**Each codec class driver must provide the following features:-**

Codec DAI and PCM configuration

Codec control IO - using RegMap API

Mixers and audio controls

Codec audio operations

DAPM description.

DAPM event handler.

Optionally, codec drivers can also provide:-

DAC Digital mute control.

**SoC DAI Drivers**

**Each SoC DAI driver must provide the following features:-**

Digital audio interface (DAI) description

Digital audio interface configuration

PCM’s description

SYSCLK configuration

Suspend and resume (optional)

**SoC DSP Drivers**

Each SoC DSP driver usually supplies the following features :-

DAPM graph

Mixer controls

DMA IO to/from DSP buffers (if applicable)

Definition of DSP front end (FE) PCM devices.

**Minimising Playback Pops and Clicks**

Startup Order :- DAC --> Mixers --> Output PGA --> Digital Unmute

Shutdown Order :- Digital Mute --> Output PGA --> Mixers --> DAC

Minimising Capture Pops and Clicks

Startup Order - Input PGA --> Mixers --> ADC

Shutdown Order - ADC --> Mixers --> Input PGA

大多数编解码器在DAC之前都具有数字静音，可用于最小化任何系统噪声。静音可以阻止任何数字数据进入DAC。当应用或释放静音时，可以为每个编解码器DAI创建常见的PCM操作模式： -一个由核心调用的回调。

模式A.

在FRAME / SYNC之后，MSB在第一个BCLK的下降沿发送。

模式B.

MSB在FRAME / SYNC的上升沿发送。

/\* dapm event types \*/

#define SND\_SOC\_DAPM\_PRE\_PMU 0x1 /\* before widget power up \*/

#define SND\_SOC\_DAPM\_POST\_PMU 0x2 /\* after widget power up \*/

#define SND\_SOC\_DAPM\_PRE\_PMD 0x4 /\* before widget power down \*/

#define SND\_SOC\_DAPM\_POST\_PMD 0x8 /\* after widget power down \*/

#define SND\_SOC\_DAPM\_PRE\_REG 0x10 /\* before audio path setup \*/

#define SND\_SOC\_DAPM\_POST\_REG 0x20 /\* after audio path setup \*/

**pcm dma**：负责把 dma buffer 中的音频数据搬运到 I2S tx FIFO，cpu dai负责把音频数据从 I2S tx FIFO 搬运到 CODEC, 音频 dma 驱动通过 snd\_soc\_register\_platform() 来注册，故也常用 platform 来指代音频 dma 驱动,

Codec 音频操作接口通过结构体 snd\_soc\_dai\_ops 描述，Codec 音频操作接口分为 5 大部分：时钟配置、格式配置、数字静音、PCM 音频接口、FIFO 延迟。codec\_dai 系统时钟设置，当上层打开 pcm 设备时，需要回调该接口设置 Codec 的系统时钟，Codec 才能正常工作；

对于 dai（codec\_dai 和 cpu\_dai），都要非常留意时钟设置，它很关键又复杂，设置错误将会导致很多问题，典型如下：

系统无声：检查 Codec 系统时钟、codec\_dai 位时钟和帧时钟是否使能；

声音失真：检查音频数据的采样率是否和 codec\_dai 帧时钟一致；

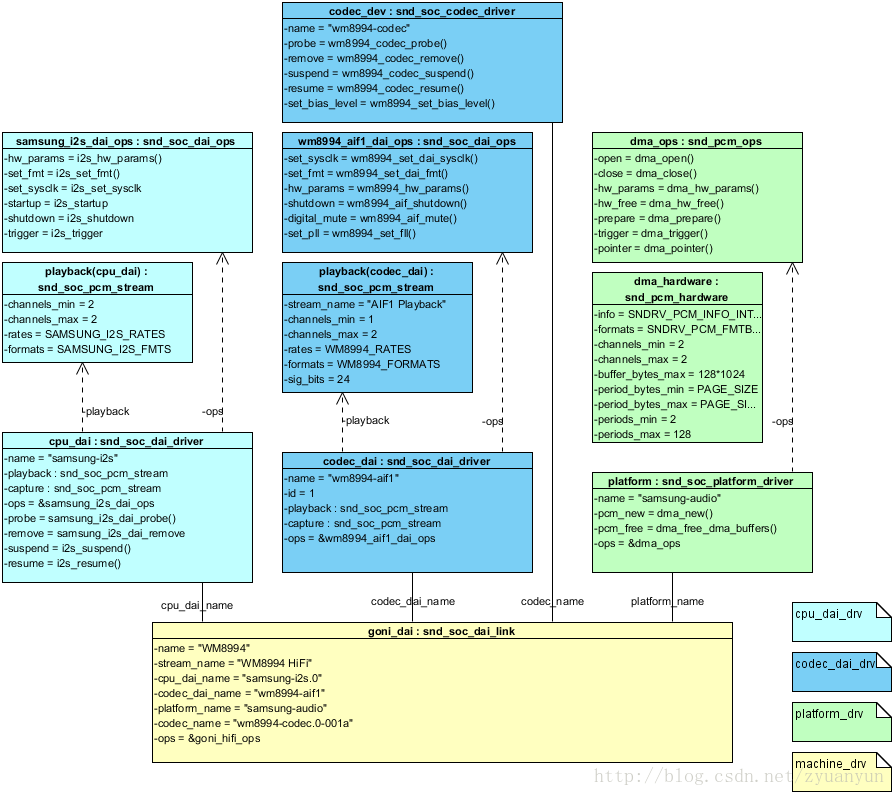
断续破音：检查 Codec 系统时钟和位时钟、帧时钟是否同步，出现这种情况，可能是因为 sysclk 和 BCLK/LRCLK 不是由同一个时钟源分频出来的。

音频部件由于上下电瞬间的瞬态冲击会产生爆破音，我们称之为 POPs。POPs 是电气特性，总的来说：上电次序是从输入端点到输出端点，下电次序是从输出端点到输入端点。

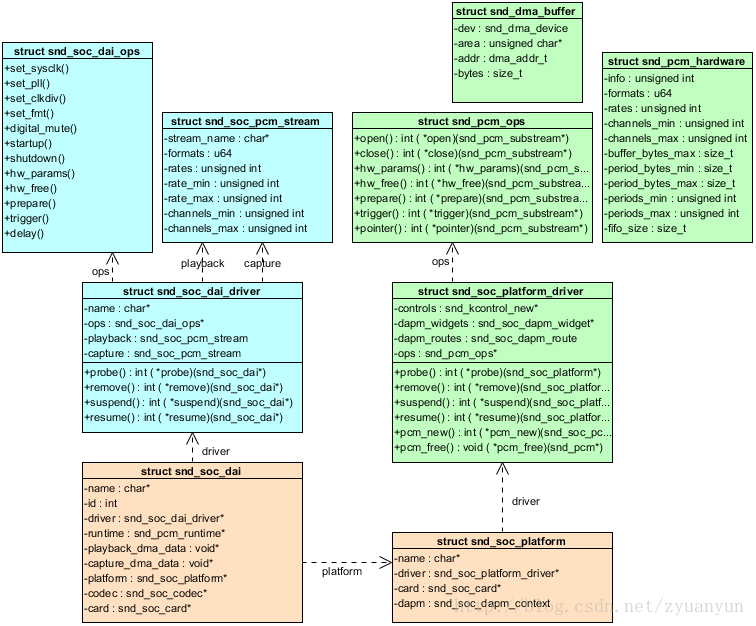
Platform 驱动主要用于音频数据传输：这里又细分为两步：

启动 dma 设备，把音频数据从 dma buffer 搬运到 cpu\_dai FIFO，这部分驱动用 snd\_soc\_platform\_driver 描述，后面分析用 pcm\_dma 指代它。

启动数字音频接口控制器（I2S/PCM/AC97），把音频数据从 cpu\_dai FIFO 传送到 codec\_dai，这部分驱动用 snd\_soc\_dai\_driver 描述，后面分析用 cpu\_dai 指代它。



codec驱动使用函数snd\_soc\_register\_codec在注册codec时（那么snd\_soc\_register\_codec啥时调用？是codec的platform\_driver和platform\_device匹配后调用probe函数，probe函数会调用这个函数），会传入该codec所支持的dai个数和记录dai信息的snd\_soc\_dai\_driver结构指针. snd\_soc\_register\_dais() 会使得ASoc把codec的dai注册到系统中，并把这些dai都挂在全局链表变量dai\_list中，然后，在codec被machine驱动匹配后，soc\_probe\_codec函数会被调用，他会通过全局链表变量dai\_list查找所有属于该codec的dai.同时在snd\_soc\_register\_codec()函数中list\_add(&codec->list, &codec\_list) 把以上 codec 实例插入到 codec\_list链表中，声卡注册时会遍历该链表（起始就是machine驱动匹配codec时），找到 dai\_link 声明的 codec 并绑定。



对于 cpu\_dai 驱动，从上面的类图我们可知，主要工作有：

实现 dai 操作函数，见 snd\_soc\_dai\_ops 定义，用于配置和操作音频数字接口控制器，如时钟配置 set\_sysclk()、格式配置 set\_fmt()、硬件参数配置 hw\_params()、启动/停止数据传输 trigger() 等；

实现 probe 函数（初始化）、remove 函数（卸载）、suspend/resume 函数（电源管理）；

初始化 snd\_soc\_dai\_driver 实例，包括回放和录制的能力描述、dai 操作函数集、probe/remove 回调、电源管理相关的 suspend/resume 回调；

通过 snd\_soc\_register\_dai() 把初始化完成的 snd\_soc\_dai\_driver 注册到 soc-core：首先创建一个 snd\_soc\_dai 实例，然后把该 snd\_soc\_dai 实例插入到 dai\_list 链表（声卡注册时会遍历该链表，找到 dai\_link 声明的 cpu\_dai 并绑定）。

cpu\_dai 驱动应该算是这个系列中最简单的一环，因此不多花费笔墨在这里了。倒是某些平台上，dma 设备信息（总线地址、通道号、传输单元大小）是在这里初始化的，这点要留意，这些 dma 设备信息在 pcm\_dma 驱动中用到。

某些平台会调用samsung\_i2s\_probe(struct platform\_device \*pdev)

{

// 从 platform\_device 中取得 resource，得到 playback dma 通道号

res = platform\_get\_resource(pdev, IORESOURCE\_DMA, 0);

dma\_pl\_chan = res->start; // dma\_pl\_chan 中的 pl 是 playback 简写

// 从 platform\_device 中取得 resource，得到 capture dma 通道号

res = platform\_get\_resource(pdev, IORESOURCE\_DMA, 1);

dma\_cp\_chan = res->start; // dma\_cp\_chan 中的 cp 是 capture 的简写

// 从 platform\_device 中取得 resource，得到 I2S 的基地址

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

pri\_dai->dma\_playback.dma\_addr = regs\_base + I2STXD; // 设置 playback dma 设备地址为 I2S tx FIFO 地址

pri\_dai->dma\_capture.dma\_addr = regs\_base + I2SRXD; // 设置 capture dma 设备地址为 I2S rx FIFO 地址

pri\_dai->dma\_playback.channel = dma\_pl\_chan; // 设置 playback dma 通道号

pri\_dai->dma\_capture.channel = dma\_cp\_chan; // 设置 capture dma 通道号

pri\_dai->dma\_playback.dma\_size = 4; // 设置 playback dma 传输单元大小为 4 个字节

pri\_dai->dma\_capture.dma\_size = 4; // 设置 capture dma 传输单元大小为 4 个字节

}

**pcm dma：**

PCM 数据管理可以说是 ALSA 系统中最核心的部分，这部分的工作有两个（回放情形）：

copy\_from\_user 把用户态的音频数据拷贝到 dma buffer 中；启动 dma 设备把音频数据从 dma buffer 传送到 I2S tx FIFO。当数据送到 I2S tx FIFO 后，剩下的是启动 I2S 控制器把数据传送到 Codec，然后 DAC 把音频数字信号转换成模拟信号，再输出到 SPK/HP。

为什么要使用 dma 传输？两个原因：首先在数据传输过程中，不需要 cpu 的参与，节省 cpu 的开销；其次传输速度快，提高硬件设备的吞吐量。对于 ARM，它不能直接把数据从 A 地址搬运到 B 地址，只能把数据从 A 地址搬运到一个寄存器，然后再从这个寄存器搬运到 B 地址；而 dma 有突发（Burst）传输能力，这种模式下一次能传输几个甚至十几个字节的数据，尤其适合大数据的高速传输。一个 dma 传输块里面，可以划分为若干个周期，每传输完一个周期产生一个中断。

对于回放来说，dma 设备把内存缓冲区的音频数据传送到 I2S tx FIFO；对于录制来说，dma 设备把 I2S rx FIFO 的音频数据传送到内存缓存区。因此在 dma 设备传输之前，必须确定 data buffer 和 I2S FIFO 的信息。

**snd\_dma\_buffer**：数据缓存区，用于保存从用户态拷贝过来的音频数据；包含 dma buffer 的物理首地址，虚拟首地址、大小等信息；其中物理地址用于设定 dma 传输的源地址（回放情形）或目的地址（录制情形），虚拟地址用于与用户态之间的音频数据拷贝。

介绍几个重要的接口：

dma buffer 的分配，一般发生在 pcm\_dma 驱动初始化阶段 probe() 或 pcm 逻辑设备创建阶段 pcm\_new()。当 soc-core 调用 soc\_new\_pcm() 创建 pcm 逻辑设备时，会回调 pcm\_new() 完成 dma buffer 内存分配，注意回放子流和录制子流有着各自的 dma buffer。

**open**：打开 pcm 逻辑设备时，回调该函数设定 dma 设备的硬件约束；并申请一个私有结构，保存 dma 设备资源如通道号、传输单元、缓冲区信息、IO 信息等，保存在 runtime->private\_data。

**hw\_params：**设置硬件参数时（cmd=SNDRV\_PCM\_IOCTL\_HW\_PARAMS），回调该函数初始化 dma 资源，包括通道号、传输单元、缓冲区信息、IO 设备信息等

**prepare**：当数据已准备好（cmd=SNDRV\_PCM\_IOCTL\_PREPARE），回调该函数告知 dma 设备数据已就绪。

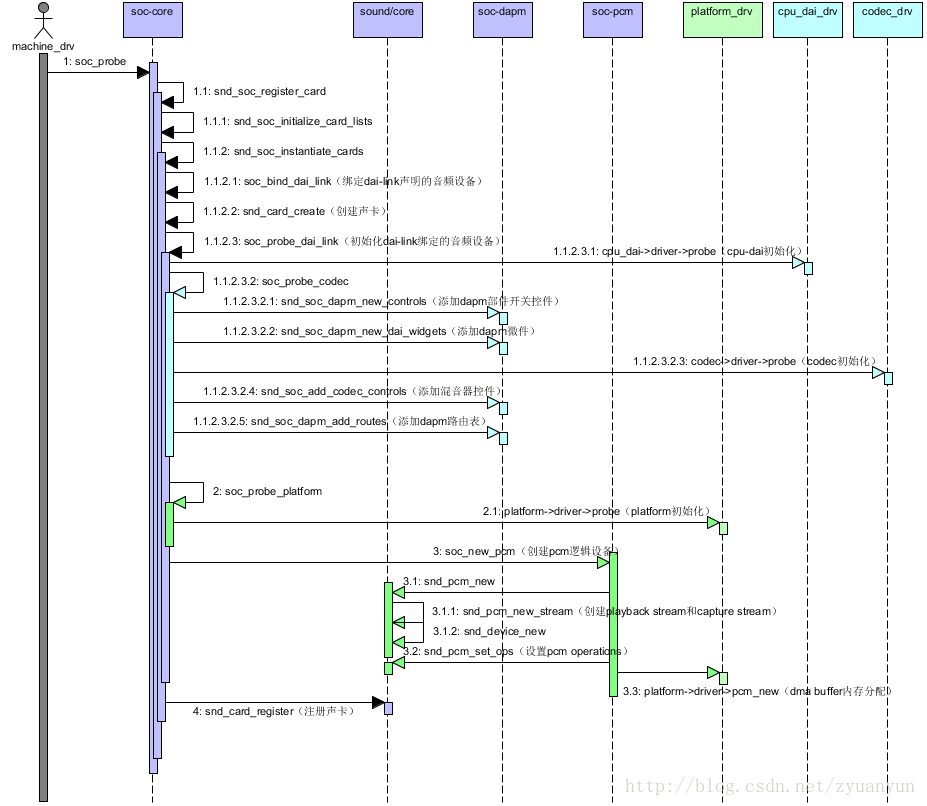
**dma\_enqueue() 函数，**把当前 dma buffer 插入到 dma 传输队列中。当触发 trigger() 启动 dma 设备传输后，将会把 dma buffer 数据传送到 FIFO（回放情形）。

注意：每次 dma 传输完一个周期的数据后，都要调用 **snd\_pcm\_period\_elapsed()** 告知 pcm native 一个周期的数据已经传送到 FIFO 上了，然后再次调用 dma\_enqueue()，dma 传输…如此循环，直到触发 trigger() 停止 dma 传输。

**trigger：**数据传送 开始/停止/暂停/恢复 时，回调该函数启动或停止 dma 传输（当上层第一次调用 pcm\_write() 时，触发 trigger() 启动 dma 传输；当上层调用 pcm\_stop() 或 pcm\_drop() 时，触发 trigger() 停止 dma 传输）。trigger() 函数里面的操作必须是原子的，不能调用可能睡眠的操作，并且应尽量简单。

**pointer**：dma 每完成一次传输，都会调用该函数获得传输数据的当前位置，这样 pcm native 可计算 dma buffer 指针位置及可用空间。该函数也是原子的。

snd\_soc\_dai\_link 结构体：**ops**：重点留意 hw\_params() 回调，一般来说这个回调是要实现的，用于配置 codec、codec\_dai、cpu\_dai 的数据格式和系统时钟。



Codec 初始化时，通过 snd\_soc\_add\_codec\_controls() 把所有定义好的音频控件注册到 alsa-core ，上层可以通过 tinymix、alsa\_amixer 等工具查看修改这些控件的设定。

创建声卡：

当platform\_device和platform\_driver匹配后，soc\_probe() 会被调用，继而调用 snd\_soc\_register\_card() 注册声卡。

snd\_soc\_register\_card() 为每个 dai\_link 分配一个 snd\_soc\_pcm\_runtime 实例，别忘了之前提过 snd\_soc\_pcm\_runtime 是 ASoC 的桥梁，保存着 codec、codec\_dai、cpu\_dai、platform 等硬件设备实例。

随后的工作都在 snd\_soc\_instantiate\_card() 进行：

遍历 dai\_list、codec\_list、platform\_list 链表，为每个音频链路找到对应的 cpu\_dai、codec\_dai、codec、platform；找到的 cpu\_dai、codec\_dai、codec、platform 保存到 snd\_soc\_pcm\_runtime ，完成音频链路的设备绑定（soc\_bind\_dai\_link）；调用 snd\_card\_create() 创建声卡；soc\_probe\_dai\_link() 依次回调 cpu\_dai、codec、platform、codec\_dai 的 probe() 函数，完成各音频设备的初始化(如：soc\_probe\_codec->{snd-soc\_dapm\_new\_controls(),snd\_soc\_dapm\_new\_dai\_widgets(),codec->driver->prob(codec初始化)，snd\_soc\_add\_codec\_controls(),snd\_soc\_dapm\_add\_routes()})，随后调用 soc\_new\_pcm() 创建 pcm 逻辑设备（因为涉及到本系列的重点内容，后面具体分析这个函数）；最后调用 snd\_card\_register() 注册声卡。

soc\_new\_pcm 源码分析：

{

// 初始化 snd\_soc\_pcm\_runtime 的 ops 字段，成员函数其实依次调用 machine、codec\_dai、cpu\_dai、platform 的回调；如 soc\_pcm\_hw\_params：

// 在这里把底层硬件的操作接口抽象起来，pcm native 不用知道底层硬件细节

soc\_pcm\_ops->open = soc\_pcm\_open;

soc\_pcm\_ops->close = soc\_pcm\_close;

soc\_pcm\_ops->hw\_params = soc\_pcm\_hw\_params;

soc\_pcm\_ops->hw\_free = soc\_pcm\_hw\_free;

soc\_pcm\_ops->prepare = soc\_pcm\_prepare;

soc\_pcm\_ops->trigger = soc\_pcm\_trigger;

soc\_pcm\_ops->pointer = soc\_pcm\_pointer;

/\* check client and interface hw capabilities \*/

// 创建 pcm 逻辑设备

ret = snd\_pcm\_new(rtd->card->snd\_card, new\_name,

num, playback, capture, &pcm);

// 回调 dma 驱动的 pcm\_new()，进行 dma buffer 内存分配和 dma 设备初始化

if (platform->driver->pcm\_new) {

ret = platform->driver->pcm\_new(rtd);  
}

snd\_pcm\_new：

创建一个 PCM 设备实例 snd\_pcm；

创建 playback stream 和 capture stream，旗下的 substream 也同时建立；

调用 snd\_device\_new() 把 PCM 设备挂到声卡的 devices 链表上。

这个 buffer 中有 4 个 period，每当 DMA 搬运完一个 period 的数据就会出生一次中断，因此搬运这个 buffer 中的数据将产生 4 次中断。ALSA 为什么这样做？因为数据缓存区可能很大，一次传输可能会导致不可接受的延迟；为了解决这个问题，alsa 把缓存区拆分成多个周期，以周期为单元传输数据。

Machine 驱动初始化时，.name = "soc-audio" 的 platform\_device 与 platform\_driver 匹配成功，触发 soc\_probe() 调用；

继而调用 snd\_soc\_register\_card()：

为每个音频物理链路找到对应的 codec、codec\_dai、cpu\_dai、platform 设备实例，完成 dai\_link 的绑定；

调用 snd\_card\_create() 创建声卡；

依次回调 cpu\_dai、codec、platform 的 probe() 函数，完成物理设备的初始化；

随后调用 soc\_new\_pcm()：

设置 pcm native 中要使用的 pcm 操作函数，这些函数用于驱动音频物理设备，包括 machine、codec\_dai、cpu\_dai、platform；

调用 snd\_pcm\_new() 创建 pcm 逻辑设备，回放子流和录制子流都在这里创建；

回调 platform 驱动的 pcm\_new()，完成音频 dma 设备初始化和 dma buffer 内存分配；

最后调用 snd\_card\_register() 注册声卡。

snd\_cards：记录着所注册的声卡实例，每个声卡实例有着各自的逻辑设备，如 PCM 设备、CTL 设备、MIDI 设备等，并一一记录到 snd\_card 的 devices 链表上

snd\_minors：记录着所有逻辑设备的上下文信息，它是声卡逻辑设备与系统调用 API 之间的桥梁；每个 snd\_minor 在逻辑设备注册时被填充，在逻辑设备使用时就可以从该结构体中得到相应的信息（主要是系统调用函数集 file\_operations）.

这个 buffer 中有 4 个 period，每当 DMA 搬运完一个 period 的数据就会出生一次中断，因此搬运这个 buffer 中的数据将产生 4 次中断。ALSA 为什么这样做？因为数据缓存区可能很大，一次传输可能会导致不可接受的延迟；为了解决这个问题，alsa 把缓存区拆分成多个周期，以周期为单元传输数据。

注：- buffer\_size = period\_size \* periods

- period\_bytes = period\_size \* bytes\_per\_frame

- bytes\_per\_frame = channels \* bytes\_per\_sample

假设我们想要使用立体声，16位，44.1 KHz流，单向（意味着，无论是在回放还是在捕获方向）。然后我们有：

- 'stereo'=频道数：2

- 1个模拟样本用16位= 2字节表示

- 1帧表示来自所有通道的1个模拟样本;这里我们有2个通道，所以：1帧=（num\_channels）\*（1个样本，以字节为单位）=（2个通道）\*（每个样本2个字节（16位））= 4个字节（32位）

- 要维持2x 44.1 KHz模拟速率 - 系统必须具有数据传输速率，以字节/秒为单位：Bps\_rate =（num\_channels）\*（1个样本，以字节为单位）\*（analog\_rate）=（1帧）\*（analog\_rate）= （2个通道）\*（2个字节/样本）\*（44100个样本/秒）= 2 \* 2 \* 44100 = 176400个字节/秒。

再说说 period bytes，对于 dma 处理来说，它直接关心的是数据大小，而非 period\_size（一个周期的帧数），有个转换关系：period\_bytes = period\_size \* sample\_bits \* channels / 8

由于 I2S 总线采样率是稳定的，我们可以计算 I2S 传输一个周期的数据所需的时间：transfer\_time = 1 \* period\_size / sample\_rate, in second

例如 period\_size = 1024，sample\_rate = 48KHz ，那么一个周期数据的传输时间是： 1 \* 1024 / 48000 = 21.3 (ms)。

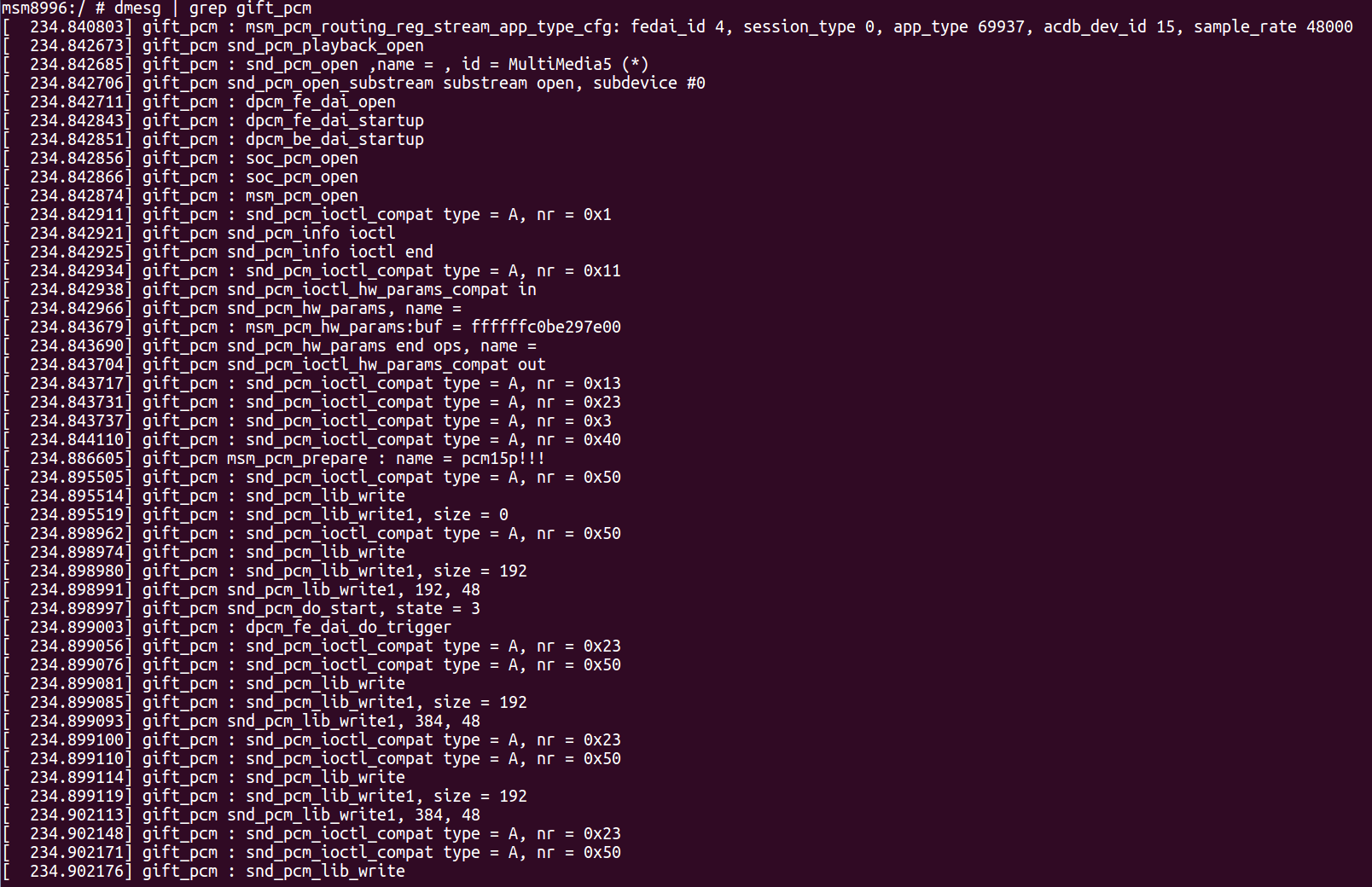
我们提到：每次 dma 传输完成一个周期的数据传输后，都要调用 snd\_pcm\_period\_elapsed() 告知 pcm native 一个周期的数据已经传送到 FIFO 上了，然后再次调用 dma 传输音频数据…如此循环。

当数据在cpu上时，是由cpu直接进行数据搬移，当数据到了adsp及codec后，数据是由adsp及codec进行搬移，但是cpu又得控制adsp、codec对数据搬移的行为，所以采用了dapm（widgets）的方式来对数据路径进行控制，但不亲自参与搬移。

ap与dsp之间采用share memory进行数据交换，ap将音频数据发送给dsp，dsp中按照配置（acdb中的配置）对音频数据进处理，并将最终处理后的音频数据送入codec或者其他audio设备，比如Bluetooth、modem等。dsp中音频数据处理链路是允许第三方增加新module的（音频数据处理单元），增加的新module可以通过.so的形式添加到dsp中也可以.lib的形式直接编译进dsp的kernel中（需要有高通dsp kernel编译环境）。

其他地方的文章中paltform的作用是控制数据在处理器内部（无论是cpu还是dsp）搬移，即：把数据流从接收端口的缓冲区中搬移到发送端口的缓冲区中，并控制dai进行数据传输。按照系统框架的设计，在cpu里面，platform的体现就是当用户打开一个pcm设备并向pcm设备写入音频数据后，platform负责把数据从pcm设备驱动的接收缓冲区中搬移到cpu dai的发送缓冲区中，并通过cpu dai进行数据传输。

但是！！！高通平台好像并不是这样，事实上cpu dai（fe），除了维护了asoc框架的一致性以及充当了dapm上面的一个端点之外并没有其他实质性的作用。作为dai（数字音频接口）最核心的功能应该是提供数据传输接口的dirver，也就是dai driver的，但是这里他并没有提供数据传输能力，真实的数据传输能力（即cpu与dsp通信的能力）是通过smem、smd以及smmu相关driver共同完成的，由platform直接调用。



总的来说，一个音频链路的启动分为5个部分：start->pcm open->hw/sw parameters->prepare->trigger->write->end

pcm open起源于逻辑pcm设备的open，也就是在linux userspace里面去fopen一个pcm设备，根据linux设备驱动框架，这里会调用注册设备时提供的open接口。这个接口在pcm\_native.c文件中。

const struct file\_operations snd\_pcm\_f\_ops[2] = {

{ .open = snd\_pcm\_playback\_open,}

open的函数调用关系：   
snd\_pcm\_playback\_open->snd\_pcm\_open->snd\_pcm\_open\_file->snd\_pcm\_open\_substream->(substream->ops->open()) ;

(所以，实质上是调用创建substream时注册的open函数, 这个函数在soc-pcm.c中,

int soc\_new\_pcm(struct snd\_soc\_pcm\_runtime \*rtd, int num)

{

……

/\* ASoC PCM operations \*/

if (rtd->dai\_link->dynamic) {

rtd->ops.open = dpcm\_fe\_dai\_open;

} else {

rtd->ops.open = soc\_pcm\_open;

if (playback)

snd\_pcm\_set\_ops(pcm, SNDRV\_PCM\_STREAM\_PLAYBACK, &rtd->ops);

if (capture)

snd\_pcm\_set\_ops(pcm, SNDRV\_PCM\_STREAM\_CAPTURE, &rtd->ops);

……

}

soc\_new\_pcm()函数，snd\_pcm\_set\_ops(pcm, SNDRV\_PCM\_STREAM\_PLAYBACK, &rtd->ops);这一句话就是把rtd的ops赋值给了substream的ops，就是前面调用的substream->ops->open()。那就是open函数到底是调用的dpcm\_fe\_dai\_open还是soc\_pcm\_open？**但事实是**在存在dsp的平台上都不可能调用soc\_pcm\_open而必须调用dpcm\_fe\_dai\_open，那么这个就涉及到dynamic pcm了. 最开始图中的打印，发现在open过程中，最终还是为了来打开跟这个stream所相关的所有dai、platform、dai link：

static int soc\_pcm\_open(struct snd\_pcm\_substream \*substream)

{

……

/\* startup the audio subsystem \*/

if (cpu\_dai->driver->ops && cpu\_dai->driver->ops->startup) {

ret = cpu\_dai->driver->ops->startup(substream, cpu\_dai);

……

}

if (platform->driver->ops && platform->driver->ops->open) {

ret = platform->driver->ops->open(substream);

……

}

for (i = 0; i < rtd->num\_codecs; i++) {

codec\_dai = rtd->codec\_dais[i];

if (codec\_dai->driver->ops && codec\_dai->driver->ops->startup) {

ret = codec\_dai->driver->ops->startup(substream,

codec\_dai);

……

}

……

}

if (rtd->dai\_link->ops && rtd->dai\_link->ops->startup) {

ret = rtd->dai\_link->ops->startup(substream);

}

……

}

platform：简单来说就是设置了hw parameter并且添加了几条rule，同时调用q6asm\_audio\_client\_alloc函数创建了一个audio\_client。这个东西就是cpu与dsp通信的接口，包括了smem、smd、smmu、apr等等。)

总结一下open到底是想干什么，感觉就是针对该stream的硬件特性以及通信接口进行初始化，所以所有跟硬件有关的初始化都应该放到open中完成。

关于hw/sw parameters:

hw parameter表示当前音频流使用的硬件参数，比如当前音频流使用的format，声道数，使用的哪个音频输出端口等等。使用的哪一种格式呢，这个问题就由上层应用来指定，比如说上层应用需要播放一个16bit的pcm，那么就会打开一个pcm逻辑设备，并通过ioctrl来通知音频驱动当前播放的音频流的参数，音频驱动根据这些配置来进行对应的操作.

这一点从前面章节图中的打印信息可以看出，android的音频框架在open了pcm设备后立即发送0x1命令查询driver info，然后就发送了0x11命令，来设置hw parameter。各driver的hw\_params调用过程：

ioctrl cmd:0x11->snd\_pcm\_ioctl\_hw\_params\_compat->snd\_pcm\_hw\_params->

(substream->ops->hw\_params)->dpcm\_fe\_dai\_hw\_params->fe/be:soc\_pcm\_hw\_params->各driver的hw\_params函数.

所谓rule，就是对hw parameter的约束，比如某个dai只支持16bit传输，结果现在上面来了一个stream是24bit的，这个时候上层应用程序通过0x11命令发送hw parameter告诉driver有一个24bit的流要来了，这个时候driver回去check自己能不能处理24bit的stream，如果能，就把自己设置为24bit模式，如果不行则配置不生效。所以，rule的添加都是音频驱动中各个driver在open的时候自己添加到该stream的rules中的.

关键文件及函数:

pcm\_native.c : snd\_pcm\_hw\_refine、snd\_pcm\_hw\_constraints\_init、snd\_pcm\_hw\_constraints\_complete

pcm\_lib.c : snd\_pcm\_hw\_rule\_add

在设置parameter的值时操作函数如下：tatic void param\_set\_mask(struct snd\_pcm\_hw\_params \*p, int n, unsigned bit)->struct snd\_mask \*param\_to\_mask(struct snd\_pcm\_hw\_params \*p,). 前面已经说过，rule是在pcm逻辑设备open时添加的，而且是由各个部分的dirver自己添加自己的rule，具体添加的接口是snd\_pcm\_hw\_rule\_add函数，所有的rule都是通过这个函数最终被添加的。 snd\_pcm\_hw\_rule\_add函数中有一个参数：runtime，这个东西是与substream对应的，其实就是substream的运行时状态信息，在runtime中有一个snd\_pcm\_hw\_constraints类型成员，这个成员就是rule的存放点，所有通过snd\_pcm\_hw\_rule\_add添加的rule都被存放到snd\_pcm\_hw\_constraints类型成员的rules中。

**最后，rule是在哪里起作用的** ? 这个问题就得说一下snd\_pcm\_hw\_refine函数了,检查参数是否满足rule, 当多条rule约束同一参数时，每条rule都会调用约束函数来处理该参数，如果参数被rule给修改了，则表示该参数被此rule约束了，一旦一个rule修改了 该参数的值，那么所有其他的rule都必须重新对该参数调用一次约束函数，直到所有约束该参数的rule都不再会修改该参数的值为止. 在be端，最终的一些输出在设计上可能希望是一个约定好的固定值，比如dsp输出的音频数据流始终是48000hz采样率，那么这个时候就需要在设置完hw params之后强行把be的hw param刷成dsp输出的实际参数，这个时候就用到了fixup。用ioctrl进行hw params的设置时其实是对fe进行的设置，be的hw params是从把fe的hw params memcpy过去的，所以如果没有fixup，则be的参数等价于fe的参数，但是如果dsp的输出端如果希望与输入端有不同的参数，则只能通过fixup来实现。 逻辑上，dsp的输出端的hw params是设备制造商自己来约定的，比如设备制造商希望dsp输出到codec的采样率统一为48000hz，所以fixup函数应该由mechine driver来提供，这就是为什么在msm8996平台上be\_hw\_params\_fixup函数都在msm8996.c文件中了。 补充一句，be\_hw\_params\_fixup是在dai link结构体中，是在mechine driver定义dai link时进行赋值的。

**Prepare:**

可以认为prepare也是设置参数，只是跟hw params不同，这里的参数基本上都是围绕音频数据流来进行的，而hw params时的参数更多的是关心硬件特性。在prepare被调用后函数内会分别调用所有的dai、platform、dai link自己的prepare函数，如果需要处理prepare的话各自就实现该函数。首先，fe的platform在prepare的时候会操作msm\_audio，向dsp设置相关的操作，be的platform在prepare的时候要打开一个adm，并对dsp内部音频链路上的处理过程进行配置. fe的cpu dai以及codec dai都没有实现，be的cpu dai在prepare时会启动dsp上对应的afe port(其实就是dsp音频链路最后的处理单元，这个处理单元后面就是音频输出的物理接口了)，be的codec dai设置了一下silm接口的宽度。

**Trigger**

启动传输，trigger只在fe platform中实现了，因为在msm8996平台上也只有cpu需要启动对dsp的数据传输，而trigger中也并不是启动的dma传输，是高通自己cpu与dsp的通信接口，所以，其实在整个音频链路中并不是绝对的dma操作，dma控制器也并不是都在cpu上, 高通平台，这里的数据搬移跟cpu关系并不到，应该是cpu与dsp通过smmu映射了一段内存，然后通过smd把内存地址通知对方，dsp再通过dma之类的操作去搬移数据, **有一点**需要说明一下，trigger并不是只有在应用层发送SNDRV\_PCM\_IOCTL\_START(0x42)命令后才会执行的，从前面那个启动打印的截图中可以看到0x42这条指令并没有发送，但是trigger依然执行了，其实这里的trigger是在write里面被执行的.

static snd\_pcm\_sframes\_t snd\_pcm\_lib\_write1(struct snd\_pcm\_substream \*substream,

unsigned long data,

snd\_pcm\_uframes\_t size,

int nonblock,

transfer\_f transfer)

{

……

if (runtime->status->state == SNDRV\_PCM\_STATE\_PREPARED &&

snd\_pcm\_playback\_hw\_avail(runtime) >= (snd\_pcm\_sframes\_t)runtime->start\_threshold) {

err = snd\_pcm\_start(substream);

if (err < 0)

goto \_end\_unlock;

}

在写数据的时候会先去判断runtime的状态，如果runtime还是prepared状态，且已经收到的数据超过了启动发送门限值了，则会调用trigger，来完成实际的数据搬移。所以上层应用往往会省略start操作，在prepare之后便直接开始写数据了。

那么如果想要创建一个pcm设备，该怎么做？那就是定义一个dai link！！！但是并不是所有dai link都会创建pcm设备.

音频链路的通路是通过widgets的路径来描述的，而不同硬件设备间的链接是通过dai link来描述的，那么根据damp的理念，在音频链路上dai link也应该对应一个widgets，该widgets分别链接两边硬件的widgets，以实现垮硬件的widgets通路。

dai widgets在声卡创建阶段会去调用soc\_probe\_link\_components函数probe所有注册进来的component(关于component看一下register相关函数和component list就明白了)，soc\_probe\_link\_components中会对每个component调用snd\_soc\_dapm\_new\_dai\_widgets。

在创建完dai widgets后会继续调用snd\_soc\_dapm\_connect\_dai\_link\_widgets函数进行dai widgets的链接：

dai widgets如和与其他widgets链接

这一块依赖于两个地方，一个地方是通过stream name去匹配，另一个是靠aif去指定。

在声卡初始化的时候，snd\_soc\_instantiate\_card中调用snd\_soc\_dapm\_link\_dai\_widgets函数来完成的。snd\_soc\_dapm\_link\_dai\_widgets函数会去遍历每一个dai widgets，然后遍历所有的非dai widgets，如果非dai widgets的stream name与dai widgets的name相同，则把两个widgets进行链接。aif的绑定则是在dai dirver被probe时完成的，同样在snd\_soc\_instantiate\_card中会调用soc\_probe\_link\_dais函数，该函数会把每个dai driver注册时提供的probe都调用一遍，如果配置了aif的话这里就应该把aif与dai widgets进行链接。这里其实也是依赖于dai widgets的name必须为stream name，由于在probe函数中，因此这个函数属于dai driver，不应该去获取widgets的信息，所以其实这里链接的两个widgets的name一个是aif的name，另一个是stream name.

所以dpcm是干什么？就是为了在数据输入流与数据输出流中间增加一个switch，允许开发者更自由的去控制音频链路，而不是像以前那样，电路板做好了音频链路也就唯一确定了.

### DPCM的工作机制

说DPCM的话，就先从dai link说起，dai link中有两个元素dynamic、no\_pcm.

几乎所有的be dai link中的no\_pcm全部都是1，也就是be不创建逻辑设备。其实我觉得这也是为什么区分be与fe的原因了，fe其实就是音频输入，所以fe都会创建对应的逻辑设备。这里提前提一句，关于为什么会有dpcm\_fe\_dai\_open()与soc\_pcm\_open()的选择，其实跟be设备没有pcm逻辑设备是有直接原因的。

dynamic成员对整个音频框架在逻辑上真正起作用的地方其实只有两个（还有几个地方不太影响整体逻辑）：一个是soc-dapm.c中的snd\_soc\_dapm\_connect\_dai\_link\_widgets函数，另一个是soc-pcm.c中的soc\_new\_pcm函数。

第一个地方的作用是区别dynamic dai link和非dynamic dai link在创建声卡时dai widgets的链接行为，对于非dynamic dai link在创建声卡时就把dai widgets链接好，而dynamic dai link是在该dai link对应的pcm逻辑设备被打开的时候才会去进行dai widgets的链接。其实这里就是dynamic和非dynamic最核心的区别的体现。

第二个地方是对整个dynamic和非dynamic最核心的区别的实现。其实soc\_new\_pcm函数最关键的就是绑定rtd->ops了，如果是dynamic的话就绑定dynamic的ops，如果是非dynamic的，则绑定普通的ops。

dynamic pcm设备的open函数为dpcm\_fe\_dai\_open：dynamic pcm里面其实就是先通过当前widgets的链接关系找到所有已经激活的be dai，然后把这个fe dai和所有激活的be dai进行链接，并且对每个be调用soc\_pcm\_open函数，最后对fe自己也调. 这里就又有一个地方可以理解了，从框架上来说，每个pcm设备都需要open之后才能使用，而由于be没有创建pcm逻辑设备，所以be的open操作得由fe来间接执行，以保证框架的一致性。同样，fe调用dpcm\_fe\_dai\_hw\_params函数进行hw params的配置，最终分别对fe和be调用soc\_pcm\_hw\_params函数完成实际的hw\_params操作。这里需要提一下的就是be的hw\_params参数其实是拷贝的fe的.

static int dpcm\_fe\_dai\_open(struct snd\_pcm\_substream \*fe\_substream)

{

\* 找到当前所有已经激活的链路（widgets） \*/ /\* 在所有已经激活的链路中找到所有的be dai，并把be dai与此fe dai链接 \*/

/\* 真正打开fe dai \*/

ret = dpcm\_fe\_dai\_startup(fe\_substream);

}

这个其实前面在分析no\_pcm的时候已经说了，只有fe dai link才会创建pcm逻辑设备，也只有创建了pcm逻辑设备的dai link（对应到这里应该是stream）才会被调用rtd->ops->open()，所以这里的dpcm\_process\_paths(fe, stream, &list, 1);这句话必然就是把当前的fe和所有已经激活的be链接，be可能可以同时存在多个。

static int dpcm\_fe\_dai\_startup(struct snd\_pcm\_substream \*fe\_substream)

{

/\* 实际open be，其实里面还是调用的soc\_pcm\_open() \*/

ret = dpcm\_be\_dai\_startup(fe, fe\_substream->stream);

/\* 还是靠soc\_pcm\_open来完成open操作 \*/

/\* start the DAI frontend \*/

ret = soc\_pcm\_open(fe\_substream)

}

dynamic pcm里面其实就是先通过当前widgets的链接关系找到所有已经激活的be dai，然后把这个fe dai和所有激活的be dai进行链接，并且对每个be调用soc\_pcm\_open函数，最后对fe自己也调用soc\_pcm\_open函数，完成pcm的实际open操作。

**关于acdb:**

**那么asm、adm及afe的管理工作或者说他们的driver都是在ap（cpu）上运行的，也就是对应的q6asm.c、q6adm.c、q6afe.c，这些driver负责提供对这些module的控制接口，以及配置接口，当有音频流需要播放时，platform driver或者dai driver里面调用对应的接口打开、配置正确的asm、adm及afe。**

**除此之外，android在打开一个linux音频逻辑设备进行播放时，还会把asm、adm及afe需要的相应配置从acdb文件中读出来，通过audio calibration这个逻辑设备配置给q6asm.c、q6adm.c、q6afe.c。这些module会将配置保存，并且在open相应module时使用这些配置。**

DSP驱动：

ASM:

**在cpu中，asm对应的其实就是stream，或者说就是一条音频数据流，asm负责完成把音频数据流写给dsp的任务以及对dsp asm的配置工作。 在dsp中，asm负责音频的编解码以及部分的音效处理。**

**asm 怎么工作的：**

**在音频流开始播放的时候，会调用open函数：msm\_pcm\_open. 跟asm相关的主要是q6asm\_audio\_client\_alloc()函数，简单来说就是向asm注册了一个client，建立了一个该substream与dsp的asm通信的通道，这个通道由q6asm.c来管理。ac->apr = apr\_register()是注册了一个到dsp中asm模块下面的对应stream的通信通道，但是这里还需要一个cpu asm到dsp asm模块的通信通道，用于cpu去控制dsp的asm模块，主要体现在两边的smmu后的共享内存上，也就是mmap，所以这里主要是为了两边协商共享内存的地址用。这之后，系统会调**prepare函数**：msm\_pcm\_playback\_prepare(/\* 按照一定的格式打开一个asm，也就是告诉dsp的asm现在的这条流的格式，下面好做准备解析 \*/, /\* 简单来说就是吧acdb配下来的参数发给asm模块 \*/.,)最后trigger和write.**

ADM: adm设备包括路由矩阵和acdb device两个东西。acdb device, 就是QACT里面的Tools->DeviceDesigner里面的一个device, 对应到dsp里面，就是链接copp和audio font port的东西. 那么fe和be具体的连接工作由dsp完成，所以dsp的driver必须告诉dsp他需要怎么连接，那么这个工作在msm8996平台上就是由be platform driver完成的，也就是msm-pcm-routing-v2.c这个文件，这个文件操作dsp中的adm模块，来实现数据的路由，所以，adm模块才是最终数据路由的执行者. 有几个东西就必须知道：1、dsp接入了那些数据；2、dsp要把这些数据从哪些端口送出去；3、每条数据需要用什么处理算法。其实整个be platform driver就是围绕这三个问题展开的，这也是为什么这个dirver的命名中有routing这个词。msm\_bedais：记录了be dai的相关信息，其中主要包括那些fe的session会送到该be dai的afe port中。

fe\_dai\_map：记录了当前所有工作的fe session的信息

fe\_dai\_app\_type\_cfg：

……

上面是qcom的文档中写的，其实这个app type就是选择copp的标志，session\_copp\_map：至于这个全局变量，不算特别重要就不记录了，简单来说就是保存了每个session要经过的copp，这个东西的作用就是给dsp发参数配置时，会从里面读取一下数据。

msm\_bedais在hw parameters里面赋值

fe\_dai\_app\_type\_cfg在Audio Stream %xx App Type Cfg的control里面赋值

cal\_data（topology相关）通过acdb loader把acdb里面的内容发下来。在adm init时会向audio calibration注册cal\_data，并且提供set函数：msm\_routing\_set\_cal，当acdb set参数时回调该函数，完成对cal\_data的写操作

fe\_dai\_map在msm\_pcm\_routing\_reg\_phy\_stream函数中赋值，该函数是msm\_pcm\_prepare时调用的，也就是fe的platform在prepare时调用

adm\_open是在msm\_pcm\_routing\_reg\_phy\_stream函数中创建的，在be platform prepare时由于fe\_dai\_map里面的stream id还没有获取（msm\_pcm\_prepare时才能获取），所以往往adm\_open并不是在be platform的prepare中创建的，也就是不是在msm\_pcm\_routing\_prepare中创建的

msm\_bedais[reg].fe\_sessions在msm\_pcm\_routing\_process\_audio()里面被设置的，msm\_pcm\_routing\_process\_audio()函数是在SLIMBUS\_0\_RX Audio Mixer MultiMedia5控件被control写后触发调用的。这个控件的定义：

AFE:

afe的作用：把adm输出的音频数据发送给codec，同时还可以进行一些音频处理。afe是在be dai prepare的时候被打开的，比如：msm-dai-q6-v2.c里面，调用了函数：int afe\_port\_start(u16 port\_id, union afe\_port\_config \*afe\_config,u32 rate){ */\* 校验参数有效性 \*/ /\* 检查afe的apr是否注册了，如果没注册，则给afe注册一个apr \*/* */\* Also send the topology id here: \*/* */\* 发送AFE\_PORT\_CMD\_DEVICE\_START消息 \*/*}整个q6afe向apr只注册了一个src port为0xFFFFFFFF的apr svc，这样一来，所有跟afe模块相关的apr消息其实都是q6afe来统一管理了. afe里面有个afe port，这个东西在打开afe(afe\_port\_start())时是dai->id确定的，在打开open adm时也会用到，open adm时的这个port id是prepare时，去激活了dai link里面定义的be id对应的msm\_bedais[]，然后再在msm\_pcm\_routing\_reg\_phy\_stream()函数调用时去查询所有active为1的msm\_bedais[]，然后进行open adm。

这里的dai link中be dai赋的值要与msm\_bedais[]中的顺序完全一致，msm\_bedais[]为

APR: apr是基于smd和smmu映射的，smd可以提供若干个通道，让不同设备进行数据交换；smmu可以把外部设备的访问映射为对一[段地址](https://www.baidu.com/s?wd=%E6%AE%B5%E5%9C%B0%E5%9D%80&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)的访问，也就是说如果外部设备有一段可以访问的内存，那么可以直接通过smmu把这段内存映射给cpu，cpu可以直接去访问这段内存。所有需要通过apr通信的模块都会把自己的一些信息保存进这个client中，然后通过handle与dsp交互数据, slimbus：qcom的一个链接dsp、codec的总线，具体总线的工作时序和协议不清楚，但是从driver里面来看，该总线设计了若干个channel（好像是32个, 上述定义是在声卡初始化dai link被创建时通过dai link的init来实现的，相关代码见：msm8996.c中的msm\_audrx\_init()函数。该函数中调用了：

snd\_soc\_dai\_set\_channel\_map(codec\_dai,ARRAY\_SIZE(tx\_ch),tx\_ch,ARRAY\_SIZE(rx\_ch), rx\_ch);来实现对codec的channel进行初始化：