**Linux Driver**

**学习笔记**

版本控制信息

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 拟稿和修改 | 说明 |
| 0.0 | 2017-09-27 | Hans Hu | 创建 |
| 0.1 | 2017-11-22 | Hans Hu | 上传至git |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[1 前言 - 5 -](#_Toc499108450)

[1.1 编写目的 - 5 -](#_Toc499108451)

[1.2 文档概述 - 5 -](#_Toc499108452)

[2 设备驱动注册 - 6 -](#_Toc499108453)

[2.1 注册入口 - 6 -](#_Toc499108454)

[2.2 设备注册 - 7 -](#_Toc499108455)

[2.2.1 xxxxx - 7 -](#_Toc499108456)

[2.2.2 xxxxx - 7 -](#_Toc499108457)

[2.3 驱动注册 - 7 -](#_Toc499108458)

[2.3.1 总线驱动注册 - 7 -](#_Toc499108459)

[2.3.2 设备驱动注册 - 11 -](#_Toc499108460)

[3 HDAC Spec - 17 -](#_Toc499108461)

[3.1 Architecture - 17 -](#_Toc499108462)

[3.1.1 HDA Architecture - 17 -](#_Toc499108463)

[3.1.2 Codec Architecture - 18 -](#_Toc499108464)

[3.1.3 Link Architecture - 19 -](#_Toc499108465)

[3.1.4 Singnal on Link - 20 -](#_Toc499108466)

[3.2 Frame Composition - 20 -](#_Toc499108467)

[3.2.1 Output Frame - 21 -](#_Toc499108468)

[3.2.2 Input Frame - 24 -](#_Toc499108469)

[3.3 Sample Rates - 26 -](#_Toc499108470)

[3.4 Codec Initialization - 27 -](#_Toc499108471)

[3.5 xxx - 27 -](#_Toc499108472)

[3.6 xxx - 28 -](#_Toc499108473)

[4 ALSA Driver - 29 -](#_Toc499108474)

[4.1 azx\_probe - 29 -](#_Toc499108475)

[4.2 hda\_codec\_driver\_probe - 29 -](#_Toc499108476)

[4.3 数据处理分析 - 29 -](#_Toc499108477)

[4.3.1 发送命令 - 29 -](#_Toc499108478)

[4.3.2 接收中断 - 29 -](#_Toc499108479)

[5 Mini\_system制作 - 31 -](#_Toc499108480)

[5.1 Mini\_system结构 - 31 -](#_Toc499108481)

[5.2 修改OS支持的initrd大小 - 31 -](#_Toc499108482)

[5.2.1 initrd生成 - 31 -](#_Toc499108483)

[5.2.2 配置bzImage - 31 -](#_Toc499108484)

[5.3 ALSA lib & utils 的安装与使用 - 31 -](#_Toc499108485)

[5.3.1 安装 - 31 -](#_Toc499108486)

[5.3.2 使用 - 31 -](#_Toc499108487)

[6 常用命令 - 34 -](#_Toc499108488)

[6.1 复制&查找 - 34 -](#_Toc499108489)

[6.1.1 查找 - 34 -](#_Toc499108490)

[6.1.2 复制 - 34 -](#_Toc499108491)

[6.2 创建 - 34 -](#_Toc499108492)

[6.3 磁盘管理 - 34 -](#_Toc499108493)

[6.4 PCI设备管理 - 34 -](#_Toc499108494)

[6.4.1 查看：lspci - 34 -](#_Toc499108495)

[6.4.2 设置：setpci - 34 -](#_Toc499108496)

[7 Linux内核相关 - 35 -](#_Toc499108497)

[7.1 current 宏 - 35 -](#_Toc499108498)

[8 附录 - 36 -](#_Toc499108499)

[8.1 xxxx - 36 -](#_Toc499108500)

[8.2 xxxxx - 36 -](#_Toc499108501)

[8.3 xxxxx - 36 -](#_Toc499108502)

# 前言

## 编写目的

本文主要记录HDAC Driver的学习内容，并以HDAC为切入点，学习Linux Driver的架构，并最终进入到内核的学习。记录的知识点方便日常使用的时候进行查看，也方便自己在回顾时，可以检阅自己在过去某一时刻记录时的知识掌握情况以及理解程度。

## 文档概述

本文档包含：

1. HDAC Spec
2. ALSA Driver源码
3. Linux Device Driver通用知识
4. ……

# 设备驱动注册

Linux Driver通用知识

## 注册入口

常见设备驱动注册函数是module\_init()，它是系统定义的kernel初始化时的入口种类之一。kernel初始化时对不同级别功能程序做了分类、识别方法，级别从0到7s共15个级别，数字越小越先初始化。其中module\_init是对device\_initcall的宏定义，因此级别为6，很靠后了，红色部分为常用的。本章就是以这四种常用的入口进行分析。

#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) // pci-driver.c pci\_driver\_init bus

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

#define arch\_initcall(fn) \_\_define\_initcall(fn, 3) // arch\x86\pci\init.c pci\_arch\_init dev

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

#define subsys\_initcall(fn) \_\_define\_initcall(fn, 4) // hda\_bus\_type.c / legacy.c hda\_bus\_init/pci\_subsys\_init bus

#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) // sound\pci\hda\hda\_intel.c azx\_driver driver

#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 \_\_initcall(fn) device\_initcall(fn)

1）常见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。module\_###\_driver的定义一般来自系统路径\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，已知驱动的注册入口为module\_init宏，因此调用module\_###\_driver同样完成了的设备模块的构造，驱动具体注册过程则由module\_driver函数中的\_\_register, \_\_unregister实现。

2）\*\*\*\_driver为具体设备的驱动函数入口结构体，该结构体作为\_\_register()的参数传入。以声卡驱动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。这些函数的实现在ALSA Driver一章展开。

3）常见总线驱动入口为postcore\_initcall、subsys\_initcall。其初始化级别为2和4，比级别6的具体设备module\_init要先初始化。这两个函数入口一般都是直接调用bus\_register()函数进行初始化，因此本章对于总线的初始化就以bus\_register函数展开。

4）常见的设备检测程序入口为arch\_initcall()，其初始化级别为3，低于postcore\_initcall但是高于subsys\_initcall和module\_init。以HDAC为例说明实际应用中的分配，HDAC用到了这四种初始化，其中HDA Controller是PCI\_BUS设备，HDA Codec是通过HDA Link挂接在HDA Controller上的下一级HDA\_BUS设备。

弄清初始化顺序对于理解设备是如何注册的前提，目前来看，几种设备对象的初始化顺序为：PCI\_Bus → PCI\_Dev → HDA\_Bus → PCI\_Dev\_Drv (controller + codec)。其中PCI\_Dev分为两步注册，一是通用部分注册PCI\_Dev，二是私有部分注册controller。

## 设备注册

主要来自这篇博客： <http://blog.csdn.net/lizuobin2/article/details/51828594>

### xxxxx

设备检测注册入口为arch\_initcall()，

### xxxxx

文本

## 驱动注册

### 总线驱动注册

#### 注册流程

无论是PCI、I2C、SPI、Platform还是自定义的总线，系统都将总线作为一个设备模型进行单独注册。bus\_type是系统给出的一个通用bus type模型，不同的bus根据需要初始化对应成员即可。总线的注册就是初始化一个总线对应该结构的实例。初始化过程会调用一个通用函数bus\_register()，源码如下：

int bus\_register(struct bus\_type \*bus)

{

int retval;

struct subsys\_private \*priv;

struct lock\_class\_key \*key = &bus->lock\_key;

priv = kzalloc(sizeof(struct subsys\_private), GFP\_KERNEL);//分配私有数据，重要！重要！重要！

if (!priv)

return -ENOMEM;

priv->bus = bus;

bus->p = priv;

BLOCKING\_INIT\_NOTIFIER\_HEAD(&priv->bus\_notifier);

retval = kobject\_set\_name(&priv->subsys.kobj, "%s", bus->name);

if (retval)

goto out;

priv->subsys.kobj.kset = bus\_kset;

priv->subsys.kobj.ktype = &bus\_ktype;

priv->drivers\_autoprobe = 1;

retval = kset\_register(&priv->subsys);

if (retval)

goto out;

retval = bus\_create\_file(bus, &bus\_attr\_uevent);

if (retval)

goto bus\_uevent\_fail;

priv->devices\_kset = kset\_create\_and\_add("devices", NULL,

&priv->subsys.kobj);

if (!priv->devices\_kset) {

retval = -ENOMEM;

goto bus\_devices\_fail;

}

priv->drivers\_kset = kset\_create\_and\_add("drivers", NULL,

&priv->subsys.kobj);

if (!priv->drivers\_kset) {

retval = -ENOMEM;

goto bus\_drivers\_fail;

}

INIT\_LIST\_HEAD(&priv->interfaces);

\_\_mutex\_init(&priv->mutex, "subsys mutex", key);

klist\_init(&priv->klist\_devices, klist\_devices\_get, klist\_devices\_put);

klist\_init(&priv->klist\_drivers, NULL, NULL);

retval = add\_probe\_files(bus);

if (retval)

goto bus\_probe\_files\_fail;

retval = bus\_add\_groups(bus, bus->bus\_groups);

if (retval)

goto bus\_groups\_fail;

pr\_debug("bus: '%s': registered\n", bus->name);

return 0;

bus\_groups\_fail:

remove\_probe\_files(bus);

bus\_probe\_files\_fail:

kset\_unregister(bus->p->drivers\_kset);

bus\_drivers\_fail:

kset\_unregister(bus->p->devices\_kset);

bus\_devices\_fail:

bus\_remove\_file(bus, &bus\_attr\_uevent);

bus\_uevent\_fail:

kset\_unregister(&bus->p->subsys);

out:

kfree(bus->p);

bus->p = NULL;

return retval;

}

#### ###\_bus\_type模块

对任意一种总线实例，都需要驱动编写着提供一个bus type的基本实例，仍以pci\_bus\_type这个结构体为例，该实例提供了一部分总线驱动函数，如下：

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\_groups = pci\_dev\_groups,

.bus\_groups = pci\_bus\_groups,

.drv\_groups = pci\_drv\_groups,

.pm = PCI\_PM\_OPS\_PTR,

};

pci\_bus\_type的设备模型是以postcore\_initcall(pci\_driver\_init);为入口开始注册的（pci-driver.c），\_bus\_type结构体给出了pci\_bus的驱动注册函数，pci总线的注册主要是完成这些函数的绑定而非调用。pci设备注册到总线时则是调用该结构体的成员函数。结合目前需要，主要分析一下.match和.probe成员。

1）.match对应的函数pci\_bus\_match，其原型如下：

static int pci\_bus\_match(struct device \*dev, struct device\_driver \*drv)

{

struct pci\_dev \*pci\_dev = to\_pci\_dev(dev);

struct pci\_driver \*pci\_drv;

const struct pci\_device\_id \*found\_id;

if (!pci\_dev->match\_driver)

return 0;

pci\_drv = to\_pci\_driver(drv);

found\_id = pci\_match\_device(pci\_drv, pci\_dev);

if (found\_id)

return 1;

return 0;

}

static const struct pci\_device\_id \*pci\_match\_device(struct pci\_driver \*drv,

struct pci\_dev \*dev)

{

struct pci\_dynid \*dynid;

const struct pci\_device\_id \*found\_id = NULL;

/\* When driver\_override is set, only bind to the matching driver \*/

if (dev->driver\_override && strcmp(dev->driver\_override, drv->name))

return NULL;

/\* Look at the dynamic ids first, before the static ones \*/ //首先通过dynid来匹配

spin\_lock(&drv->dynids.lock);

list\_for\_each\_entry(dynid, &drv->dynids.list, node) {

if (pci\_match\_one\_device(&dynid->id, dev)) {

found\_id = &dynid->id;

break;

}

}

spin\_unlock(&drv->dynids.lock);

if (!found\_id)

found\_id = pci\_match\_id(drv->id\_table, dev);//此处传入了device设备和pci\_driver的id\_table，此处就是azx\_ids

/\* driver\_override will always match, send a dummy id \*/

if (!found\_id && dev->driver\_override)

found\_id = &pci\_device\_id\_any;

return found\_id;

}

const struct pci\_device\_id \*pci\_match\_id(const struct pci\_device\_id \*ids,

struct pci\_dev \*dev)

{

if (ids) {

while (ids->vendor || ids->subvendor || ids->class\_mask) {

if (pci\_match\_one\_device(ids, dev))//匹配成功则返回

return ids;

ids++;

}

}

return NULL;

}

static inline const struct pci\_device\_id \*

pci\_match\_one\_device(const struct pci\_device\_id \*id, const struct pci\_dev \*dev)

{

if ((id->vendor == PCI\_ANY\_ID || id->vendor == dev->vendor) &&

(id->device == PCI\_ANY\_ID || id->device == dev->device) &&

(id->subvendor == PCI\_ANY\_ID || id->subvendor == dev->subsystem\_vendor) &&

(id->subdevice == PCI\_ANY\_ID || id->subdevice == dev->subsystem\_device) &&

!((id->class ^ dev->class) & id->class\_mask))

return id;

return NULL;

}

该函数的功能是实现设备与驱动之间的匹配。函数的传入参数为device和device\_driver两个结构体，从名字可以看出，其与具体的总线（PCI、I2C、SPI…）无关。但是其内部则利用to\_pci\_dev(dev);和to\_pci\_driver(drv);两个函数索引到了pci\_dev和pci\_driver两个结构体，而这两个结构体则是我们编写pci设备驱动时注册函数的入口参数(dev&drv)。最后通过比较驱动和设备的vendor 、subvendor 、device 、subdevice 来匹配，匹配成功则返回设备匹配到的驱动id。

2）.probe对应的pci\_device\_probe()，原型如下：

static int pci\_device\_probe(struct device \*dev)

{

int error;

struct pci\_dev \*pci\_dev = to\_pci\_dev(dev);

struct pci\_driver \*drv = to\_pci\_driver(dev->driver);

error = pcibios\_alloc\_irq(pci\_dev);

if (error < 0)

return error;

pci\_dev\_get(pci\_dev);

error = \_\_pci\_device\_probe(drv, pci\_dev);

if (error) {

pcibios\_free\_irq(pci\_dev);

pci\_dev\_put(pci\_dev);

}

return error;

}

static int \_\_pci\_device\_probe(struct pci\_driver \*drv, struct pci\_dev \*pci\_dev)

{

const struct pci\_device\_id \*id;

int error = 0;

if (!pci\_dev->driver && drv->probe) {// pci\_dev->driver为空（说明该设备的驱动未注册）并且drv->probe不空（说明pci设备必须给出probe成员）

error = -ENODEV;

id = pci\_match\_device(drv, pci\_dev);//在.match部分中已分析

if (id)

error = pci\_call\_probe(drv, pci\_dev, id);

if (error >= 0)

error = 0;

}

return error;

}

static int pci\_call\_probe(struct pci\_driver \*drv, struct pci\_dev \*dev,

const struct pci\_device\_id \*id)

{

int error, node;

struct drv\_dev\_and\_id ddi = { drv, dev, id };

/\*

\* Execute driver initialization on node where the device is

\* attached. This way the driver likely allocates its local memory

\* on the right node.

\*/

node = dev\_to\_node(&dev->dev);

if (node >= 0 && node != numa\_node\_id()) {

int cpu;

get\_online\_cpus();

cpu = cpumask\_any\_and(cpumask\_of\_node(node), cpu\_online\_mask);

if (cpu < nr\_cpu\_ids)

error = work\_on\_cpu(cpu, local\_pci\_probe, &ddi);

else

error = local\_pci\_probe(&ddi);

put\_online\_cpus();

} else

error = local\_pci\_probe(&ddi);

return error;

}

static long local\_pci\_probe(void \*\_ddi)

{

struct drv\_dev\_and\_id \*ddi = \_ddi;

struct pci\_dev \*pci\_dev = ddi->dev;

struct pci\_driver \*pci\_drv = ddi->drv;

struct device \*dev = &pci\_dev->dev;

int rc;

pm\_runtime\_get\_sync(dev);

pci\_dev->driver = pci\_drv;

rc = pci\_drv->probe(pci\_dev, ddi->id);

if (!rc)

return rc;

if (rc < 0) {

pci\_dev->driver = NULL;

pm\_runtime\_put\_sync(dev);

return rc;

}

dev\_warn(dev, "Driver probe function unexpectedly returned %d\n", rc);

return 0;

}

最终调用的pci\_drv->probe(pci\_dev, ddi->id);就是hda\_intel.c文件中提供的入口函数static int azx\_probe(struct pci\_dev \*pci,const struct pci\_device\_id \*pci\_id)。

### 设备驱动注册

由2.1节已知驱动具体注册过程则由module\_driver函数中的\_\_register, \_\_unregister实现。本节关于设备驱动注册就从\_\_register对应函数开始分析。

#### ###\_register\_driver分析

仍以pci\_register\_driver(azx\_driver)为例，\_\_register 对应的\_\_pci\_register\_driver源码为：

int \_\_pci\_register\_driver(struct pci\_driver \*drv, struct module \*owner,

const char \*mod\_name)

{

/\* initialize common driver fields \*/

drv->driver.name = drv -> name;

drv->driver.bus = &pci\_bus\_type;

drv->driver.owner = owner;

drv->driver.mod\_name = mod\_name;

spin\_lock\_init(&drv->dynids.lock);

INIT\_LIST\_HEAD(&drv->dynids.list);

/\* register with core \*/

return driver\_register(&drv->driver);

}

该函数主要分为对drv->driver的赋值和注册两部分，比较重要的是drv->driver.bus = &pci\_bus\_type;和driver\_register(&drv->driver);这两句。

###\_bus\_type参见2.3.1.2节。

driver\_register()

通过查看pci、i2c、spi等不同总线的驱动注册函数###\_register\_driver()，可以看到，最后都是调用了driver\_register函数，顾名思义，注册带有总线的驱动（还有不带总线了喽？），源码如下：

/\*\*

\* driver\_register - register driver with bus

\* @drv: driver to register

\*

\* We pass off most of the work to the bus\_add\_driver() call,

\* since most of the things we have to do deal with the bus

\* structures.

\*/

int driver\_register(struct device\_driver \*drv)

{

int ret;

struct device\_driver \*other;

//私有数据是在总线初始化时分配，而总线初始化级别高于设备，因此此处私有数据是必然存在的，，否则应该报个错的。。。难道有例外？

BUG\_ON(!drv->bus->p);

//这个我理解的比较复杂，drv->bus->probe其实和dev->bus->probe对应的，一般在设备注册的时候已经给dev->bus赋了probe等值。在本程序后续的判断中会有一个dev->bus->probe和drv->probe二选一的过程，而且是优先选择dev->bus->probe。如果dev->bus->probe存在，而drv->probe又存在，这说明写驱动人不想用dev->bus->probe而是想使用自定义的drv->probe，但实际上后续的优先选择机制又不让你用……所以这里给出了一个警告，就是说如果你不用我给你提供的默认probe，就说明你想更新旧的，那我建议你还是采用修改bus\_type的方式吧，，，否则，发现没变化时，别怪我没给你警告！

if ((drv->bus->probe && drv->probe) ||

(drv->bus->remove && drv->remove) ||

(drv->bus->shutdown && drv->shutdown))

printk(KERN\_WARNING "Driver '%s' needs updating - please use "

"bus\_type methods\n", drv->name);

//每注册一个驱动，会在bus结构的私有数据上添加一个以名字为记的索引。对比，如已注册则中断，并给出错误。问题在于为什么是给出错误而不是警告？再猜一下，这个不像前面的更新驱动，实际中如果发生这种情况，说明不同模块采用了相同的名字（或重复加载？）这……多么明显的一个错误。

other = driver\_find(drv->name, drv->bus);

if (other) {

printk(KERN\_ERR "Error: Driver '%s' is already registered, "

"aborting...\n", drv->name);

return -EBUSY;

}

ret = bus\_add\_driver(drv);//将driver添加到一个bus上，下面会主要分析

if (ret)

return ret;

ret = driver\_add\_groups(drv, drv->groups); //如果grop不为空的话，将在驱动文件夹下创建以group名字的子文件夹，然后在子文件夹下添加group的属性文件

if (ret) {

bus\_remove\_driver(drv);

return ret;

}

kobject\_uevent(&drv->p->kobj, KOBJ\_ADD);

return ret;

}

分析bus\_add\_driver函数，该函数的原形如下：

int bus\_add\_driver(struct device\_driver \*drv)

{

struct bus\_type \*bus;

struct driver\_private \*priv;

int error = 0;

bus = bus\_get(drv->bus); //增加该bus->p->subsys->kobject->kref的引用计数，bus还是那个bus

if (!bus)

return -EINVAL;

pr\_debug("bus: '%s': add driver %s\n", bus->name, drv->name);

priv = kzalloc(sizeof(\*priv), GFP\_KERNEL); //分配driver\_private结构

if (!priv) {

error = -ENOMEM;

goto out\_put\_bus;

}

klist\_init(&priv->klist\_devices, NULL, NULL); //初始化priv->klist\_devices

priv->driver = drv;

drv->p = priv;

priv->kobj.kset = bus->p->drivers\_kset;

error = kobject\_init\_and\_add(&priv->kobj, &driver\_ktype, NULL, //驱动的kobject初始化和添加dir到sysfs中

"%s", drv->name);

if (error)

goto out\_unregister;

klist\_add\_tail(&priv->knode\_bus, &bus->p->klist\_drivers); //将priv->knode\_bus添加到bus->p->klist\_drivers

if (drv->bus->p->drivers\_autoprobe) {//看来要弄清楚这个p是哪里来的了，即使else了居然也没报错，看来是特殊驱动才有，保证不会出错

if (driver\_allows\_async\_probing(drv)) {//这个放在if位没也要弄清楚才行，先分析else位吧

pr\_debug("bus: '%s': probing driver %s asynchronously\n",

drv->bus->name, drv->name);

async\_schedule(driver\_attach\_async, drv);

} else {

error = driver\_attach(drv);//此行的重点函数

if (error)

goto out\_unregister;

}

}

module\_add\_driver(drv->owner, drv); //添加drv的module

error = driver\_create\_file(drv, &driver\_attr\_uevent); //在sysfs的目录下创建文件uevent属性文件

if (error) {

printk(KERN\_ERR "%s: uevent attr (%s) failed\n",

\_\_func\_\_, drv->name);

}

error = driver\_add\_groups(drv, bus->drv\_groups); //添加组

if (error) {

/\* How the hell do we get out of this pickle? Give up \*/

printk(KERN\_ERR "%s: driver\_create\_groups(%s) failed\n",

\_\_func\_\_, drv->name);

}

if (!drv->suppress\_bind\_attrs) {

error = add\_bind\_files(drv); //添加绑定文件

if (error) {

/\* Ditto \*/

printk(KERN\_ERR "%s: add\_bind\_files(%s) failed\n",

\_\_func\_\_, drv->name);

}

}

return 0;

out\_unregister:

kobject\_put(&priv->kobj);

kfree(drv->p);

drv->p = NULL;

out\_put\_bus:

bus\_put(bus);

return error;

}

这个函数是driver\_register中核心函数，真正的功能实现都在这个函数里面。这个函数首先找到该drv所属的bus，然后为driver\_private结构分配空间，然后初始化priv，把driver，bus，priv联系在一块，然后添加驱动的kobject到kobject的层次中，也就是添加驱动文件夹到sysfs，然后根据drivers\_autoprobe决定是否去bus上寻找与driver匹配的device。然后将driver添加到bus上的驱动列表中。然后添加驱动的模块，再然后就是生成sysfs下面的一些属性文件。

结合目前需要，我们分析driver\_attach()函数，

int driver\_attach(struct device\_driver \*drv)

{

return bus\_for\_each\_dev(drv->bus, NULL, drv, \_\_driver\_attach);//这是循环函数，与bus上的全部设备进行逐一匹配，找到匹配成功的那个dev

}

int bus\_for\_each\_dev(struct bus\_type \*bus, struct device \*start,

void \*data, int (\*fn)(struct device \*, void \*))

{

struct klist\_iter i;

struct device \*dev;

int error = 0;

if (!bus)

return -EINVAL;

klist\_iter\_init\_node(&bus->p->klist\_devices, &i,

(start ? &start->p->knode\_bus : NULL)); //将bus中的已注册的device列表放到迭代器中，列表在私有数据中

while ((dev = next\_device(&i)) && !error) //将驱动逐个地与列表中每一个的device匹配，可能一个驱动匹配好几个设备

error = fn(dev, data); //这个fn就是上面传下来的\_\_driver\_attach，逐个匹配

klist\_iter\_exit(&i);

return error;

}

该函数内主要是执行了\_\_driver\_attach()函数，将挂接在该bus上的dev与drv进行匹配，其参数为匹配到的drv->bus->dev和需要绑定的drv，

static int \_\_driver\_attach(struct device \*dev, void \*data)

{

struct device\_driver \*drv = data;

/\*

\* Lock device and try to bind to it. We drop the error

\* here and always return 0, because we need to keep trying

\* to bind to devices and some drivers will return an error

\* simply if it didn't support the device.

\*

\* driver\_probe\_device() will spit a warning if there

\* is an error.

\*/

if (!driver\_match\_device(drv, dev))//进行drv和dev的匹配，但是还未绑定

return 0;

if (dev->parent) /\* Needed for USB \*/ //对于usb设备需要进行父设备的锁定

device\_lock(dev->parent);

device\_lock(dev);

if (!dev->driver)//如果驱动还未绑定

driver\_probe\_device(drv, dev);//尝试绑定设备和驱动

device\_unlock(dev);

if (dev->parent)

device\_unlock(dev->parent);

return 0;

}

static inline int driver\_match\_device(struct device\_driver \*drv, struct device \*dev)

{

return drv->bus->match ? drv->bus->match(dev, drv) : 1;

}

//此处调用的match函数就是上一节pci\_bus\_type中介绍的pci\_bus\_match，因此，通过该函数实现了回溯，虽然从driver\_register传入的参数为同一的不包含具体驱动信息的结构，但是通过此处的match函数则又重新找到了上一层结构。执行这个函数的前提是drv->bus->match存在。由此可见###\_bus\_type结构体中.match成员存在的重要性。

int driver\_probe\_device(struct device\_driver \*drv, struct device \*dev)

{

int ret = 0;

if (!device\_is\_registered(dev))

return -ENODEV;

pr\_debug("bus: '%s': %s: matched device %s with driver %s\n",

drv->bus->name, \_\_func\_\_, dev\_name(dev), drv->name);

if (dev->parent)

pm\_runtime\_get\_sync(dev->parent);

pm\_runtime\_barrier(dev);

ret = really\_probe(dev, drv);//真正是probe功能函数

pm\_request\_idle(dev);

if (dev->parent)

pm\_runtime\_put(dev->parent);

return ret;

}

static int really\_probe(struct device \*dev, struct device\_driver \*drv)

{

int ret = 0;

int local\_trigger\_count = atomic\_read(&deferred\_trigger\_count);

atomic\_inc(&probe\_count);

pr\_debug("bus: '%s': %s: probing driver %s with device %s\n",

drv->bus->name, \_\_func\_\_, drv->name, dev\_name(dev));

WARN\_ON(!list\_empty(&dev->devres\_head));

dev->driver = drv;//将驱动挂载到对应设备上

/\* If using pinctrl, bind pins now before probing \*/

ret = pinctrl\_bind\_pins(dev);

if (ret)

goto probe\_failed;

if (driver\_sysfs\_add(dev)) {

printk(KERN\_ERR "%s: driver\_sysfs\_add(%s) failed\n",

\_\_func\_\_, dev\_name(dev));

goto probe\_failed;

}

if (dev->pm\_domain && dev->pm\_domain->activate) {

ret = dev->pm\_domain->activate(dev);

if (ret)

goto probe\_failed;

}

/\*

\* Ensure devices are listed in devices\_kset in correct order

\* It's important to move Dev to the end of devices\_kset before

\* calling .probe, because it could be recursive and parent Dev

\* should always go first

\*/

devices\_kset\_move\_last(dev);

if (dev->bus->probe) {

ret = dev->bus->probe(dev); //首先判断bus->probe是否存在，存在的话则执行。

if (ret)

goto probe\_failed;

} else if (drv->probe) { //如果bus->probe不存在但是drv->probe存在，则执行drv->probe。

ret = drv->probe(dev);

if (ret)

goto probe\_failed;

}

if (dev->pm\_domain && dev->pm\_domain->sync)

dev->pm\_domain->sync(dev);

driver\_bound(dev);

ret = 1;

pr\_debug("bus: '%s': %s: bound device %s to driver %s\n",

drv->bus->name, \_\_func\_\_, dev\_name(dev), drv->name);

goto done;

probe\_failed:

devres\_release\_all(dev);

driver\_sysfs\_remove(dev);

dev->driver = NULL;

dev\_set\_drvdata(dev, NULL);

if (dev->pm\_domain && dev->pm\_domain->dismiss)

dev->pm\_domain->dismiss(dev);

switch (ret) {

case -EPROBE\_DEFER:

/\* Driver requested deferred probing \*/

dev\_dbg(dev, "Driver %s requests probe deferral\n", drv->name);

driver\_deferred\_probe\_add(dev);

/\* Did a trigger occur while probing? Need to re-trigger if yes \*/

if (local\_trigger\_count != atomic\_read(&deferred\_trigger\_count))

driver\_deferred\_probe\_trigger();

break;

case -ENODEV:

case -ENXIO:

pr\_debug("%s: probe of %s rejects match %d\n",

drv->name, dev\_name(dev), ret);

break;

default:

/\* driver matched but the probe failed \*/

printk(KERN\_WARNING

"%s: probe of %s failed with error %d\n",

drv->name, dev\_name(dev), ret);

}

/\*

\* Ignore errors returned by ->probe so that the next driver can try

\* its luck.

\*/

ret = 0;

done:

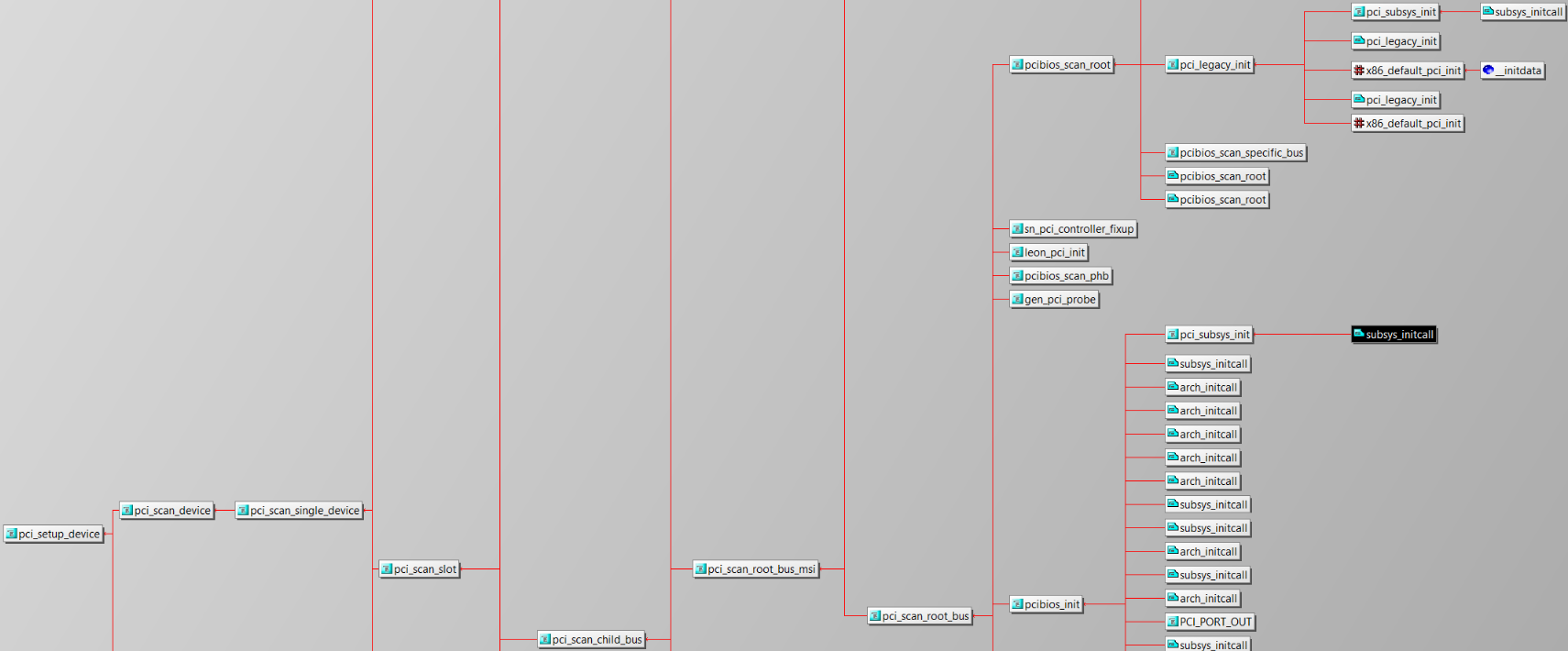
atomic\_dec(&probe\_count);

wake\_up(&probe\_waitqueue);

return ret;

}

一路分析来看，drv->probe并不存在，那么要执行的必然是dev->bus->probe，dev是通过对bus上已注册的全部设备进行vender\_id匹配到的，因此dev->bus->probe应该是在系统检测到device之后进行设置的。通过对前文pci\_bus\_type中probe函数的分析，结合dev->bus的数据类型，不难猜出dev->bus->probe也是通过dev.bus = &pci\_bus\_type;这一赋值操作得到。全面分析暂时还做不到，目前也是仅根据需要分析，通过查找，找到了pci\_setup\_device函数，其内部包含了dev->dev.bus = &pci\_bus\_type;这句话。而pci\_setup\_device()函数往上回溯的调用逻辑如下：



逐个分析又是一个浩大的工程，择日再进行分析吧，但是总结来说，就是在系统进行PCI设备检测时，一旦检测到就已经完成了dev->bus->probe的赋值。

至此，就暂时分析完了系统是如何调用azx\_driver结构体中的成员进行操作的了！

但是，凡事都有个但是不是么？既然系统已经在设备检测时就完成了dev->bus->probe的赋值，那么if(){}else if(){}的意义何在？刚才的分析只是对于PCI设备，对于非PCI设备的驱动是如何实现的？

HDAC作为常规PCI设备，采用本节分析的流程即可实现驱动加载。下一节针对挂接在HDAC上的codec进行分析。

#### HDAC Codec驱动注册

codec实例驱动多以patch文件存在，否则就使用generic.c中的模型。现以patch\_via.c为例，其驱动入口为module\_hda\_codec\_driver(via\_driver)。下面针对\_\_register对应的\_\_hda\_codec\_driver\_register函数进行分析，函数原型如下：

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

struct module \*owner)

{

drv->core.driver.name = name;

drv->core.driver.owner = owner;

drv->core.driver.bus = &snd\_hda\_bus\_type;

drv->core.driver.probe = hda\_codec\_driver\_probe;

drv->core.driver.remove = hda\_codec\_driver\_remove;

drv->core.driver.shutdown = hda\_codec\_driver\_shutdown;

drv->core.driver.pm = &hda\_codec\_driver\_pm;

drv->core.type = HDA\_DEV\_LEGACY;

drv->core.match = hda\_codec\_match;

drv->core.unsol\_event = hda\_codec\_unsol\_event;

return driver\_register(&drv->core.driver);

}

struct bus\_type snd\_hda\_bus\_type = {

.name = "hdaudio",

.match = hda\_bus\_match,

};

可以看出，该\_\_register函数同样绑定了一个bus\_type结构体，因此，与pci\_bus\_type一样，snd\_hda\_bus\_type作为一个总线模型，它也是作为一个总线模块单独注册，使用的注册入口为subsys\_initcall，在文件hda\_bus\_type.c文件中实现，同pci\_bus\_type一样也是调用bus\_register()函数完成。

不同于pci\_bus\_type的是，snd\_hda\_bus\_type中只定义了.name和.match成员。而且对drv->core.driver.probe等成员进行了赋值，那么当程序走到上节really\_probe()函数时，则会执行else部分的ret = drv->probe(dev);语句，也就是此时绑定的hda\_codec\_driver\_probe()函数，因此，对codec分析的入口就是这个函数。

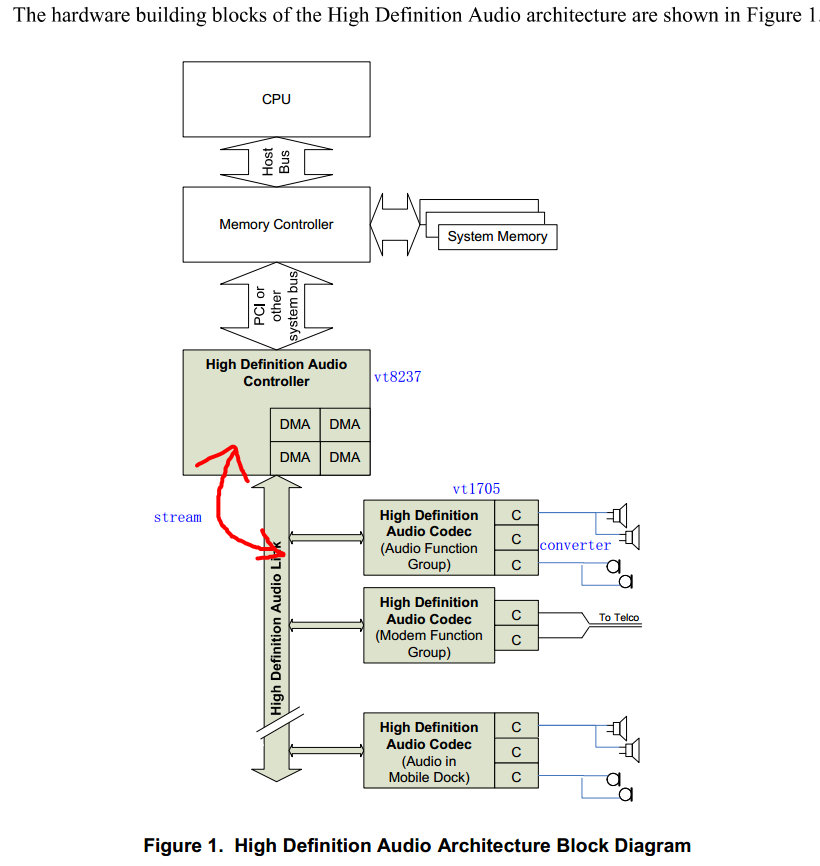
# HDAC Spec

## Architecture

### HDA Architecture



图2



上图1来自HDAC Spec，图2来自T-spec。结合两幅图可以看出，HDA Controller是集成在SB中的一个PCI设备(现SB与NB已集成为ChipSet)。HDA codec作为外接的编解码芯片，通过HDA Link与HDA Controller相连。除了这三样设备，系统架构中相关的主要设备还包括声学装置（麦克风耳机等）、存储器、存储控制器和CPU。其中声卡驱动的开发中，我们主要关注三个对象：controller、codec和link。对于controller、codec是通过寄存器设置来控制，控制过程是通过link的Bus协议实现。

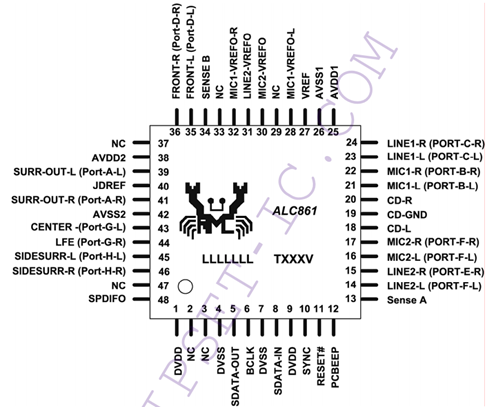
1）controller理论上提供了15路DMA IN和15路DMA OUT，不同厂家的NB芯片提供了不同的个数，我们采用的为4 In 4 Out。controller利用DMA提供了在存储器和codec之间进行数据交换的控制功能。对其控制的协议为PCI，寄存器由PCI-spec和IRS\_CND001\_D20F0\_HDAC\_R050提供，controller寄存器说明由HDAC spec提供。

2）link为controller和codec之间通信的桥梁，提供了硬件连接线路和数据协议。link协议的实现由controller控制，只要提供给controller一些配置参数即可。

3）codec为编解码芯片，主要提供模拟信号的编码与数字信号的解码。每个codec上可以有多路转换器，对应实例的情况就是多声道及播录一体。

### Codec Architecture

下图是一个Realtek的ALC861 codec芯片实例。



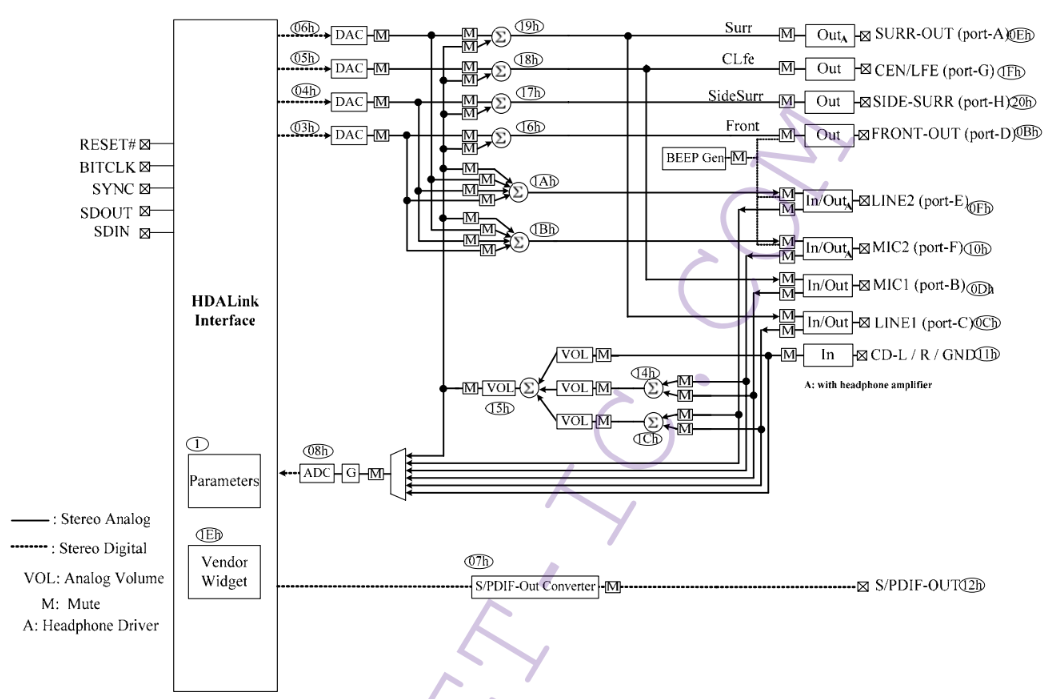
ADCC

DAC

Link

codec按照逻辑功能划分成了多个Widget，给每个部件提供了Node ID，spec或者datasheet提供了这些ID的描述，通过这些NID可以分别实现对每个部件的访问控制，NID长度一般为7bit，特殊codec的ND为15bits。

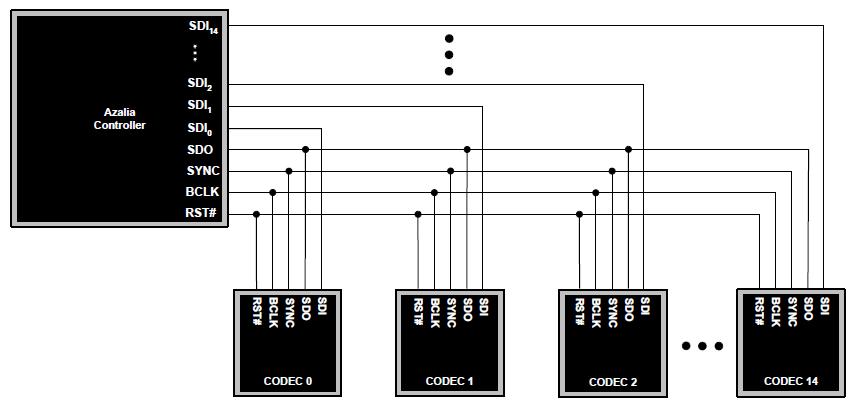
codec内部的Widget的Node ID是唯一的，每个codec的ID也是唯一的，因此利用这两个ID，系通过就可以访问任一Widget。一般厂家提供的codec已经对各Widget的Node ID进行了初始化设置，如下图所示：



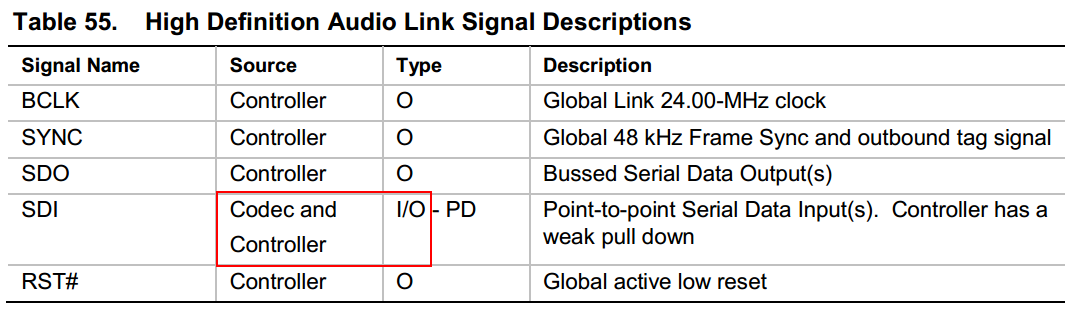
每个Widget的定义及访问情况，参见hdac spec chapter 7，或者ALC861 Datasheet chapter 8

### Link Architecture

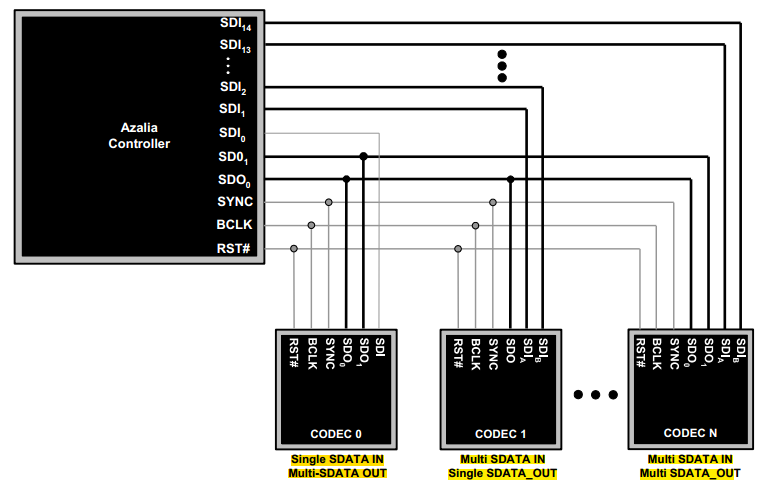
controller与codec之间是一对多的关系，二者之间基本的控制通信线连接情况如下图所示。



从图中不难看出，controller的输出为广播模式，而输入为单播模式，对于不同模式下他们是如何识别，DMA如何分配，后面再讲。图中各个连接线的引脚含义如下表所示，Type是相对于controller：

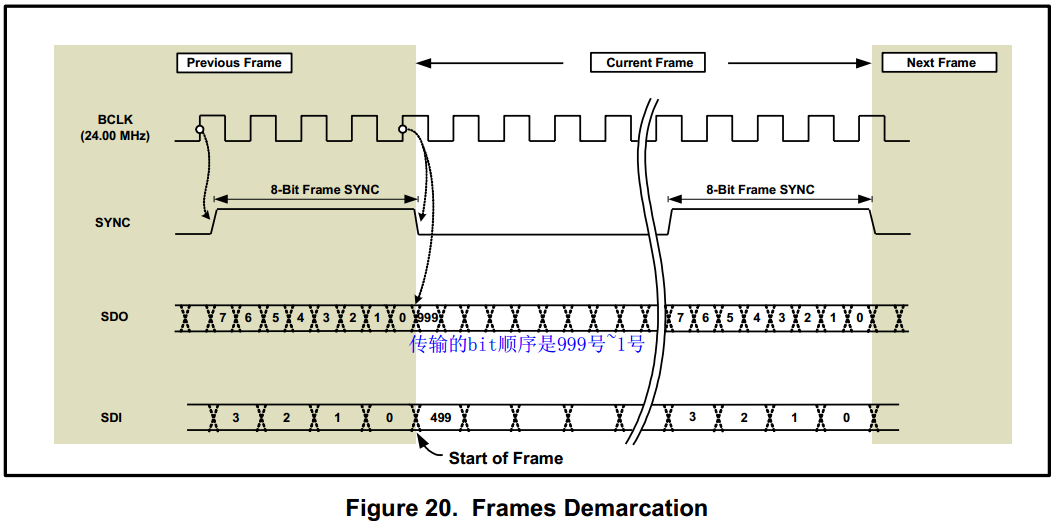


除了基本的接线方式外，还有一种拓展方式，如下图所示。



图中可以看出，输入和输出均可以进行扩展，但是无论如何至少要有SDO0和一路SDI与codec相连，保证基本的cmd和response。

### Singnal on Link



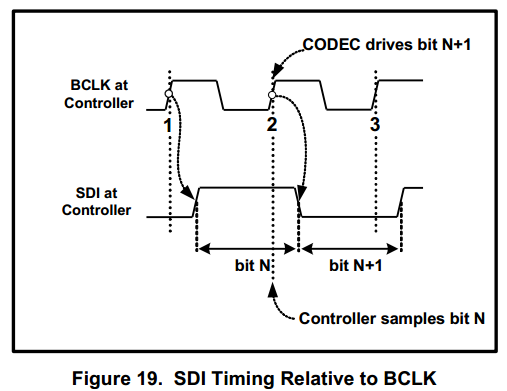
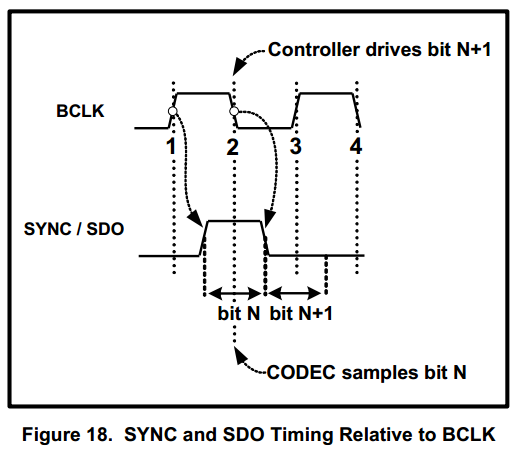
1）线路上数据传输时钟信号来自BCLK：Bit Clock。由系统控制固定输出的24M信号，是其他信号的基础时钟。

2）SYNC是一个固定周期为48KHz的同步信号。计算24M/48k = 500，可得每个周期内包含500个BCLK周期，而如果每个变化沿算一个信号，那么就会产生1000个信号位。SYNC的同步信号依据的是变化沿，每个信号包含8个位。数据传输时就是以SYNC为一个周期进行的。每帧数据的开始信号为同步信号的下降沿。

3）SDO是以变化沿为信号，因此每个周期内可包含1000数据位。

4）SDI是以上升沿为信号，因此每个周期内可包含500数据位。

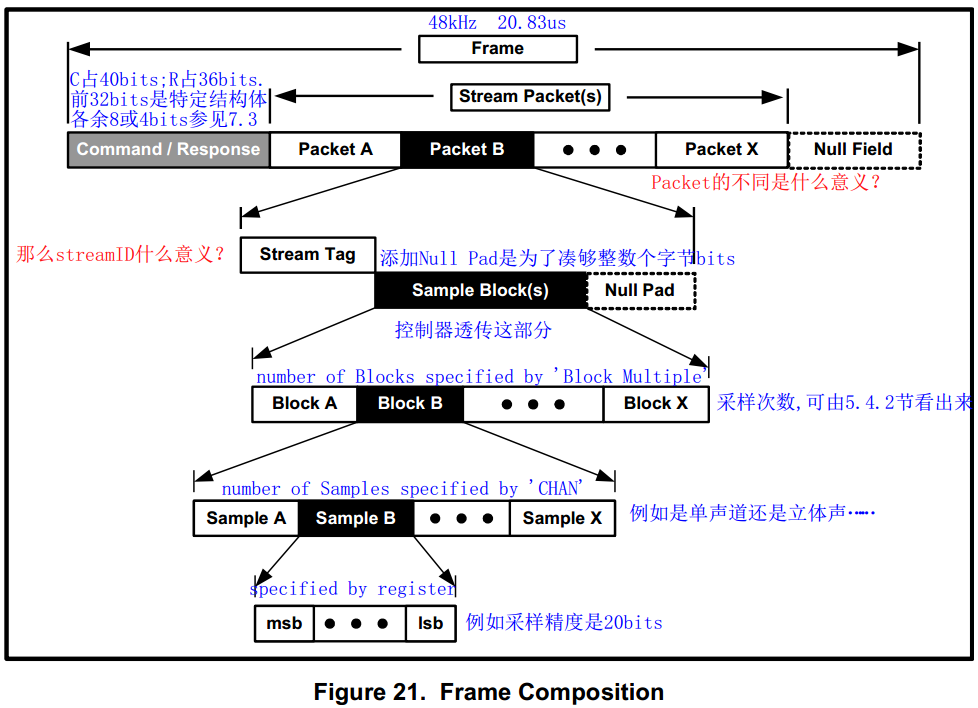
时序图如下图所示：



## Frame Composition

每帧数据按逻辑划分成了三个不同的组件。如下图所示，分别为：

A single Command/Response Field  
Zero or more Stream Packets  
A Null Field to fill out the frame



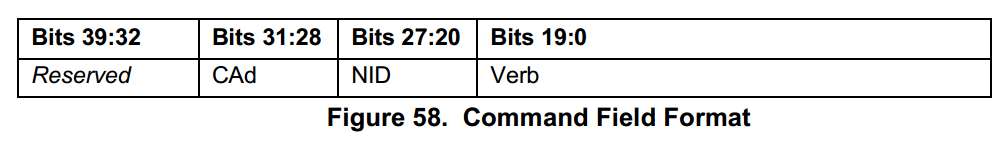
一个帧周期为48KHz，窗口时间为1 / 48k = 20.833us。其中Command/Response Field指示的是该数据的传输方向以及后面数据的含义，例如该CMD是对codec寄存器的读写操作还是对音频数据的传输，该Field为必选项，本节分析的均是音频数据的传输；Stream Packets为可由可无，根据要Command/Response内容决定；Null Field为补零域，以助填满1000或500 bits。

下面以Output和Input两种情况对这些组件功能进行介绍。

### Output Frame

#### Command Field

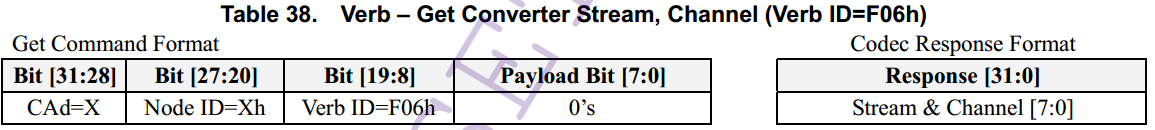
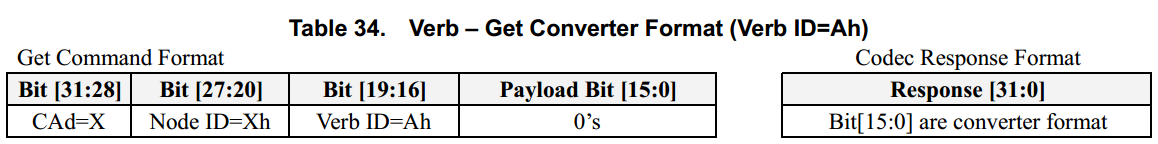
cmd域包含40个位，其结构如下图



1）CAd为4-bit codec address；因为每个stream有且仅有一个Command Field，因此每个stream只能作用于一个codec，Command Frame使用广播模式，但codec会根据CAd决定要不要响应。

2）NID为an 8-bit node ID that identifies the target node within the codec，这是由Codec又细分为多个节点组件所致，例如AD是一个Node，DA也是一个Node；一个codec可能包含多个Node，每个Node对应一个NID；每个codec里的NID唯一，而且一般都是在codec datasheet中分配好了。

3）Verb就是要操作的内容，Verb内分为Verb ID和Payload两个，Verb的格式又分为两种，如下两个实例：



可以看出，主要是依据Verb ID长度为12bits和4bits划分成两种，这样分的原因也很简单，就是有一些需要传送16bits的Payload使用Verb ID长度为4 bits格式，这种格式并不常用。小技巧：codec操作的内容，一般分为Get和Set两种，对于Verb ID长度为16 bits格式，Get操作的Verb ID为0xF##H格式，Set操作的Verb ID为0x7##H格式。这也导致4bits格式的Verb ID不能使用0x7和0xF。

4）存储上，cmd域包含的40bit中，有效字节数位后面连续的32bit，刚好与CORB Entry大小相等。直接存储

#### Stream Packets

从Frame结构看，每个Stream包含多个Packets，每个Packets包括一个stream\_tag和多个block，每个block又包含多个sample，每个sample包含多个数据位。下面从下往上分别从物理意义上解释：

1）sample/block

采样值，例如采样精度为20 bits时，每个sample就是20 bits。spec上说明了HDAC支持的8-, 16-, 20-, 24-, and 32-bit sample resolution per stream。多个sample表示的是多声道采样或者播放，例如stereo有2声道，就有两个sample。每次在多声道上采集的数据组成一个block。

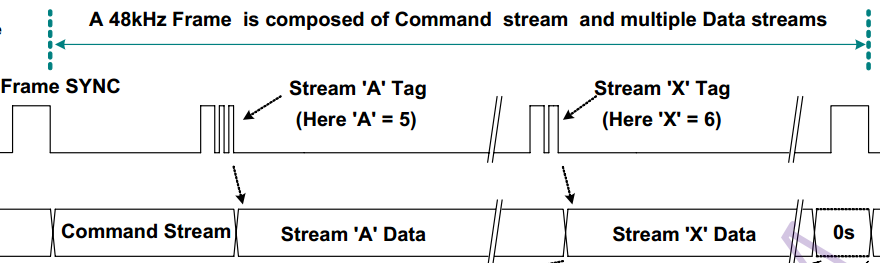
2）block/sample blocks

采样点，每个block代表一个周期（48kHz）内的采样次数。spec上说明了HDAC Sample rate support ranging from 6 kHz to 192 kHz，意味着支持在一个周期内的采样次数可以是从1/8 ~ 4个。这是指的采用一条line。当codec的采样速度很高>192 kHz时，就要多line支持了；播放时，一般HDAC只有2条output line，因此最多支持的rate为384kHz，这些都是冷知识的感觉……每个周期内多个block组成一个sample blocks。

3）Packet/stream\_tag

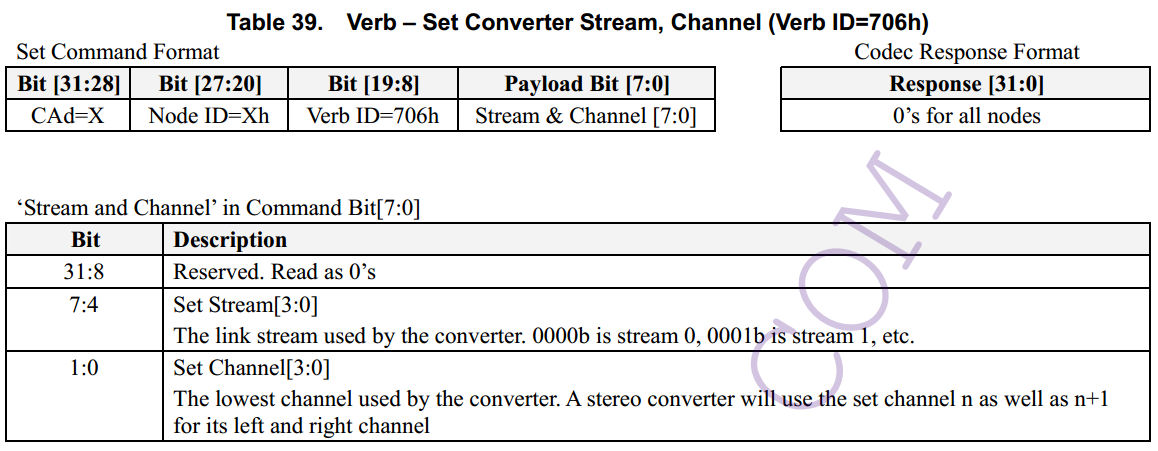
每一组sample blocks 加上一个stream\_tag组成一个Packet，Packet的物理意义表示的是每个codec上的converter或者S/PDIF的数据包。以ALC861为例，它有4个DAC、1个ADC、1个S/PDIF。这里需要注意的是，多个converter与多声道并不完全一回事，比如一个DAC即可实现两声道的立体声播放，但是影院使用的7.1声道则是利用了4个DAC。虽然codec每次传送一条bit数据流，但是实际是传送了多个逻辑数据流，通过stream\_ID与NID绑定，每个Packet对应不同的逻辑数据流（目前只是猜测，代码中应该有体现，还没看到），下面所说的stream均指逻辑数据流。

每个Packet对应一个stream\_tag，每个stream\_tag分为两部分，第一部分为4bit的前导码，第二部分为4bit的stream\_ID，其中前导码为4位高电平。Output Frame中stream\_tag是在SYNC信号线上传送，第一个Packet的stream\_tag是在Command Field的后8bit对应的位置上，其后每个Packet的stream\_tag都是在前一个Packet的后8bit对应位置。如下图所示：



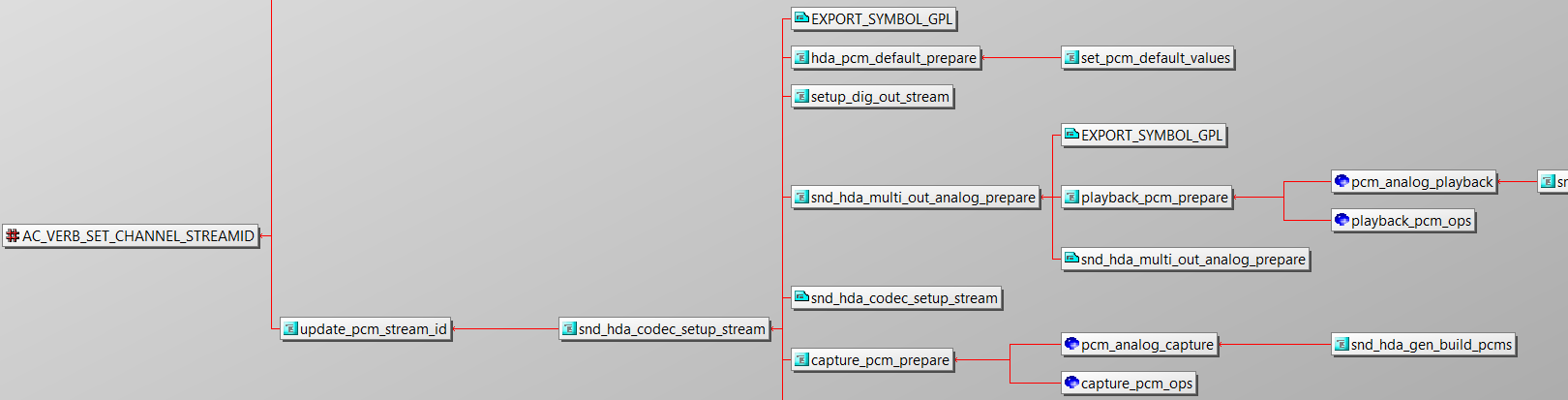
每个Packet均对应一个stream\_ID，4bit stream\_ID的范围为1~15。stream\_ID的分配是在stream初始化函数中进行设置，通过azx\_init\_streams()函数可以看出，stream\_ID就是与流对应的一个标记。标记的方法有两个，一个是根据chip中的所有stream顺序，从1开始递增计数；另一种方法是将所有的stream分成两种，即play与capture，分别计数。从源码上看，系统支持的stream总数量由chipset芯片决定，普通的芯片为4路capture和4路playback。此处stream\_ID最大为15，与HDA Controller支持15个codec并没有任何关系，二者都是15纯属巧合。一般一个codec用不到那么多stream，以ALC861为例，4个DAC和1个ADC占5个stream，1个S/PDIF也占不了几个。

stream\_ID驱动注册时根据chipset芯片决定，它与converter的一一对应是利用一条控制Output Frame实现的，对应的Verb ID为0x706，其命令格式如下图所示：



这是一条Set命令，对应Get命令的VerbID=F06H。命令的含义是设置convertor的声道编号和stream\_ID，其中Payload的bit 7:4为stream\_ID，bit 1:0为声道编号。如果为双声道，则此编号n为左声道，n+1为右声道。这个编号在代码中的体现还未看到，尚属猜测。

我们的PC都有外部功放和耳机两个播放口，我们播放时系统根据是否插入耳机决定是是用哪个口。这里的两个口看着好像是通过硬件即可实现，但是实际上两个口是使用了两个不同的DAC，耳机插入仅是作为一个DAC切换的信号。由软件来实现切换，两种方案可以实现：1、待播放音频数据是存储在steam中的，只要利用上述Set Converter stream命令将两个converter NID对应的stream ID调换一下即可。2、将待播放的音频数据放到切换后的converter对应的stream中，那么DMA将传输该stream中的数据。具体是哪一种不太确定，这部分可能是在ALSA Lib源码中实现。但是通过通过追踪Set Verb=AC\_VERB\_SET\_CHANNEL\_STREAMID（706H）可以看到ALSA Driver中提供了update\_pcm\_stream\_id底层函数以及hda\_pcm\_default\_prepare默认设置函数。同时提供的capture\_pcm\_prepare和两个playback\_pcm\_prepare两个函数中均调用了update\_pcm\_stream\_id函数。暂时认为是第一种情况。

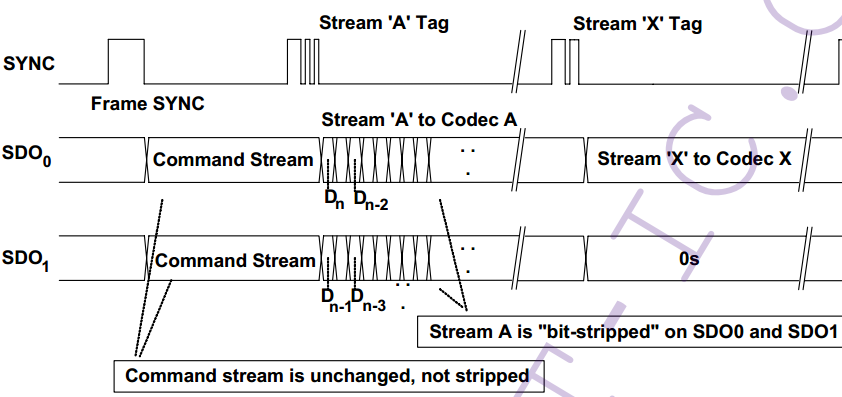


#### Multiple SDO

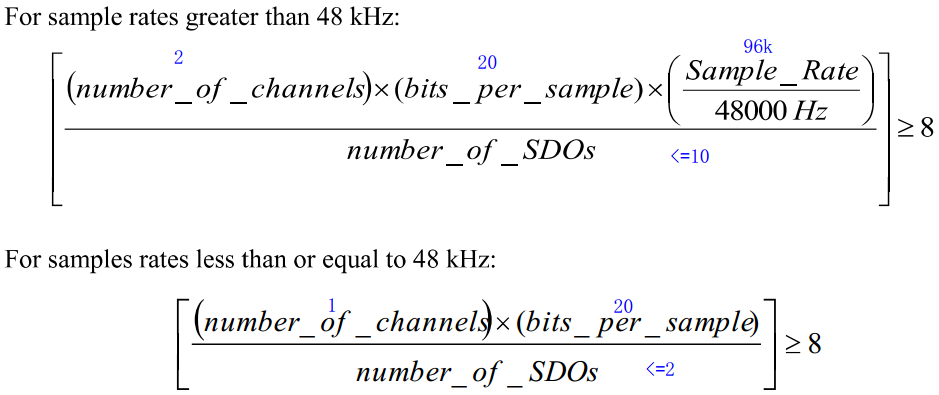
多条SDO线的情况，只用在专业设备上，普通的设备上只用一条SDO足以，此处也分析一下，便于阅读Driver Code。硬件接线情况可参见3.1.2节“controller与codec”。要注意的地方有这几处：

1）多条线的情况需要使能Output Stream Descriptor Control register的Stripe Control位。

2）每个SDO上的Command Field以及Stream Tag一样，但是Packet数据则被多线分掉，两条线为例，数据的最高位放在SDO0，次高位放在SDO1，然后再SDO0……如下图所示：



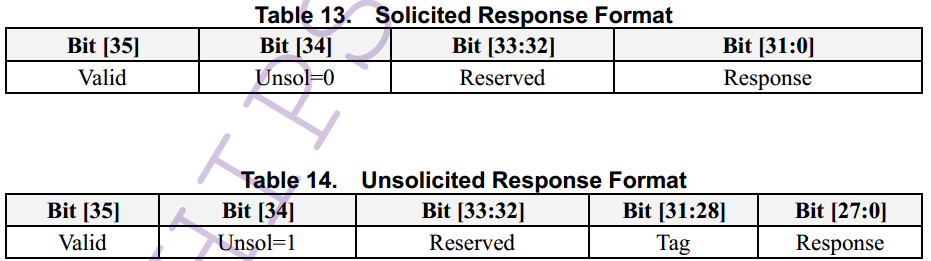
3）因为每个Stream Tag处于前一个Packet的后8bit对应位置处，这就要求每个Packet长度必须要≥8bit。因此需要使用几根SDO，是需要根据sample bits、channels、rate共同决定。下面这个是公式：



### Input Frame

#### Response Feild

1）Response Field包含36 bits数据，其格式分为被动响应与主动上传两种，如下图所示：

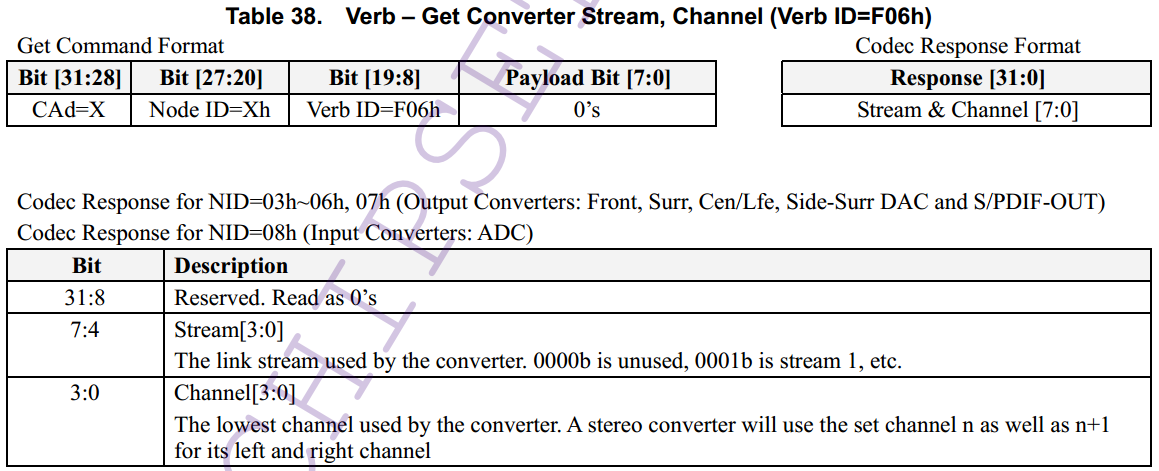


bit[35] = 1代表该response有效，HDAC会将之放入RIRB，否则表示无响应；

bit[34] = 1代表该response为Unsolicited，即主动上送，否则为某个command的响应；

bit[33:32] = 0，未用的默认值；

bit[31:0]响应的具体值，以Get操作的Verb ID为0xF06H为例，cmd与resp返回结果如下：

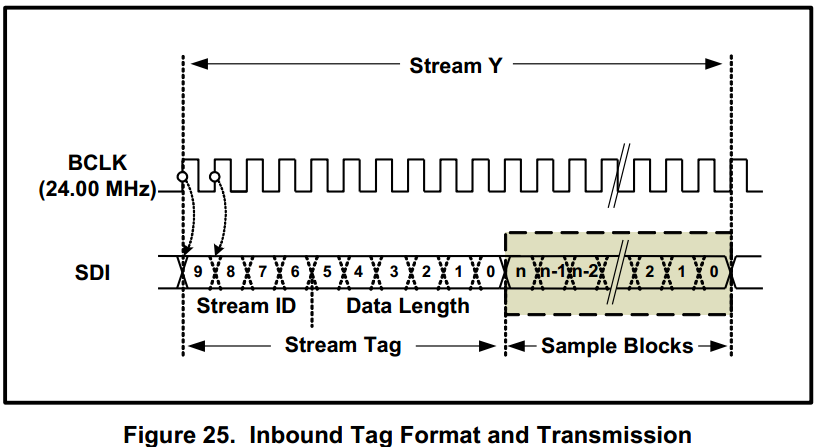


未给出的bit[35:32] = 1000b。

2）在存储方面，每个Response的36位数据存储在有64字节的RIRB Entry中，其中，前32字节存储Response的32数据位，余下4位存储在后32位区中。

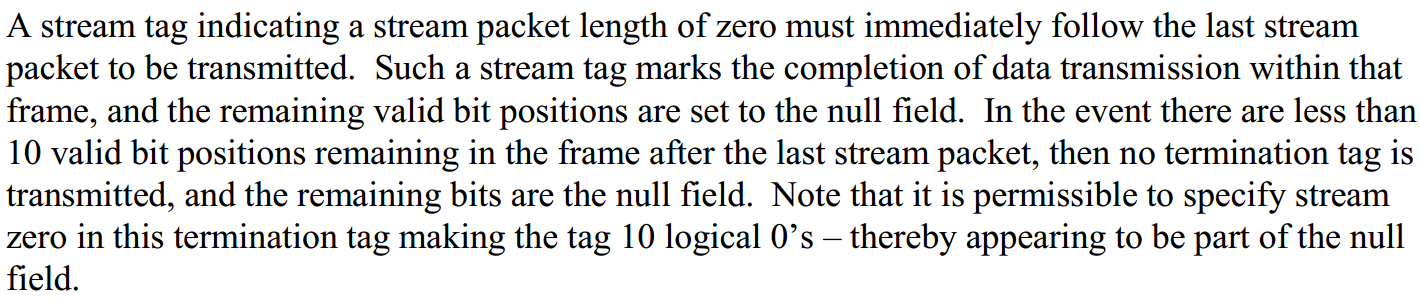
#### Stream Packets

Input Frame的Stream Packets结构与Output Frame的Stream Packets一样，每个Stream包含多个Packets，每个Packets包括一个stream\_tag和多个block，每个block又包含多个sample，每个sample包含多个数据位。需要注意的是Input Frame的stream\_tag，它在SDI上传输，位于每个stream的开头，包含10个bits，前4bits为stream\_id，后6 bits为data\_length，如下图所示：



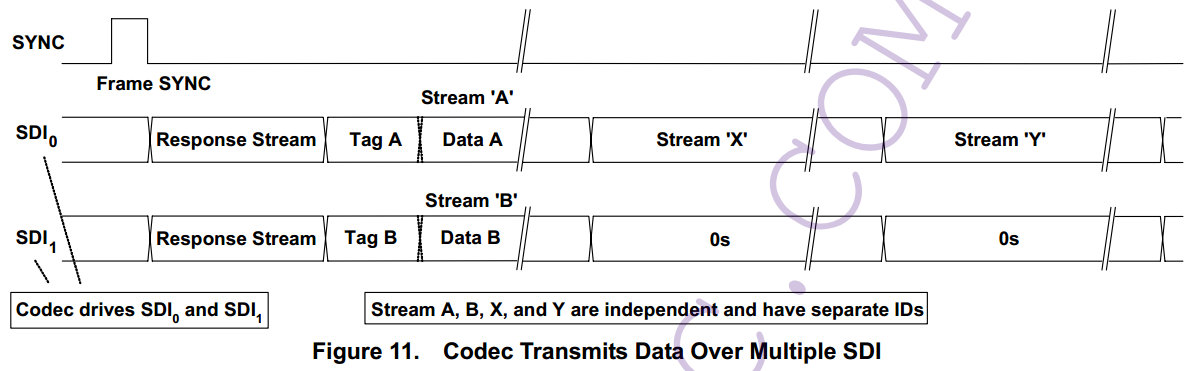
stream\_id前面已分析过。data\_length表示该stream packet中包含的所有block所占的字节数，这就要求blocks中的数据bits总数必须要是8的整数倍才能与data\_length匹配，但实际情况下有效的blocks中的bit总数不满足，例如48-kHz mono stream using 20-bit samples，每个block只有20个bit，这时就需要进行在最后补4个0凑齐24 = 8 \* 3。

还有一部分暂时没读懂，大概是讲如何结束的，如下：



#### Multiple SDI

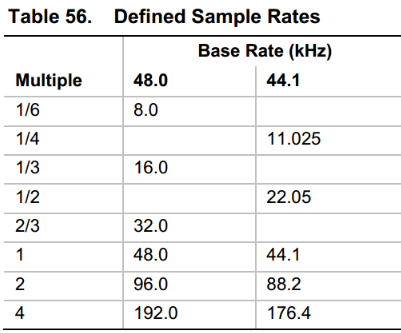
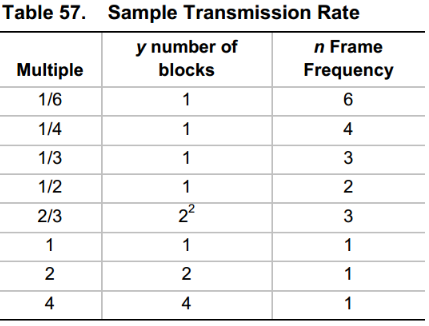
多条SDI可以提升带宽，其实现的接线可参见3.1.2节“controller与codec”。在数据传输时，多条线上的数据传输相对独立，各自传输不同的stream\_tag和packet，类似多个codec。如下图所示：



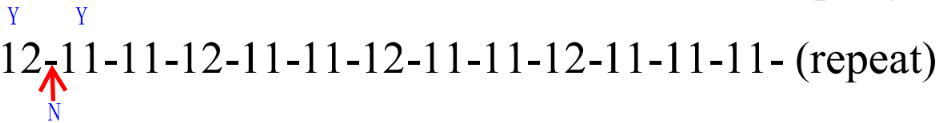
软件会根据stream\_id对各packet进行后续处理。

## Sample Rates

手册上支持的采样速率为6kHz ~ 192 kHz，一般codec提供的有以下几个



采样速率的基数为48kHz或者44.1kHz，已知link的周期为固定48kHz，当采用不同的 Base Rate时，根据采样速率与Base Rate的关系，可以是每个周期内传送多个blocks，也可能是多个周期传送一个blocks。当采用44.1kHz及其Multiple倍数时，不是一目了然的传送情况，以44.1kHz为例，其传送的情况如下：



它表示的就是传12个有blocks的→1个没blocks的→11个有blocks的→1个没blocks的→……多次以后就开始重复了。

tips：此处44.1kHz，由香农采样定律，采样频率应该不小于模拟信号频谱中最高频率的2倍。而普通人耳听力频谱范围在20Hz ~ 20kHz之间，因此要相对人耳不失真，采样率必须要大于2 \* 20k。但为什么是44.1而不是40？这是个录音发展史上的遗留问题，链接： <http://blog.csdn.net/Openking/article/details/42501287> 。简单说起来如下：

最早的录音制式有2个：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 地区 | 制式 | 场频 | 行频 | 扫描线 | 可用扫描线 | 每条线数据块 |
| 欧洲 | PAL | 50 | 15625 | 625 | 294 | 3 |
| 美日 | NTSC | 59.94 | 15575 | 525 | 245 | 3 |

可用扫描线是利用隔行扫描和过扫描技术对扫描线的处理之后得到。

计算公式：频率 = 可用扫描线 \* 场频 \* 数据块。因此：

44100 = 294 \* 50 \* 3

44056 = 245 \* 59.94 \* 3

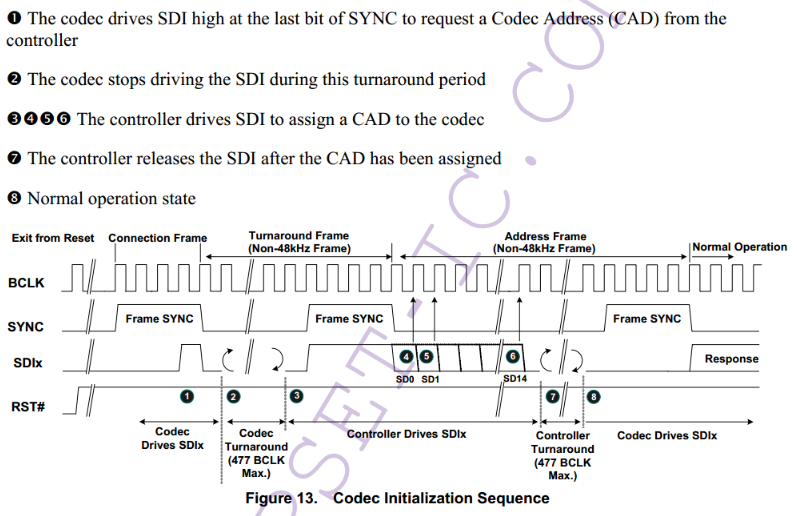
后来PAL发展强大了，所以就沿用了下来。

那么48kHz又是怎么得来的呢C:\Users\hanshu\AppData\LocalLow\Baidu\BAIDUP~1\dict\Default\FA6547~1.PNG

此外，S/PDIF audio or modems是由源同步的输入信号，而非BCLK，这部分的介绍等需要时再结合对应datasheet学习吧！

## Codec Initialization

初始化顺序如下图所示，这个没什么特别要讲的，按照流程去设置就行。



唯一要注意的是步骤，对应codec的ID是根据SDIx控制权第一次翻转之后，其上面的高电平持续到哪一位来算的，非二进制数。

## xxx

## xxx

# ALSA Driver

## azx\_probe

文本

## hda\_codec\_driver\_probe

文本

## 数据处理分析

### 发送命令

/\* send a command \*/

static int azx\_corb\_send\_cmd(struct hda\_bus \*bus, u32 val)

{

struct azx \*chip = bus->private\_data;

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

unsigned int wp, rp;

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

/\* add command to corb \*/

wp = azx\_readw(chip, CORBWP);//获取写指针

if (wp == 0xffff) {

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

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

return -EIO;

}

wp++;

wp %= ICH6\_MAX\_CORB\_ENTRIES;

rp = azx\_readw(chip, CORBRP);//获取读指针

if (wp == rp) {

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

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

return -EAGAIN;

}

chip->rirb.cmds[addr]++;//待回应命令数增1

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

azx\_writew(chip, CORBWP, wp);//修改写指针，DMA便会自动将数据发出

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

return 0;

}

### 接收中断

/\* retrieve RIRB entry - called from interrupt handler \*/

static void azx\_update\_rirb(struct azx \*chip)

{

unsigned int rp, wp;

unsigned int addr;

u32 res, res\_ex;

wp = azx\_readw(chip, RIRBWP);

if (wp == 0xffff) {

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

return;

}

if (wp == chip->rirb.wp)//指针未发生变化，说明无数据

return;

chip->rirb.wp = wp;//更新当前写指针位置

while (chip->rirb.rp != wp) {//循环至所有数据读取并处理完

chip->rirb.rp++;

chip->rirb.rp %= ICH6\_MAX\_RIRB\_ENTRIES;//

rp = chip->rirb.rp << 1; /\* an RIRB entry is 8-bytes \*/ // rirb.rp相当于索引x，rirb.buf为该索引对应的数值y、y1，因此y=2x、y1=2x+1

res\_ex = le32\_to\_cpu(chip->rirb.buf[rp + 1]);

res = le32\_to\_cpu(chip->rirb.buf[rp]);

addr = res\_ex & 0xf;//获取codec #，即SDIx

if ((addr >= AZX\_MAX\_CODECS) || !(chip->codec\_mask & (1 << addr))) {//codec addr不对

dev\_err(chip->card->dev, "spurious response %#x:%#x, rp = %d, wp = %d",

res, res\_ex,

chip->rirb.rp, wp);

snd\_BUG();

}

else if (res\_ex & ICH6\_RIRB\_EX\_UNSOL\_EV)//如果是主动上送

snd\_hda\_queue\_unsol\_event(chip->bus, res, res\_ex);//放到工作队列等待处理

else if (chip->rirb.cmds[addr]) {//如果是正常cmd的回应

chip->rirb.res[addr] = res;//将回应值传出，

smp\_wmb();//内存屏障函数，适用于多处理器的写内存屏障。读写屏障像一堵墙，所有在设置读写屏障之前发起的内存访问，必须先于在设置屏障之后发起的内存访问之前完成，确保内存访问按程序的顺序完成。？？？保证之前的内存写完的意思？k4.0中已经去掉这句

chip->rirb.cmds[addr]--;//待回应命令数减1

} else if (printk\_ratelimit()) {//混乱回应，报错

dev\_err(chip->card->dev, "spurious response %#x:%#x, last cmd=%#08x\n",

res, res\_ex,

chip->last\_cmd[addr]);

}

}

}

# Mini\_system制作

## Mini\_system结构

mini\_system是精简的内核加上busybox为基础的文件系统构成的小型Linux系统，

参考制作方法：

<http://blog.csdn.net/mybelief321/article/details/9995199>

## 修改OS支持的initrd大小

### initrd生成

### 配置bzImage

在配置阶段（make menuconfig），选中

## ALSA lib & utils 的安装与使用

### 安装

### 使用

常用命令有aplay、arecord和amixer三个。aplay、arecord命令用法相似。如果不用参数<-c, -D 等>指定声卡，则默认值来自于/usr/share/alsa/alsa.conf文件。

#### aplay & arecord

顾名思义，aplay用来播放，arecord用来录音。

举例：#arecord -D plughw:0,0 -r 32000 -f S16\_LE -c 2 32k\_16bit\_2ch\_record.wav

解释：该命令用于录制采样率为32k、采样格式为16bit小端格式存储、双通道（即立体声stereo）格式的音频文件，并保存为32k\_16bit\_2ch\_record.wav文件；直接使用CTRL+C结束录制。录制结束后使用aplay命令播放录制的音频文件，测试录制的音频文件播放是否正常。plughw:0,0表示采用的硬件device是card 0上的device 0。系统的device信息可通过aplay -l命令读取。

arecord命令可选参数为：

-r：32000、44100、48000

-f：S16\_LE、S32\_LE

-c：1（即Mono）、2（即Stereo）

可用选项可用过arecord -h查看，如下：

-h, --help help

--version print current version

-l, --list-devices list all soundcards and digital audio devices

-L, --list-pcms list device names

-D, --device=NAME select PCM by name

-q, --quiet quiet mode

-t, --file-type TYPE file type (voc, wav, raw or au)

-c, --channels=# channels

-f, --format=FORMAT sample format (case insensitive)

-r, --rate=# sample rate

-d, --duration=# interrupt after # seconds

-M, --mmap mmap stream

-N, --nonblock nonblocking mode

-F, --period-time=# distance between interrupts is # microseconds

-B, --buffer-time=# buffer duration is # microseconds

--period-size=# distance between interrupts is # frames

--buffer-size=# buffer duration is # frames

-A, --avail-min=# min available space for wakeup is # microseconds

-R, --start-delay=# delay for automatic PCM start is # microseconds

(relative to buffer size if <= 0)

-T, --stop-delay=# delay for automatic PCM stop is # microseconds from xrun

-v, --verbose show PCM structure and setup (accumulative)

-V, --vumeter=TYPE enable VU meter (TYPE: mono or stereo)

-I, --separate-channels one file for each channel

-i, --interactive allow interactive operation from stdin

-m, --chmap=ch1,ch2,.. Give the channel map to override or follow

--disable-resample disable automatic rate resample

--disable-channels disable automatic channel conversions

--disable-format disable automatic format conversions

--disable-softvol disable software volume control (softvol)

--test-position test ring buffer position

--test-coef=# test coefficient for ring buffer position (default 8)

expression for validation is: coef \* (buffer\_size / 2)

--test-nowait do not wait for ring buffer - eats whole CPU

--max-file-time=# start another output file when the old file has recorded

for this many seconds

--process-id-file write the process ID here

--use-strftime apply the strftime facility to the output file name

--dump-hw-params dump hw\_params of the device

--fatal-errors treat all errors as fatal

Recognized sample formats are: S8 U8 S16\_LE S16\_BE U16\_LE U16\_BE S24\_LE S24\_BE U24\_LE U24\_BE S32\_LE S32\_BE U32\_LE U32\_BE FLOAT\_LE FLOAT\_BE FLOAT64\_LE FLOAT64\_BE IEC958\_SUBFRAME\_LE IEC958\_SUBFRAME\_BE MU\_LAW A\_LAW IMA\_ADPCM MPEG GSM SPECIAL S24\_3LE S24\_3BE U24\_3LE U24\_3BE S20\_3LE S20\_3BE U20\_3LE U20\_3BE S18\_3LE S18\_3BE U18\_3LE U18\_3BE G723\_24 G723\_24\_1B G723\_40 G723\_40\_1B DSD\_U8 DSD\_U16\_LE

Some of these may not be available on selected hardware

The available format shortcuts are:

-f cd (16 bit little endian, 44100, stereo)

-f cdr (16 bit big endian, 44100, stereo)

-f dat (16 bit little endian, 48000, stereo)

#### amixer

该命令用来进行播放/录音参数的设定，包括音量、开关、声道选择等等，命令格式如下：

amixer <options> [command]

举例：#amixer -c 0 cget numid=28

解释：-c 0为option，表示选择card0；cget numid=28为命令，表示获取numid=28的参数值。所有参数都有默认值，可以通过cget命令读取，cset命令设置。如果不用-c指定card id，默认为card0。全部option和command可通过amixer –h获取，如下所示：

Available options:

-h,--help this help

-c,--card N select the card

-D,--device N select the device, default 'default'

-d,--debug debug mode

-n,--nocheck do not perform range checking

-v,--version print version of this program

-q,--quiet be quiet

-i,--inactive show also inactive controls

-a,--abstract L select abstraction level (none or basic)

-s,--stdin Read and execute commands from stdin sequentially

-R,--raw-volume Use the raw value (default)

-M,--mapped-volume Use the mapped volume

Available commands:

scontrols show all mixer simple controls

scontents show contents of all mixer simple controls (default command)

sset sID P set contents for one mixer simple control

sget sID get contents for one mixer simple control

controls show all controls for given card

contents show contents of all controls for given card

cset cID P set control contents for one control

cget cID get control contents for one control

# 常用命令

## 复制&查找

### 查找

locate：

find：find ./ -name ‘\*.ko’

grep：

### 复制

1）cp

2）find + cp

find . –name ‘\*.ko’ | xargs –I {} cp {} /home/hx

## 创建

mkdir

## 磁盘管理

查看u盘文件类型以及分盘： lsblk ; fdisk -l

vitrualbox extension pack安装，使能usb功能

## PCI设备管理

### 查看：lspci

常用顺序：

1. 先用lspci查看全部设备信息，例如某设备的pci编号为00:14.0
2. 使用lspci –s 14.0 ###，查看指定信息，其中###为选项，可单独可组合，例如sudo lspci –s 14.0 –kxxx，表示的意义为显示slot=14的PCI设备的驱动名称以及前256字节的配置信息。ps:14.0为不严格的设定，当系统中多条pci总线上都有slot为14的设备，则会全部显示出来。

### 设置：setpci

常用方法：setpci -s 14.0 42.B=08；其中-s 14.0为device、42为reg、.B为格式byte、08为values。表示的意义即为将slot为14的pci设备的配置空间地址为0x42的值设置为08。ps：该处的14.0同样不严格。

该命令便于调试，不用每次重设BIOS即可修改设备的属性。例如snoop path问题。

# Linux内核相关

## current 宏

current表示当前正在运行的进程

# 附录

文本

## xxxx

## xxxxx

## xxxxx

/\*\* @brief Temperature Measurement Cluster Server Init

\*

\* Server Init

\*

\* @param endpoint Endpoint that is being initialized Ver.: always

\*/

void emberAfTempMeasurementClusterServerInitCallback(int8u endpoint)

{

//schedule the next tick

emberAfScheduleClusterTick(endpoint,

ZCL\_TEMP\_MEASUREMENT\_CLUSTER\_ID,

EMBER\_AF\_SERVER\_CLUSTER\_TICK,

5000,

EMBER\_AF\_STAY\_AWAKE);

}

/\*\* @brief Temperature Measurement Cluster Server Tick

\*

\* Server Tick

\*

\* @param endpoint Endpoint that is being served Ver.: always

\*/

void emberAfTempMeasurementClusterServerTickCallback(int8u endpoint)

{

/\*\*/

int16s temValue;

temValue = sht\_getht(0xe3);

emberAfWriteAttribute(endpoint,

ZCL\_TEMP\_MEASUREMENT\_CLUSTER\_ID,

ZCL\_TEMP\_MEASURED\_VALUE\_ATTRIBUTE\_ID,

CLUSTER\_MASK\_SERVER,

(int8u \*)&temValue,

ZCL\_INT16S\_ATTRIBUTE\_TYPE);

//schedule the next tick

emberAfScheduleClusterTick(endpoint,

ZCL\_TEMP\_MEASUREMENT\_CLUSTER\_ID,

EMBER\_AF\_SERVER\_CLUSTER\_TICK,

5000,

EMBER\_AF\_OK\_TO\_HIBERNATE);

}