动态音频电源管理的意思，DAPM是为了使基于Linux的移动设备上的音频子系统，在任何时候都工作在最小功耗状态下。DAPM对用户空间的应用程序来说是透明的，所有与电源相关的开关都在ASoc core中完成。用户空间的应用程序无需对代码做出修改，也无需重新编译，DAPM根据当前激活的音频流（playback/capture）和声卡中的mixer等的配置来决定那些音频控件的电源开关被打开或关闭。

一个kcontrol代表着一个mixer（混音器），或者是一个mux（多路开关），又或者是一个音量控制器等等。 从上述文章中我们知道，定义一个kcontrol主要就是定义一个snd\_kcontrol\_new结构. 对于每个控件，我们需要定义一个和他对应的snd\_kcontrol\_new结构，这些snd\_kcontrol\_new结构会在声卡的初始化阶段，通过snd\_soc\_add\_codec\_controls函数注册到系统中.

们会为mixer的每个输入端都单独定义一个简单控件来控制该路输入的开启和关闭，反应在代码上，就是定义一个soc\_kcontrol\_new数组。

mux控件与mixer控件类似，也是多个输入端和一个输出端的组合控件，与mixer控件不同的是，mux控件的多个输入端同时只能有一个被选中。因此，mux控件所对应的寄存器，通常可以设定一段连续的数值，每个不同的数值对应不同的输入端被打开，与上述的mixer控件不同，ASoc用soc\_enum结构来描述mux控件的寄存器信息。

kcontrol。利用kcontrol，我们可以完成对音频系统中的mixer，mux，音量控制，音效控制，以及各种开关量的控制，通过对各种kcontrol的控制，使得音频硬件能够按照我们预想的结果进行工作

kcontrol还是有以下几点不足：

• 只能描述自身，无法描述各个kcontrol之间的连接关系；

• 没有相应的电源管理机制；

• 没有相应的时间处理机制来响应播放、停止、上电、下电等音频事件；

• 为了防止pop-pop声，需要用户程序关注各个kcontrol上电和下电的顺序；

• 当一个音频路径不再有效时，不能自动关闭该路径上的所有的kcontrol；

DAPM框架为了解决这些问题，引入了widget这一概念, 在DAPM框架中，把各种不同的widget分为不同的种类，snd\_soc\_dapm\_widget结构中的id字段用来表示该widget的种类.

一个widget是有输入和输出的，而且widget之间是可以动态地进行连接的，那它们是用什么来连接两个widget的呢？DAPM为我们提出了path这一概念，path相当于电路中的一根跳线，它把一个widget的输出端和另一个widget的输入端连接在一起，path用snd\_soc\_dapm\_path结构来描述, 连接的路径是这样的：起始端widget的输出-->path的输入-->path的输出-->到达端widget输入。一个路径的连接至少包含以下几个元素：起始端widget，跳线path，到达端widget，在DAPM中，用snd\_soc\_dapm\_route结构来描述这样一个连接关系. 该结构的意义很明显就是：source通过一个kcontrol，和sink连接在一起，现在是否处于连接状态，请调用connected回调函数检查。

所有定义好的route，最后都要注册到dapm系统中，dapm会根据这些名字找出相应的widget，并动态地生成所需要的snd\_soc\_dapm\_path结构，正确地处理各个链表和指针的关系，实现两个widget之间的连接. 其中snd\_soc\_dapm\_path无需我们自己定义，它会在注册snd\_soc\_dapm\_route时动态地生成，但是对于系统中的widget和route，我们是需要自己进行定义的，另外，widget所包含的kcontrol与普通的kcontrol有所不同，它们的定义方法与标准的kcontrol也有所不同。

根据widget的类型，按照它们的电源所在的域，被分为了几个域，他们分别是：

• codec域 比如VREF和VMID等提供参考电压的widget，这些widget通常在codec的probe/remove回调中进行控制，当然，在工作中如果没有音频流时，也可以适当地进行控制它们的开启与关闭。

• platform域 位于该域上的widget通常是针对平台或板子的一些需要物理连接的输入/输出接口，例如耳机、扬声器、麦克风，因为这些接口在每块板子上都可能不一样，所以通常它们是在machine驱动中进行定义和控制，并且也可以由用户空间的应用程序通过某种方式来控制它们的打开和关闭。

• 音频路径域 一般是指codec内部的mixer、mux等控制音频路径的widget，这些widget可以根据用户空间的设定连接关系，自动设定他们的电源状态。

• 音频数据流域 是指那些需要处理音频数据流的widget，例如ADC、DAC等等。

每个codec有多个dai，而cpu（通常就是指某个soc cpu芯片）也会有多个dai，dai注册时，dapm系统会为每个dai创建一个snd\_soc\_dapm\_dai\_in或snd\_soc\_dapm\_dai\_out类型的widget，通常，这两种widget会和codec中具有相同的stream name的widget进行连接。dapm利用这些kcontrol完成音频路径的控制。不过，对于widget来说，它的任务还不止这些，dapm还要动态地管理这些音频路径的连结关系，以便可以根据这些连接关系来控制这些widget的电源状态..

对于音频路径上的mixer或mux类型的widget，它们包含了若干个kcontrol，这些被包含的kcontrol实际上就是我们之前讨论的mixer和mux等，dapm利用这些kcontrol完成音频路径的控制。不过，对于widget来说，它的任务还不止这些，dapm还要动态地管理这些音频路径的连结关系，以便可以根据这些连接关系来控制这些widget的电源状态，如果按照普通的方法定义这些kcontrol，是无法达到这个目的的.

相比普通的kcontrol控件，dapm的kcontrol控件只是把info，get，put回调函数换掉了。dapm kcontrol的put回调函数不仅仅会更新控件本身的状态，他还会把这种变化传递到相邻的dapm kcontrol，相邻的dapm kcontrol又会传递这个变化到他自己相邻的dapm kcontrol，知道音频路径的末端，通过这种机制，只要改变其中一个widget的连接状态，与之相关的所有widget都会被扫描并测试一下自身是否还在有效的音频路径中，从而可以动态地改变自身的电源状态，这就是dapm的精髓所在。

从代码入手，分析dapm的详细工作原理：

• 如何注册widget

• 如何连接两个widget

• 一个widget的状态裱画如何传递到整个音频路径中

dapm把整个音频系统，按照功能和偏置电压级别，划分为若干个电源域，每个域包含各自的widget，每个域中的所有widget通常都处于同一个偏置电压级别上，而一个电源域就是一个dapm context，通常会有以下几种dapm context：

• 属于codec中的widget位于一个dapm context中

• 属于platform的widget位于一个dapm context中

• 属于整个声卡的widget位于一个dapm context中

对于音频系统的硬件来说，通常要提供合适的偏置电压才能正常地工作，有了dapm context这种组织方式，我们可以方便地对同一组widget进行统一的偏置电压管理，ASoc用snd\_soc\_dapm\_context结构来表示一个dapm context, snd\_soc\_dapm\_context被内嵌到代表codec、platform、card、dai的结构体中, 代表widget结构snd\_soc\_dapm\_widget中，有一个snd\_soc\_dapm\_context结构指针，指向所属的codec、platform、card、或dai的dapm结构。同时，所有的dapm结构，通过它的list字段，链接到代表声卡的snd\_soc\_card结构的dapm\_list链表头字段。

通常，我们会根据音频硬件的组成，分别在声卡的codec驱动、platform驱动和machine驱动中定义一组widget，这些widget用数组进行组织，我们一般会使用dapm框架提供的大量的辅助宏来定义这些widget数组.

codec驱动中注册 我们知道，我们会通过ASoc提供的api函数snd\_soc\_register\_codec来注册一个codec驱动，该函数的第二个参数是一个snd\_soc\_codec\_driver结构指针，这个snd\_soc\_codec\_driver结构需要我们在codec驱动中显式地进行定义，其中有几个与dapm框架有关的字段.

我们只要把我们定义好的snd\_soc\_dapm\_widget结构数组的地址和widget的数量赋值到dapm\_widgets和num\_dapm\_widgets字段即可，这样，经过snd\_soc\_register\_codec注册codec后，在machine驱动匹配上该codec时，系统会判断这两个字段是否被赋值，如果有，它会调佣dapm框架提供的api来创建和注册widget，注意这里我说还要创建这个词，你可能比较奇怪，既然代表widget的snd\_soc\_dapm\_widget结构数组已经在codec驱动中定义好了，为什么还要在创建？事实上，我们在codec驱动中定义的widget数组只是作为一个模板，dapm框架会根据该模板重新申请内存并初始化各个widge.

有时候我们为了代码的清晰，可能会根据功能把不同的widget定义成多个数组，但是snd\_soc\_codec\_driver中只有一个dapm\_widgets字段，无法设定多个widget数组，这时候，我们需要主动在codec的probe回调中调用dapm框架提供的api来创建这些widget.

platform驱动中注册 和codec驱动一样.

有些widget可能不是位于codec中，例如一个独立的耳机放大器，或者是喇叭功放等，这种widget通常需要在machine驱动中注册，通常他们的dapm context也从属于声卡（snd\_soc\_card）域。做法依然和codec驱动类似，只要把定义好的widget数组和数量赋值给dapm\_widgets指针和num\_dapm\_widgets即可，注册声卡使用的api：snd\_soc\_register\_card()，也会通过snd\_soc\_dapm\_new\_controls来完成widget的创建工作。

系统中注册的各种widget需要互相连接在一起才能协调工作，连接关系通过snd\_soc\_dapm\_route结构来定义/和widget一样，路径信息也分别存在与codec驱动，machine驱动和platform驱动中，我们一样有两种方式来注册音频路径信息：

• 通过snd\_soc\_codec\_driver/snd\_soc\_platform\_driver/snd\_soc\_card结构中的dapm\_routes和num\_dapm\_routes字段；

• 在codec、platform的的probe回调中主动注册音频路径，machine驱动中则通过snd\_soc\_dai\_link结构的init回调函数来注册音频路径；

在dapm框架中，还有另外一种widget，它代表了一个dai（数字音频接口）, dai按所在的位置，又分为cpu dai和codec dai，在硬件上，通常一个cpu dai会连接一个codec dai，而在machine驱动中，我们要在snd\_soc\_card结构中指定一个叫做snd\_soc\_dai\_link的结构，该结构定义了声卡使用哪一个cpu dai和codec dai进行连接。

dai由codec驱动和平台代码中的iis或pcm接口驱动注册，machine驱动负责找到snd\_soc\_dai\_link中指定的一对cpu/codec dai，并把它们进行绑定。不管是cpu dai还是codec dai，通常会同时传输播放和录音的音频流的能力，所以我们可以看到，snd\_soc\_dai中有两个widget指针，分别代表播放流和录音流。

这两个dai widget是何时创建的呢？

首先，codec驱动在注册codec时，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. 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.

实际上dai widget是一条完整dapm音频路径的重要元素，没有她，我们无法完成dapm的动态电源管理工作，因为它是音频流和其他widget的纽带.

* snd\_soc\_dapm\_new\_controls

实际上，这个函数只是创建widget的第一步，它为每个widget分配内存，初始化必要的字段，然后把这些widget挂在代表声卡的snd\_soc\_card的widgets链表字段中。要使widget之间具备连接能力，我们还需要第二个函数：

* snd\_soc\_dapm\_new\_widgets

这个函数会根据widget的信息，创建widget所需要的dapm kcontrol，这些dapm kcontol的状态变化，代表着音频路径的变化，从而影响着各个widget的电源状态。

当音频路径发生变化时，power\_check回调会被调用，用于检查该widget的电源状态是否需要更新。power\_check设置完成后，需要设置widget所属的codec、platform和dapm context，几个用于音频路径的链表也需要初始化，然后，把该widget加入到声卡的widgets链表中。

再次强调一下snd\_soc\_dapm\_new\_controls函数所完成的主要功能：

* 为widget分配内存，并拷贝参数中传入的在驱动中定义好的模板
* 设置power\_check回调函数
* 把widget挂在声卡的widgets链表中

dapm框架在声卡的初始化阶段，等所有的widget（包括machine、platform、codec）都创建好之后，通过snd\_soc\_dapm\_new\_widgets函数，创建widget内包含的dapm kcontrol，并初始化widget的初始电源状态和音频路径的初始连接状态。

如果widget之间没有连接关系，dapm就无法实现动态的电源管理工作，正是widget之间有了连结关系，这些连接关系形成了一条所谓的完成的音频路径，dapm可以顺着这条路径，统一控制路径上所有widget的电源状态，前面我们已经知道，widget之间是使用snd\_soc\_path结构进行连接的，驱动要做的是定义一个snd\_soc\_route结构数组，该数组的每个条目描述了目的widget的和源widget的名称，以及控制这个连接的kcontrol的名称，最终，驱动程序使用api函数snd\_soc\_dapm\_add\_routes来注册这些连接信息.

我们为声卡创建并初始化好了所需的widget，各个widget也通过path连接在了一起，接下来，dapm等待用户的指令，一旦某个dapm kcontrol被用户空间改变，利用这些连接关系，dapm会重新创建音频路径，脱离音频路径的widget会被下电，加入音频路径的widget会被上电，所有的上下电动作都会自动完成，用户空间的应用程序无需关注这些变化，它只管按需要改变某个dapm kcontrol即可。

设计dapm的主要目的之一，就是希望声卡上的各种部件的电源按需分配，需要的就上电，不需要的就下电，使得整个音频系统总是处于最小的耗电状态，最主要的就是，这一切对用户空间的应用程序是透明的，也就是说，用户空间的应用程序无需关心那个部件何时需要电源，它只要按需要设定好音频路径，播放音频数据，暂停或停止，dapm框架会根据音频路径，完美地对各种部件的电源进行控制，而且精确地按某种顺序进行，防止上下电过程中产生不必要的pop-pop声。

dapm要给一个widget上电的其中一个前提条件是：这个widget位于一条完整的音频路径上，而一条完整的音频路径的两头，必须是输入/输出引脚，或者是一个外部音频设备，又或者是一个处于激活状态的音频流widget。

在代表声卡的snd\_soc\_card结构中，有一个链表字段：dapm\_dirty，所有状态发生了改变的widget，dapm不会立刻处理它的电源状态，而是需要先挂在该链表下面，等待后续的进一步处理：或者是上电，或者是下电。

我们知道，在创建widget的时候，widget的power\_check回调函数会根据widget的类型，设置不同的回调函数。当widget的状态改变后，dapm会遍历dapm\_dirty链表，并通过power\_check回调函数，决定该widget是否需要上电。

当一个widget的状态改变后，该widget会被加入dapm\_dirty链表，然后通过dapm\_power\_widgets函数来改变整个音频路径上的电源状态, 该函数通过遍历dapm\_dirty链表，对每个链表中的widget调用dapm\_power\_one\_widget，dapm\_power\_one\_widget函数除了处理自身的状态改变外，还把自身的变化传递到和它相连的邻居widget中，结果就是，所有需要上电的widget会被放在up\_list链表中，而所有需要下电的widget会被放在down\_list链表中.

一个widget的状态改变，邻居widget会受到“感染”而被加入到dapm\_dirty链表的末尾，所以扫描到链表的末尾时，邻居widget也会执行同样的操作，从而“感染”邻居的邻居，直到没有新的widget被加入dapm\_dirty链表为止，这时，所有受到影响的widget都被加入到up\_list或down\_li链表中，等待后续的上下电操作。

最终的上电或下电操作需要通过codec的寄存器来实现，因为定义widget时，如果这是一个带电源控制的widget，我们必须提供reg/shift等字段的设置值，如果该widget无需寄存器控制电源状态，则reg字段必须赋值为：

* SND\_SOC\_NOPM        (该宏定义的实际值是-1)

这些过程总是需要一个起始点：是谁触动了dapm，使得它需要执行上述的扫描和上电过程？事实上，以下几种情况可以触发dapm发起一次扫描操作：

* 声卡初始化阶段，snd\_soc\_dapm\_new\_widgets函数创建widget包含的kcontrol后，会触发一次扫描操作。
* 用户空间的应用程序修改了widget中包含的dapm kcontrol的配置值时，会触发一次扫描操作。
* pcm的打开或关闭，会通过音频流widget触发一次扫描操作。
* 驱动程序在改变了某个widget并把它加入到dapm\_dirty链表后，主动调用snd\_soc\_dapm\_sync函数触发扫描操作。

dapm框架中的另一个机制：事件机制。通过dapm事件机制，widget可以对它所关心的dapm事件做出反应，这种机制对于扩充widget的能力非常有用，例如，对于那些位于codec之外的widget，好像喇叭功放、外部的前置放大器等等，由于不是使用codec内部的寄存器进行电源控制，我们就必须利用dapm的事件机制，获得相应的上下电事件，从而可以定制widget自身的电源控制功能。

在所有需要处理电源变化的widget被分别放入up\_list和down\_list链表后，会相应地发出各种dapm事件, 由此可见，dai widget和stream widget是通过stream name进行匹配的，所以，我们在定义codec的stream widget时，它们的stream name必须要包含dai的stream name，这样才能让ASoc自动把这两种widget连接在一起，只有把它们连接在一起，ASoc中的播放、录音和停止等事件，才能通过dai widget传递到codec中，使得codec中的widget能根据目前的播放状态，动态地开启或关闭音频路径上所有widget的电源。

 ASoC 硬件设备驱动的三大构成：Codec、Platform 和 Machine，下面列举各驱动的功能构成：

**ASoC Codec Driver**：

* Codec DAI 和 PCM 的配置信息
* Codec 的控制接口，如 I2C/SPI
* Mixer 和其他音频控件
* Codec 的音频接口函数，见 snd\_soc\_dai\_ops 结构体定义
* DAPM 描述信息
* DAPM 事件处理句柄
* DAC 数字静音控制

**ASoC Platform Driver**： 包括 dma 和 cpu\_dai 两部分：

* dma 驱动实现音频 dma 操作，具体见 snd\_pcm\_ops 结构体定义
* cpu\_dai 驱动实现音频数字接口控制器的描述和配置

**ASoC Machine Driver**：

* 作为链结 Platform 和 Codec 的载体，它必须配置 dai\_link 为音频数据链路指定 Platform 和 Codec
* 处理机器特有的音频控件和音频事件，例如回放时打开外放功放