3288为声卡设备号，driver\_data是该设备驱动的私有数据，用法Z主要标识卡的类型type或者说能力capabilities，具体初始化中会用到。

该宏生成一个名为\_\_mod\_pci\_device\_table的局部变量，该变量指向第二个参数。内核构建时，depmod程序会在所有模块中搜索符号\_\_mod\_pci\_device\_table，把数据（设备列表）从模块中抽出，添加到映射文件/lib/modules/KERNEL\_VERSION/modules.pcimap中，当depmod结束之后，所有的PCI设备连同他们的模块名字都被该文件列出。当内核告知热插拔系统一个新的PCI设备被发现时，热插拔系统使用modules.pcimap文件来找寻恰当的驱动程序。

4.4 controller与codec关系建立

HH

## 5、ALSA相关底层函数

### 5.1 verb发送接收直接函数

codec相关操作值的宏定义在…\include\sound\hda\_verbs.h中，例如verb, Parameter IDs等等

1）发送：

/\*\*

\* snd\_hdac\_bus\_send\_cmd - send a command verb via CORB

\* @bus: HD-audio core bus

\* @val: encoded verb value to send

\*

\* Returns zero for success or a negative error code.

\*/

int snd\_hdac\_bus\_send\_cmd(struct hdac\_bus \*bus, unsigned int val)

{

unsigned int addr = azx\_command\_addr(val);

unsigned int wp, rp;

spin\_lock\_irq(&bus->reg\_lock);

bus->last\_cmd[azx\_command\_addr(val)] = val;

/\* add command to corb \*/

wp = snd\_hdac\_chip\_readw(bus, CORBWP);

if (wp == 0xffff) {

/\* something wrong, controller likely turned to D3 \*/

spin\_unlock\_irq(&bus->reg\_lock);

return -EIO;

}

wp++;

wp %= AZX\_MAX\_CORB\_ENTRIES;

rp = snd\_hdac\_chip\_readw(bus, CORBRP);

if (wp == rp) {

/\* oops, it's full \*/

spin\_unlock\_irq(&bus->reg\_lock);

return -EAGAIN;

}

bus->rirb.cmds[addr]++;

bus->corb.buf[wp] = cpu\_to\_le32(val);

snd\_hdac\_chip\_writew(bus, CORBWP, wp);//最终调用的是hdac\_bus结构体中的io\_ops绑定的函数writew

spin\_unlock\_irq(&bus->reg\_lock);

return 0;

}

2）接收

/\*\*

\* snd\_hdac\_bus\_get\_response - receive a response via RIRB

\* @bus: HD-audio core bus

\* @addr: codec address

\* @res: pointer to store the value, NULL when not needed

\*

\* Returns zero if a value is read, or a negative error code.

\*/

int snd\_hdac\_bus\_get\_response(struct hdac\_bus \*bus, unsigned int addr,

unsigned int \*res)

{

unsigned long timeout;

unsigned long loopcounter;

timeout = jiffies + msecs\_to\_jiffies(1000);

for (loopcounter = 0;; loopcounter++) {

spin\_lock\_irq(&bus->reg\_lock);

if (!bus->rirb.cmds[addr]) {

if (res)

\*res = bus->rirb.res[addr]; /\* the last value \*/

spin\_unlock\_irq(&bus->reg\_lock);

return 0;

}

spin\_unlock\_irq(&bus->reg\_lock);

if (time\_after(jiffies, timeout))

break;

if (loopcounter > 3000)

msleep(2); /\* temporary workaround \*/

else {

udelay(10);

cond\_resched();

}

}

return -EIO;

}

上述两个函数被封装到snd\_hda\_codec\_read() and snd\_hda\_codec\_write()中调用。

### 5.2 Register读写函数

1）azx\_readw(chip, GCAP) 等同于snd\_hdac\_chip\_readw(chip, GCAP);

函数参数中的GCAP为字符串，该函数是一个宏定义，最后是将该字符串拼接成了AZX\_REG\_GCAP，这又是一个宏定义，在../include/sound/had\_register.h中定义，其值为0x00就是HAD Controler的Global Capabilities寄存器值。不同版本下的名字以及位置可能不一样，但采用相同的拼接规则。had\_register.h是处于用户空间的文件，因此，说证明了azx\_readw不能被用户空间直接调用。

2）写函数用法相似。

### 5.3 三个底层操作接口映射

//直接操作到pci基本字、字节等的读写。对controller的操作

static const struct hdac\_io\_ops pci\_hda\_io\_ops = {

.reg\_writel = pci\_azx\_writel,

.reg\_readl = pci\_azx\_readl,

.reg\_writew = pci\_azx\_writew,

.reg\_readw = pci\_azx\_readw,

.reg\_writeb = pci\_azx\_writeb,

.reg\_readb = pci\_azx\_readb,

.dma\_alloc\_pages = dma\_alloc\_pages,//映射到DMA

.dma\_free\_pages = dma\_free\_pages,

};

//PCI内存页的管理、中断管理。

static const struct hda\_controller\_ops pci\_hda\_ops = {

.disable\_msi\_reset\_irq = disable\_msi\_reset\_irq,//失能msi复位中断

.substream\_alloc\_pages = substream\_alloc\_pages,//为substream分配页

.substream\_free\_pages = substream\_free\_pages,

.pcm\_mmap\_prepare = pcm\_mmap\_prepare,//准备pcm虚拟映射空间

.position\_check = azx\_position\_check,//检查chip位置

.link\_power = azx\_intel\_link\_power,

};

//以pci\_hda\_io\_ops及协议格式为基础的封装。对codec操作

static const struct hdac\_bus\_ops bus\_core\_ops = {

.command = azx\_send\_cmd,

.get\_response = azx\_get\_response,

.link\_power = azx\_link\_power,

};

## 6、驱动文件分析

### 6.1 四个重要文件hdac\_bus.c、hdac\_controller.c、hdac\_stream.c 、hdac\_devices.c

这两个文件主要包含了以下两组函数

**/\*hdac\_bus.c\*/**

//主要是绑定verb操作函数、底层IO操作函数

int snd\_hdac\_bus\_init(struct hdac\_bus \*bus, struct device \*dev, const struct hdac\_bus\_ops \*ops, const struct hdac\_io\_ops \*io\_ops);

void snd\_hdac\_bus\_exit(struct hdac\_bus \*bus);

//在指定bus上执行Verb操作，带不带锁

int snd\_hdac\_bus\_exec\_verb(struct hdac\_bus \*bus, unsigned int addr, unsigned int cmd, unsigned int \*res);

int snd\_hdac\_bus\_exec\_verb\_unlocked(struct hdac\_bus \*bus, unsigned int addr, unsigned int cmd, unsigned int \*res);

//添加一个主动上送的事件，该函数在主动上送的中断处理函数中被调用

void snd\_hdac\_bus\_queue\_event(struct hdac\_bus \*bus, u32 res, u32 res\_ex);

//往bus上增减codec设备

int snd\_hdac\_bus\_add\_device(struct hdac\_bus \*bus, struct hdac\_device \*codec);

void snd\_hdac\_bus\_remove\_device(struct hdac\_bus \*bus, struct hdac\_device \*codec);

**/\*hdac\_controller.c\*/**

//通过CORB发送一个cmd verb，包含指针修改，寄存器填充等过程

int snd\_hdac\_bus\_send\_cmd(struct hdac\_bus \*bus, unsigned int val);

//接收一个response

int snd\_hdac\_bus\_get\_response(struct hdac\_bus \*bus, unsigned int addr, unsigned int \*res);

//复位并启动控制器寄存器，会调用azx\_reset，snd\_hdac\_bus\_init\_cmd\_io，中断使能等函数

bool snd\_hdac\_bus\_init\_chip(struct hdac\_bus \*bus, bool full\_reset);

//失能控制器的全部中断和IO

void snd\_hdac\_bus\_stop\_chip(struct hdac\_bus \*bus);

//初始化设置CORB/RIRB 相关寄存器

void snd\_hdac\_bus\_init\_cmd\_io(struct hdac\_bus \*bus);

//清除CORB/RIRB相关buffer，主要是通过设置\*CTL实现失能DMA

void snd\_hdac\_bus\_stop\_cmd\_io(struct hdac\_bus \*bus);

//将hdac复位状态的进入与退出, GCTL寄存器的bit0

void snd\_hdac\_bus\_enter\_link\_reset(struct hdac\_bus \*bus);

void snd\_hdac\_bus\_exit\_link\_reset(struct hdac\_bus \*bus);

//检索RIRB entreis，通常在中断处理函数中被调用

void snd\_hdac\_bus\_update\_rirb(struct hdac\_bus \*bus);

//数据流的中断处理函数

void snd\_hdac\_bus\_handle\_stream\_irq(struct hdac\_bus \*bus, unsigned int status, void (\*ack)(struct hdac\_bus \*,struct hdac\_stream \*));

//分配和释放Buffer Descriptor List以及相关缓冲区

int snd\_hdac\_bus\_alloc\_stream\_pages(struct hdac\_bus \*bus);

void snd\_hdac\_bus\_free\_stream\_pages(struct hdac\_bus \*bus);

**/\* hdac\_stream.c \*/**

//初始化每个流设备，初始化为playback或者capture设备；还包括方向，stream\_tag等等，并将各stream绑定到hdac\_bus上

void snd\_hdac\_stream\_init(struct hdac\_bus \*bus, struct hdac\_stream \*azx\_dev, int idx, int direction, int tag);

//为PCM分配数据流空间，先查找之前用过目前空闲的，没有的话找其它空闲的，再不行就返回NULL

struct hdac\_stream \*snd\_hdac\_stream\_assign(struct hdac\_bus \*bus, struct snd\_pcm\_substream \*substream);

//释放数据流空间

void snd\_hdac\_stream\_release(struct hdac\_stream \*azx\_dev);

//返回指定stream\_tag和dir的数据流

struct hdac\_stream \*snd\_hdac\_get\_stream(struct hdac\_bus \*bus, int dir, int stream\_tag);

//设置数据流传输控制参数

int snd\_hdac\_stream\_setup(struct hdac\_stream \*azx\_dev);

//清除数据流传输控制参数

void snd\_hdac\_stream\_cleanup(struct hdac\_stream \*azx\_dev);

//Set up the buffer descriptor table of the given stream based on the period and buffer sizes of the assigned PCM substream.

int snd\_hdac\_stream\_setup\_periods(struct hdac\_stream \*azx\_dev);

//Setup the HD-audio core stream parameters from substream of the stream and passed format value

int snd\_hdac\_stream\_set\_params(struct hdac\_stream \*azx\_dev, unsigned int format\_val);

//开始一个数据流的传输，包括DMA和中断

void snd\_hdac\_stream\_start(struct hdac\_stream \*azx\_dev, bool fresh\_start);

//停止传输的DMA

void snd\_hdac\_stream\_clear(struct hdac\_stream \*azx\_dev);

//停止当前数据流的DMA和中断

void snd\_hdac\_stream\_stop(struct hdac\_stream \*azx\_dev);

//复位一个数据流

void snd\_hdac\_stream\_reset(struct hdac\_stream \*azx\_dev);

//开关流同步寄存器0x38 ssync

void snd\_hdac\_stream\_sync\_trigger(struct hdac\_stream \*azx\_dev, bool set, unsigned int streams, unsigned int reg);

//启停触发操作的同步

void snd\_hdac\_stream\_sync(struct hdac\_stream \*azx\_dev, bool start, unsigned int streams);

//初始化时间计数器

void snd\_hdac\_stream\_timecounter\_init(struct hdac\_stream \*azx\_dev, unsigned int streams);

**/\* hdac\_devices.c \*/**

//初始化hd-audio codec base device

int snd\_hdac\_device\_init(struct hdac\_device \*dev, struct hdac\_bus \*bus, const char \*name, unsigned int addr);

//清除hd-audio codec base device

void snd\_hdac\_device\_exit(struct hdac\_device \*dev);

//注册hd-audio codec base device

int snd\_hdac\_device\_register(struct hdac\_device \*codec);

//注销hd-audio codec base device

void snd\_hdac\_device\_unregister(struct hdac\_device \*codec);

//复位组件的开始结束节点

int snd\_hdac\_refresh\_widgets(struct hdac\_device \*codec);

//复位codec组件，重新初始化codec系统文件

int snd\_hdac\_refresh\_widget\_sysfs(struct hdac\_device \*codec);

//生成一个用来发送给hdac控制器的32bits的命令字

unsigned int snd\_hdac\_make\_cmd(struct hdac\_device \*codec, hda\_nid\_t nid, unsigned int verb, unsigned int parm);

//执行一个编码之后的verb

int snd\_hdac\_exec\_verb(struct hdac\_device \*codec, unsigned int cmd, unsigned int flags, unsigned int \*res);

//执行一个读verb

int snd\_hdac\_read(struct hdac\_device \*codec, hda\_nid\_t nid, unsigned int verb, unsigned int parm, unsigned int \*res);

//读取codec参数

int \_snd\_hdac\_read\_parm(struct hdac\_device \*codec, hda\_nid\_t nid, int parm, unsigned int \*res);

//不用缓存，直接读取codec参数

int snd\_hdac\_read\_parm\_uncached(struct hdac\_device \*codec, hda\_nid\_t nid, int parm);

//重写只读参数

int snd\_hdac\_override\_parm(struct hdac\_device \*codec, hda\_nid\_t nid, unsigned int parm, unsigned int val);

//获取组件的链接列表

int snd\_hdac\_get\_connections(struct hdac\_device \*codec, hda\_nid\_t nid, hda\_nid\_t \*conn\_list, int max\_conns);

//从开始节点获取其包含的子节点

int snd\_hdac\_get\_sub\_nodes(struct hdac\_device \*codec, hda\_nid\_t nid, hda\_nid\_t \*start\_id);

//启停codec的链路供能？

int snd\_hdac\_link\_power(struct hdac\_device \*codec, bool enable);

//通过指定的参数计算格式的设置值format bitset

unsigned int snd\_hdac\_calc\_stream\_format(unsigned int rate,

unsigned int channels,

unsigned int format,

unsigned int maxbps,

unsigned short spdif\_ctls);

//查询支持的PCM速率和格式

int snd\_hdac\_query\_supported\_pcm(struct hdac\_device \*codec, hda\_nid\_t nid, u32 \*ratesp, u64 \*formatsp, unsigned int \*bpsp);

//检查格式的有效性，Check whether the given node supports the format value.

bool snd\_hdac\_is\_supported\_format(struct hdac\_device \*codec, hda\_nid\_t nid, unsigned int format);

### 5.4 又四个重要文件hda\_controller.c、hda\_bind.c、hda\_codec.c、hda\_intel.c

先后顺序：

/\* **hda\_controller.c** \*/

unsigned int azx\_get\_position(struct azx \*chip, struct azx\_dev \*azx\_dev);

//获取link position in buffer，缓冲区中的链路位置，LPIB是控制器的寄存器

unsigned int azx\_get\_pos\_lpib(struct azx \*chip, struct azx\_dev \*azx\_dev);

//获取position buffer pointer，位置缓冲区指针

unsigned int azx\_get\_pos\_posbuf(struct azx \*chip, struct azx\_dev \*azx\_dev);

/\* Stream control. \*/

void azx\_stop\_all\_streams(struct azx \*chip);

//停止全部数据流的DMA和中断

/\* Allocation functions. \*/

//分配和释放Buffer Descriptor List以及相关缓冲区,在所有的流分配之后调用这个（Call this after assigning the all streams）

#define azx\_alloc\_stream\_pages(chip) \

snd\_hdac\_bus\_alloc\_stream\_pages(azx\_bus(chip))

#define azx\_free\_stream\_pages(chip) \

snd\_hdac\_bus\_free\_stream\_pages(azx\_bus(chip))

/\* Low level azx interface \*/

//复位重启控制器的寄存器

void azx\_init\_chip(struct azx \*chip, bool full\_reset);

//停止所有中断和IO

void azx\_stop\_chip(struct azx \*chip);

//将hdac复位状态的进入, GCTL寄存器的bit0

#define azx\_enter\_link\_reset(chip) \

snd\_hdac\_bus\_enter\_link\_reset(azx\_bus(chip))

//中断的处理，返回值处理后的状态，分为NONE,HANDLED,WAKE\_THREAD三种

irqreturn\_t azx\_interrupt(int irq, void \*dev\_id);

/\* Codec interface \*/

int azx\_bus\_init(struct azx \*chip, const char \*model, const struct hdac\_io\_ops \*io\_ops);

//创建所有的codec实例

int azx\_probe\_codecs(struct azx \*chip, unsigned int max\_slots);

//配置每个codec实例

int azx\_codec\_configure(struct azx \*chip);

//初始化数据流，主要内容为为每个逻辑流分配数据空间azx\_dev并初步初始化其值，例如stream\_tag

int azx\_init\_streams(struct azx \*chip);

//顾名思义，与init相反，释放空间及相关信息

void azx\_free\_streams(struct azx \*chip);

/\* **hda\_bind.c** \*/

//这个文件主要是在pach文件和驱动文件之间建立关系

int \_\_hda\_codec\_driver\_register(struct hda\_codec\_driver \*drv, const char \*name,

struct module \*owner);

#define hda\_codec\_driver\_register(drv) \

\_\_hda\_codec\_driver\_register(drv, KBUILD\_MODNAME, THIS\_MODULE)

void hda\_codec\_driver\_unregister(struct hda\_codec\_driver \*drv);

#define module\_hda\_codec\_driver(drv) \

module\_driver(drv, hda\_codec\_driver\_register, \

hda\_codec\_driver\_unregister)

//配置所有的codec实例

int snd\_hda\_codec\_configure(struct hda\_codec \*codec);

/\* hda\_codec.c\*/

//该文件包含了codec相关的一些函数，主要分为以下几种

/\*

\* constructors 构造函数

int snd\_hda\_codec\_new(struct hda\_bus \*bus, struct snd\_card \*card, unsigned int codec\_addr, struct hda\_codec \*\*codecp);

int snd\_hda\_codec\_configure(struct hda\_codec \*codec);

int snd\_hda\_codec\_update\_widgets(struct hda\_codec \*codec);

\*

\* low level functions 底层相关功能函数

unsigned int snd\_hda\_codec\_read(struct hda\_codec \*codec, hda\_nid\_t nid, int flags, unsigned int verb, unsigned int parm);

int snd\_hda\_codec\_write(struct hda\_codec \*codec, hda\_nid\_t nid, int flags,unsigned int verb, unsigned int parm);

#define snd\_hda\_param\_read(codec, nid, param) snd\_hdac\_read\_parm(&(codec)->core, nid, param)

#define snd\_hda\_get\_sub\_nodes(codec, nid, start\_nid) snd\_hdac\_get\_sub\_nodes(&(codec)->core, nid, start\_nid)

int snd\_hda\_get\_connections(struct hda\_codec \*codec, hda\_nid\_t nid, hda\_nid\_t \*conn\_list, int max\_conns);

……

\*

\* unsolicited event主动事件

\*

\* cached write缓存写

\*

\* codec pin configs编解码器引脚控制

\*

\* SPIDF controls

\*

\* Mixer 混音器

int snd\_hda\_codec\_build\_controls(struct hda\_codec \*codec);

\*

\* PCM

int snd\_hda\_codec\_parse\_pcms(struct hda\_codec \*codec);

int snd\_hda\_codec\_build\_pcms(struct hda\_codec \*codec);

void snd\_hda\_codec\_pcm\_put(struct hda\_pcm \*pcm);

int snd\_hda\_codec\_prepare(struct hda\_codec \*codec,

struct hda\_pcm\_stream \*hinfo,

unsigned int stream,

unsigned int format,

struct snd\_pcm\_substream \*substream);

void snd\_hda\_codec\_cleanup(struct hda\_codec \*codec, struct hda\_pcm\_stream \*hinfo, struct snd\_pcm\_substream \*substream);

void snd\_hda\_codec\_setup\_stream(struct hda\_codec \*codec, hda\_nid\_t nid, u32 stream\_tag, int channel\_id, int format);

void \_\_snd\_hda\_codec\_cleanup\_stream(struct hda\_codec \*codec, hda\_nid\_t nid, int do\_now);

#define snd\_hda\_codec\_cleanup\_stream(codec, nid)\_\_snd\_hda\_codec\_cleanup\_stream(codec, nid, 0)

#define snd\_hda\_query\_supported\_pcm(codec, nid, ratesp, fmtsp, bpsp)snd\_hdac\_query\_supported\_pcm(&(codec)->core, nid, ratesp, fmtsp, bpsp)

#define snd\_hda\_is\_supported\_format(codec, nid, fmt)snd\_hdac\_is\_supported\_format(&(codec)->core, nid, fmt)

int snd\_hda\_attach\_pcm\_stream(struct hda\_bus \*\_bus, struct hda\_codec \*codec, struct hda\_pcm \*cpcm);

\*

\* Misc 杂

void snd\_hda\_get\_codec\_name(struct hda\_codec \*codec, char \*name, int namelen);

void snd\_hda\_codec\_set\_power\_to\_all(struct hda\_codec \*codec, hda\_nid\_t fg, unsigned int power\_state);

int snd\_hda\_lock\_devices(struct hda\_bus \*bus);

void snd\_hda\_unlock\_devices(struct hda\_bus \*bus);

void snd\_hda\_bus\_reset(struct hda\_bus \*bus);

void snd\_hda\_bus\_reset\_codecs(struct hda\_bus \*bus);

\*

\* power management电源管理

\*

\* power saving 省电节能

\*

\* patch firmware补丁固件

int snd\_hda\_load\_patch(struct hda\_bus \*bus, size\_t size, const void \*buf);

\*

\*/

/\* hda\_intel.c \*/

//最重要的文件，最开始的文件

### 5.4 几个辅助文件hda\_sysfs.c、hda\_proc.c、hda\_jack.c、hda\_beep.c、hda\_auto\_parser.c

### 5.5补丁文件patch\_\*.c

该文件的作用是提供了hda\_codec\_preset这样一个结构体信息

struct hda\_codec\_preset {

unsigned int id;

unsigned int rev;

const char \*name;

int (\*patch)(struct hda\_codec \*codec);

};

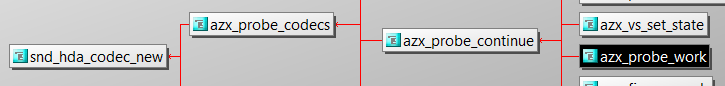
其中的id和name成员用来进行驱动补丁的匹配，rev一般用不到，patch则是补丁的内容。通过查看patch函数实例，它的传入参数是一个hda\_codec结构体，其内部创建了一个第一个成员为hda\_gen\_spec结构的spec结构体，函数的作用一是将两个结构与操作绑定（codec->spec = spec;），二是对这两个结构体进行了部分特殊信息初始化。

因此，驱动与patch之间是通过id与name匹配，通过hda\_codec实现数据交互，通过via\_patch\_ops实现特殊操作绑定。via\_patch\_ops也是hda\_codec.c文件中很多函数实现时调用的接口。而via\_patch\_ops中的函数也是通过调用hda\_codec.c -> hda\_device.c -> hdac\_bus.c -> ……来实现的。

## 7、 ALSA中的重要结构体

### 7.1 hda\_codec

该结构体在snd\_hda\_codec\_new函数中申请到内存，AZX\_DEFAULT\_CODECS = 4，每一个codec对应一个结构体实例



/\* codec information \*/

struct hda\_codec {

struct hdac\_device core;

struct hda\_bus \*bus;

struct snd\_card \*card;

unsigned int addr; /\* codec addr\*/

u32 probe\_id; /\* overridden id for probing \*/

/\* detected preset \*/

const struct hda\_codec\_preset \*preset;

const char \*modelname; /\* model name for preset \*/

/\* set by patch \*/

struct hda\_codec\_ops patch\_ops;

/\* PCM to create, set by patch\_ops.build\_pcms callback \*/

struct list\_head pcm\_list\_head;

/\* codec specific info \*/

void \*spec;

/\* beep device \*/

struct hda\_beep \*beep;

unsigned int beep\_mode;

/\* widget capabilities cache \*/

u32 \*wcaps;

struct snd\_array mixers; /\* list of assigned mixer elements \*/

struct snd\_array nids; /\* list of mapped mixer elements \*/

struct list\_head conn\_list; /\* linked-list of connection-list \*/

struct mutex spdif\_mutex;

struct mutex control\_mutex;

struct snd\_array spdif\_out;

unsigned int spdif\_in\_enable; /\* SPDIF input enable? \*/

const hda\_nid\_t \*slave\_dig\_outs; /\* optional digital out slave widgets \*/

struct snd\_array init\_pins; /\* initial (BIOS) pin configurations \*/

struct snd\_array driver\_pins; /\* pin configs set by codec parser \*/

struct snd\_array cvt\_setups; /\* audio convert setups \*/

struct mutex user\_mutex;

#ifdef CONFIG\_SND\_HDA\_RECONFIG

struct snd\_array init\_verbs; /\* additional init verbs \*/

struct snd\_array hints; /\* additional hints \*/

struct snd\_array user\_pins; /\* default pin configs to override \*/

#endif

#ifdef CONFIG\_SND\_HDA\_HWDEP

struct snd\_hwdep \*hwdep; /\* assigned hwdep device \*/

#endif

/\* misc flags \*/

unsigned int in\_freeing:1; /\* being released \*/

unsigned int registered:1; /\* codec was registered \*/

unsigned int spdif\_status\_reset :1; /\* needs to toggle SPDIF for each

\* status change

\* (e.g. Realtek codecs)

\*/

unsigned int pin\_amp\_workaround:1; /\* pin out-amp takes index

\* (e.g. Conexant codecs)

\*/

unsigned int single\_adc\_amp:1; /\* adc in-amp takes no index

\* (e.g. CX20549 codec)

\*/

unsigned int no\_sticky\_stream:1; /\* no sticky-PCM stream assignment \*/

unsigned int pins\_shutup:1; /\* pins are shut up \*/

unsigned int no\_trigger\_sense:1; /\* don't trigger at pin-sensing \*/

unsigned int no\_jack\_detect:1; /\* Machine has no jack-detection \*/

unsigned int inv\_eapd:1; /\* broken h/w: inverted EAPD control \*/

unsigned int inv\_jack\_detect:1; /\* broken h/w: inverted detection bit \*/

unsigned int pcm\_format\_first:1; /\* PCM format must be set first \*/

unsigned int cached\_write:1; /\* write only to caches \*/

unsigned int dp\_mst:1; /\* support DP1.2 Multi-stream transport \*/

unsigned int dump\_coef:1; /\* dump processing coefs in codec proc file \*/

unsigned int power\_save\_node:1; /\* advanced PM for each widget \*/

unsigned int auto\_runtime\_pm:1; /\* enable automatic codec runtime pm \*/

#ifdef CONFIG\_PM

unsigned long power\_on\_acct;

unsigned long power\_off\_acct;

unsigned long power\_jiffies;

#endif

/\* filter the requested power state per nid \*/

unsigned int (\*power\_filter)(struct hda\_codec \*codec, hda\_nid\_t nid,

unsigned int power\_state);

/\* codec-specific additional proc output \*/

void (\*proc\_widget\_hook)(struct snd\_info\_buffer \*buffer,

struct hda\_codec \*codec, hda\_nid\_t nid);

/\* jack detection \*/

struct snd\_array jacktbl;

unsigned long jackpoll\_interval; /\* In jiffies. Zero means no poll, rely on unsol events \*/

struct delayed\_work jackpoll\_work;

/\* jack detection \*/

struct snd\_array jacks;

int depop\_delay; /\* depop delay in ms, -1 for default delay time \*/

/\* fix-up list \*/

int fixup\_id;

const struct hda\_fixup \*fixup\_list;

const char \*fixup\_name;

/\* additional init verbs \*/

struct snd\_array verbs;

};

### 7.2 hda\_intel

struct hda\_intel {

struct azx chip;

/\* for pending irqs \*/

struct work\_struct irq\_pending\_work;

/\* sync probing \*/

struct completion probe\_wait;

struct work\_struct probe\_work;

/\* card list (for power\_save trigger) \*/

struct list\_head list;

/\* extra flags \*/

unsigned int irq\_pending\_warned:1;

unsigned int probe\_continued:1;

/\* VGA-switcheroo setup \*/

unsigned int use\_vga\_switcheroo:1;

unsigned int vga\_switcheroo\_registered:1;

unsigned int init\_failed:1; /\* delayed init failed \*/

/\* secondary power domain for hdmi audio under vga device \*/

struct dev\_pm\_domain hdmi\_pm\_domain;

bool need\_i915\_power:1; /\* the hda controller needs i915 power \*/

};

### 7.3 snd\_card

/\* main structure for soundcard \*/

struct snd\_card {

int number; /\* number of soundcard (index to snd\_cards) \*/

char id[16]; /\* id string of this card \*/

char driver[16]; /\* driver name \*/

char shortname[32]; /\* short name of this soundcard \*/

char longname[80]; /\* name of this soundcard \*/

char mixername[80]; /\* mixer name \*/

char components[128]; /\* card components delimited with space \*/

struct module \*module; /\* top-level module \*/

void \*private\_data; /\* private data for soundcard \*/

void (\*private\_free) (struct snd\_card \*card); /\* callback for freeing of private data \*/

struct list\_head devices; /\* devices \*/

struct device ctl\_dev; /\* control device \*/

unsigned int last\_numid; /\* last used numeric ID \*/

struct rw\_semaphore controls\_rwsem; /\* controls list lock \*/

rwlock\_t ctl\_files\_rwlock; /\* ctl\_files list lock \*/

int controls\_count; /\* count of all controls \*/

int user\_ctl\_count; /\* count of all user controls \*/

struct list\_head controls; /\* all controls for this card \*/

struct list\_head ctl\_files; /\* active control files \*/

struct mutex user\_ctl\_lock; /\* protects user controls against

concurrent access \*/

struct snd\_info\_entry \*proc\_root; /\* root for soundcard specific files \*/

struct snd\_info\_entry \*proc\_id; /\* the card id \*/

struct proc\_dir\_entry \*proc\_root\_link; /\* number link to real id \*/

struct list\_head files\_list; /\* all files associated to this card \*/

struct snd\_shutdown\_f\_ops \*s\_f\_ops; /\* file operations in the shutdown state \*/

spinlock\_t files\_lock; /\* lock the files for this card \*/

int shutdown; /\* this card is going down \*/

struct completion \*release\_completion;

struct device \*dev; /\* device assigned to this card \*/

struct device card\_dev; /\* cardX object for sysfs \*/

const struct attribute\_group \*dev\_groups[4]; /\* assigned sysfs attr \*/

bool registered; /\* card\_dev is registered? \*/

#ifdef CONFIG\_PM

unsigned int power\_state; /\* power state \*/

struct mutex power\_lock; /\* power lock \*/

wait\_queue\_head\_t power\_sleep;

#endif

#if defined(CONFIG\_SND\_MIXER\_OSS) || defined(CONFIG\_SND\_MIXER\_OSS\_MODULE)

struct snd\_mixer\_oss \*mixer\_oss;

int mixer\_oss\_change\_count;

#endif

};

### 7.4 hda\_codec\_driver

### 7.5 snd\_pcm\_ops

static struct snd\_pcm\_ops azx\_pcm\_ops = {

.open = azx\_pcm\_open,

.close = azx\_pcm\_close,

.ioctl = snd\_pcm\_lib\_ioctl,

.hw\_params = azx\_pcm\_hw\_params,

.hw\_free = azx\_pcm\_hw\_free,

.prepare = azx\_pcm\_prepare,

.trigger = azx\_pcm\_trigger,

.pointer = azx\_pcm\_pointer,

.get\_time\_info = azx\_get\_time\_info,

.mmap = azx\_pcm\_mmap,

.page = snd\_pcm\_sgbuf\_ops\_page,

};

该结构体为PCM操作的接口，被patch\_\*.c文件所注册

## 7. 其它Spec

### 7.1 IRS\_CND001\_D20F0\_HDAC\_R050

IRS：Internal Register Set内部寄存器组

该文件给出了与南桥中的bus 0, device 20, function 0模块HDAC有关的IRS。

主要包含三个部分：

HDAC的基本PCI 配置空间0x00 ~ 0x3F

HDAC的扩展PCI配置空间0x40 ~ 0xFF ~ \*，厂家、设备自定义区。

HDAC的MMIO配置空间0x00 ~ 0x2167，PCI设备内部寄存器说明

对于每个寄存器给出如下结构的描述：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit | **Attribute** | HW priority | **Default** | **Description** | Mnemonic | **ChipRev** | **PwrDm** | **S** | **P** | **E** |

其中：

Bit:数据位

Attribute： 读写属性

HW priority： 硬件优先级，不明白

Default: 默认值

Description: 寄存器描述

Mnemonic: 助记符\*

ChipRev: 芯片版本

PwrDm: 电源域

S: 设置为最佳稳定性和/或可靠性的值。

P: 提升性能

E: 用于节能

**((Internal Content)):** Used to represent internal registers or contents that cannot be released to customers.

这种灰色覆盖的寄存器不能开放给用户

### 7.2 P-spec

HDAC Programming Guide 编程手册

每个寄存器都在驱动中给出了相应的默认值，但是在实际使用中，通过调整相应寄存器的值可以改变对HDAC的控制。

该手册按功能对寄存器进行了分组，操作分为两部分：

1. 给出寄存器设置顺序
2. 给出部分寄存器的设置值。

### 7.3 T-spec

TARGET SPECIFICATION

按章节：

系统架构框图

特性列表

时钟方案

HDAC架构框图

引脚说明

用于配置寄存器的下游接口

snoop接口

no-snoop接口

与codec之间的link接口

复位与初始化

电源管理

在C3/C4期间降低输出流的阈值

状态切换到C2时阈值的设置公式为（N+1）\* Frame + 4，最小FIFO的计算公式为（N+2）\* Frame + 4。

IO的虚拟化 IOV

## x、 杂

### x.1 关于stream\_tag

stream\_tag是在stream初始化函数中进行设置，通过阅读azx\_init\_streams()函数可以看出，stream\_tag就是与流对应的一个标记。标记的方法有两个，一个是根据chip中的所有stream顺序，从1开始递增计数；另一种方法是将所有的stream分成两种，即play与capture，分别计数。从源码上看，系统支持的stream总数量由chipset芯片决定，普通的芯片为4路capture和4路playback。

为什么要设置这样一个标记？

### x.2 不清楚的疑问

#### 1、各路DMA的数据是存储在HDA Controler吗？

hdac\_bus结构体中有一个snd\_dma\_buffer结构体成员，其结构为

struct snd\_dma\_buffer {

struct snd\_dma\_device dev; /\* device type \*/

unsigned char \*area; /\* virtual pointer \*/

dma\_addr\_t addr; /\* physical address \*/

size\_t bytes; /\* buffer size in bytes \*/

void \*private\_data; /\* private for allocator; don't touch \*/

};

int snd\_hdac\_bus\_alloc\_stream\_pages(struct hdac\_bus \*bus)

{

struct hdac\_stream \*s;

int num\_streams = 0;

int err;

list\_for\_each\_entry(s, &bus->stream\_list, list) {//罗列所有的s

/\* allocate memory for the BDL for each stream该函数为每个流分配了BDL，\*/

err = bus->io\_ops->dma\_alloc\_pages(bus, SNDRV\_DMA\_TYPE\_DEV, BDL\_SIZE, &s->bdl);

num\_streams++;

if (err < 0)

return -ENOMEM;

}

if (WARN\_ON(!num\_streams)) return -EINVAL;

/\* allocate memory for the position buffer 分配\*/

err = bus->io\_ops->dma\_alloc\_pages(bus, SNDRV\_DMA\_TYPE\_DEV, num\_streams \* 8, &bus->posbuf);//64bites 3.6.1节

if (err < 0) return -ENOMEM;

list\_for\_each\_entry(s, &bus->stream\_list, list)

s->posbuf = (\_\_le32 \*)(bus->posbuf.area + s->index \* 8);

/\* single page (at least 4096 bytes) must suffice for both ringbuffes分配了一个总线缓冲区供所有缓冲区公用 \*/

return bus->io\_ops->dma\_alloc\_pages(bus, SNDRV\_DMA\_TYPE\_DEV, PAGE\_SIZE, &bus->rb);

}

该结构体为dma分配数据区的描述结构，其中area为数据区的首地址，bytes为数据区大小。

还有两个hdac\_rb结构体ring buffer

struct hdac\_rb {

\_\_le32 \*buf; /\* virtual address of CORB/RIRB buffer \*/

dma\_addr\_t addr; /\* physical address of CORB/RIRB buffer \*/

unsigned short rp, wp; /\* RIRB read/write pointers \*/

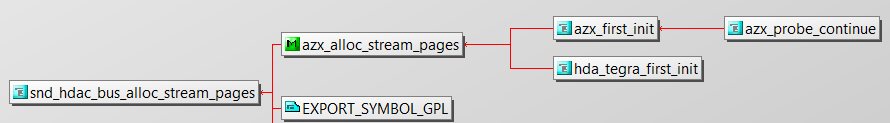
int cmds[HDA\_MAX\_CODECS]; /\* number of pending requests \*/

u32 res[HDA\_MAX\_CODECS]; /\* last read value \*/

};

snd\_dma\_buffer为向内存申请的数据区（snd\_hdac\_bus\_alloc\_stream\_pages函数），其大小为BDL\_SIZE bytes，分给两个hdac\_rb共享（snd\_hdac\_bus\_init\_cmd\_io函数），即将snd\_dma\_buffer. area分两部分赋值给hdac\_rb.buf，每份2048字节。

#### 2、从snd\_hdac\_bus\_alloc\_stream\_pages函数来看，是根据hdac\_bus中的stream\_list索引出hdac\_stream，然后再allocate memory for the BDL for each stream。那么，得出hdac\_stream数量之前，是否需要probe codec？

从函数调用关系来看，，其中，在azx\_first\_init函数之后才调用azx\_probe\_codecs函数进行codec检测，说明不需要。那么得出数据流数量的依据又是什么？是从芯片厂家，azx\_init\_streams函数之前。

#### 3、外部传入参数是由哪传入？如何传入的？

hda\_codec的数量为4个、

#### 4、数据流的工作流程

相关主要函数：

static struct snd\_pcm\_ops azx\_pcm\_ops = {

.open = azx\_pcm\_open,

.close = azx\_pcm\_close,

.ioctl = snd\_pcm\_lib\_ioctl,

.hw\_params = azx\_pcm\_hw\_params,

.hw\_free = azx\_pcm\_hw\_free,

.prepare = azx\_pcm\_prepare,

.trigger = azx\_pcm\_trigger,

.pointer = azx\_pcm\_pointer,

.get\_time\_info = azx\_get\_time\_info,

.mmap = azx\_pcm\_mmap,

.page = snd\_pcm\_sgbuf\_ops\_page,

};

static const struct hda\_pcm\_stream pcm\_analog\_playback = {

.substreams = 1,

.channels\_min = 2,

.channels\_max = 8,

/\* NID is set in build\_pcms \*/

.ops = {

.open = playback\_pcm\_open,

.close = playback\_pcm\_close,

.prepare = playback\_pcm\_prepare,

.cleanup = playback\_pcm\_cleanup

},

};

static const struct hda\_pcm\_stream pcm\_analog\_capture = {

.substreams = 1,

.channels\_min = 2,

.channels\_max = 2,

/\* NID is set in build\_pcms \*/

.ops = {

.open = capture\_pcm\_open,

.close = capture\_pcm\_close,

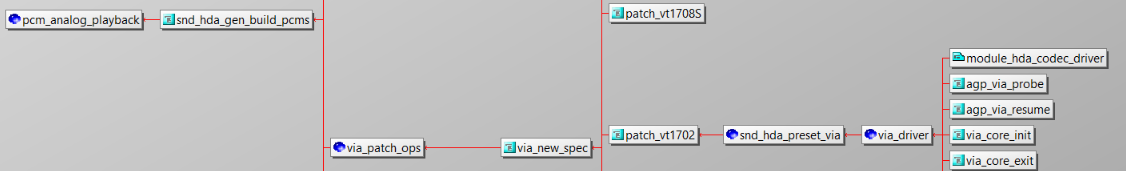
.prepare = capture\_pcm\_prepare,

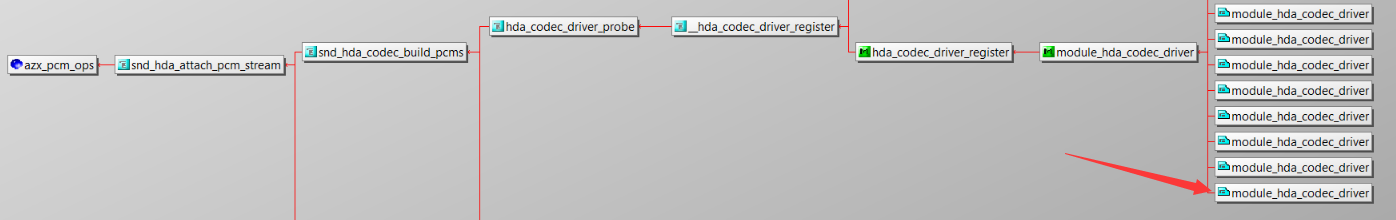
.cleanup = capture\_pcm\_cleanup

},

};

azx\_pcm\_ops：PCM的操作函数，并未指明是playback还是capture





### x.3 list与container\_of

linux中的链表采用的是无数据域的形式，链表定义及初始化如下：

struct list\_head {

struct list\_head \*next, \*prev;

};

static inline void INIT\_LIST\_HEAD(struct list\_head \*list)

{

list->next = list;

list->prev = list;

}

可以看出，链表为双向循环链表。因此，在进行链表索引时，如果索引到自身时则说明遍历结束。此外，linux中不是在链表结构中包含数据，而是在数据结构中包含链表节点，即链表结构为结构体的一个成员。那么想要根据链表所引导数据则要利用container\_of宏定义。该宏的原型如下：

#define offsetof(TYPE, MEMBER) ((unsigned int)&((TYPE \*)0)->MEMBER)

#define container\_of(ptr, type, member) ({ \

const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

其中：

ptr:表示结构体中member的地址

type:表示结构体类型

member:表示结构体中的成员

该宏的作用，就是通过ptr的地址返回type型结构体的首地址。为了方便，又对该宏进行了宏定义包装

#define list\_entry(ptr, type, member) \

container\_of(ptr, type, member)

### x.4 codec的检测

stream的数量由南桥芯片类型决定，但是codec的数量及型号则是检测出来的，那么codec是在哪里检出来的？first\_init函数中根据azx\_bus(chip)->codec\_mask为真时认为有codecs那么，codec\_mask是在哪里赋值？查找结果显示两处：check\_probe\_mask函数和azx\_probe\_codecs函数。先是check\_probe\_mask函数，codec\_mask由codec\_probe\_mask决定，codec\_probe\_mask由外部参数probe\_mask[dev]决定。azx\_probe\_codecs函数中的赋值仅仅是在探测不到时将对应codec标记为error。

因此，codec的数量及型号由驱动外部参数传入借用函数为check\_probe\_mask。驱动要做的是探测外部传入参数对应的codec是否存在，存在则做进一步操作，借用的函数为azx\_probe\_codecs。

### x.5 snoop概念与操作

azx\_init\_pci(chip);函数名的意思是初始化pci，注释的意思是初始化chip（/\* initialize chip \*/）。其函数源码如下：

static void azx\_init\_pci(struct azx \*chip)

{

int snoop\_type = azx\_get\_snoop\_type(chip);

/\* Clear bits 0-2 of PCI register TCSEL (at offset 0x44)

\* TCSEL == Traffic Class Select Register, which sets PCI express QOS

\* Ensuring these bits are 0 clears playback static on some HD Audio

\* codecs.

\* The PCI register TCSEL is defined in the Intel manuals.

\*/

if (!(chip->driver\_caps & AZX\_DCAPS\_NO\_TCSEL)) {

dev\_dbg(chip->card->dev, "Clearing TCSEL\n");

update\_pci\_byte(chip->pci, AZX\_PCIREG\_TCSEL, 0x07, 0);

}

/\* For ATI SB450/600/700/800/900 and AMD Hudson azalia HD audio,

\* we need to enable snoop.

\*/

if (snoop\_type == AZX\_SNOOP\_TYPE\_ATI) {

dev\_dbg(chip->card->dev, "Setting ATI snoop: %d\n",

azx\_snoop(chip));

update\_pci\_byte(chip->pci,

ATI\_SB450\_HDAUDIO\_MISC\_CNTR2\_ADDR, 0x07,

azx\_snoop(chip) ? ATI\_SB450\_HDAUDIO\_ENABLE\_SNOOP : 0);

}

/\* For NVIDIA HDA, enable snoop \*/

if (snoop\_type == AZX\_SNOOP\_TYPE\_NVIDIA) {

dev\_dbg(chip->card->dev, "Setting Nvidia snoop: %d\n",

azx\_snoop(chip));

update\_pci\_byte(chip->pci,

NVIDIA\_HDA\_TRANSREG\_ADDR,

0x0f, NVIDIA\_HDA\_ENABLE\_COHBITS);

update\_pci\_byte(chip->pci,

NVIDIA\_HDA\_ISTRM\_COH,

0x01, NVIDIA\_HDA\_ENABLE\_COHBIT);

update\_pci\_byte(chip->pci,

NVIDIA\_HDA\_OSTRM\_COH,

0x01, NVIDIA\_HDA\_ENABLE\_COHBIT);

}

可见，该函数的实现是根据不同芯片平台，来控制是否enable snoop。那么所谓snoop的意义是什么？

### x.6 Makefile 与 Kconfig 编写说明

1）Makefile

内核的Makefile分为5个组成部分：

1. 最顶层的Makefile

2. 内核的.config配置文件

3. 在arch/$(ARCH) 目录下的体系结构相关的Makefile

4. 在s目录下的 Makefile.\* 文件，是一些Makefile的通用规则

5. 各级目录下的大概约500个kbuild Makefile文件

一个典型可执行的Makefile文件如下：KERNELRELEASE在刚进来时并未定义，走到后面，根据参数决定是defalut还是clean，如果是default，那么先KDIR中读取Makefile，其中KERNELRELEASE被定义；然后回到PWD当前目录继续执行，这时则执行obj-m := demo.o，

ifneq ($(KERNELRELEASE),)

obj-m := demo.o

else

KVERS:= $(shell uname -r)

KDIR := /lib/modules/$(KVERS)/build

default:

$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:

rm -f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers

endif

2）Kconfig

一个典型的内核配置菜单如下：

menu "Network device support"

config NETDEVICES

bool "Enable Net Devices"

depends on NET

default y

select NOTHER\_DEV

help

This is help desciption。

...

endmenu

包含在menu/endmenu中的内容会成为Network device support的子菜单。每一个子菜单项都是由config来定义的。congfig下方的那些bool、depends on、default、help等为config的属性，用于定义该菜单项的类型、依赖项、默认值、帮助信息等，select 用于配置相关联另一个选项。

每个config菜单项都要有类型定义: bool布尔类型、 tristate三态(内建、模块、移除)、string字符串、 hex十六进制、 integer整型。例如：

config HELLO\_MODULE

bool "hello test module"

bool 类型的只能选中或不选中，显示为[ ]; tristate类型的菜单项多了编译成内核模块的选项，显示为< > , 假如选择编译成内核模块，则会在.config中生成一个 CONFIG\_HELLO\_MODULE=m的配置，假如选择内建，就是直接编译成内核影响，就会在.config中生成一个 CONFIG\_HELLO\_MODULE=y的配置. hex十六进制类型显示为（ ）。

在Kconfig中有类似语句：source "drivers/usb/Kconfig"



用来包含（或嵌套）新的Kconfig文件，这样便可以使各个目录管理各自的配置内容，不必把那些配置都写在同一个文件里，方便修改和管理。

## end of this file