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

1. Codec DAI and PCM configuration
2. Codec control IO - using RegMap API
3. Mixers and audio controls
4. Codec audio operations
5. DAPM description.
6. DAPM event handler.

Optionally, codec drivers can also provide:-

1. DAC Digital mute control.

**SoC DAI Drivers**

Each SoC DAI driver must provide the following features:-

1. Digital audio interface (DAI) description
2. Digital audio interface configuration
3. PCM’s description
4. SYSCLK configuration
5. Suspend and resume (optional)

## SoC DSP Drivers

Each SoC DSP driver usually supplies the following features :-

1. DAPM graph
2. Mixer controls
3. DMA IO to/from DSP buffers (if applicable)
4. 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

The audio driver processes this as follows :-

1. Machine driver receives Jack removal event.
2. Machine driver OR audio HAL disables the Headset path.
3. DPCM runs the PCM trigger(stop), hw\_free(), shutdown() operations on DAI0 for headset since the path is now disabled.
4. Machine driver or audio HAL enables the speaker path.
5. DPCM runs the PCM ops for startup(), hw\_params(), prepapre() and trigger(start) for DAI1 Speakers since the path is enabled.

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 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。

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。

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：

**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。

。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。

回放情形下，pcm\_dma 设备负责把 dma buffer 中的数据搬运到 I2S tx FIFO，I2S 总线控制器负责把 I2S tx FIFO 中的数据传送到 Codec。

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。

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；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，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）