static u8 lc1160\_codec\_reg[LC1160\_CACHEREGNUM] = {

0x00, /\*R0\*/

0xAD, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x72, 0xFA, /\*R01~R10\*/

0x3C, 0x72, 0x1F, 0xE7, 0x00, 0x28, 0x00, 0x32, 0x00, 0x3F, /\*R11~R20\*/

0x67, 0x3F, 0xE7, 0x00, 0x28, 0x00, 0x32, 0x00, 0x3F, 0xFF, /\*R21~R30\*/

0xE0, 0x60, 0x67, 0x67, 0x00, 0x07, 0x23, 0x7F, 0x00, 0x7F, /\*R31~R40\*/

0x00, 0x32, 0x00, 0x02, 0x02, 0x02, 0x02, 0x02, 0x33, 0x00, /\*R41~R50\*/

0x07, 0x23, 0xE7, 0x67, 0x7F, 0x00, 0x7F, 0x00, 0x32, 0x00, /\*R51~R60\*/

0x1F, 0x07, 0x26, 0xD5, 0x8F, 0x82, 0x0A, 0x34, 0x2C, 0x00, /\*R61~R70\*/

0x1B, 0x1B, 0x19, 0x19, 0x07, 0x08, 0x27, 0x00, 0x00, 0x00, /\*R71~R80\*/

0x00, /\*R81 Read Only\*/

0x4c, //\*LDOA15 For Codec,addr:0x3b\* /\*\*/

/\* Jack & Hookswitch,addr:0xa7,0xa8,0xa9,0xaa,0xab\*/

0x00,0x04,0x04,0x04,0x00,

0x00, /\*DBB\_PCM\_SWITCH\*/

0x00, /\*PA\_ENABLE\*/

};

static struct snd\_soc\_codec\_driver soc\_codec\_dev\_lc1160 = {

.probe = lc1160\_probe,

.remove = lc1160\_remove,

.read = lc1160\_read\_reg\_cache,

.write = lc1160\_write,

.set\_bias\_level = lc1160\_set\_bias\_level,

.reg\_cache\_size = sizeof(lc1160\_codec\_reg),

.reg\_word\_size = sizeof(u8),

.reg\_cache\_default = lc1160\_codec\_reg,

.ignore\_pmdown\_time = true,

.controls = lc1160\_snd\_controls,

.num\_controls = ARRAY\_SIZE(lc1160\_snd\_controls),

.dapm\_widgets = lc1160\_dapm\_widgets,

.num\_dapm\_widgets = ARRAY\_SIZE(lc1160\_dapm\_widgets),

.dapm\_routes = intercon,

.num\_dapm\_routes = ARRAY\_SIZE(intercon),

};

lc1160\_codec\_probe

--->1:snd\_soc\_register\_codec(&pdev->dev,&soc\_codec\_dev\_lc1160,

lc1160\_dai, ARRAY\_SIZE(lc1160\_dai));

static struct snd\_soc\_dai\_driver lc1160\_dai[] = {

{

.name = "comip\_hifi",

.playback = {

.stream\_name = "Playback",

.channels\_min = 1,

.channels\_max = 2,

.rates = COMIP\_1160\_RATES,

.formats = COMIP\_1160\_FORMATS,

},

.capture = {

.stream\_name = "Capture",

.channels\_min = 1,

.channels\_max = 2,

.rates = COMIP\_1160\_RATES,

.formats = COMIP\_1160\_FORMATS,

},

.ops = &lc1160\_dai\_ops,

},

{

.name = "comip\_voice",

.playback = {

.stream\_name = "VxDL",

.channels\_min = 1,

.channels\_max = 2,

.rates = COMIP\_1160\_RATES,

.formats = COMIP\_1160\_FORMATS,

},

.capture = {

.stream\_name = "VxUL",

.channels\_min = 1,

.channels\_max = 2,

.rates = COMIP\_1160\_RATES,

.formats = COMIP\_1160\_FORMATS,

},

.ops = &lc1160\_dai\_ops,

},

{

.name = "virtual\_codec",

.playback = {

.stream\_name = "Play",

.channels\_min = 1,

.channels\_max = 2,

.rates = COMIP\_1160\_RATES,

.formats = COMIP\_1160\_FORMATS,

},

.capture = {

.stream\_name = "Cap",

.channels\_min = 1,

.channels\_max = 2,

.rates = COMIP\_1160\_RATES,

.formats = COMIP\_1160\_FORMATS,

},

}

};

static struct snd\_soc\_dai\_ops lc1160\_dai\_ops = {

.shutdown = lc1160\_shutdown,

.hw\_params = lc1160\_hw\_params,

.set\_sysclk = lc1160\_set\_dai\_sysclk,

.set\_fmt = lc1160\_set\_dai\_fmt,

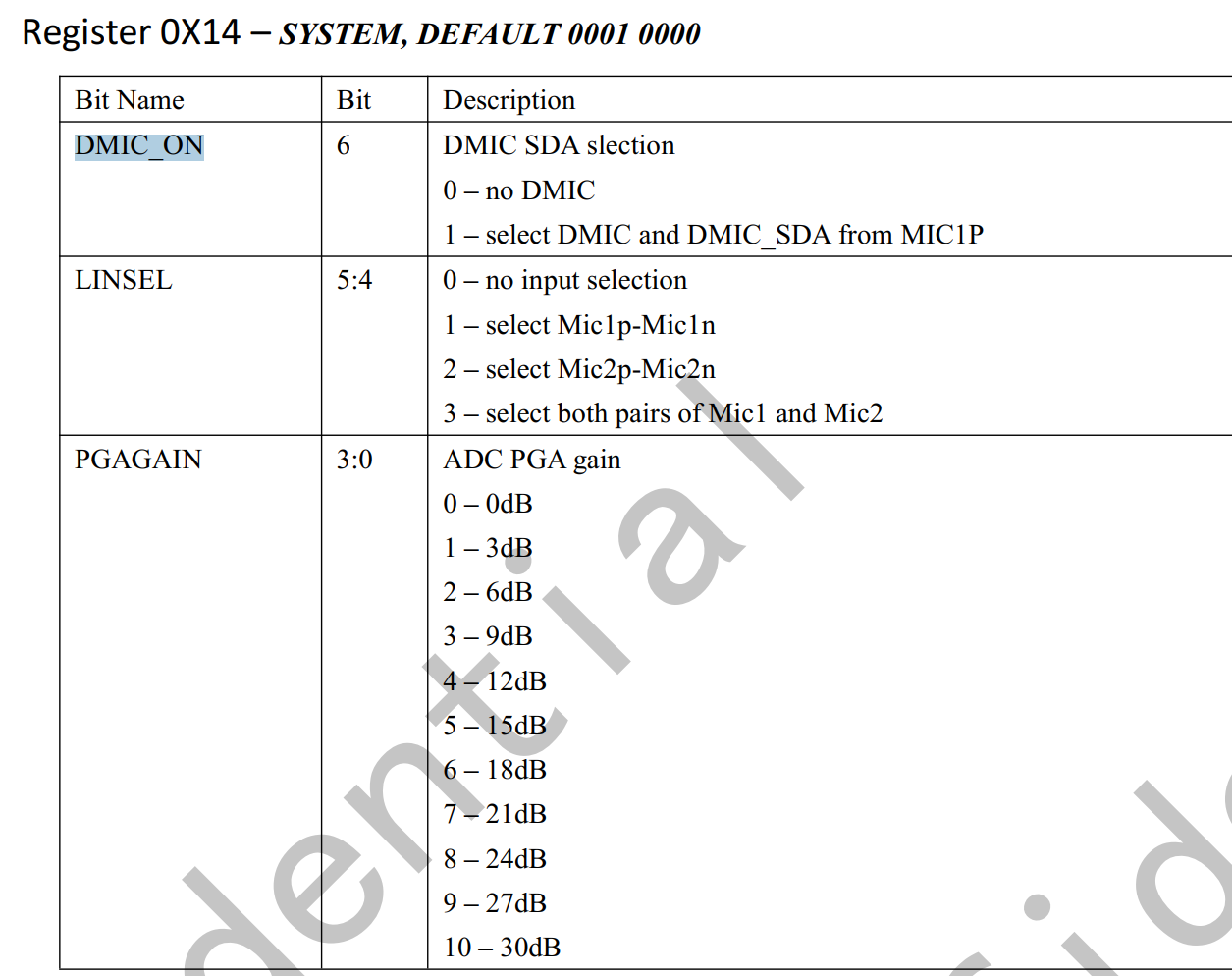
};

tinymix  -D 1  25  150 ==> ok DAC===》 150

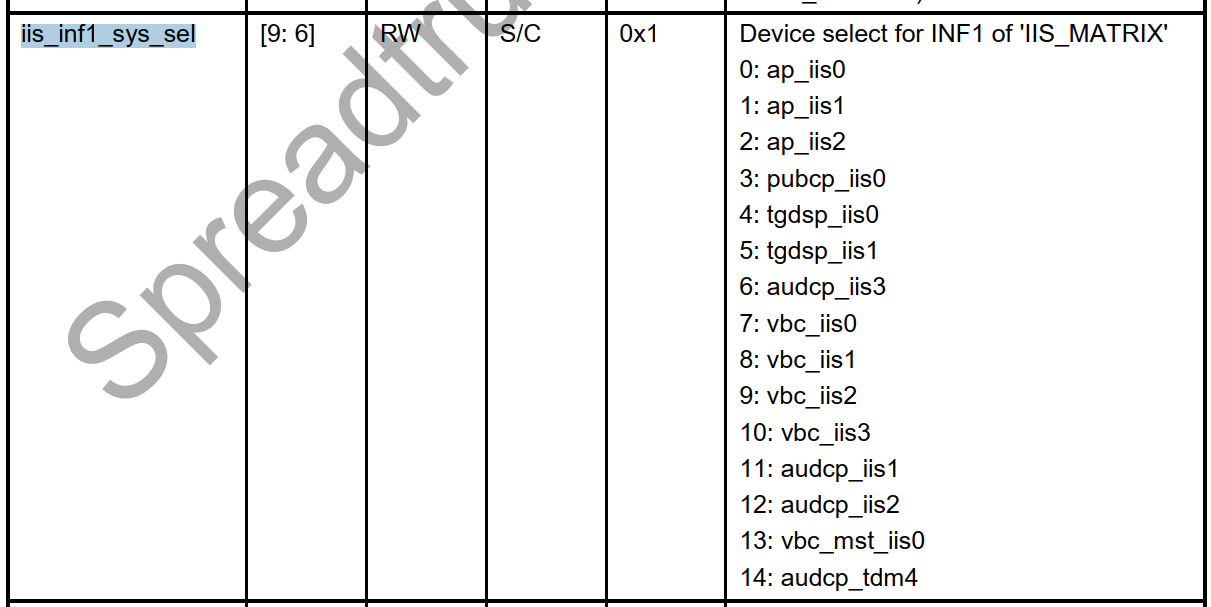
tinymix  -D 1  0 0  ===> ok  adc pga 设置为0

Es8311的回声串绕有改善。

===



====



iis1 默认接通ap iis1

添加iis1:

                                         <&g12\_pll CLK\_TWPLL\_153M6>;

                        };

+

+                       i2s1: i2s@70d00000 {

+                               compatible = "sprd,i2s";

+                               reg = <0 0x70d00000 0 0x1000>;

+                               sprd,dai\_name = "i2s\_bt\_sco1";

+                               sprd,hw\_port = <1>;

+                               sprd,syscon-ap-apb = <&ap\_apb\_regs>;

+                               #sound-dai-cells = <0>;

+                               status = "disable";

+                               clock-names = "clk\_iis1",

+                                             "clk\_twpll\_128m",

+                                             "clk\_twpll\_153m6";

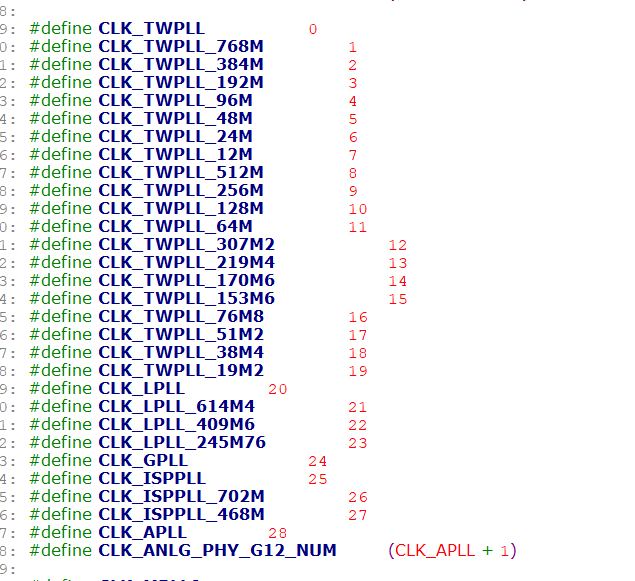
+                               clocks = <&ap\_clk CLK\_AP\_IIS1>,

+                                        <&g12\_pll CLK\_TWPLL\_128M>,

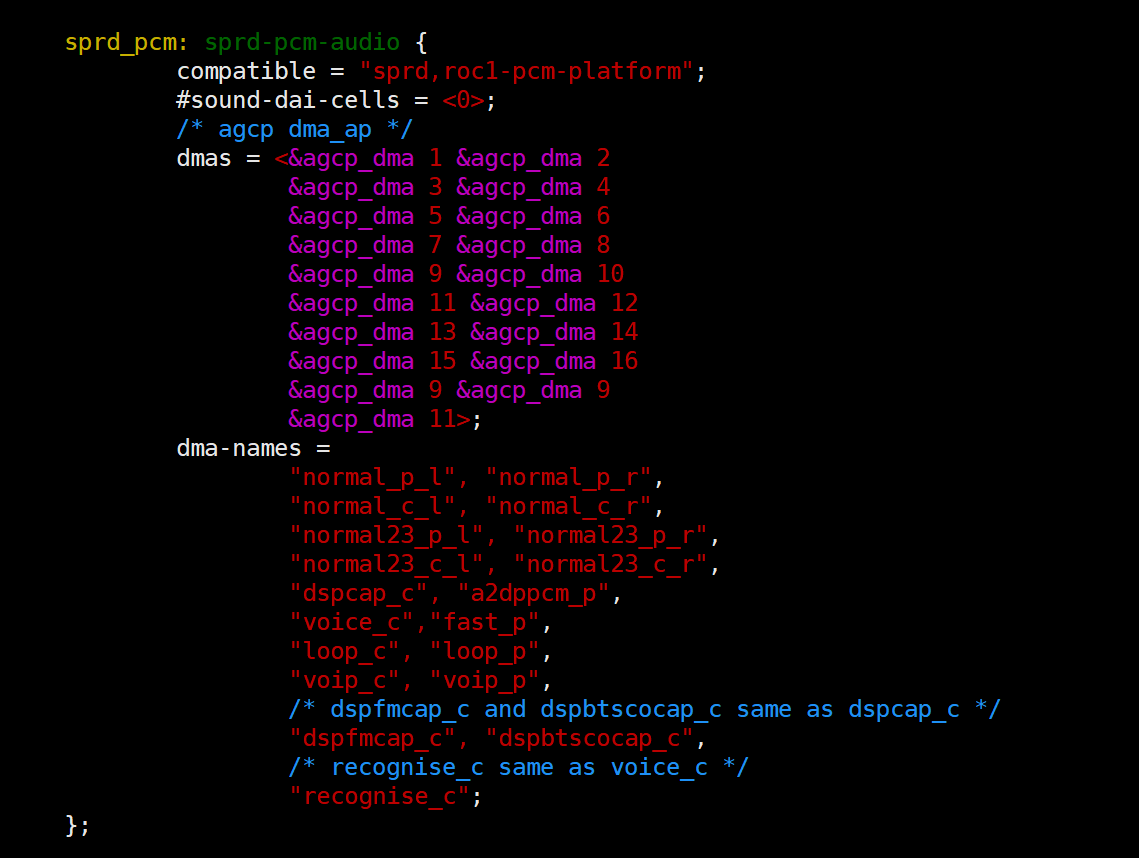
+                                        <&g12\_pll CLK\_TWPLL\_153M6>;

+                       };

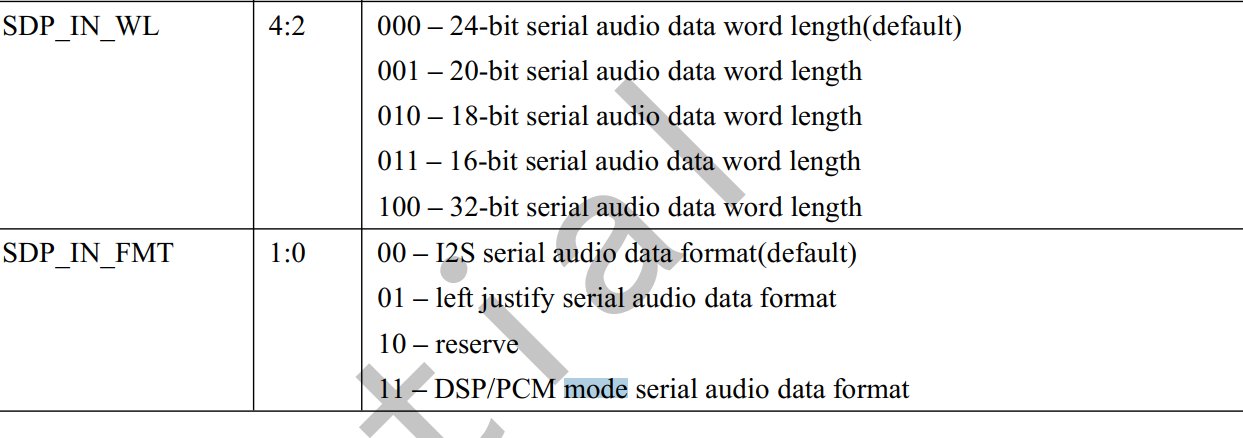
====

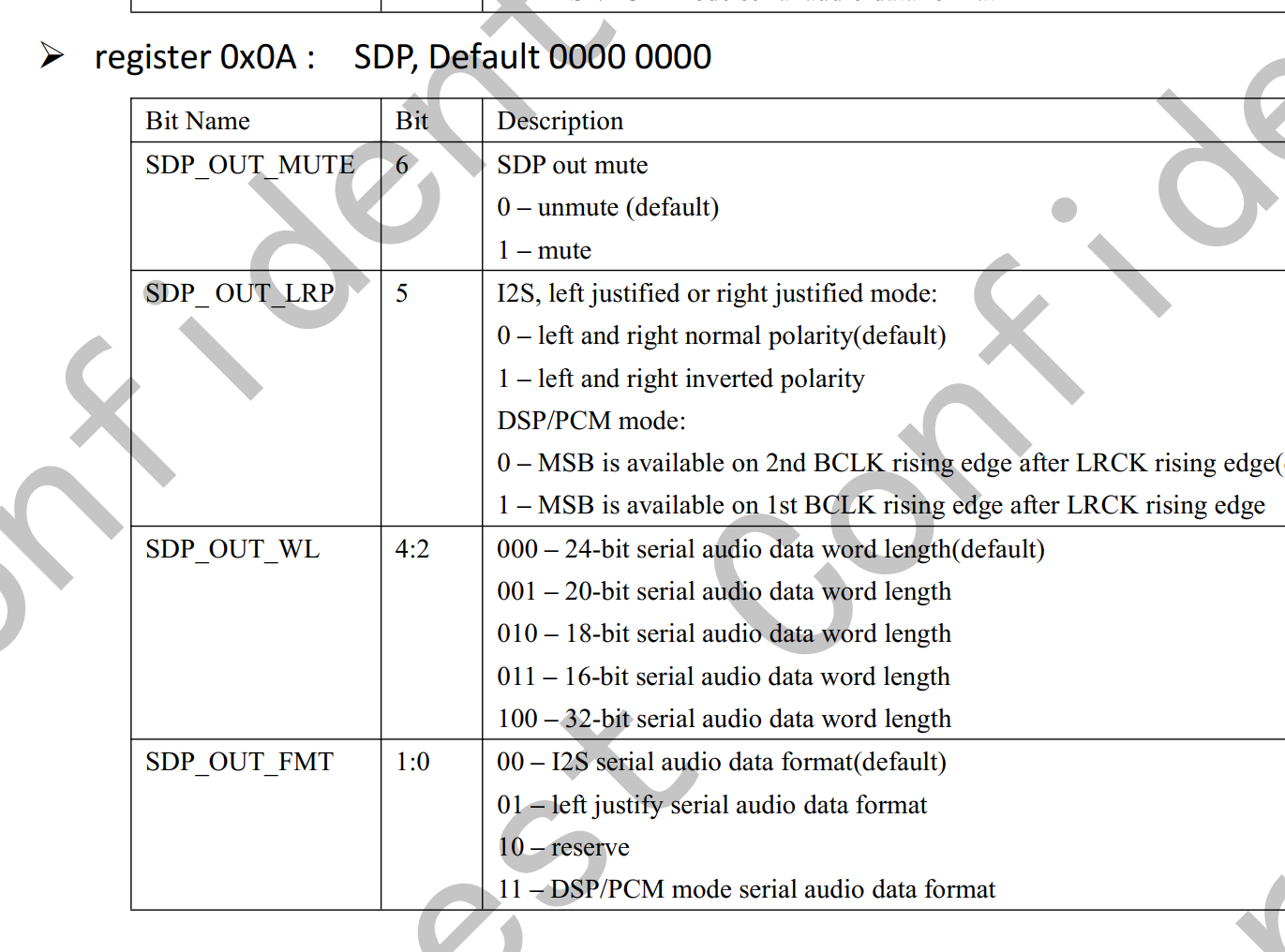


pcm  audio

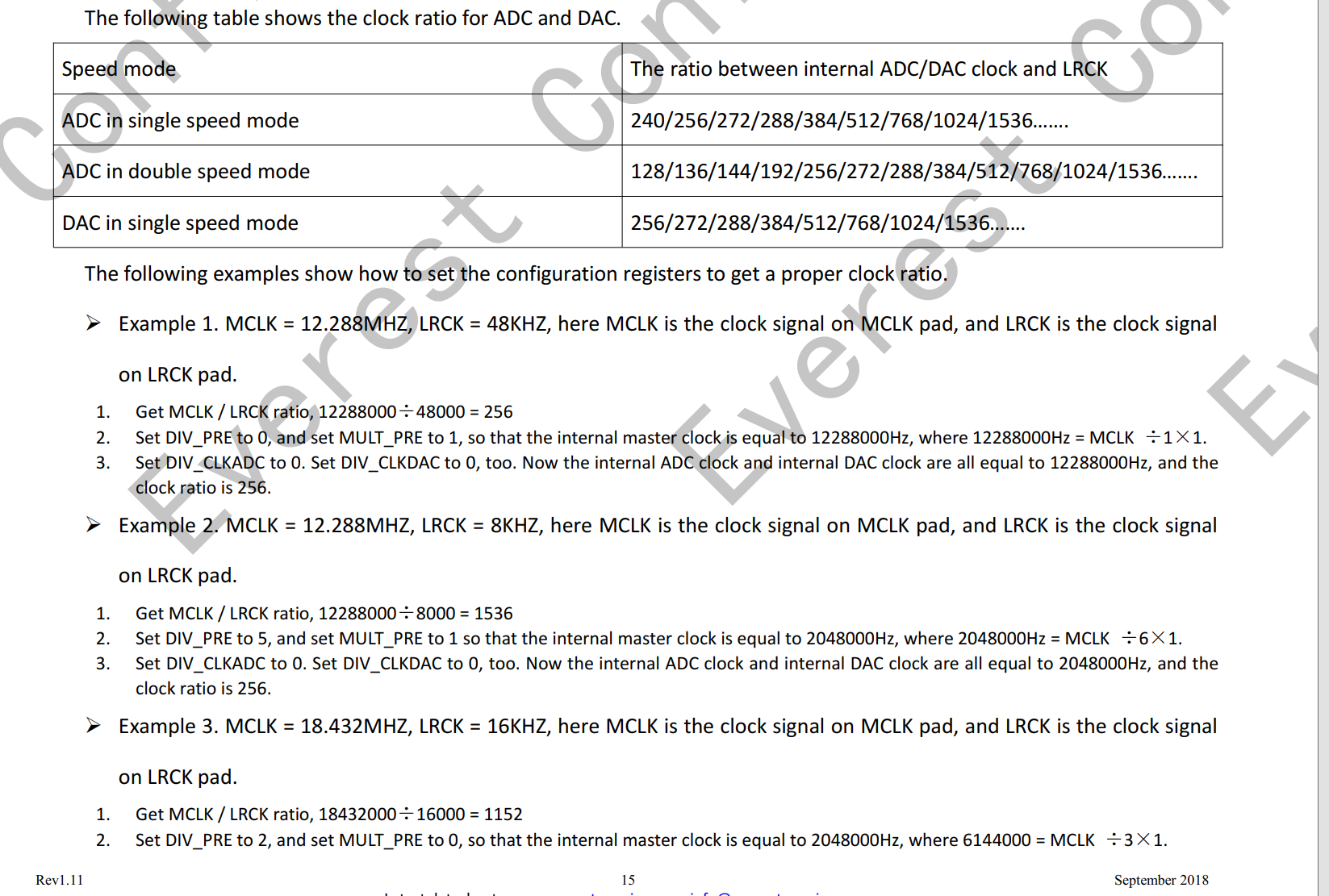


模式选择：





时钟设置：



==

&i2s0 {

        status = "okay";

        sprd,config\_type = "i2s";//i2s和pcm

        sprd,slave\_timeout = <0xf11>;//slave模式使用  超时时间长度

        sprd,\_hw\_port = <0>;// i2s hardware port number

        sprd,fs = <48000>;//采用率

        sprd,bus\_type = <0>;//总线模式 0：i2s 1:pcm

        sprd,rtx\_mode = <3>;//0 idle 1:只接收 2：只发送  3：双工收发

        sprd,byte\_per\_chan = <1>;//每个i2s通道有多少个byte  0 8bit, 1 16 bit, 2 32 bit, 3 32 bit

        sprd,slave\_mode = <0>;//0 master, 1 slave

        sprd,lsb = <0>;//0 :msb is first bit to transmit, 1 lsb is first bit to transmit

        sprd,lrck = <1>; /\*sync\_mode\*/ /0 output lrck, 1 sync. 0:输出的是lrck的信号  1：长帧信号

        sprd,low\_for\_left = <0>; /\*lrck\_inv\*/ //0 low\_for\_left, 1 high\_for\_left 左声道的位置---->默认为1

        sprd,clk\_inv = <1>;//0 not invert, 1 invert 时钟是否反正

        sprd,pcm\_short\_frame = <0>; /\*pcm\_bus\_mode\*/  0:I2S\_LONG\_FRAME     1:I2S\_SHORT\_FRAME

        sprd,pcm\_slot = <0x1>;//1：一个i2s通道  最多3个 pcm\_slot means channel count, support 3 slots. i2s 通道的个数

        sprd,pcm\_cycle = <1>;//slots use how many cycle, for exsample 8bit for each slot, 3 slot, 4 cycle. There will 4 8bit output, 3 8bit is valid, 1 8bit is dymmy.

        sprd,tx\_watermark = <12>;//iis tx watermark for dma request or interrupt, not large than 32

        sprd,rx\_watermark = <20>;//iis rx watermark for dma request or interrupt, not large than 32

        sprd,i2s\_compatible= <1>;//i2s\_compatible the same bit for pcm\_short\_frame. And for i2s bus mode we call this bit as i2s\_compatible.

};

0011 1111 1000

采样率8000  16000 32000  时钟： devm\_clk\_get(i2s->dev, "clk\_twpll\_128m");

采样率:9600  12000 24000  48000 时钟：devm\_clk\_get(i2s->dev, "clk\_twpll\_153m6");

Example:

i2s configured as 96k 32bit

i2s0: i2s@70c00000  {

    compatible = "sprd,i2s";

    reg = <0x70c00000 0x100000>;

    sprd,dai\_name = "i2s\_bt\_sco0";

    sprd,hw\_port = <0>;

    sprd,syscon-ap-apb = <&ap\_apb\_controller>;

    sprd,config\_type = "i2s";

    sprd,slave\_timeout = <0xF11>;

    sprd,fs = <96000>;

    sprd,bus\_type = <0>;

    sprd,rtx\_mode = <3>;

    sprd,byte\_per\_chan = <2>; //32bit  对应i2s left justified模式

    sprd,slave\_mode = <0>;

    sprd,lsb = <0>;

    sprd,lrck = <1>;

    sprd,low\_for\_left = <0>;

    sprd,clk\_inv = <1>;

    sprd,pcm\_short\_frame = <0>;

    sprd,pcm\_slot = <0x1>;

    sprd,pcm\_cycle = <1>;

    sprd,tx\_watermark = <24>;

    sprd,rx\_watermark = <24>;

    sprd,i2s\_compatible = <0>;

};

默认的i2c和iis设置：

/\* default pcm config \*/

static const struct i2s\_config def\_pcm\_config = {

    .hw\_port = 0,

    .fs = 8000,

    .slave\_timeout = 0xF11,

    .bus\_type = PCM\_BUS,

    .byte\_per\_chan = I2S\_BPCH\_16,

    .mode = I2S\_MASTER,

    .lsb = I2S\_MSB,

    .rtx\_mode = I2S\_RTX\_MODE,

    .sync\_mode = I2S\_SYNC,/\* I2S\_SYNC better! \*/

    .lrck\_inv = I2S\_L\_LEFT,

    .clk\_inv = I2S\_CLK\_N,

    .pcm\_bus\_mode = I2S\_SHORT\_FRAME,

    .pcm\_slot = 0x1,

    .pcm\_cycle = 1,

    .tx\_watermark = 12,

    .rx\_watermark = 20,

};

/\* default i2s config \*/

static const struct i2s\_config def\_i2s\_config = {

    .hw\_port = 0,

    .fs = 32000,

    .slave\_timeout = 0xF11,

    .bus\_type = I2S\_BUS,

    .byte\_per\_chan = I2S\_BPCH\_16,

    .mode = I2S\_SLAVE,

    .lsb = I2S\_MSB,

    .rtx\_mode = I2S\_RX\_MODE,

    .sync\_mode = I2S\_LRCK,

    .lrck\_inv = I2S\_L\_LEFT,

    .clk\_inv = I2S\_CLK\_N,

    .i2s\_bus\_mode = I2S\_MSBJUSTFIED,

    .tx\_watermark = 12,

    .rx\_watermark = 20,

};

=====

es7210  iis设置:

设备树:

&i2s0 {

    status = "okay";

    sprd,config\_type = "i2s";

    sprd,slave\_timeout = <0xf11>;

    sprd,\_hw\_port = <0>;

    sprd,fs = <16000>;

    sprd,bus\_type = <0>;

    sprd,rtx\_mode = <3>;

    sprd,byte\_per\_chan = <1>;

    sprd,slave\_mode = <0>;

    sprd,lsb = <0>;

    sprd,lrck = <0>; /\*sync\_mode\*/

    sprd,low\_for\_left = <0>; /\*lrck\_inv\*/

    sprd,clk\_inv = <1>;//

    sprd,pcm\_short\_frame = <0>; /\*pcm\_bus\_mode\*/

    sprd,pcm\_slot = <1>;

    sprd,pcm\_cycle = <1>;

    sprd,tx\_watermark = <16>;

    sprd,rx\_watermark = <16>;

    sprd,i2s\_compatible= <1>;

};

hal 层:

static const int ext\_codec\_configs[I2S\_CONFIG\_MAX] = {

    16000, /\* fs \*/

    0, /\* hw\_port \*/

    0xf11, /\* slave\_timeout \*/

    0, /\* bus\_type \*/

    1, /\* byte\_per\_chan \*/

    0, /\* mode(master/slave mode) \*/

    0, /\* lsb \*/

    3, /\* rtx\_mode \*/

    0, /\* lrck\_inv(low\_for\_left) \*/

    0, /\* sync\_mode(lrck) \*/

    1, /\* clk\_inv \*/

    1, /\* i2s\_bus\_mode \*/

    0, /\* pcm\_bus\_mode \*/

    0, /\* pcm\_slot \*/

    0, /\* pcm\_cycle \*/

    16, /\* tx\_watermark \*/

    16, /\* rx\_watermark \*/

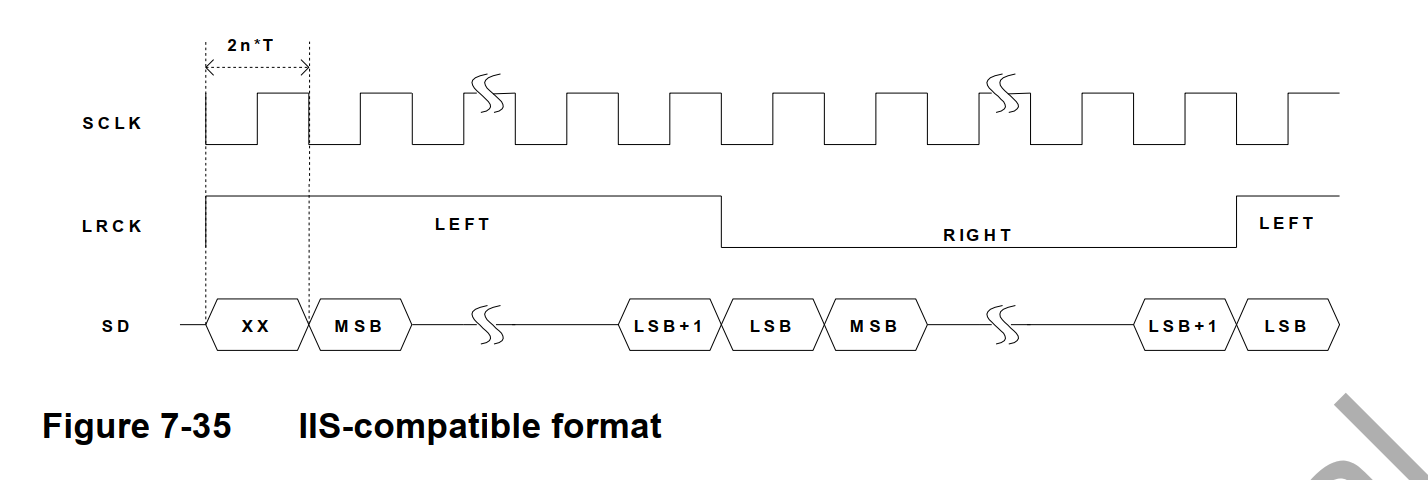
};

====

sync模式

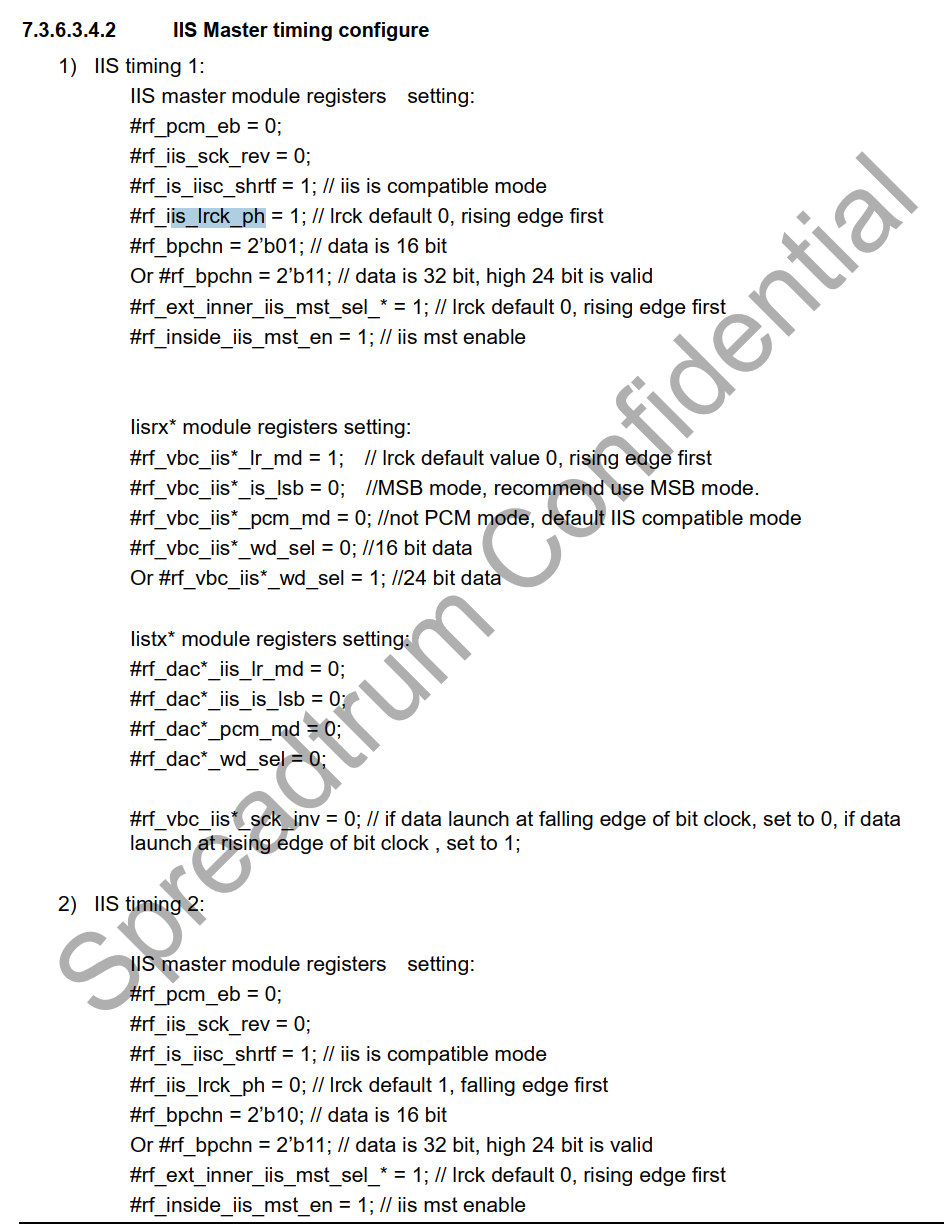


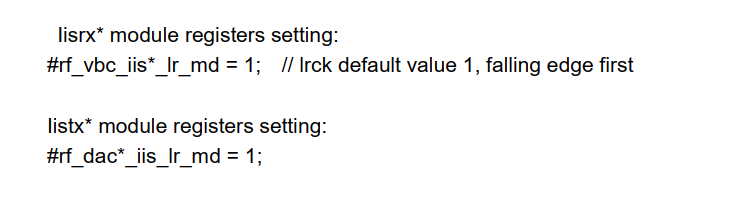




串行时钟 SCLK，也叫做位时钟BCLK，即对应数字音频的每一位数据，SCLK的频率=2×采样频率×采样位数 ,呵呵，现在问题来了，有人会问这些东西到底是什么意思呢？其实，I2S一般是传输立体声，有两个声道channel，采样频率指得是采样数率，多久去采集一个点，每个点是几个bit组成。  
帧时钟LRCK，用于切换左右声道的数据，LRCK为“0”表示正在传输的是左声道的数据，为“1”表示正在传输的是右声道的数据。LRCLK == FS,就是采样频率  
串行数据SDATA，就是用二进制补码表示的音频数据，有时为了使系统间能够更好的同步，还需要另外传输一个信号MCLK，称为主时钟，也叫系统时钟（System Clock），是采样频率的256或384倍

iis timing:

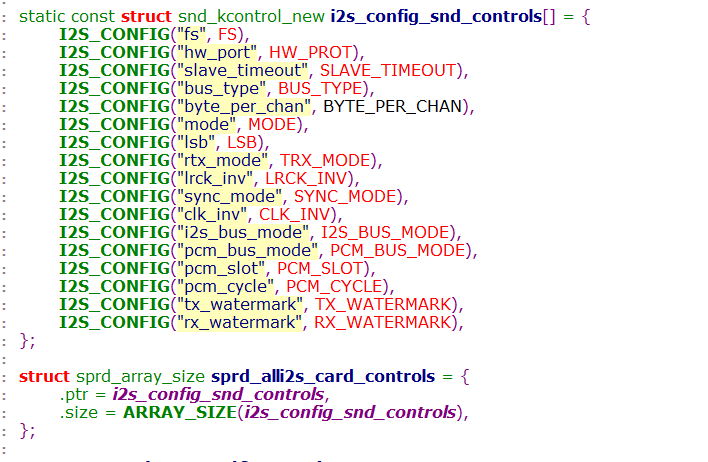




=======

I2s-r0p0-dummy-codec.c (sound\soc\sprd)    3992    2022/2/28

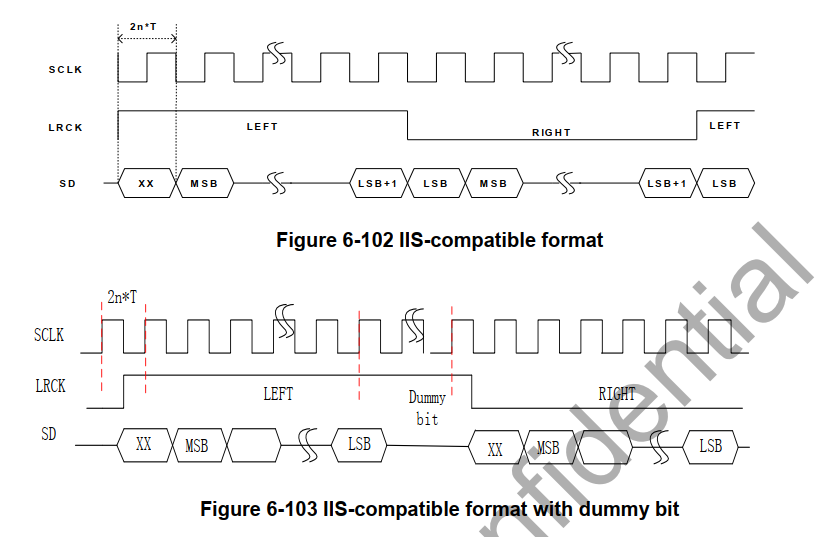
hal层设置i2s 模式的接口:



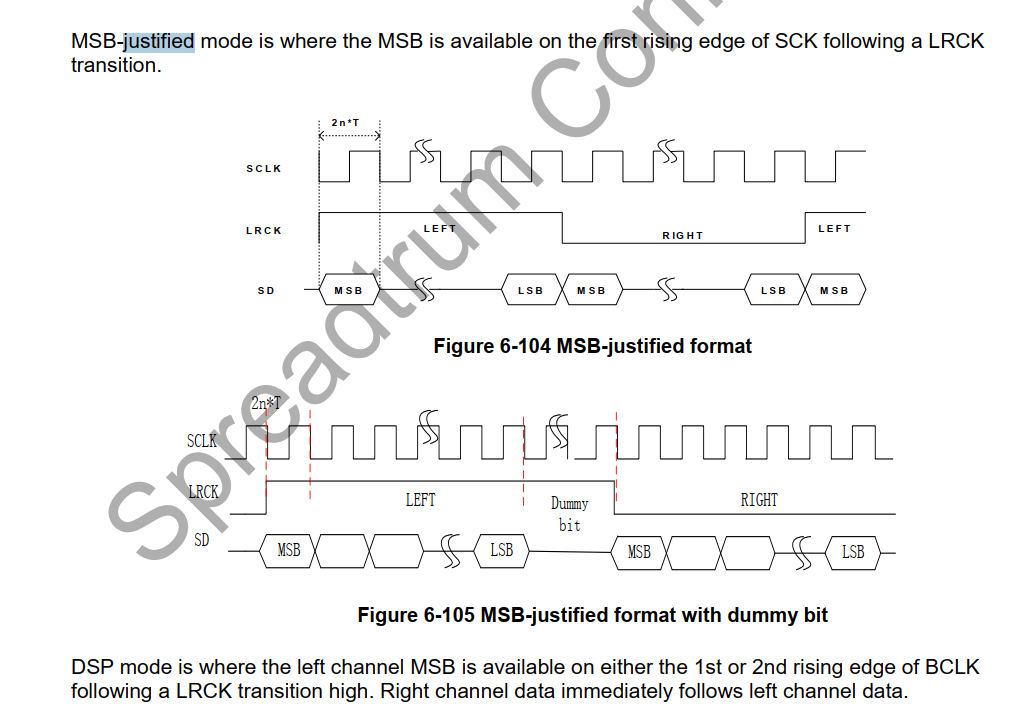
iis 的模式

iis mode 和 MSB-justified mode

IMG_268



MSB-justified mode



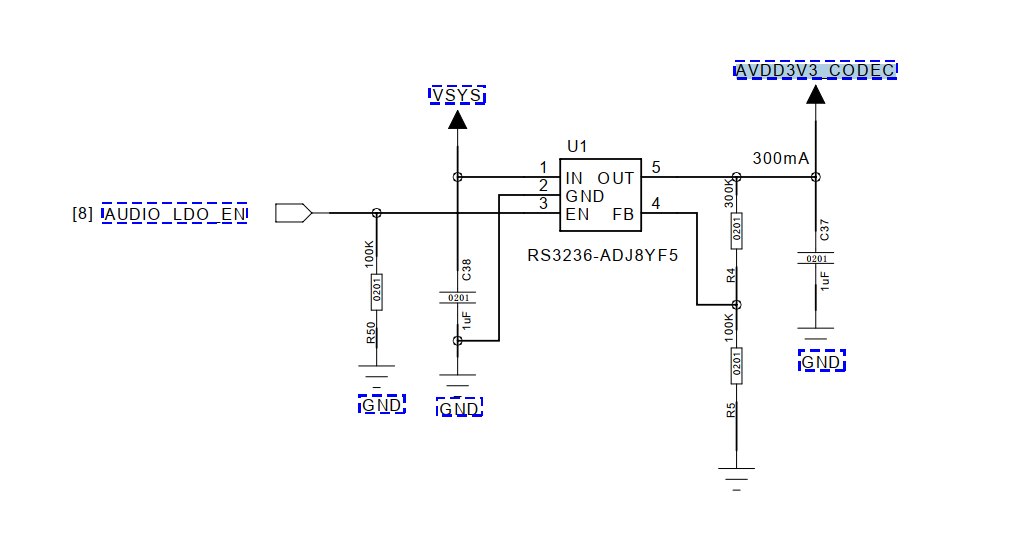
====



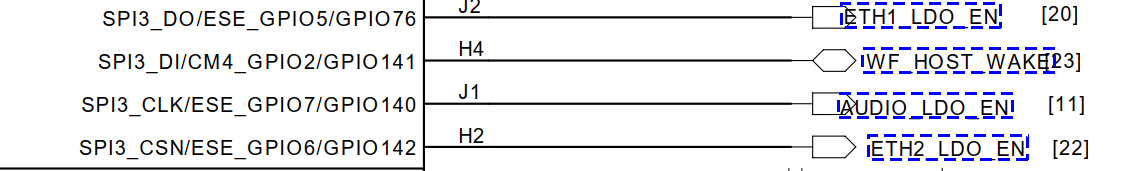
供电:

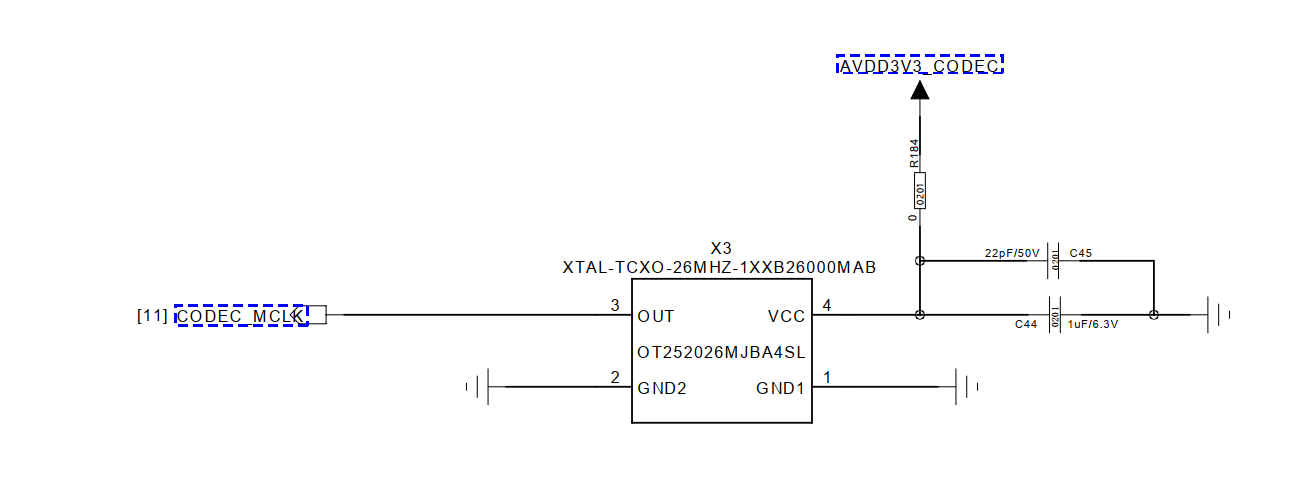
DVDD1V8\_CODEC

AVDD3V3\_CODEC



AUDIO\_LDO\_EN---->gpio140



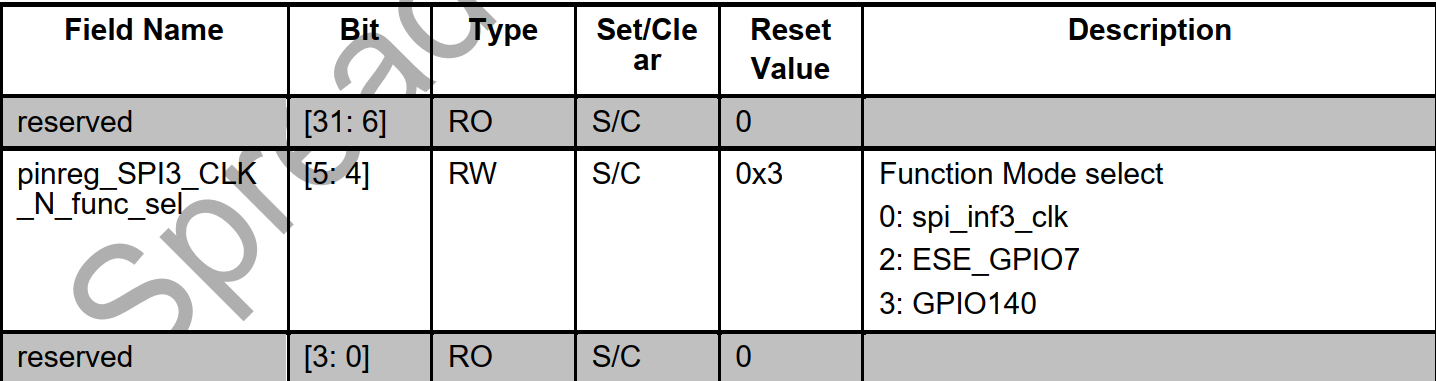


==

gpio140



IMG_276



===

[   15.082789] c6 i2s-null-codec sound@1:     name : i2s\_bt\_sco0-snd-soc-dummy-dai

[   15.089728] c6 i2s-null-codec sound@1:     stream\_name : i2s\_bt\_sco0-snd-soc-dummy-dai

[   15.097424] c6 i2s-null-codec sound@1:     name : i2s\_bt\_sco0-ES8311 HiFi

[   15.103843] c6 i2s-null-codec sound@1:     stream\_name : i2s\_bt\_sco0-ES8311 HiFi

[   15.110941] c6 [ASoC:BOARD] sprd\_asoc\_card\_parse\_hook\_spk hook aw87xx i2c pa

[   15.117990] c6 [Audio:AGDSP\_ACCESS] agdsp\_access\_enable out

[   15.123767] c6 Enter into es8311\_probe()

[   15.213085] c4 modem\_ctrl mdm\_ctrl: crash send to cp.

[   15.218058] c4 pcie res: wait resource, val=-62.

[   15.222625] c4 mpm: sipc-nr-mpm-1 wait resource, ret=-62, timeout=-1.

[   15.229035] c4 smsg\_ch\_open: channel 1-120 send open msg error = -62!

[   15.235444] c4 sipa\_dele: conn\_thread sipa\_delegator failed to open dst 1 channel 120

[   15.240897] c6 Enter into es8311\_set\_bias\_level(), level = 1

[   15.253718] c4 i2s-null-codec sound@1: snd-soc-dummy-dai <-> i2s\_bt\_sco0 mapping ok

[   15.261266] c4 Enter into es8311\_set\_dai\_fmt()

[   15.267063] c4 ES8311 in Slave mode

[   15.270673] c4 es8311 4-0018: ASoC: Failed to set DAI format: -22

[   15.277240] c4 i2s-null-codec sound@1: ES8311 HiFi <-> i2s\_bt\_sco0 mapping ok

[   15.285773] c0 Enter into es8311\_set\_bias\_level(), level = 1

[   15.299568] c4 ALSA device list:

[   15.302717] c4   #0: sprdphone-sc2730

[   15.306334] c4   #1: all-i2s

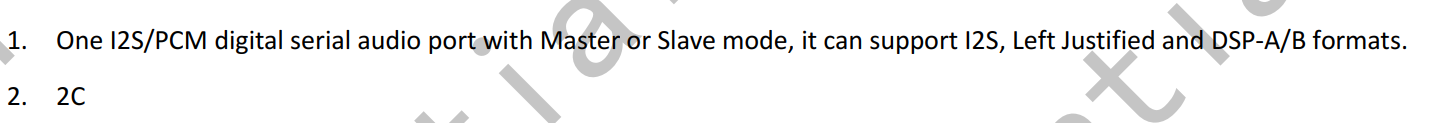
=====

0x00069040

0110    1001    0000    0100    0000--->0x01-->iis1-->ap iis1

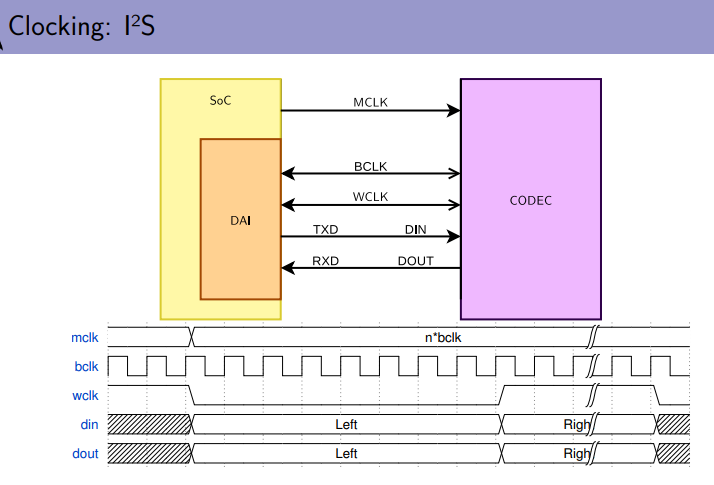
0110    1001    0000    0000    0000--->0x00-->iis1-->ap iis0

====



支持i2s left justified   dsb-a/b 模式

时钟相关



MCLK ---> 给codec的时钟

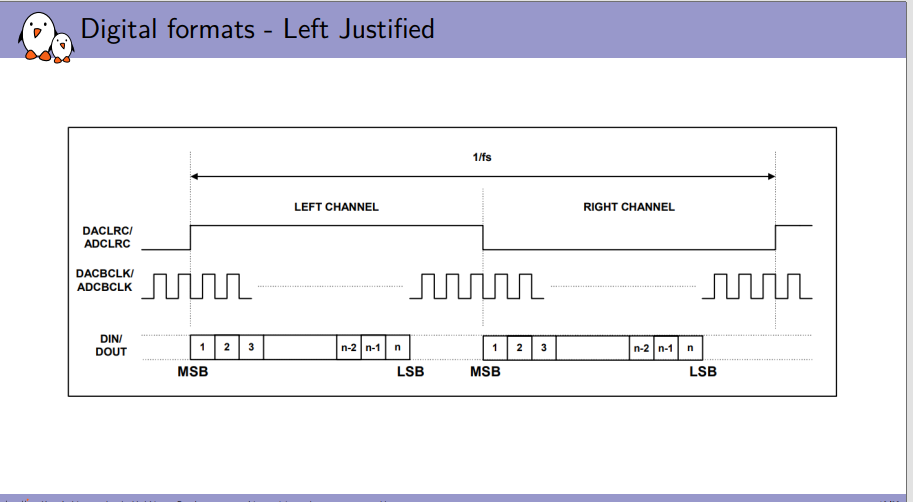
BCLK ---> 数据bit时钟

WCLK--> word 字时钟 LRCLK左右声道时钟 FCLK/FSCLK帧时钟

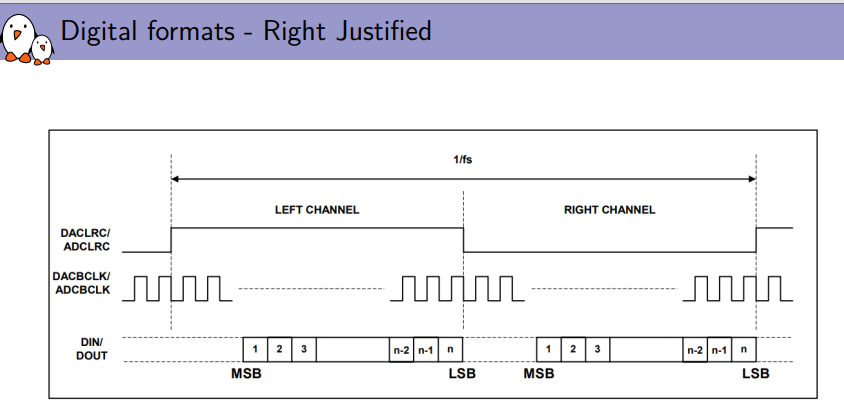
Bclk = Wclk \* Nchannels \* BitDepth

codec的BLCK需要通过MCLK的分频

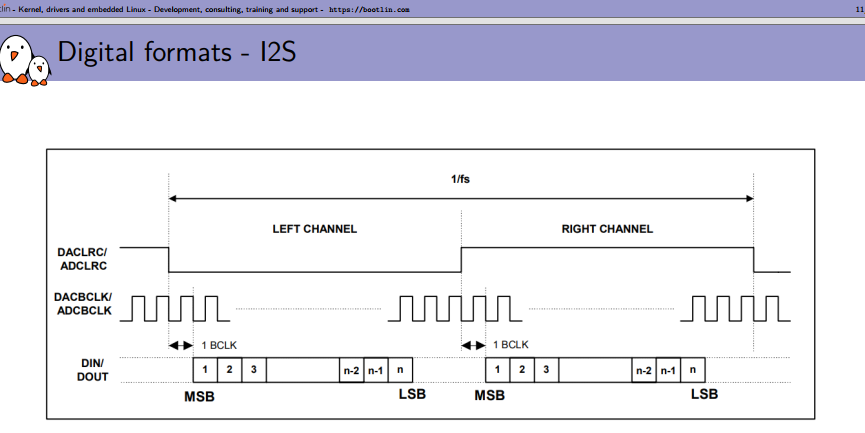
iis的左对齐模式:左声道为高



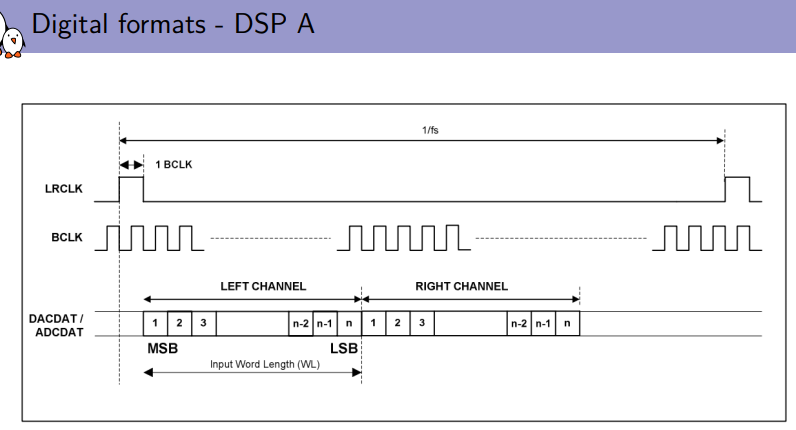
iis 右对齐模式



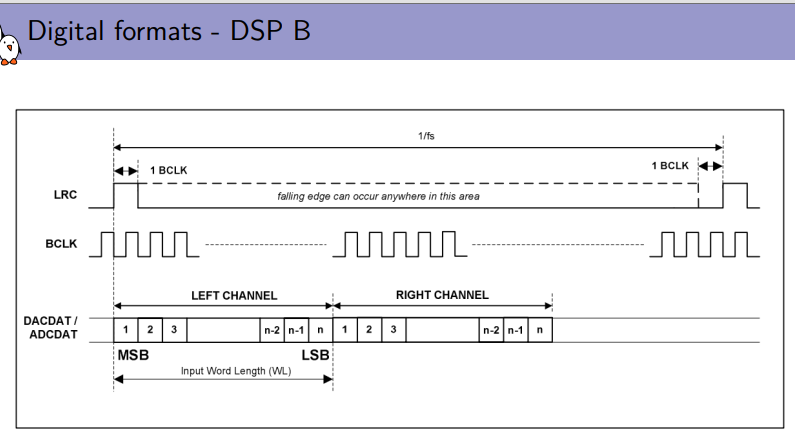
iis的标准模式: 左声道为低



iis  dsp-a 模式

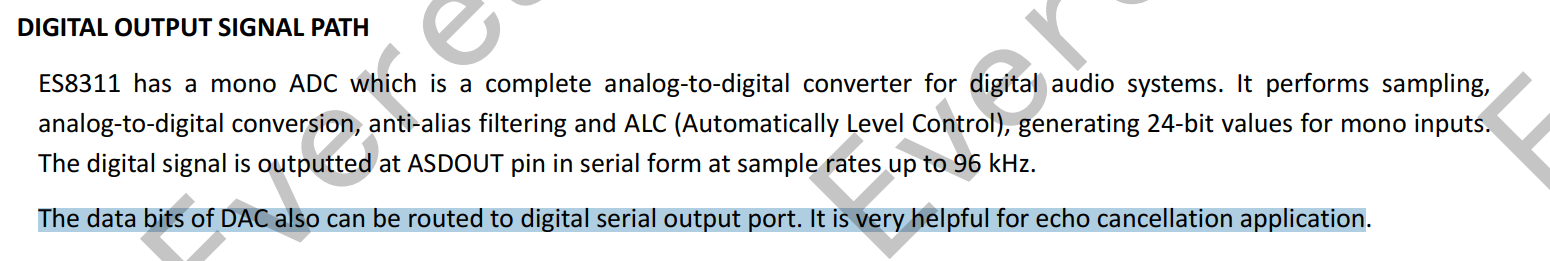


iis dsp-b 模式



===

具有回声消除功能：



==

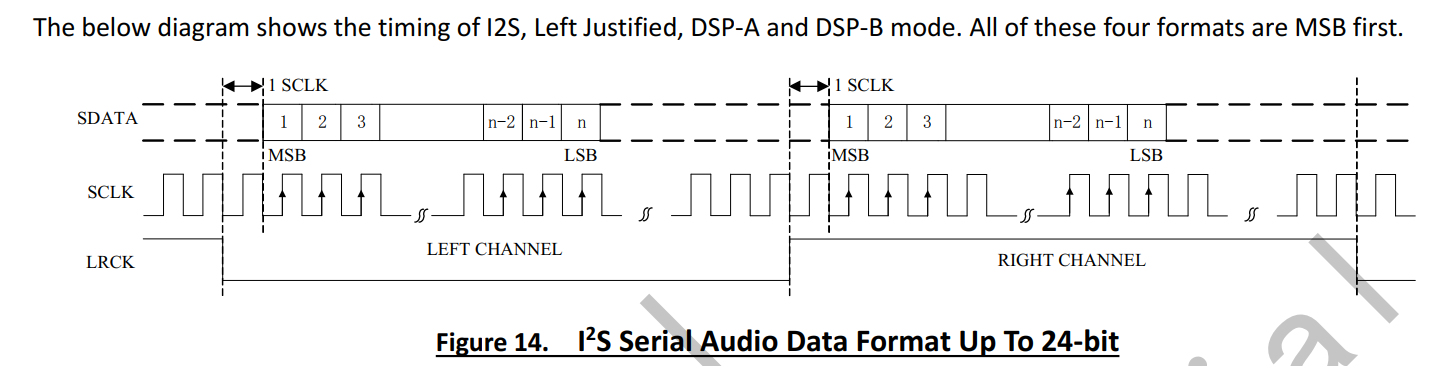
es8311只支持 4种数据格式:

1:标准的I2S格式数据。

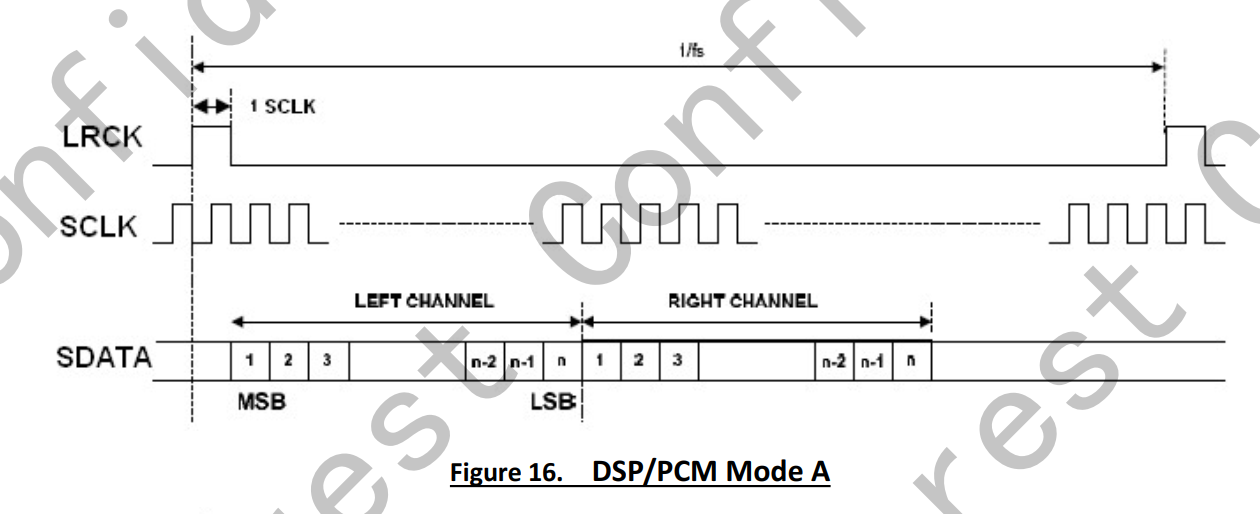
2：左对齐 LEFT Justified

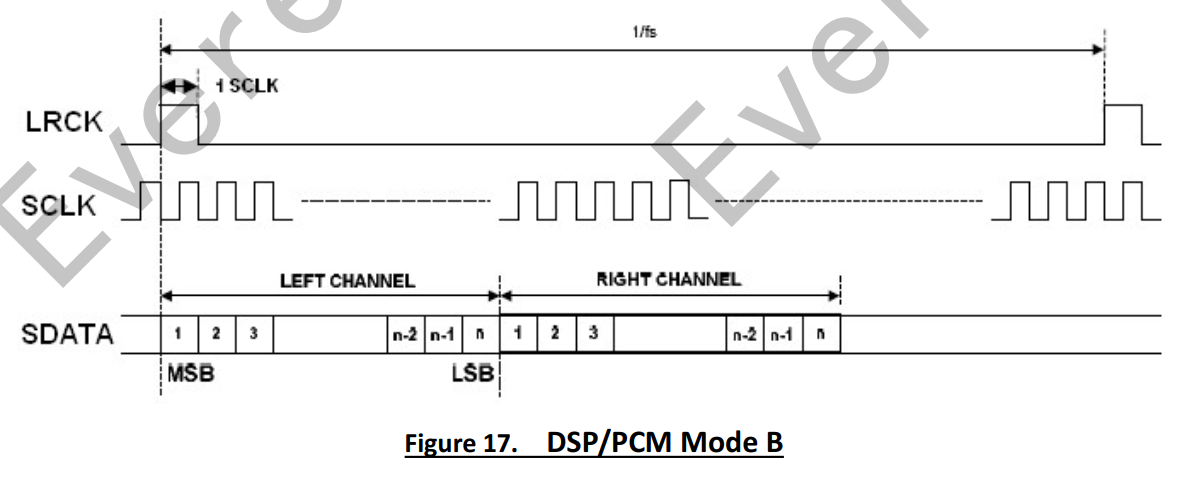
3:DSP mode A

4:DSP mode B.









1：寄存器0x09和寄存器0x0a 选择adc和dac的格式

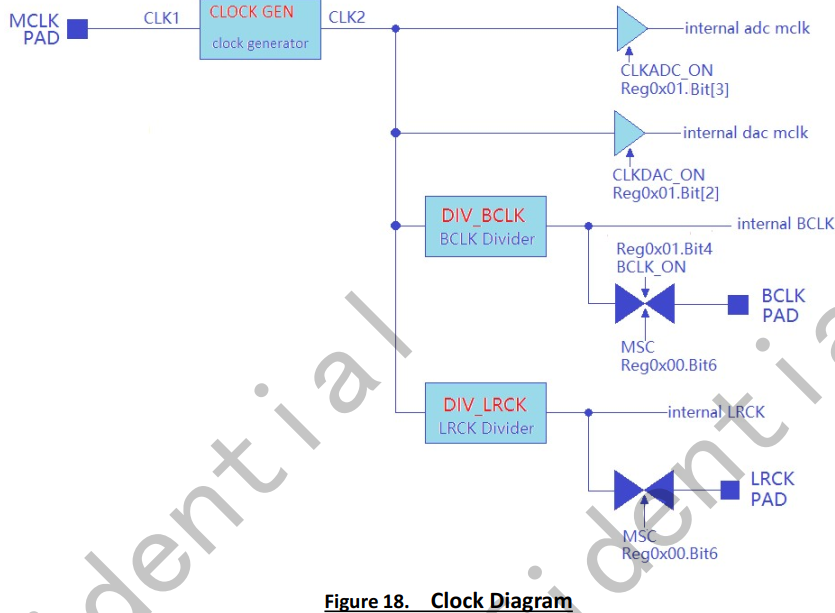
a: DAC的数据通道选择 LEFT和right通道。--->这个左右通道应该是i2s的数据通道

b:    设置极性是否反正

c:     设置数据宽度:    24bit    20bit    18bit    16bit    32bit

d:设置数据格式：i2s left dsp/pcm

时钟架构:



==

寄存器0x00-->MSC  选择Master或者Slave

1:选择master或者slave

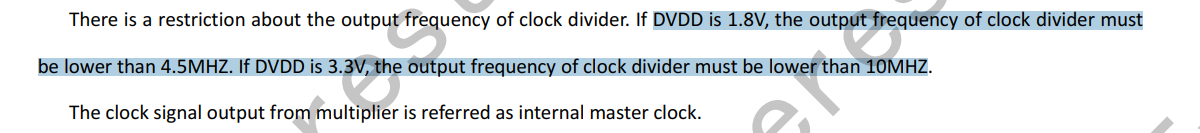
时钟选择 可选择mclk pin  或者sclk

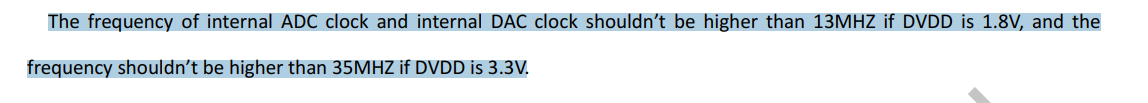
a:通过寄存器的Reg 0x00 Bit7

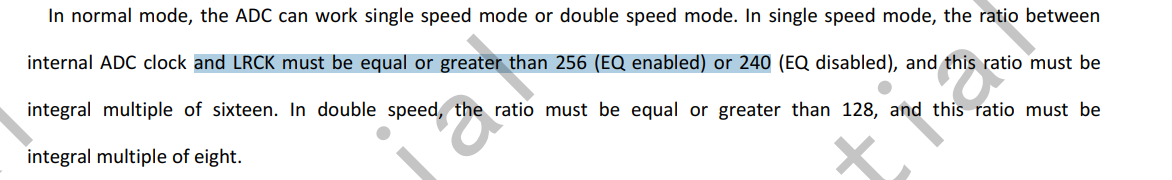
b:寄存器Reg0x01 Bit[5] 设置为1 MCLK ON 和sclk没有关系

==

时钟个电压的关系

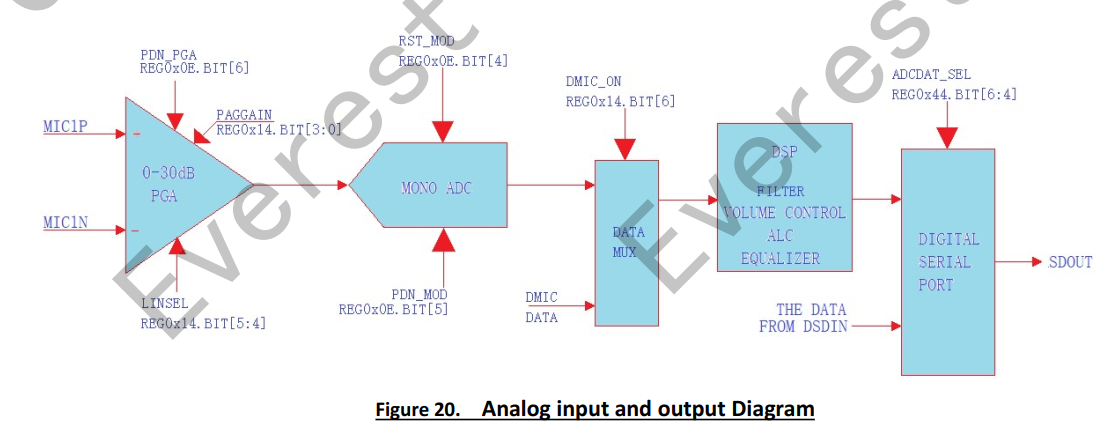


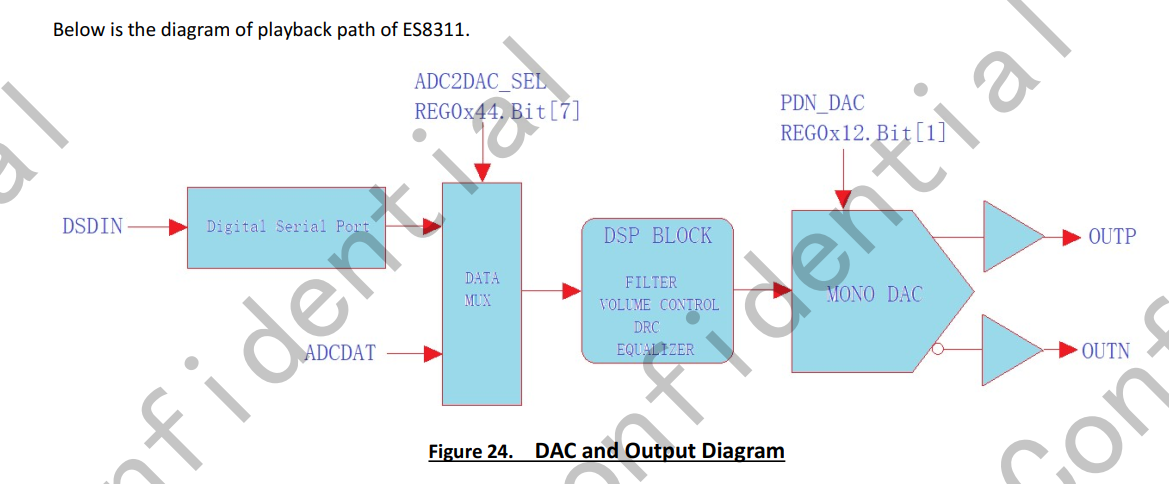




mclk时钟只能做输入

\*\*



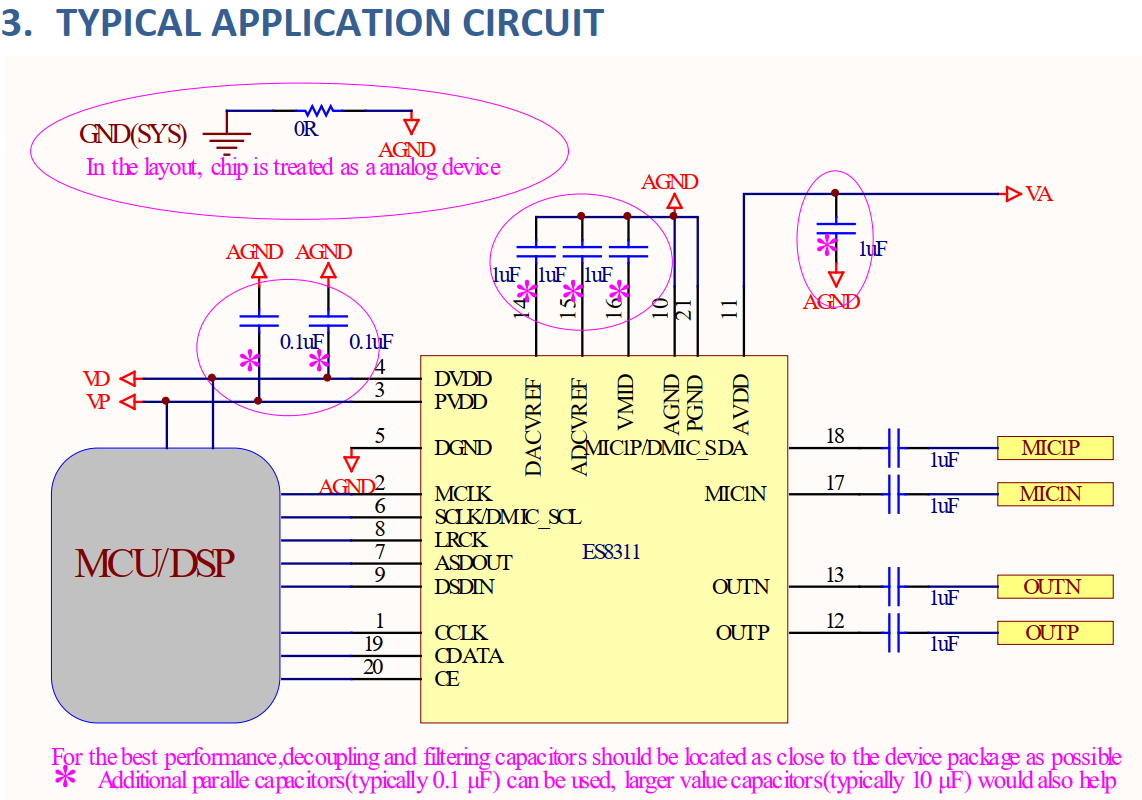


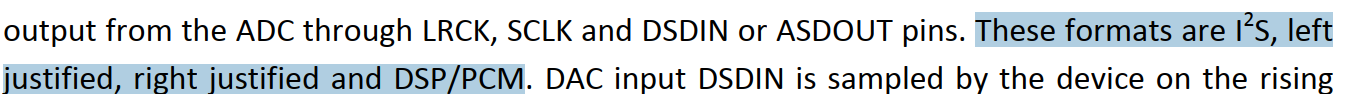
====

引脚设置:



电路





在设备的上升沿接收数据

T10

====

t710端的iis 和pcm的时序

ctl0 ----->  1000 1101 0001

0：bit[15] = 0 i2s模式

0:bit[14] = 0  没使用dma

00:bit[13:12] = 00  没使用

1：bit[11] = 1 sck bit时钟翻转

0:bit10=0 low for left 左声道为低

0：bit9 = 0 输出LRCK

0：bit8  =0 msb justified模式

11:bit[7:6] =11 收发功能

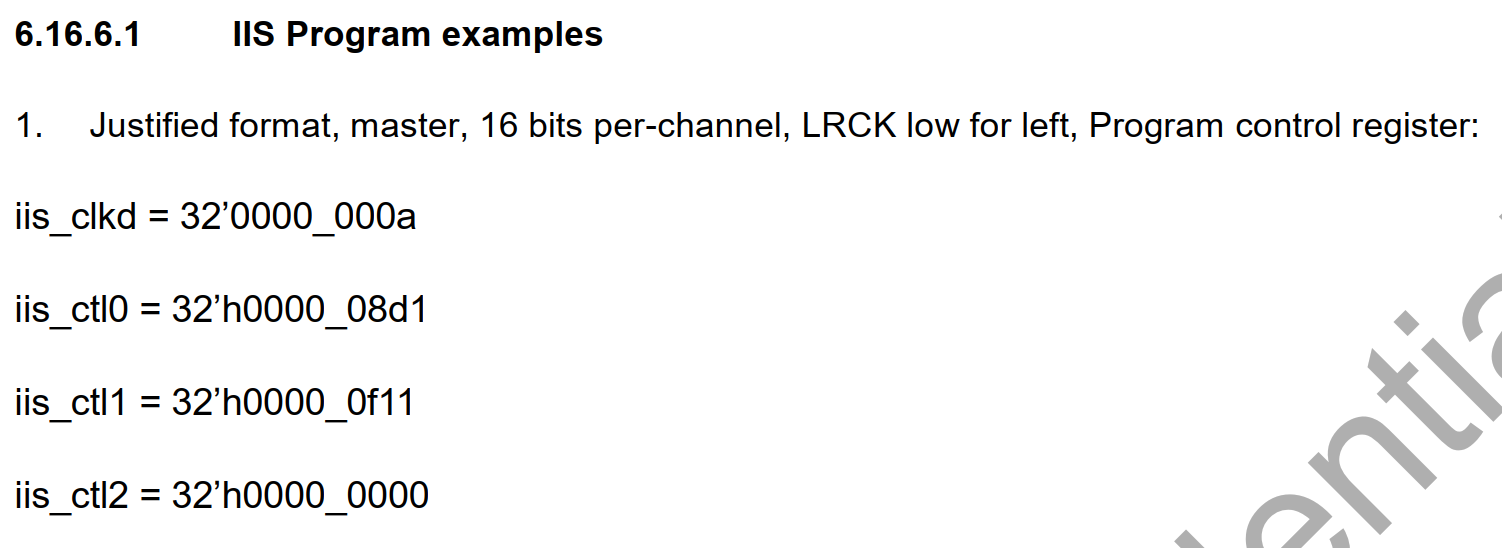
01:bit[5:4] =01 16bit每个channel

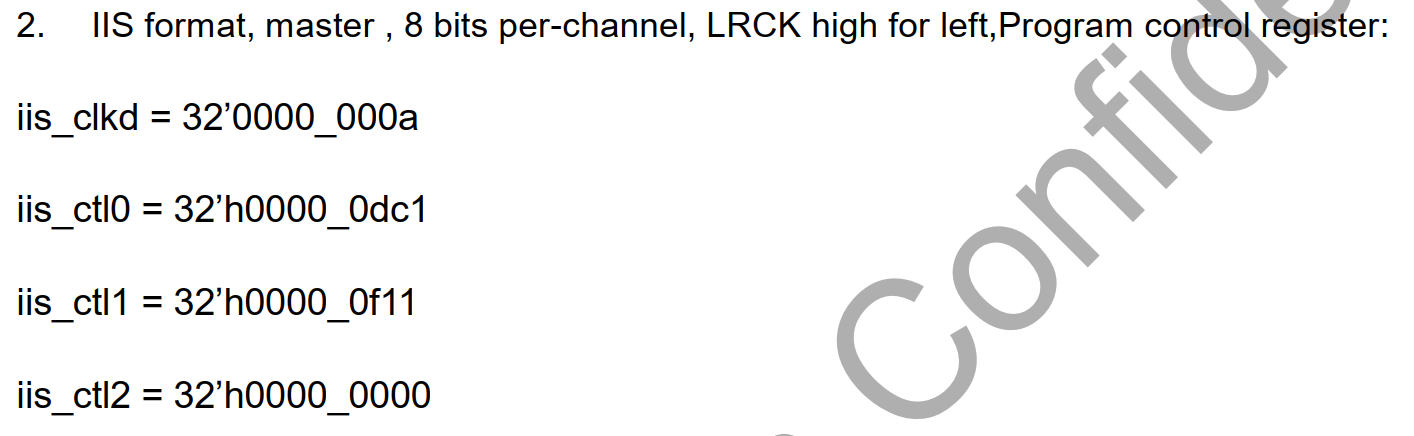
0:bit3:iis master

0: bit2:MSB

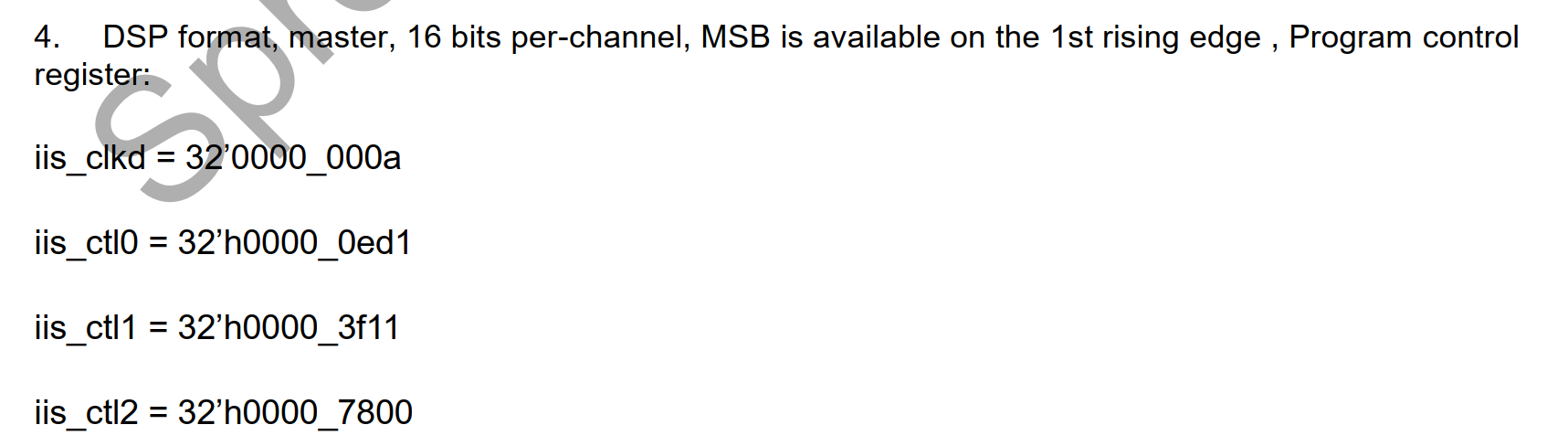
0：bit1

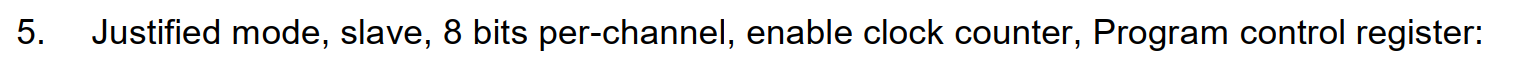
1:bit0 NG\_RX  设置接收数据的移位极性

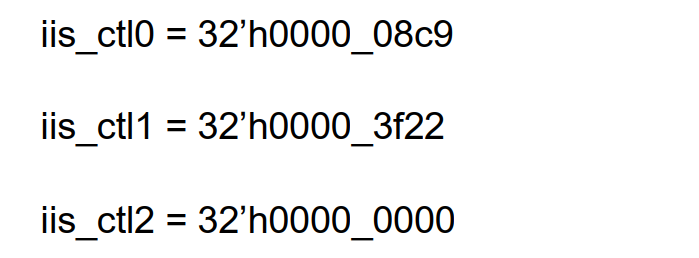




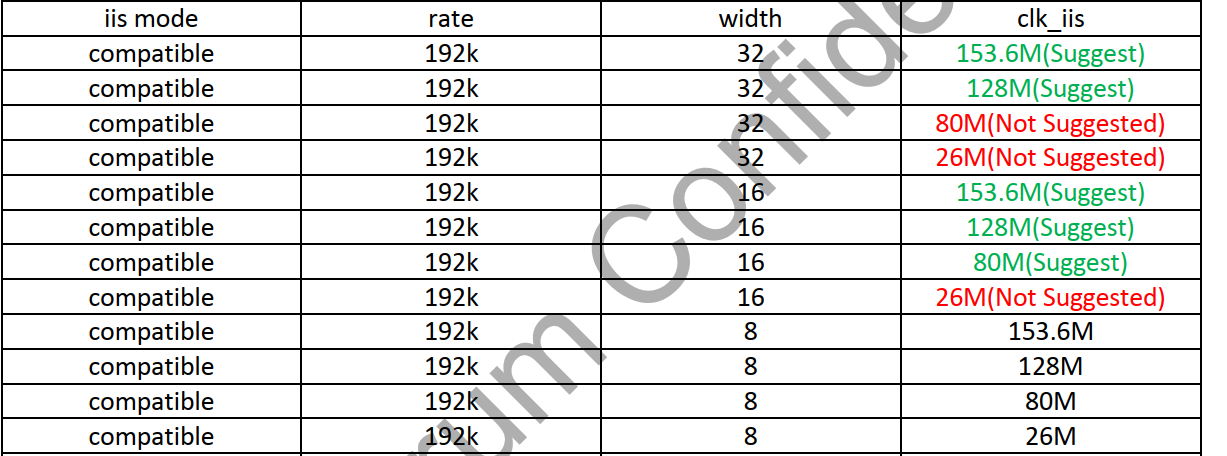


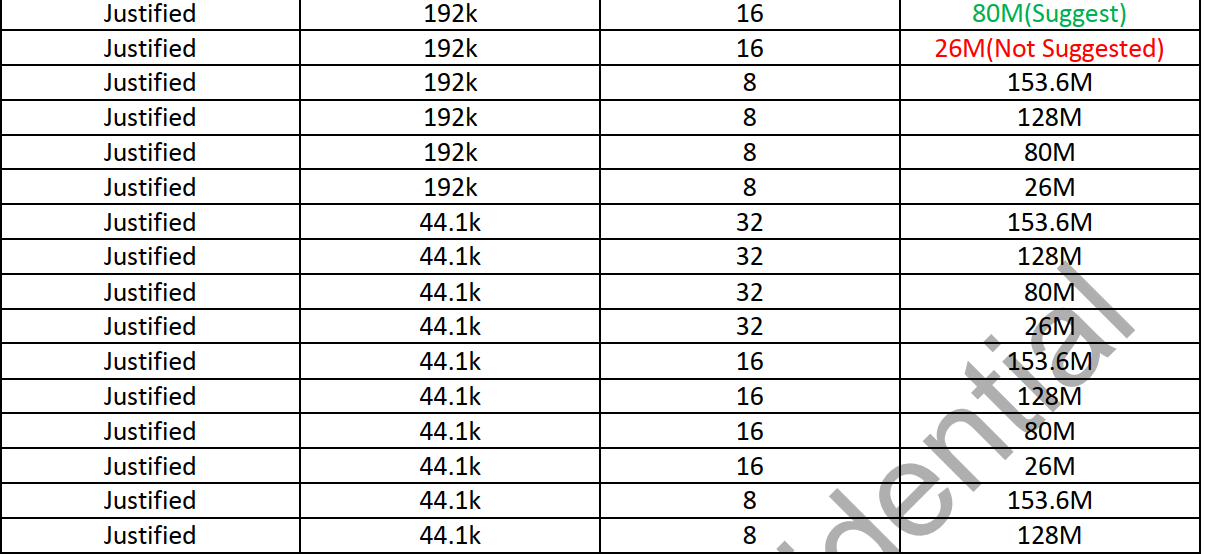


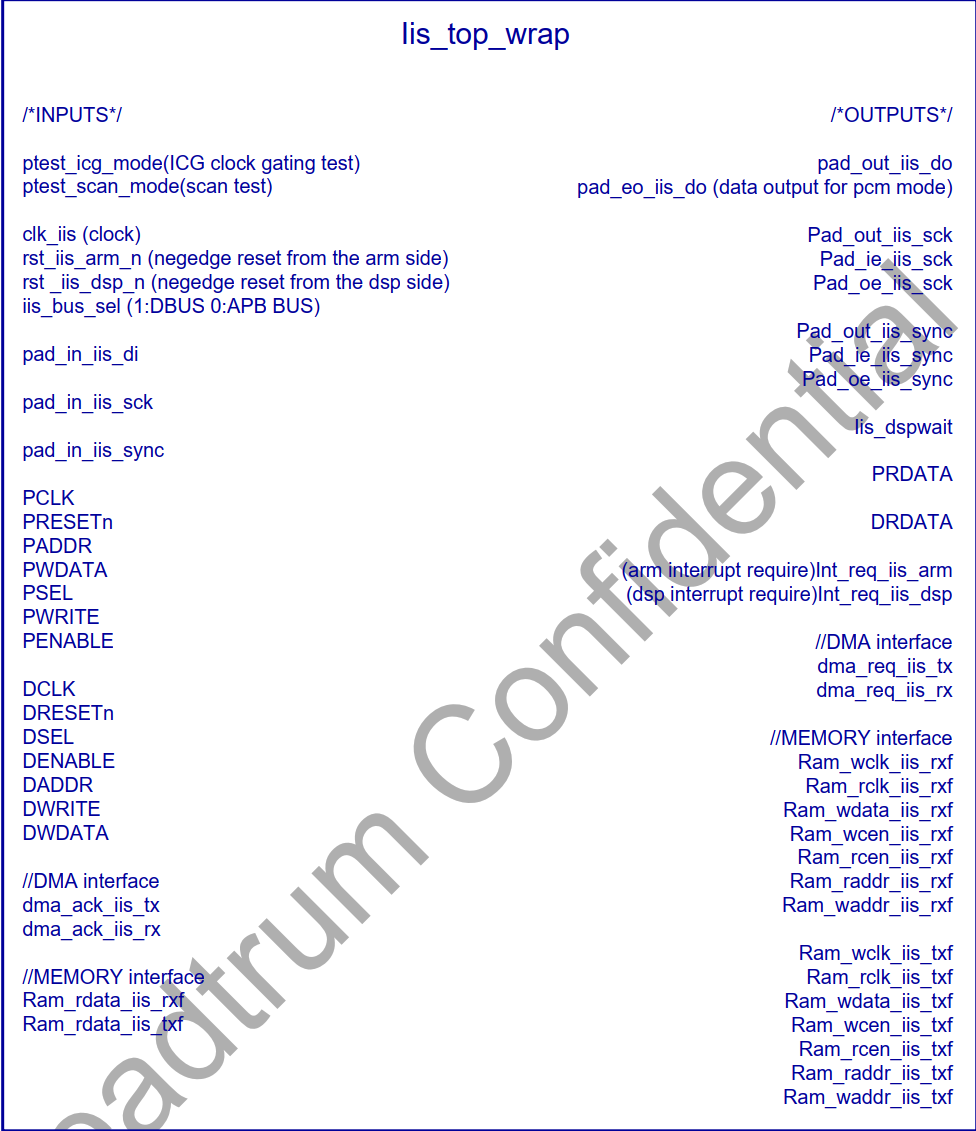




时钟设置：





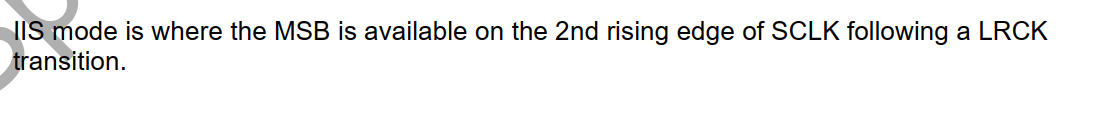


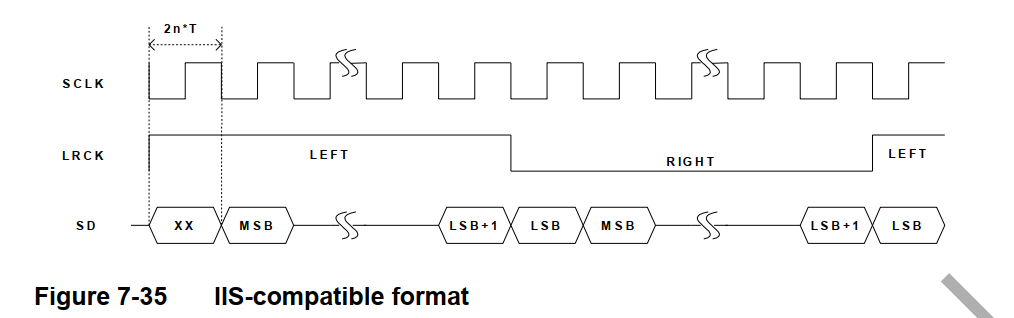
i2s模式选择：

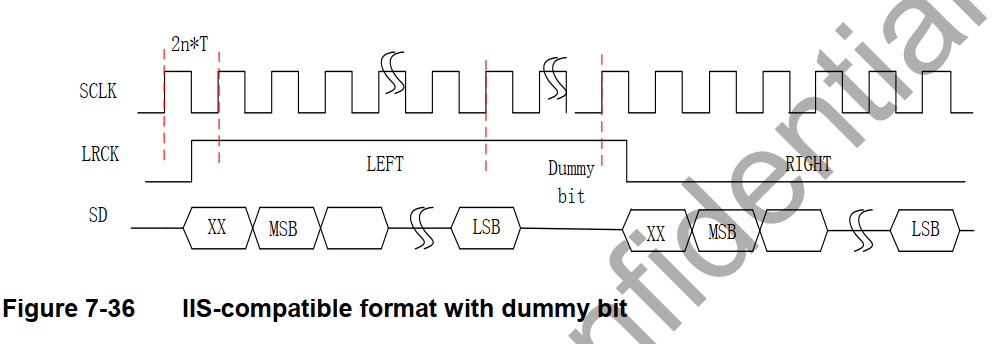
IIS 三种模式：

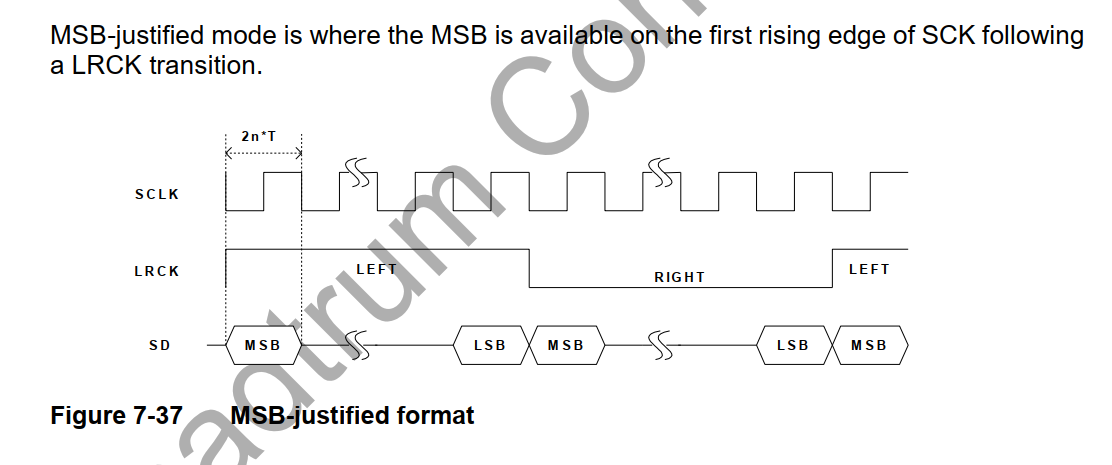
1：iis  2:MSB-Justified 3:DSP

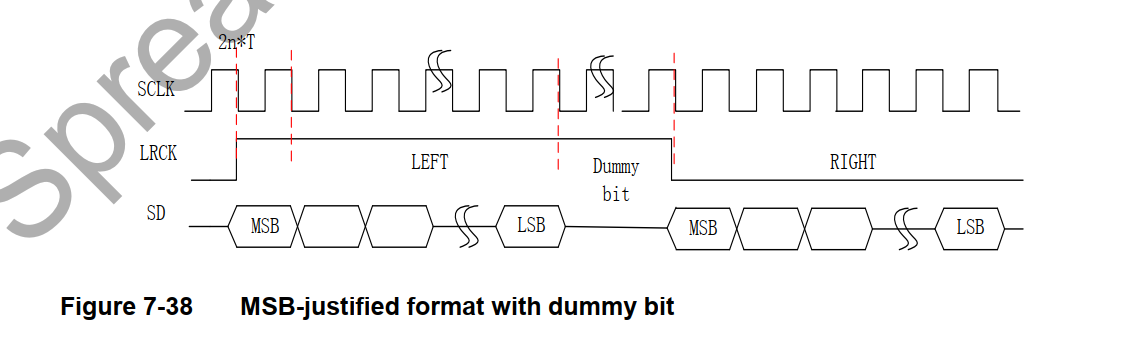
2:字宽： 8 16 32bit

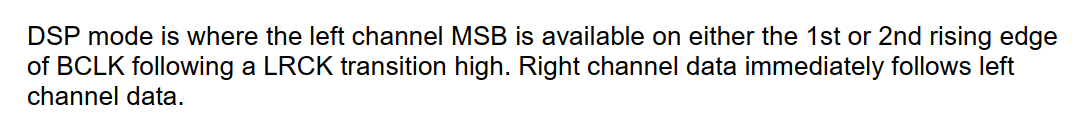


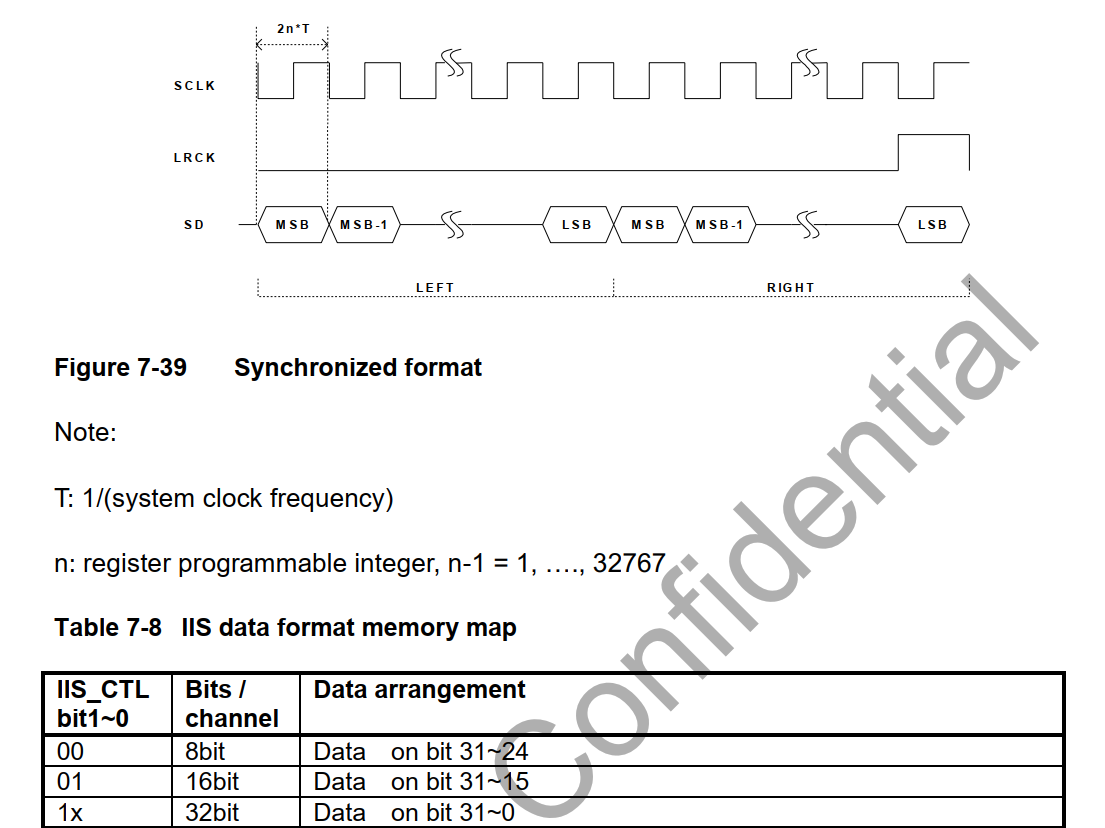






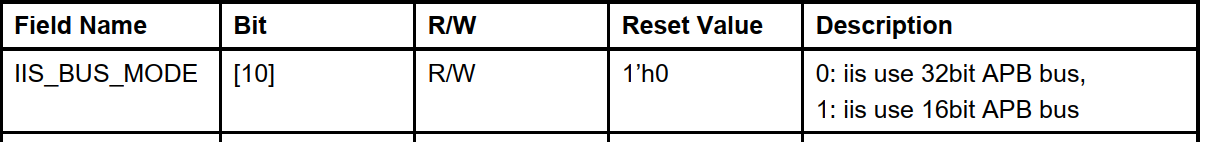






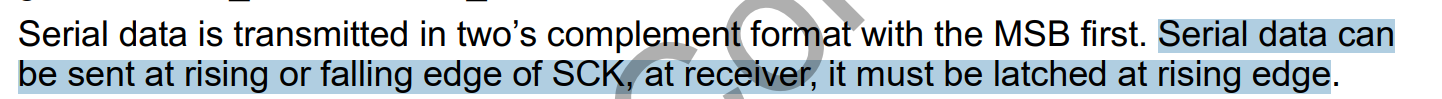
==

iis 总线模式



发送数据可以在上升沿或者下降沿

接收方必须在上升沿采样



## 如果系统的字长比传输的iis 传输字长 要多，则数据进行截断处理

###如果发给接收方的数据比字长要多  lsb部分要忽略掉

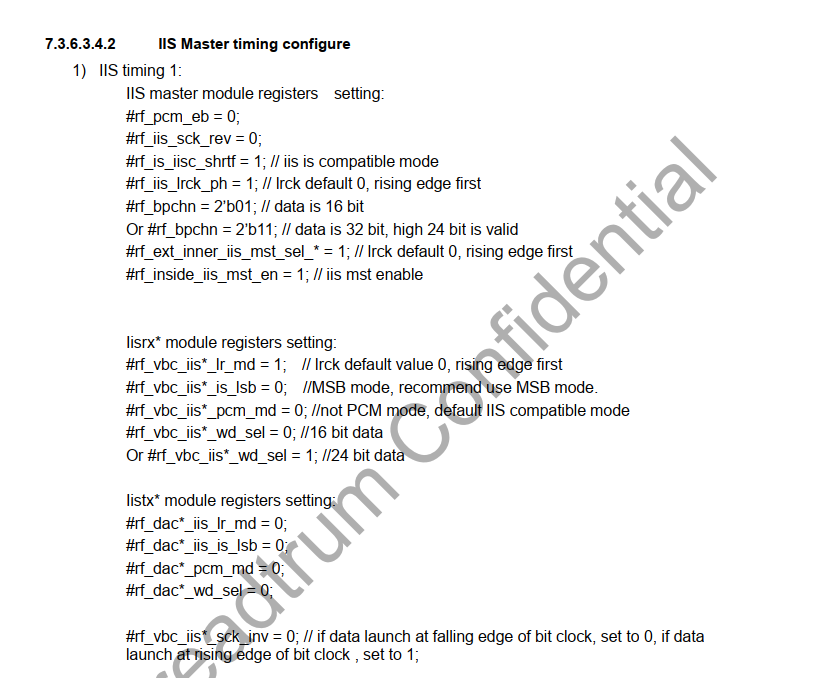
###如果接收方的数据比字长少  少的部分用0补充

###LRCK的极性可以编程

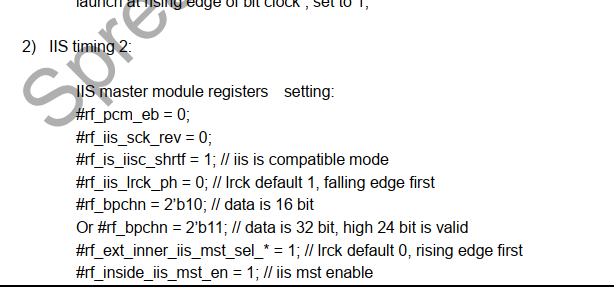
###

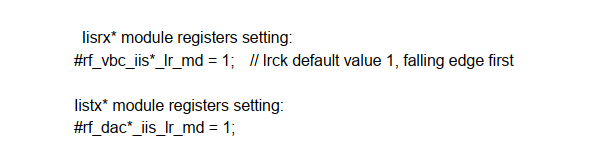
vbc  iis时序设置:

iis 标准格式



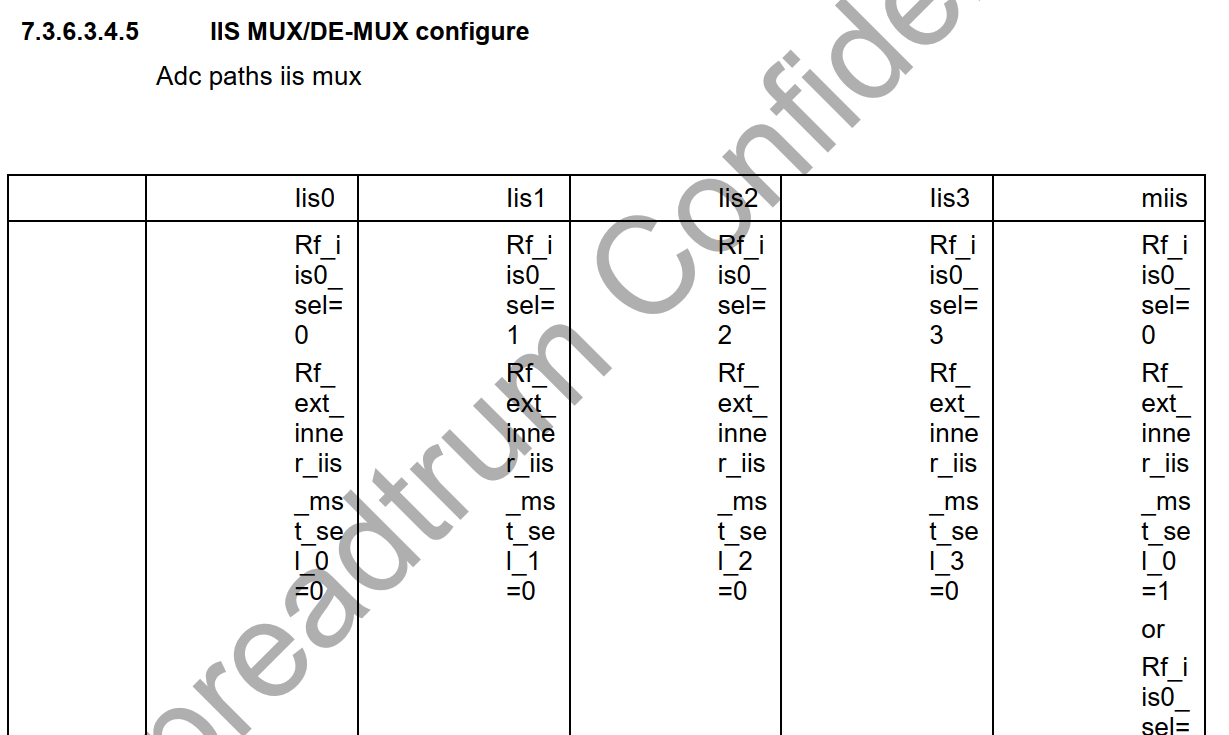
iis  格式：



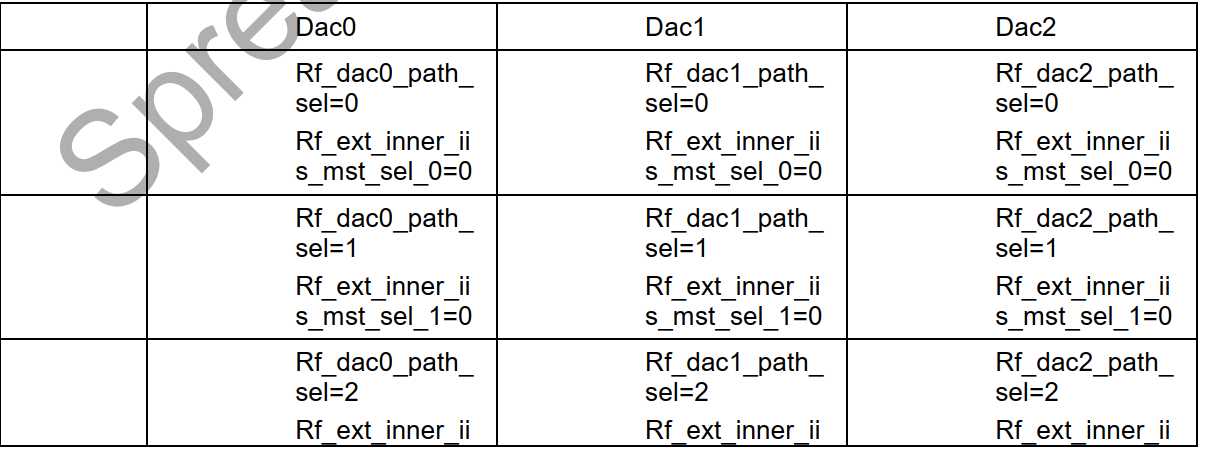


===

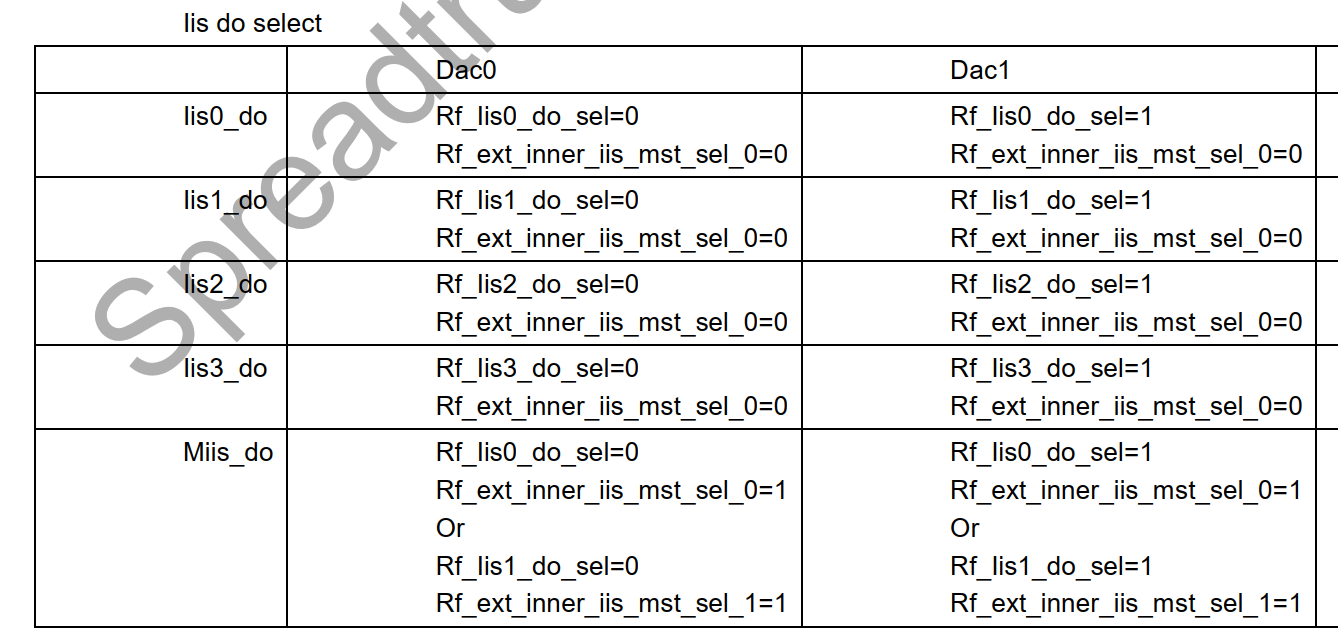
vbc iis 的mux和demux的多路复用和解复用的使用:



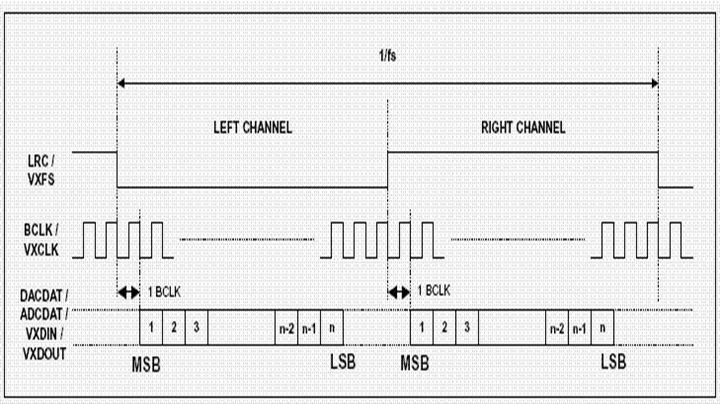
dac 路径 iis clk和 lrck 选择



iis do 选择



===

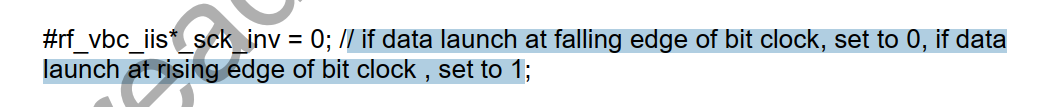


标准的i2s格式 低是左声道    高是右声道

===

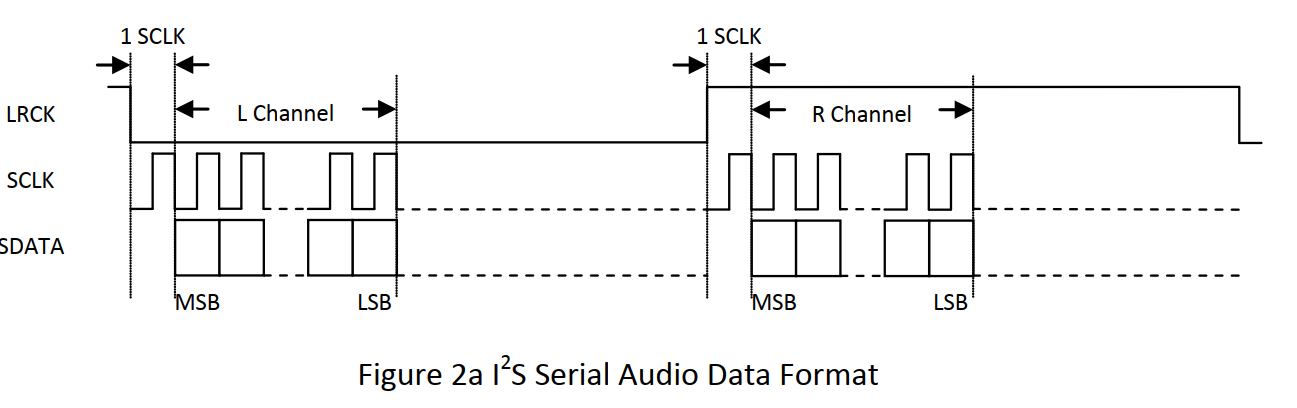
sck  和lrck的关系

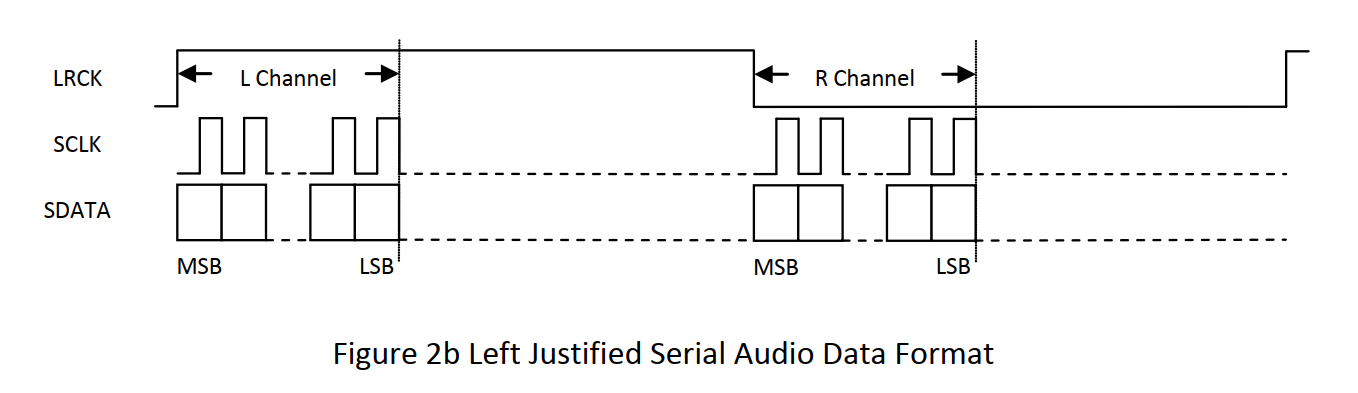
sck\_inv是否要设置为0 Or 1?

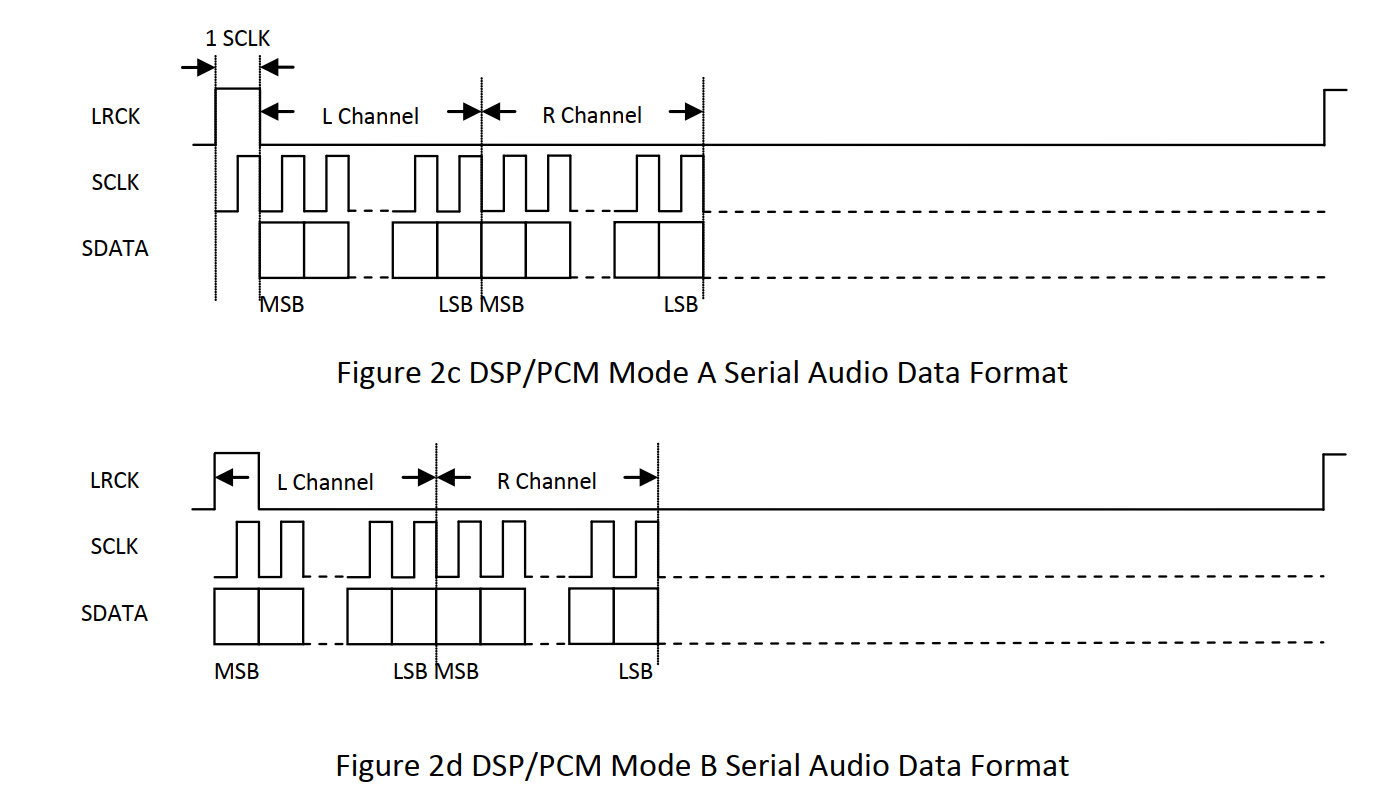


es8311的数据 在接收端于上升沿采样   发送数据在下降沿









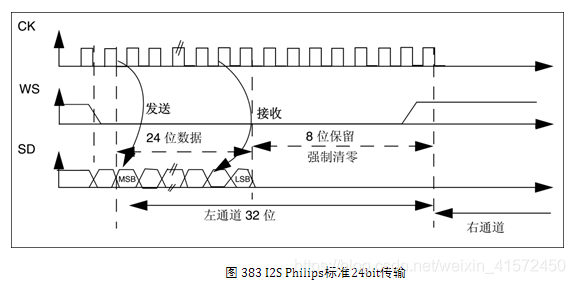
<https://blog.csdn.net/weixin_41572450/article/details/103582662、>

音频数据传输协议标准

使用WS信号来指示当前正在发送的数据所属的通道（即一帧的开始），数据传输从MSB到LSB，发送方在时钟信号（CK）的下降沿改变数据，接收方在上升沿读取数据，WS也在CK的下降沿变化，有3种标准

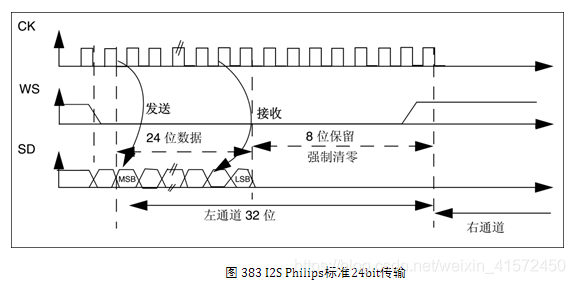
I2S Philips标准

串行数据（SD）在WS变化后的第2个时钟信号（CK）边沿开始发送MSB，下面是为24bit数据封装在32bit帧传输波形。



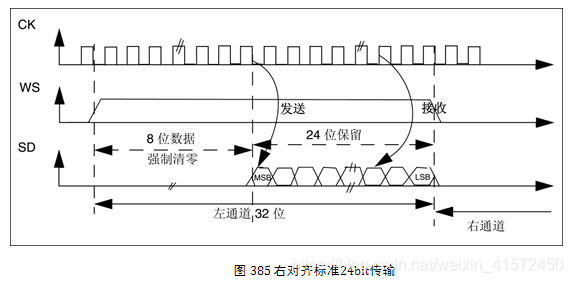
左对齐标准

在WS发生翻转同时开始传输数据，参考图 384，为24bit数据封装在32bit帧传输波形。该标准较少使用。注意此时WS为1时，传输的是左声道数据，这刚好与I2S Philips标准相反。

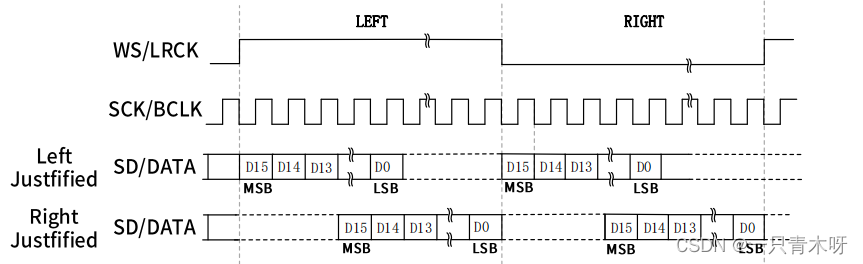


右对齐标准

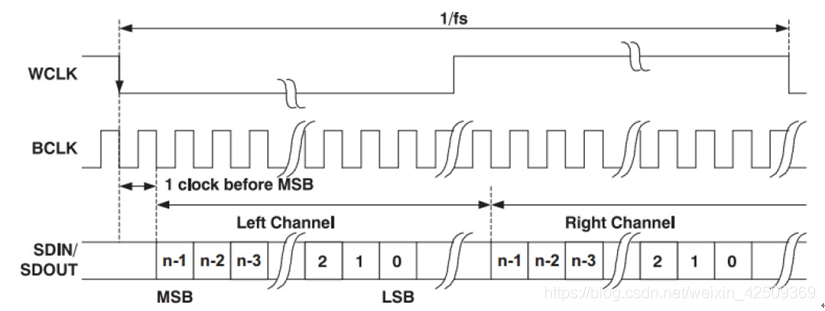
与左对齐标准类似，参考图 385，为24bit数据封装在32bit帧传输波形。



左对齐：数据的MSB在LRCLK边沿起第一个BCLK上升沿，用的比较少。  
右对齐：数据的LSB靠左LRCLK的上升沿，Sony使用这种格式。



IS的操作模式分为三种：标准IIS模式、左对齐模式和右对齐模式。



标准IIS模式   Phillips Standard  
IIS模式是标准左对齐格式再延迟一个时钟位变化来的，时序如下所示：

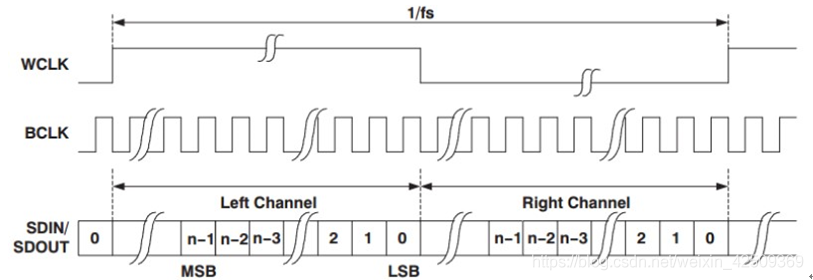
左右通道的数据MSB均是在WS变化后第二个SCK/BCLK上升沿有效。

左对齐模式  Left Justified Standard  
标准左对齐格式的数据的MSB没有相对于BCLK延迟一个时钟。左对齐格式的左右声道数据的MSB在WS边沿变化后SCK/BCLK的第一个上升沿有效。具体如下图所示：



支持16~32bit字长格式；

右边对齐模式  Right Justified Standard     
也叫日本格式，sony格式，具体对齐方式如下图所示：



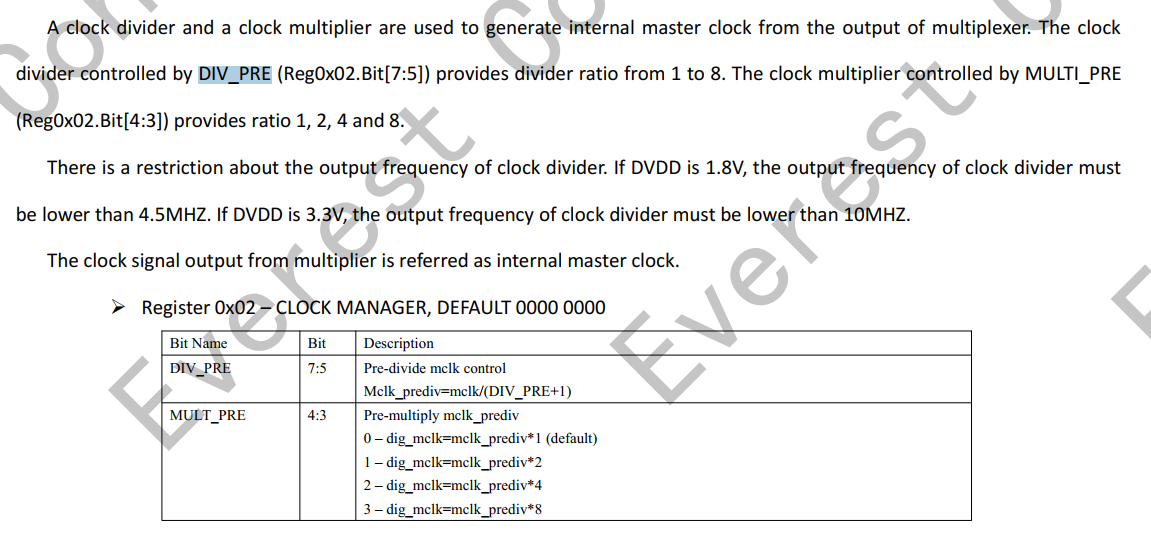
接收设备必须事先知道待传数据的字长。

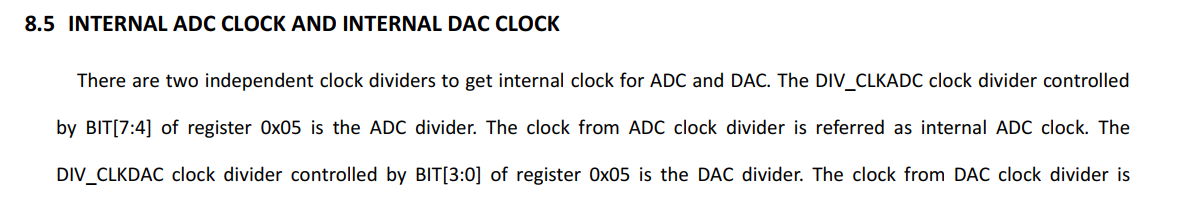
注意左右对齐模式的WS时钟高电平为左声道，低电平为右声道，刚好与标准IIS相反。

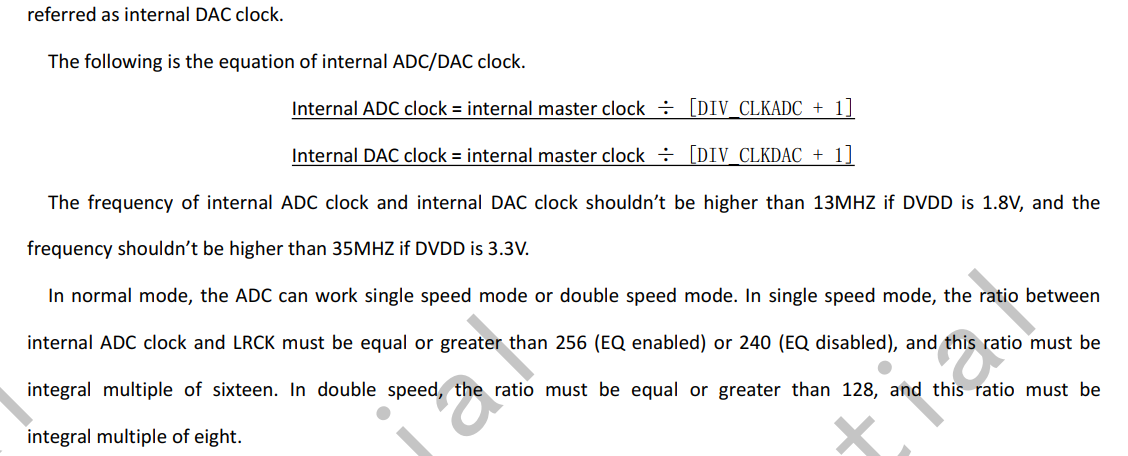
SCK = 采样率（48K、44.1K、16K等） x  字长（16bit、24bit、32bit） x 2（左右两通道）

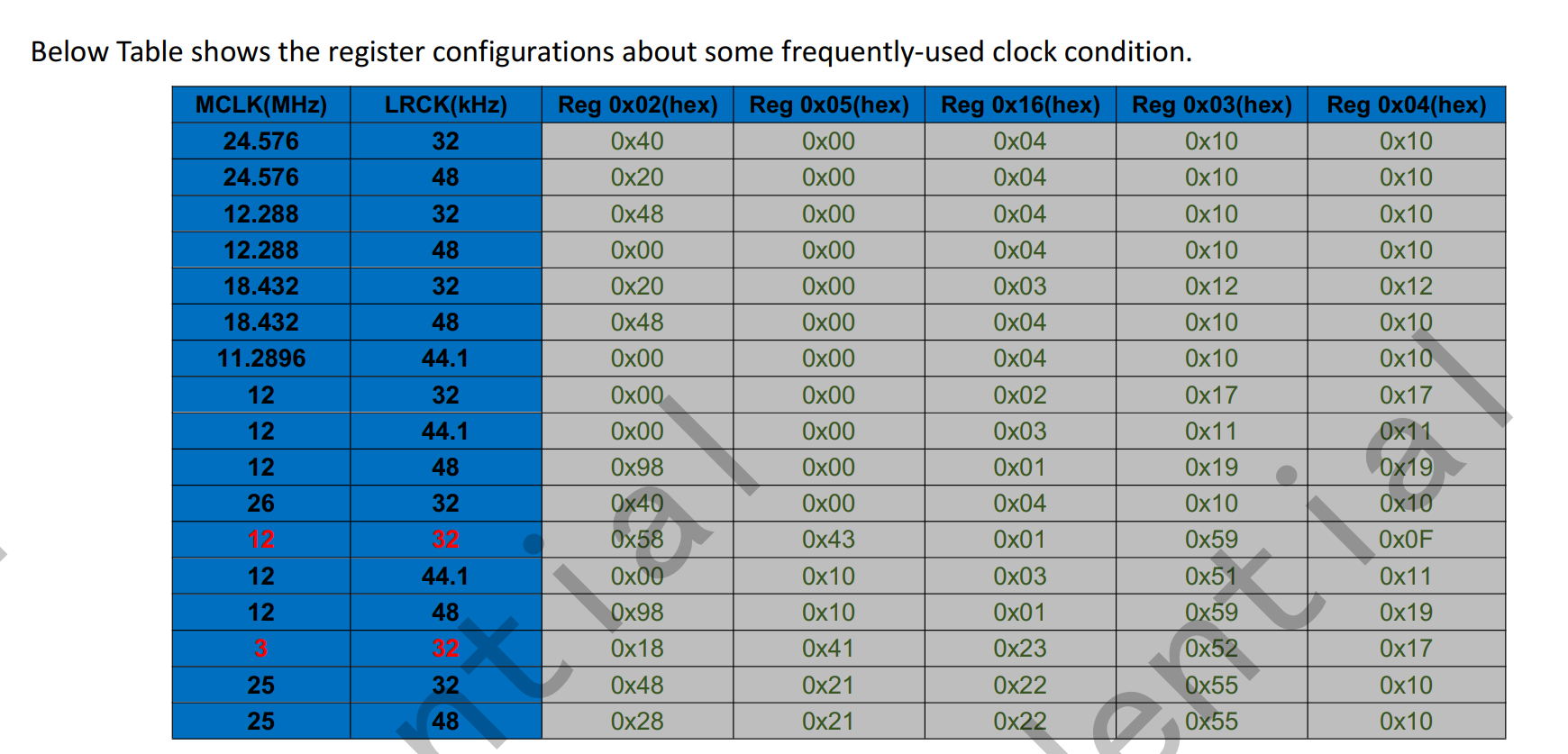
MCLK/SCK =  384 、256 等需要参考手册说明支持哪种；

来源： <https://bbs.huaweicloud.com/blogs/309313>









=====

{12288000, 8000 , 0x06, 0x01, 0x01, 0x01, 0x00, 0x00, 0xff, 0x04, 0x10, 0x10}

12288000, //mclk=12.288M

8000 , //rate  lrck/rate >= 256 --->内部master clk = 8000 \* 256 = 2048000  = mclk / 6 ---> 6 = prediv /premulti -->prediv:0x6   premulti:0x1

0x06, //prediv DIV\_PRE的取值:1 to 8; 寄存器reg02 bit[7:5] bit000--->bit111

0x01,//premulti  MULTI\_PRE的取值:1,2,4,8,寄存器：reg02 bit[4:3] bit00--->bit11

 0x01, //adcdiv    reg05 bit[7:4] 取值1->16 bit0000 --> bit1111 adc\_mclk=dig\_mclk/(DIV\_CLKADC+1)   dig\_mclk=mclk\_prediv\*MULT\_PRE= mclk/(DIV\_PRE+1)\*MULT\_PRE  =

0x01,//dacdiv    reg05 bit[3:0] 取值1->16 bit0000 --> bit1111 dac\_mclk=dig\_mclk/(DIV\_CLKDAC+1)    dac\_mclk/lrck >= 256

 0x00, //fsmode    0:单模式  1:双模式

0x00,//lrck\_h LRCK(master)=dig\_mclk/(LRCK\_DIV+1) ====》 master模式

 0xff, //lrck\_l LRCK(master)=dig\_mclk/(LRCK\_DIV+1)

0x04, //bclkdiv  dig\_mclk/(DIV\_BCLK+1) (default 3)

0x10, //adcosr 积分频率:64fs

0x10//dacosr 积分64fs

===

/\* 64k \*/

{

18432000, // mclk = 18432000

64000,  //rate --->radio = 18432000/64000 = 288

0x03, //--->3

0x04, //4

0x03, //adc mclk =mclk/3 \* 4 /3 =   --->rate = adc\_mclk/64000 = 128

0x01, //dac\_clk = mclk/3 \*4 =24,576,000   ---> rate = dac\_clk/64000 = 384

0x01,

 0x7f,

0x06,

0x10,

0x10

},

==

/\* 44.1k \*/

    {

11289600, //mclk = 11.2896M

 44100, //rate = 44100  ;internal master clk = rate \* 256 = 44100  \* 256 = 11,289,600 = mclk /1 ==> 1 =  prediv /premulti  = 1/1

0x01,  //prediv  =1

0x01,  //premulti   =1

0x01,    //

0x01,

0x00,

0x00,

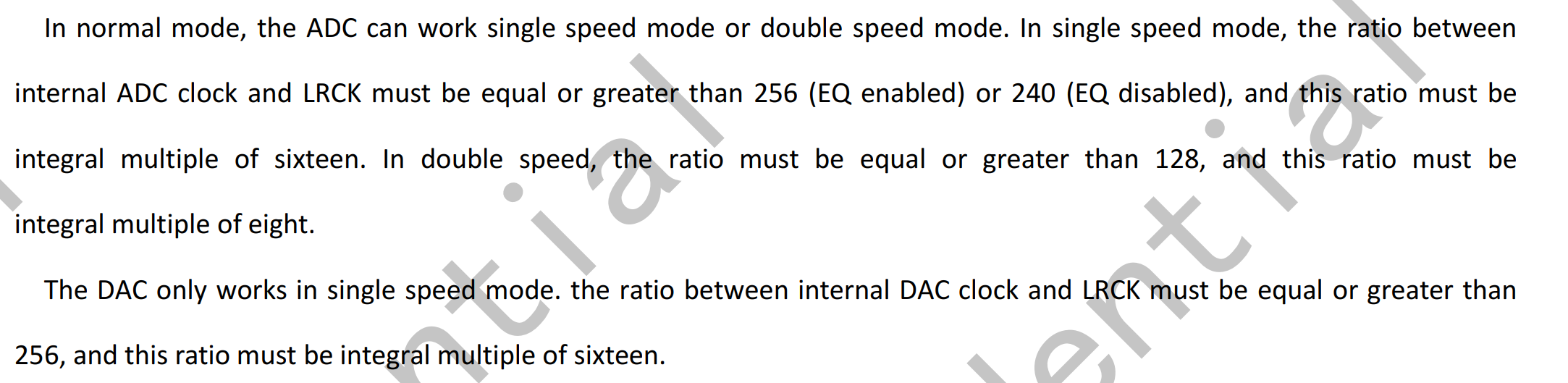
 0xff,

0x04,

0x10,

0x10},

=====





26000000,//mclk = 26M

48000// fs = 48000    internal master clk = 48000 \* 256 = 12288000 ; --> 12288000 = mclk/x == 26000000/x ---> x =

====

reg0x2 0x40--->   MULT\_PRE:1    DIV\_PRE:3

reg0x05 0x00 ----> 设置DIV\_CLKADC和DIV\_CLKDAC为0

reg0x16 0x04 ---> adc 增益 24db

reg0x03 0x10 ADC\_OSR:16-64fs

reg0x04 0x10 DAC\_OSR:16-64fs

26000000,//mclk = 26M

32000 ，// fs = 32000  --->    internal master clk = mclk \*  MULT\_PRE/(DIV\_PRE + 1) = mclk \* 2/(2+1) = mclk \* 2/3 =

3， //prediv bit[7:5] = 0x2  :reg0x2 0x40--->   MULT\_PRE:1    DIV\_PRE:3

1,// ---> 2 premulti bit[4:3] = 0x0

0x1, //adcdiv  reg0x05 0x00 ----> 设置DIV\_CLKADC和DIV\_CLKDAC为0

0x1,//dacdiv

0x00, //fsmode    0:单模式  1:双模式

0x10, //adcosr 积分频率:64fs

0x10//dacosr 积分64fs

{26000000, 32000, 0x03, 0x01, 0x01, 0x01, 0x00, 0x00, 0xff, 0x04, 0x10, 0x10},

{12288000, 32000, 0x03, 0x02, 0x01, 0x01, 0x00, 0x00, 0xff, 0x04, 0x10, 0x10},

{

12288000, //mclk = 12288000

44100, //rate = 44100   mclk/rate=12288000/44100 = 278.639

1,

1,

0x1,

0x1,

0x00,

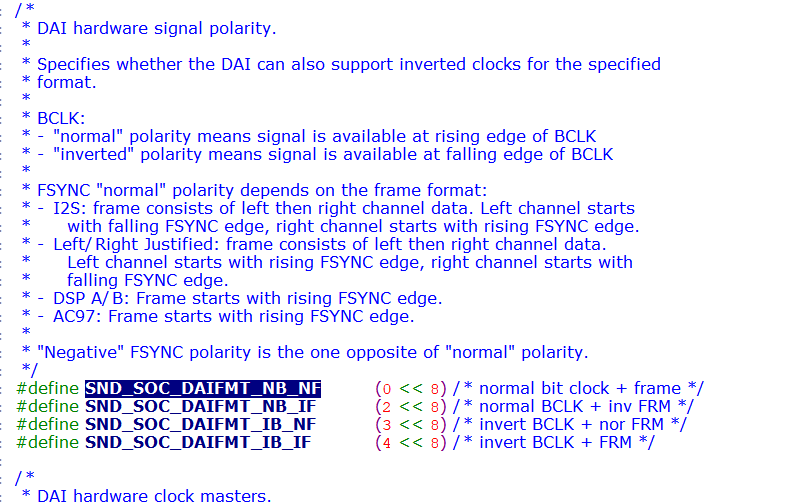
0x10,

0x10

}

===

关于时钟反正



26000000

==

其中的reg字段被设置为SND\_SOC\_NOPM（-1），表明这些widget是没有寄存器控制位来控制widget的电源状态的。麦克风，耳机，扬声器，线路输入接口这几

<https://zhuanlan.zhihu.com/p/537564029>

PRODUCT\_PACKAGES += \

    lights.$(TARGET\_BOARD\_PLATFORM) \

    sensors.$(TARGET\_BOARD\_PLATFORM) \

    tinymix \

PRODUCT\_PACKAGES += \

tinycap \

tinyplay \

tinyhostless \

tinypcminfo

ud710\_2h10:/proc/asound/card1 # tinymix -D 1

Mixer name: 'all-i2s'

Number of controls: 57

ctl type num name value

0 INT 1 MIC PGA GAIN 10

1 INT 1 ADC SCALE 4

2 ENUM 1 DMIC TYPE dmic at high level

3 INT 1 ADC RAMP RATE 0

4 BOOL 1 ADC SDP MUTE On

5 BOOL 1 ADC INVERTED Off

6 BOOL 1 ADC SYNC Off

7 BOOL 1 ADC RAM CLR Off

8 INT 1 ADC VOLUME 191

9 BOOL 1 ALC ENABLE Off

10 ENUM 1 ALC AUTOMUTE TYPE automute disabled

11 INT 1 ALC WIN SIZE 0

12 INT 1 ALC MAX LEVEL 0

13 INT 1 ALC MIN LEVEL 0

14 INT 1 ALC AUTOMUTE WINSIZE 0

15 INT 1 ALC AUTOMUTE GATE THRESHOLD 0

16 INT 1 ALC AUTOMUTE VOLUME 0

17 BOOL 1 ADC FS MODE Off

18 INT 1 ADC OSR 16

19 BOOL 1 DAC SDP MUTE Off

20 BOOL 1 DAC DEM MUTE Off

21 BOOL 1 DAC INVERT Off

22 BOOL 1 DAC RAM CLR Off

23 ENUM 1 DAC DSM MUTE mute to 8

24 INT 1 DAC OFFSET 0

25 INT 1 DAC VOLUME 0

26 BOOL 1 DRC ENABLE Off

27 INT 1 DRC WIN SIZE 0

28 INT 1 DRC MAX LEVEL 0

29 INT 1 DRC MIN LEVEL 0

30 INT 1 DAC RAMP RATE 0

31 INT 1 DAC OSR 16

32 ENUM 1 AEC MODE adc left, adc right

33 ENUM 1 ADC DATA TO DAC TEST MODE disable

34 BOOL 1 MCLK INVERT Off

35 BOOL 1 BCLK INVERT Off

36 ENUM 1 MCLK SOURCE from mclk pin

37 INT 1 fs 32000

38 INT 1 hw\_port 0

39 INT 1 slave\_timeout 3857

40 INT 1 bus\_type 0

41 INT 1 byte\_per\_chan 1

42 INT 1 mode 0

43 INT 1 lsb 0

44 INT 1 rtx\_mode 3

45 INT 1 lrck\_inv 1

46 INT 1 sync\_mode 1

47 INT 1 clk\_inv 0

48 INT 1 i2s\_bus\_mode 1

49 INT 1 pcm\_bus\_mode 0

50 INT 1 pcm\_slot 0

51 INT 1 pcm\_cycle 0

52 INT 1 tx\_watermark 12

53 INT 1 rx\_watermark 20

54 ENUM 1 DMIC MUX DMIC DISABLE

55 ENUM 1 SDP OUT MUX FROM ADC OUT

56 ENUM 1 DAC SDP SRC MUX SELECT SDP LEFT DATA

录音前:

ud710\_2h10:/data # tinymix -D 1

Mixer name: 'all-i2s'

Number of controls: 57

ctl type num name value

0 INT 1 MIC PGA GAIN 10

1 INT 1 ADC SCALE 4

2 ENUM 1 DMIC TYPE dmic at high level

3 INT 1 ADC RAMP RATE 0

4 BOOL 1 ADC SDP MUTE On

5 BOOL 1 ADC INVERTED Off

6 BOOL 1 ADC SYNC Off

7 BOOL 1 ADC RAM CLR Off

8 INT 1 ADC VOLUME 191

9 BOOL 1 ALC ENABLE Off

10 ENUM 1 ALC AUTOMUTE TYPE automute disabled

11 INT 1 ALC WIN SIZE 0

12 INT 1 ALC MAX LEVEL 0

13 INT 1 ALC MIN LEVEL 0

14 INT 1 ALC AUTOMUTE WINSIZE 0

15 INT 1 ALC AUTOMUTE GATE THRESHOLD 0

16 INT 1 ALC AUTOMUTE VOLUME 0

17 BOOL 1 ADC FS MODE Off

18 INT 1 ADC OSR 16

19 BOOL 1 DAC SDP MUTE Off

20 BOOL 1 DAC DEM MUTE Off

21 BOOL 1 DAC INVERT Off

22 BOOL 1 DAC RAM CLR Off

23 ENUM 1 DAC DSM MUTE mute to 8

24 INT 1 DAC OFFSET 0

25 INT 1 DAC VOLUME 0

26 BOOL 1 DRC ENABLE Off

27 INT 1 DRC WIN SIZE 0

28 INT 1 DRC MAX LEVEL 0

29 INT 1 DRC MIN LEVEL 0

30 INT 1 DAC RAMP RATE 0

31 INT 1 DAC OSR 16

32 ENUM 1 AEC MODE adc left, adc right

33 ENUM 1 ADC DATA TO DAC TEST MODE disable

34 BOOL 1 MCLK INVERT Off

35 BOOL 1 BCLK INVERT Off

36 ENUM 1 MCLK SOURCE from mclk pin

37 INT 1 fs 32000

38 INT 1 hw\_port 0

39 INT 1 slave\_timeout 3857

40 INT 1 bus\_type 0

41 INT 1 byte\_per\_chan 1

42 INT 1 mode 0

43 INT 1 lsb 0

44 INT 1 rtx\_mode 3

45 INT 1 lrck\_inv 1

46 INT 1 sync\_mode 1

47 INT 1 clk\_inv 0

48 INT 1 i2s\_bus\_mode 1

49 INT 1 pcm\_bus\_mode 0

50 INT 1 pcm\_slot 0

51 INT 1 pcm\_cycle 0

52 INT 1 tx\_watermark 12

53 INT 1 rx\_watermark 20

54 ENUM 1 DMIC MUX DMIC DISABLE

55 ENUM 1 SDP OUT MUX FROM ADC OUT

56 ENUM 1 DAC SDP SRC MUX SELECT SDP LEFT DATA

录音后

ud710\_2h10:/ # tinymix -D 1

Mixer name: 'all-i2s'

Number of controls: 57

ctl type num name value

0 INT 1 MIC PGA GAIN 10

1 INT 1 ADC SCALE 4

2 ENUM 1 DMIC TYPE dmic at high level

3 INT 1 ADC RAMP RATE 4

4 BOOL 1 ADC SDP MUTE On

5 BOOL 1 ADC INVERTED Off

6 BOOL 1 ADC SYNC Off

7 BOOL 1 ADC RAM CLR Off

8 INT 1 ADC VOLUME 191

9 BOOL 1 ALC ENABLE Off

10 ENUM 1 ALC AUTOMUTE TYPE automute disabled

11 INT 1 ALC WIN SIZE 0

12 INT 1 ALC MAX LEVEL 0

13 INT 1 ALC MIN LEVEL 0

14 INT 1 ALC AUTOMUTE WINSIZE 0

15 INT 1 ALC AUTOMUTE GATE THRESHOLD 0

16 INT 1 ALC AUTOMUTE VOLUME 0

17 BOOL 1 ADC FS MODE Off

18 INT 1 ADC OSR 16

19 BOOL 1 DAC SDP MUTE Off

20 BOOL 1 DAC DEM MUTE Off

21 BOOL 1 DAC INVERT Off

22 BOOL 1 DAC RAM CLR Off

23 ENUM 1 DAC DSM MUTE mute to 8

24 INT 1 DAC OFFSET 0

25 INT 1 DAC VOLUME 191

26 BOOL 1 DRC ENABLE Off

27 INT 1 DRC WIN SIZE 0

28 INT 1 DRC MAX LEVEL 0

29 INT 1 DRC MIN LEVEL 0

30 INT 1 DAC RAMP RATE 4

31 INT 1 DAC OSR 16

32 ENUM 1 AEC MODE adc left, adc right

33 ENUM 1 ADC DATA TO DAC TEST MODE disable

34 BOOL 1 MCLK INVERT Off

35 BOOL 1 BCLK INVERT Off

36 ENUM 1 MCLK SOURCE from mclk pin

37 INT 1 fs 32000

38 INT 1 hw\_port 0

39 INT 1 slave\_timeout 3857

40 INT 1 bus\_type 0

41 INT 1 byte\_per\_chan 1

42 INT 1 mode 0

43 INT 1 lsb 0

44 INT 1 rtx\_mode 3

45 INT 1 lrck\_inv 1

46 INT 1 sync\_mode 1

47 INT 1 clk\_inv 0

48 INT 1 i2s\_bus\_mode 1

49 INT 1 pcm\_bus\_mode 0

50 INT 1 pcm\_slot 0

51 INT 1 pcm\_cycle 0

52 INT 1 tx\_watermark 12

53 INT 1 rx\_watermark 20

54 ENUM 1 DMIC MUX DMIC DISABLE

55 ENUM 1 SDP OUT MUX FROM ADC OUT

56 ENUM 1 DAC SDP SRC MUX SELECT SDP LEFT DATA

====

0x69000

iis\_intf1  ----> ap\_iis1

bit[9:6] = 0x1

110 1001 0000 0100 0000

[   77.834979] c5 libprocessgroup: Successfully killed process cgroup uid 1000 pid 5358 in 0ms

[   82.603364] c0 [ASoC: I2S ] i2s config hw\_port 1

[   82.608492] c4 Bad mode in Error handler detected on CPU4, code 0xbe000011 -- SError

[   82.616127] c4 Internal error: Oops - bad mode: 0 [#1] PREEMPT SMP

[   82.622268] c4 dump\_die\_cb in.

[   82.625294] c4 dump\_die\_cb save pregs\_die\_g ok .

[   82.629879] c4 dump\_die\_cb out.

===

iis0:

{REG\_PIN\_IIS0DI,                        BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS0DI,                   BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//BT\_PCM\_OUT

{REG\_PIN\_IIS0DO,                        BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS0DO,                   BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AP|BIT\_PIN\_SLP\_NUL|BIT\_PIN\_SLP\_OE},//BT\_PCM\_IN

{REG\_PIN\_IIS0CLK,                       BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS0CLK,                  BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//BT\_PCM\_CLK

{REG\_PIN\_IIS0LRCK,                      BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS0LRCK,                 BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//BT\_PCM\_SYNC

iis1:

ud710-20c10:

{REG\_PIN\_IIS1DI,                        BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1DI,                   BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//SPKR\_I2S\_DIN

{REG\_PIN\_IIS1DO,                        BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1DO,                   BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_NUL|BIT\_PIN\_SLP\_OE},//SPKR\_I2S\_DOUT

{REG\_PIN\_IIS1CLK,                       BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1CLK,                  BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//SPKR\_I2S\_BCK

{REG\_PIN\_IIS1LRCK,                      BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1LRCK,                 BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//SPKR\_I2S\_WS

iis3:

{REG\_PIN\_RFCTL\_5,                       BITS\_PIN\_AF(1)},

{REG\_MISC\_PIN\_RFCTL\_5,                  BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//AP\_I2S3\_SDI

{REG\_PIN\_RFCTL\_6,                       BITS\_PIN\_AF(3)},

{REG\_MISC\_PIN\_RFCTL\_6,                  BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_WPU|BIT\_PIN\_SLP\_AP|BIT\_PIN\_SLP\_WPU|BIT\_PIN\_SLP\_OE},////audio pa(GPIO25)

{REG\_PIN\_RFCTL\_7,                       BITS\_PIN\_AF(1)},

{REG\_MISC\_PIN\_RFCTL\_7,                  BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_NUL|BIT\_PIN\_SLP\_OE},//AP\_I2S3\_LRCK

{REG\_PIN\_RFCTL\_8,                       BITS\_PIN\_AF(1)},

{REG\_MISC\_PIN\_RFCTL\_8,                  BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//AP\_I2S3\_SCLK

===

0x69000

110 1001 0000 0000 0000

bit[3:0] = 0000 ---> iis\_inf0 = ap\_iis0

bit4 = 0  intf0--> intf1 loop=0

bit5 = 0  intf0 ---> intf2 loop = 0

bit[9:6] = 0000 ---> iis\_intf1 = ap\_iis0

bit[10] = 0 ---> iis\_inf1 ---> intf2 loop = 0

bit[14:11] = 0010 ---> iis\_inf2 ---> ap\_iis2

bit[18:15] = 1101 --->iis\_intf3-->audcp\_iis0

---->

修改为

0x69001

====================

0x69040-->110 1001 0000 0100 0000

==

主mic录音

tinymix -D 0 "VBC ADCL DG Switch" 1

CONFIG\_SND\_SOC\_SPRD\_AUDIO\_TWO\_STAGE\_DMAENGINE\_SURPPORT

作用？？

===

{REG\_PIN\_IIS\_MATRIX\_MTX\_CFG,0x00069001},//IIS0->AP\_IIS1; IIS1->AP\_IIS0; IIS2->AP\_IIS2; IIS3->AUDCP\_IIS0;

{REG\_PIN\_IIS\_MATRIX\_MTX\_CFG,0x00069001},//IIS0->AP\_IIS1; IIS1->AP\_IIS0; IIS2->AP\_IIS2; IIS3->AUDCP\_IIS0;

==

问题：音频i2s mclk和lrck没有时钟输出

使用iis1(对应gpio130 gpio131 gpio132 gpio133 )接音频codec     t710通过这个音频codec 与天通做语音交互

uboot设置REG\_PIN\_IIS\_MATRIX\_MTX\_CFG的IIS1->AP\_IIS0

{REG\_PIN\_IIS\_MATRIX\_MTX\_CFG,0x00069001},//IIS0->AP\_IIS1; IIS1->AP\_IIS0; IIS2->AP\_IIS2; IIS3->AUDCP\_IIS0;

设置gpio为iis

{REG\_PIN\_IIS1DI,                        BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1DI,                   BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_WPD|BIT\_PIN\_SLP\_ALL|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//FTID\_INT --->  codec iis iis1di

{REG\_PIN\_IIS1DO,                        BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1DO,                   BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_ALL|BIT\_PIN\_SLP\_NUL|BIT\_PIN\_SLP\_OE},//codec iis IIS1DO

{REG\_PIN\_IIS1CLK,                       BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1CLK,                  BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_ALL|BIT\_PIN\_SLP\_NUL|BIT\_PIN\_SLP\_OE},//codec iis IIS1CLK

{REG\_PIN\_IIS1LRCK,                      BITS\_PIN\_AF(0)},

{REG\_MISC\_PIN\_IIS1LRCK,                 BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_ALL|BIT\_PIN\_SLP\_NUL|BIT\_PIN\_SLP\_OE},//codec iis IIS1LRCLK

在内核把codec注册在声卡1上:

&sound\_sprd\_ap\_alliis {

        status = "okay";

        /\* dai-links \*/

        sprd-audio-card,dai-link@1 {

                plat {

                        sound-dai = <&sprd\_pcm\_iis>;

                };

                cpu {

                        sound-dai = <&i2s0>;

                };

                codec {

                        sound-dai = <&es8311 0>;

                };

        };

};

设置了i2s0的格式

&i2s0 {

        status = "okay";

        sprd,config\_type = "i2s";

        sprd,slave\_timeout = <0xf11>;

        sprd,hw\_port = <0>;

        //sprd,fs = <48000>;

        sprd,fs = <32000>;

        sprd,bus\_type = <0>;

        sprd,rtx\_mode = <3>;

        sprd,byte\_per\_chan = <1>;

        sprd,slave\_mode = <0>;

        sprd,lsb = <0>;

        sprd,lrck = <0>; /\*sync\_mode\*/

        sprd,low\_for\_left = <0>; /\*lrck\_inv\*/

        //sprd,clk\_inv = <1>;

        sprd,clk\_inv = <0>;

        sprd,pcm\_short\_frame = <0>; /\*pcm\_bus\_mode\*/

        sprd,pcm\_slot = <0x1>;

        sprd,pcm\_cycle = <1>;

       // sprd,tx\_watermark = <12>;

       // sprd,rx\_watermark = <20>;

        sprd,tx\_watermark = <16>;

        sprd,rx\_watermark = <16>;

        sprd,i2s\_compatible= <1>;

};

内核sprd\_roc1\_defconfig添加

CONFIG\_SND\_SOC\_SPRD\_I2S=y

CONFIG\_SND\_SOC\_SPRD\_I2S\_DUMMY=y

开机后检测到声卡注册成功了：

ud710\_2h10:/proc/asound/card1/pcm1c # cat info

card: 1

device: 1

subdevice: 0

stream: CAPTURE

id: i2s\_bt\_sco0-ES8311 HiFi ES8311 HiFi-1

name:

subname: subdevice #0

class: 0

subclass: 0

subdevices\_count: 1

subdevices\_avail: 1

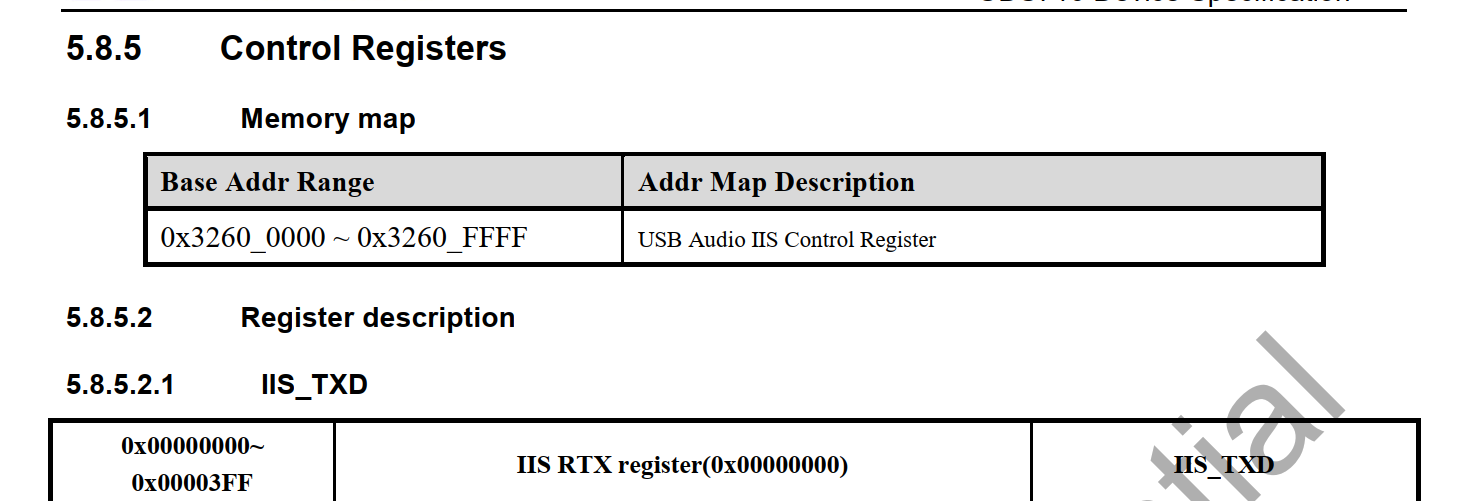
使用tinycap录音 也没报错

tinycap  /data/rec5.wav -D 1 -d 1 -c 2 -r 32000  -T 80

但是录音到的数据 播放不了，同时示波器量 mclk和lrck没有时钟输出。

==

usb 音频控制:



AON 上的MST\_IIS

arch\_audio\_iis\_to\_audio\_top\_enable

/\* AGCP IIS multiplexer setting.

 \* @iis: the iis channel to be set.

 \* @en:

 \*   0: AG\_IIS0\_EXT\_SEL to whale2 top

 \*   1: AG\_IIS0\_EXT\_SEL to audio top

 \*/

static inline int arch\_

===

sprd,syscon-agcp-ahb = <&audcp\_ahb\_regs>;

audcp\_ahb\_regs: syscon@0x335e0000 {

IMG_344

adb  push F:\work\2022-work\t710-pad\tinycap  /system/bin/tinycap

FE\_NORMAL\_AP01

.stream\_name = "FE\_DAI\_NORMAL\_AP01\_P",

            .aif\_name = "FE\_IF\_NORMAL\_AP01\_P",

udx710

udx710\_4h10\_nse

252 {REG\_PIN\_IIS0DI, BITS\_PIN\_AF(0)},

253 {REG\_MISC\_PIN\_IIS0DI, BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//I2S0\_SDO(GPIO45)

254 {REG\_PIN\_IIS0DO, BITS\_PIN\_AF(0)},

255 {REG\_MISC\_PIN\_IIS0DO, BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_NUL|BIT\_PIN\_SLP\_OE},//I2S0\_SDI(GPIO44)

256 {REG\_PIN\_IIS0CLK, BITS\_PIN\_AF(3)},

257 {REG\_MISC\_PIN\_IIS0CLK, BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_WPD|BIT\_PIN\_SLP\_AP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_IE},//I2S0\_SCLK(GPIO46) mofiy by yangjia

258 {REG\_PIN\_IIS0LRCK, BITS\_PIN\_AF(0)},

259 {REG\_MISC\_PIN\_IIS0LRCK, BITS\_PIN\_DS(1)|BIT\_PIN\_NULL|BIT\_PIN\_NUL|BIT\_PIN\_SLP\_AUDCP|BIT\_PIN\_SLP\_WPD|BIT\_PIN\_SLP\_OE},//I2S0\_LRCK(GPIO47)

260 {REG\_PIN\_SCL0, BITS\_PIN\_AF(0)},

CONFIG\_SND\_SMARTPA\_AW881XX

CONFIG\_SND\_SOC\_AW87XX\_IIC\_PA

==

tinycap /data/rec.wav -D -0 -d 0 -r 16000 -b 16 -T 10

tinycap /data/rec.wav -D -0 -d 0 -r 16000 -b 16 -T 10

<double-mic device="0x80000084">

<on>

<ctl name="ADC LRCLK Select" val="invert"/>

<ctl name="ADC1 LRCLK Select" val="invert"/>

<ctl name="MIC Boost" val="1"/>

<ctl name="AUXMIC Boost" val="1"/>

<ctl name="ADCR Mixer AuxMICADCR Switch" val="1"/>

<ctl name="ADCL Mixer MainMICADCL Switch" val="1"/>

<ctl name="AUD ADC0L Switch" val="1" />

<ctl name="AUD ADC0R Switch" val="1" />

<ctl name="Aux Mic Function" val="1"/>

<ctl name="Mic Function" val="1"/>

</on>

tinymix "ADC1 LRCLK Select" "invert"

tinymix "ADC LRCLK Select" "invert"

tinymix "ADC1 LRCLK Select" "invert"

tinymix "MIC Boost" "1"

tinymix "AUXMIC Boost" "1"

tinymix "ADCR Mixer AuxMICADCR Switch" "1"

tinymix "ADCL Mixer MainMICADCL Switch" "1"

tinymix "AUD ADC0L Switch" "1"

tinymix "AUD ADC0R Switch" "1"

tinymix "Aux Mic Function" "1"

tinymix "Mic Function" "1"

tinymix "ADC LRCLK Select"                  "invert"

tinymix "ADC1 LRCLK Select"              "invert"

tinymix "MIC Boost"                      "1"

tinymix "AUXMIC Boost"                      "1"

tinymix "ADCR Mixer AuxMICADCR Switch"   "1"

tinymix "ADCL Mixer MainMICADCL Switch"  "1"

tinymix "AUD ADC0L Switch"                  "1"

tinymix "AUD ADC0R Switch"                  "1"

tinymix "Aux Mic Function"                  "1"

tinymix "Mic Function"                      "1"

==

audio\_pcm.xml

<mm\_normal channels="2" rate="48000" period\_size="960" period\_count="4" format="0" card="sprdphone-sc2730"  device="3"/>

format:

audio\_format\_t

AUDIO\_FORMAT\_MP3

AUDIO\_FORMAT\_PCM\_SUB\_16\_BIT

card:声卡

device：pcm

//

====

**声卡测试：**  
① 编译工具：**mmm external/tinyalsa/**  
② 播放：tinyplay file.wav [-D card] [-d device] [-p period\_size] [-n n\_periods]  
　　        **tinyplay /sdcard/test.wav -D 0 -d 0 -p 1024 -n 3**  
③ 录音：**tinycap /sdcard/rec.wav -D 0 -d 0 -c 2 -r 44100 -b 16 -p 1024 -n  3**

**来源：**<https://www.cnblogs.com/blogs-of-lxl/p/6538769.html>

enum {

AUDIO\_DEVICE\_NONE = 0x0,

/\* reserved bits \*/

AUDIO\_DEVICE\_BIT\_IN = 0x80000000,

AUDIO\_DEVICE\_BIT\_DEFAULT = 0x40000000,

/\* output devices \*/

AUDIO\_DEVICE\_OUT\_EARPIECE = 0x1, // 听筒

AUDIO\_DEVICE\_OUT\_SPEAKER = 0x2, // 扬声器

AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET = 0x4, // 线控耳机，可以通过耳机控制远端播放、暂停、音量调节等功能的耳机

AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE = 0x8, // 普通耳机，只能听，不能操控播放

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO = 0x10, // 单声道蓝牙耳机，十进制32

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET = 0x20, // 车载免提蓝牙设备，十进制64

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT = 0x40, // 立体声蓝牙耳机

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP = 0x80, // 十进制128

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES = 0x100, // 十进制256

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER = 0x200, // 十进制512

AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL = 0x400, // 十进制1024

AUDIO\_DEVICE\_OUT\_ANLG\_DOCK\_HEADSET = 0x800, // 十进制2048

AUDIO\_DEVICE\_OUT\_DGTL\_DOCK\_HEADSET = 0x1000, // 十进制4096

AUDIO\_DEVICE\_OUT\_USB\_ACCESSORY = 0x2000,

AUDIO\_DEVICE\_OUT\_USB\_DEVICE = 0x4000,

AUDIO\_DEVICE\_OUT\_REMOTE\_SUBMIX = 0x8000,

AUDIO\_DEVICE\_OUT\_DEFAULT = AUDIO\_DEVICE\_BIT\_DEFAULT,

AUDIO\_DEVICE\_OUT\_ALL = (AUDIO\_DEVICE\_OUT\_EARPIECE |

AUDIO\_DEVICE\_OUT\_SPEAKER |

AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET |

AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER |

AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL |

AUDIO\_DEVICE\_OUT\_ANLG\_DOCK\_HEADSET |

AUDIO\_DEVICE\_OUT\_DGTL\_DOCK\_HEADSET |

AUDIO\_DEVICE\_OUT\_USB\_ACCESSORY |

AUDIO\_DEVICE\_OUT\_USB\_DEVICE |

AUDIO\_DEVICE\_OUT\_REMOTE\_SUBMIX |

AUDIO\_DEVICE\_OUT\_DEFAULT),

AUDIO\_DEVICE\_OUT\_ALL\_A2DP = (AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER),

AUDIO\_DEVICE\_OUT\_ALL\_SCO = (AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT),

AUDIO\_DEVICE\_OUT\_ALL\_USB = (AUDIO\_DEVICE\_OUT\_USB\_ACCESSORY |

AUDIO\_DEVICE\_OUT\_USB\_DEVICE),

/\* input devices \*/

AUDIO\_DEVICE\_IN\_COMMUNICATION = AUDIO\_DEVICE\_BIT\_IN | 0x1,

AUDIO\_DEVICE\_IN\_AMBIENT = AUDIO\_DEVICE\_BIT\_IN | 0x2,

AUDIO\_DEVICE\_IN\_BUILTIN\_MIC = AUDIO\_DEVICE\_BIT\_IN | 0x4,　　//手机自带MIC

AUDIO\_DEVICE\_IN\_BLUETOOTH\_SCO\_HEADSET = AUDIO\_DEVICE\_BIT\_IN | 0x8,

AUDIO\_DEVICE\_IN\_WIRED\_HEADSET = AUDIO\_DEVICE\_BIT\_IN | 0x10,　　//耳机

AUDIO\_DEVICE\_IN\_AUX\_DIGITAL = AUDIO\_DEVICE\_BIT\_IN | 0x20,

AUDIO\_DEVICE\_IN\_VOICE\_CALL = AUDIO\_DEVICE\_BIT\_IN | 0x40,

AUDIO\_DEVICE\_IN\_BACK\_MIC = AUDIO\_DEVICE\_BIT\_IN | 0x80,

AUDIO\_DEVICE\_IN\_REMOTE\_SUBMIX = AUDIO\_DEVICE\_BIT\_IN | 0x100,

AUDIO\_DEVICE\_IN\_ANLG\_DOCK\_HEADSET = AUDIO\_DEVICE\_BIT\_IN | 0x200,

AUDIO\_DEVICE\_IN\_DGTL\_DOCK\_HEADSET = AUDIO\_DEVICE\_BIT\_IN | 0x400,

AUDIO\_DEVICE\_IN\_USB\_ACCESSORY = AUDIO\_DEVICE\_BIT\_IN | 0x800,

AUDIO\_DEVICE\_IN\_USB\_DEVICE = AUDIO\_DEVICE\_BIT\_IN | 0x1000,

AUDIO\_DEVICE\_IN\_DEFAULT = AUDIO\_DEVICE\_BIT\_IN | AUDIO\_DEVICE\_BIT\_DEFAULT,

AUDIO\_DEVICE\_IN\_ALL = (AUDIO\_DEVICE\_IN\_COMMUNICATION |

AUDIO\_DEVICE\_IN\_AMBIENT |

AUDIO\_DEVICE\_IN\_BUILTIN\_MIC |

AUDIO\_DEVICE\_IN\_BLUETOOTH\_SCO\_HEADSET |

AUDIO\_DEVICE\_IN\_WIRED\_HEADSET |

AUDIO\_DEVICE\_IN\_AUX\_DIGITAL |

AUDIO\_DEVICE\_IN\_VOICE\_CALL |

AUDIO\_DEVICE\_IN\_BACK\_MIC |

AUDIO\_DEVICE\_IN\_REMOTE\_SUBMIX |

AUDIO\_DEVICE\_IN\_ANLG\_DOCK\_HEADSET |

AUDIO\_DEVICE\_IN\_DGTL\_DOCK\_HEADSET |

AUDIO\_DEVICE\_IN\_USB\_ACCESSORY |

AUDIO\_DEVICE\_IN\_USB\_DEVICE |

AUDIO\_DEVICE\_IN\_DEFAULT),

AUDIO\_DEVICE\_IN\_ALL\_SCO = AUDIO\_DEVICE\_IN\_BLUETOOTH\_SCO\_HEADSET,

};

==========

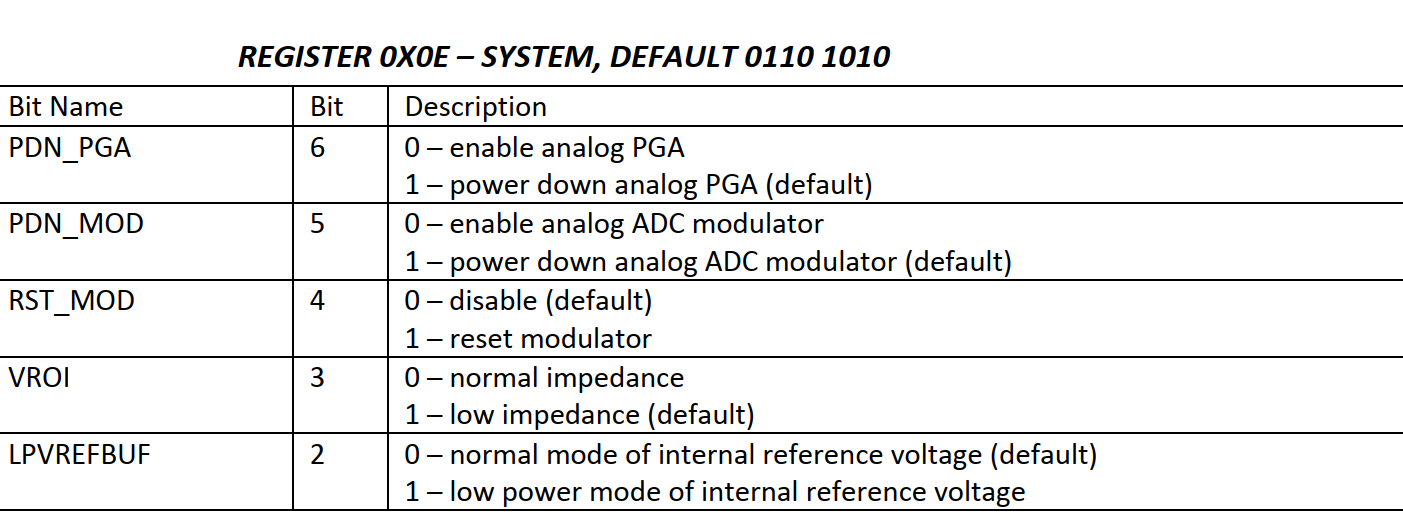
es8311的adc 增益相关

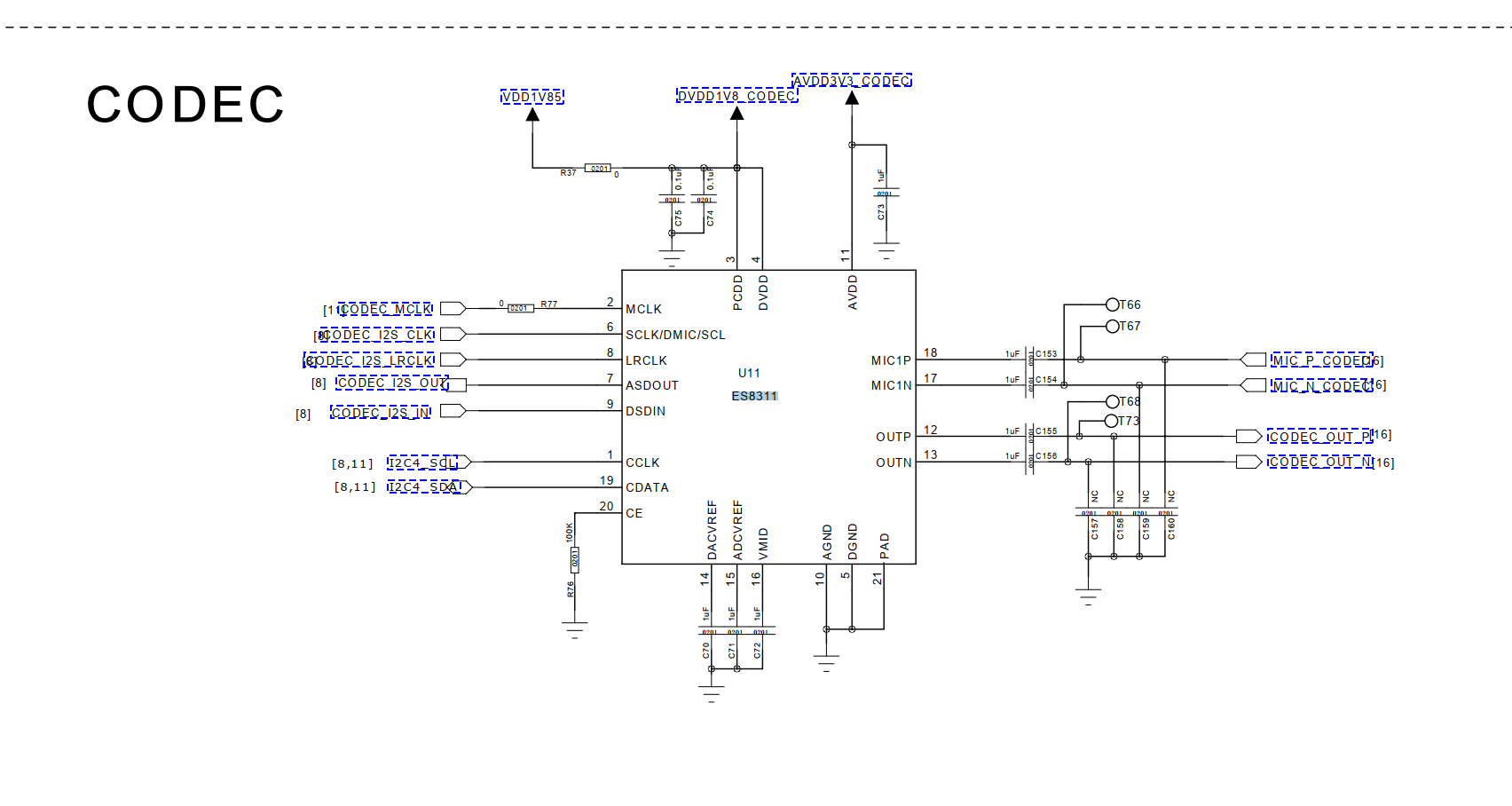
====

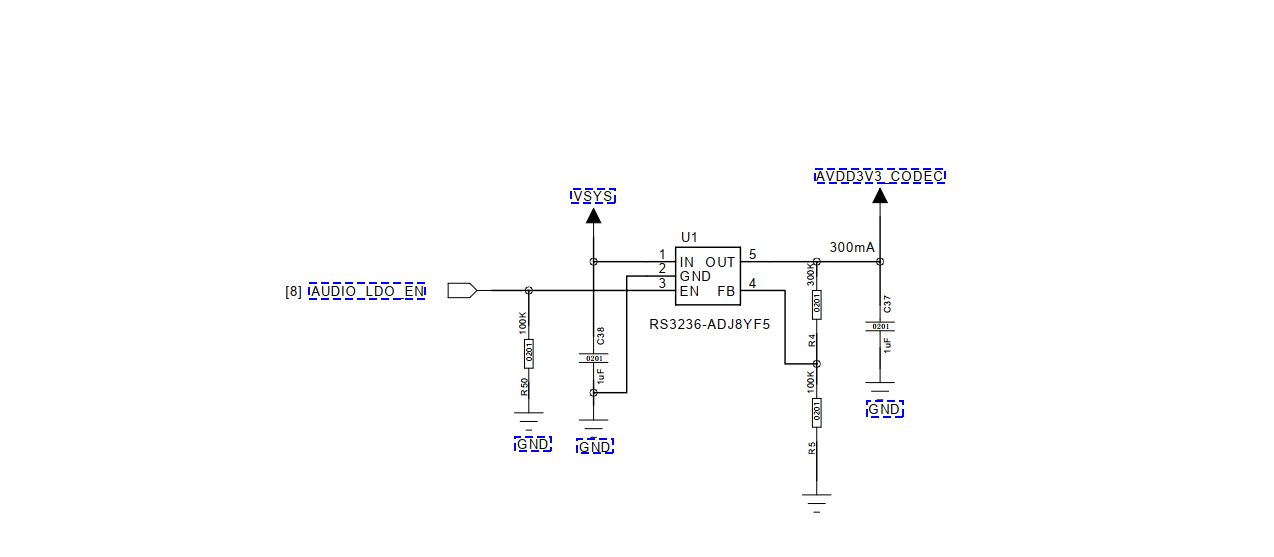
不能播放的原因：

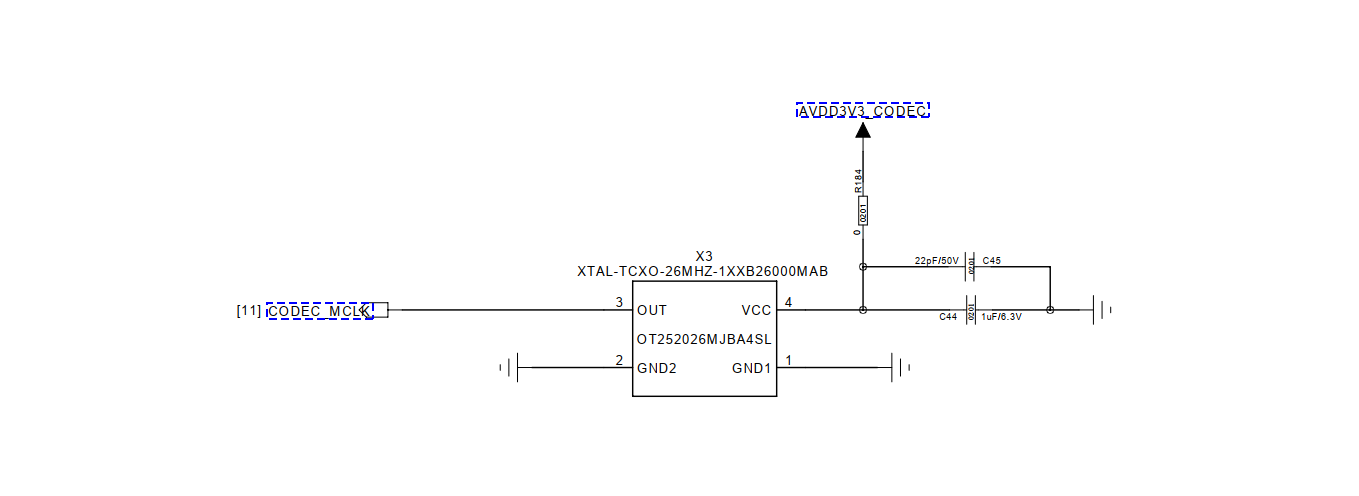
正确的设置为02









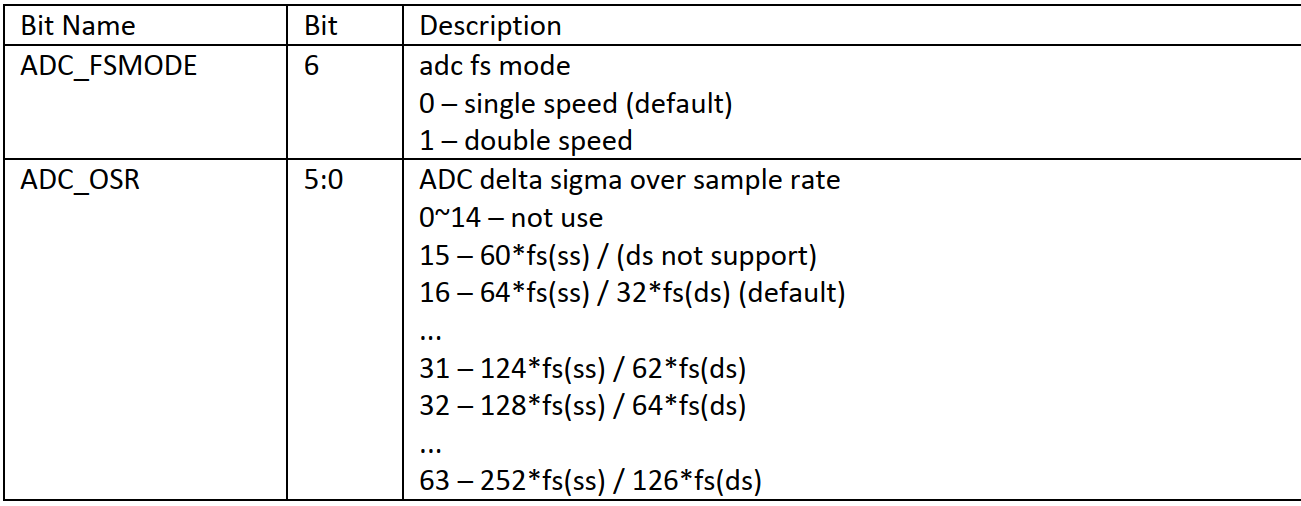


====

[ 2751.435173] c4 Enter into es8311\_pcm\_hw\_params()

[ 2751.439692] c4 stephen es8311\_pcm\_hw\_params es8311->mclk:12288000,rate:16000

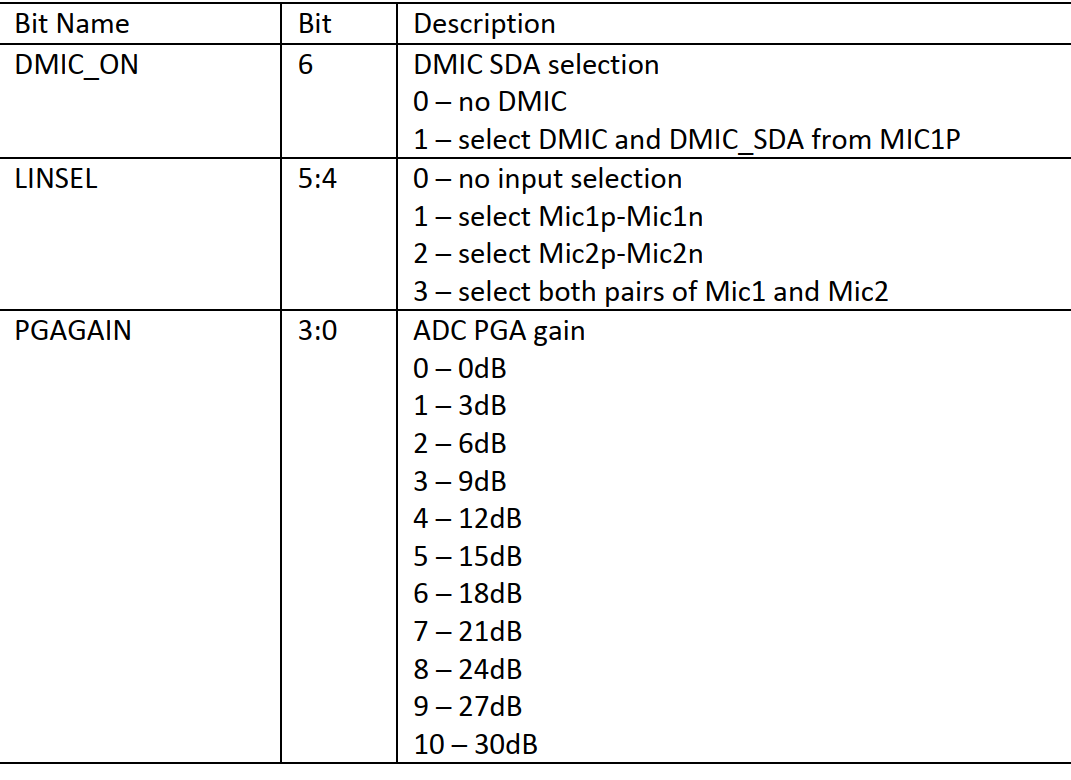
==\



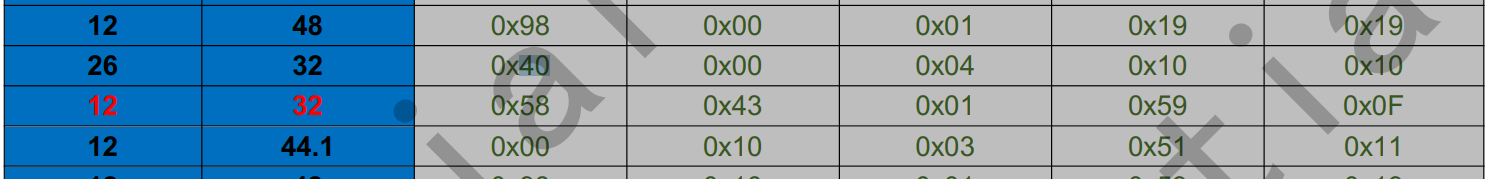
echo 1033f > es8311  ===>没有效果

03: 3f

===



echo 1143a > es8311



====

adc ---> dac设置

你把0x44 改为0x88

从8311的输出端听听

===

把BCLK当MCLK用

这几个寄存器改一下

0X00---0X80

0x01----0x9F

0X02---0X10

echo 1019f > es8311

echo 10210 > es8311

echo 10080 > es8311

===

./tinycap cap.wav -D 1 -d 1 -c 2 -r 48000 -b 16

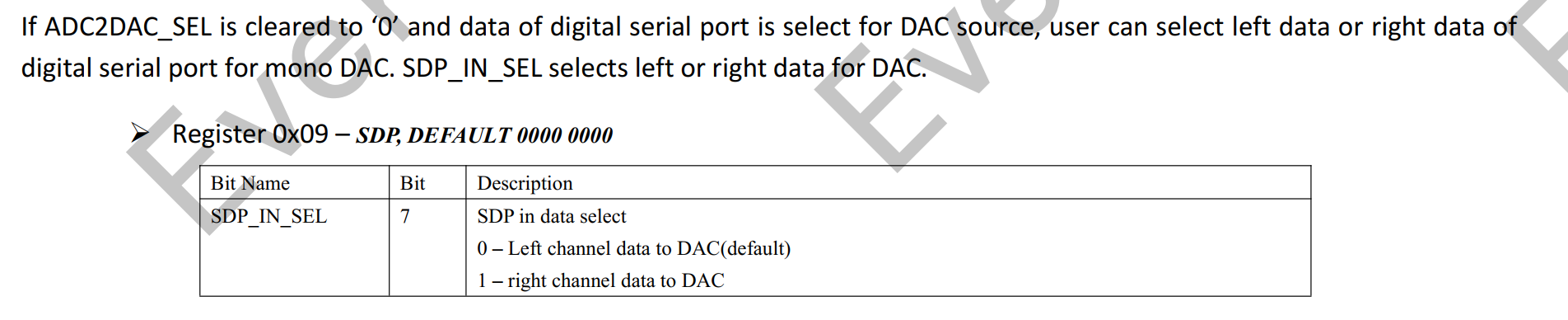
./tinyplay music-48000.wav  -D 1 -d 1

====

0x0d -->0x1

echo 10d01 > es8311

===



==

1100

1000 1100

echo 1098c > es8311

1：写

2:09寄存器

3:8c寄存器的值

====

es8311操作:

ud710\_2h10:/sys # find  ./  -name  \*es8311\*

./kernel/debug/asoc/all-i2s/codec:es8311.4-0018

./devices/platform/soc/soc:ap-apb/70700000.i2c/i2c-4/4-0018/es8311\_debug

./devices/platform/soc/soc:ap-apb/70700000.i2c/i2c-4/4-0018/es8311\_debug/es8311

./bus/i2c/drivers/es8311

./firmware/devicetree/base/soc/ap-apb/i2c@70700000/es8311@0x18

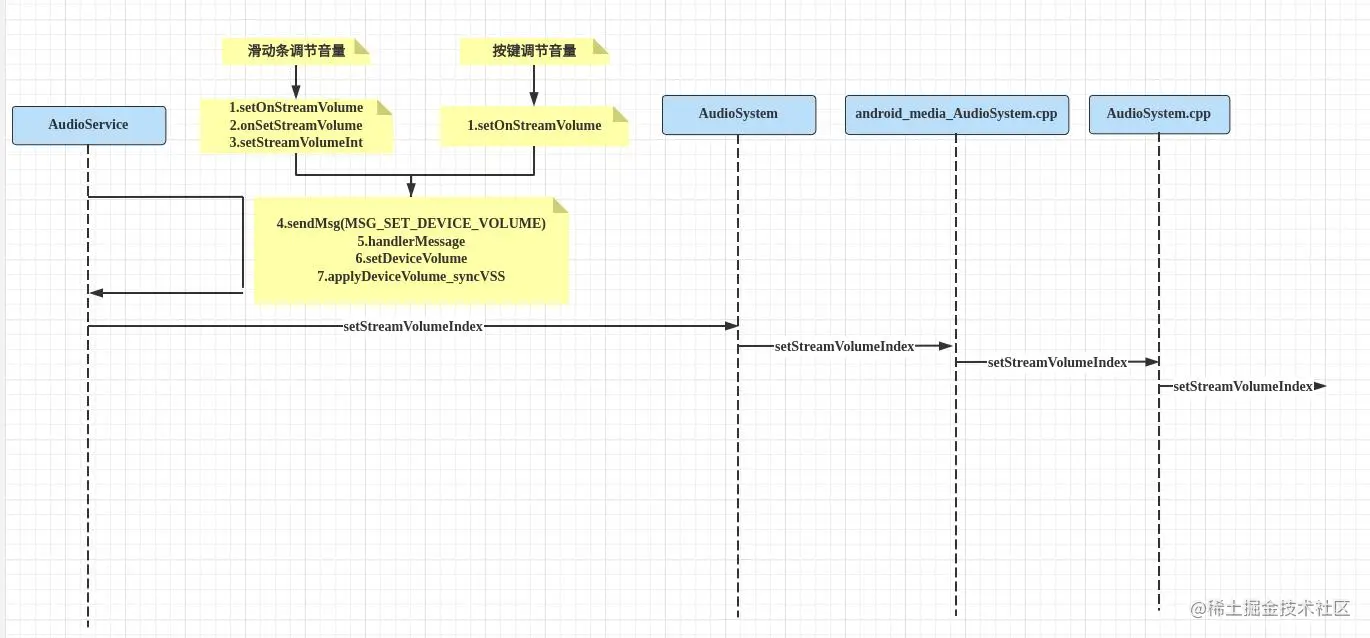
./firmware/devicetree/base/\_\_symbols\_\_/es8311

===

# **Android音视频六：音量调节流程**

# 音量调节JAVA层

按键调节 和 滑动条调节 整体流程区别不大，就将图和在一起了



## 按键调节

protected void adjustStreamVolume(int streamType, int direction, int flags,

String callingPackage, String caller, int uid, boolean hasModifyAudioSettings,

int keyEventMode) {

// lyh 检查调节方向 和 流类型

ensureValidDirection(direction);

ensureValidStreamType(streamType);

// lyh 是否是关于静音的调节（滑动条触发的）

boolean isMuteAdjust = isMuteAdjust(direction);

if (isMuteAdjust && !isStreamAffectedByMute(streamType)) {

return;

}

// lyh 流类型的组名

int streamTypeAlias = mStreamVolumeAlias[streamType];

// lyh 根据流类型的别名在流类型列表中找到对应的流信息

VolumeStreamState streamState = mStreamStates[streamTypeAlias];

// lyh 拿device

final int device = getDeviceForStream(streamTypeAlias);

// lyh 当前音量下标值

int aliasIndex = streamState.getIndex(device);

boolean adjustVolume = true;

int step;

// lyh 计算步长

flags &= ~AudioManager.FLAG\_FIXED\_VOLUME;

if (streamTypeAlias == AudioSystem.STREAM\_MUSIC && isFixedVolumeDevice(device)) {

flags |= AudioManager.FLAG\_FIXED\_VOLUME;

if (mSafeMediaVolumeState == SAFE\_MEDIA\_VOLUME\_ACTIVE &&

mSafeMediaVolumeDevices.contains(device)) {

step = safeMediaVolumeIndex(device);

} else {

step = streamState.getMaxIndex();

}

if (aliasIndex != 0) {

aliasIndex = step;

}

} else {

// convert one UI step (+/-1) into a number of internal units on the stream alias

step = rescaleStep(10, streamType, streamTypeAlias);

}

// lyh 看起来是调节音量时能否改变ringermode状态(组别是：STREAM\_SYSTEM 第二个条件成立)

if (((flags & AudioManager.FLAG\_ALLOW\_RINGER\_MODES) != 0) ||

(streamTypeAlias == getUiSoundsStreamType())) {

int ringerMode = getRingerModeInternal();

// lyh 重要方法

final int result = checkForRingerModeChange(aliasIndex, direction, step,

streamState.mIsMuted, callingPackage, flags);

adjustVolume = (result & FLAG\_ADJUST\_VOLUME) != 0;

}

// lyh 音量下标值，不知道和aliasIndex有什么区别

int oldIndex = mStreamStates[streamType].getIndex(device);

// lyh 第一次点击 direction = 0 ，第二次 = 1

if (adjustVolume

&& (direction != AudioManager.ADJUST\_SAME) && (keyEventMode != VOL\_ADJUST\_END)) {

mAudioHandler.removeMessages(MSG\_UNMUTE\_STREAM);

// lyh 滑动条的，设置同一组下的音量状态

if (isMuteAdjust) {

boolean state;

if (direction == AudioManager.ADJUST\_TOGGLE\_MUTE) {

state = !streamState.mIsMuted;

} else {

state = direction == AudioManager.ADJUST\_MUTE;

}

if (streamTypeAlias == AudioSystem.STREAM\_MUSIC) {

setSystemAudioMute(state);

}

for (int stream = 0; stream < mStreamStates.length; stream++) {

if (streamTypeAlias == mStreamVolumeAlias[stream]) {

if (!(readCameraSoundForced()

&& (mStreamStates[stream].getStreamType()

== AudioSystem.STREAM\_SYSTEM\_ENFORCED))) {

mStreamStates[stream].mute(state);

}

}

}

} else if ((direction == AudioManager.ADJUST\_RAISE) &&

!checkSafeMediaVolume(streamTypeAlias, aliasIndex + step, device)) {

// lyh 安全音量提示

mVolumeController.postDisplaySafeVolumeWarning(flags);

} else if (!isFullVolumeDevice(device)

&& (streamState.adjustIndex(direction \* step, device, caller,

hasModifyAudioSettings)

|| streamState.mIsMuted)) {

if (streamState.mIsMuted) {

if (direction == AudioManager.ADJUST\_RAISE) {

streamState.mute(false);

} else if (direction == AudioManager.ADJUST\_LOWER) {

if (mIsSingleVolume) {

sendMsg(mAudioHandler, MSG\_UNMUTE\_STREAM, SENDMSG\_QUEUE,

streamTypeAlias, flags, null, UNMUTE\_STREAM\_DELAY);

}

}

}

// lyh 发送Handler消息

sendMsg(mAudioHandler,

MSG\_SET\_DEVICE\_VOLUME,

SENDMSG\_QUEUE,

device,

0,

streamState,

0);

}

}

// lyh 调整后的音量新下标

final int newIndex = mStreamStates[streamType].getIndex(device);

// lyh HDMI 相关

if (adjustVolume) {

synchronized (mHdmiClientLock) {

if (mHdmiManager != null) {

...

}

}

}

// lyh 更新

sendVolumeUpdate(streamType, oldIndex, newIndex, flags, device);

}

复制代码

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_SET\_DEVICE\_VOLUME:

setDeviceVolume((VolumeStreamState) msg.obj, msg.arg1);

break;

}

}

复制代码

void setDeviceVolume(VolumeStreamState streamState, int device) {

synchronized (VolumeStreamState.class) {

// Apply volume

// lyh 设置音量

streamState.applyDeviceVolume\_syncVSS(device);

// Apply change to all streams using this one as alias

// 修改同一组下的流音量

int numStreamTypes = AudioSystem.getNumStreamTypes();

for (int streamType = numStreamTypes - 1; streamType >= 0; streamType--) {

if (streamType != streamState.mStreamType &&

mStreamVolumeAlias[streamType] == streamState.mStreamType) {

// Make sure volume is also maxed out on A2DP device for aliased stream

// that may have a different device selected

int streamDevice = getDeviceForStream(streamType);

if ((device != streamDevice) && mAvrcpAbsVolSupported

&& AudioSystem.DEVICE\_OUT\_ALL\_A2DP\_SET.contains(device)) {

mStreamStates[streamType].applyDeviceVolume\_syncVSS(device);

}

mStreamStates[streamType].applyDeviceVolume\_syncVSS(streamDevice);

}

}

}

// Post a persist volume msg

sendMsg(mAudioHandler,

MSG\_PERSIST\_VOLUME,

SENDMSG\_QUEUE,

device,

0,

streamState,

PERSIST\_DELAY);

}

复制代码

void applyDeviceVolume\_syncVSS(int device) {

int index;

if (isFullyMuted()) {

index = 0;

} else if (AudioSystem.DEVICE\_OUT\_ALL\_A2DP\_SET.contains(device)

&& mAvrcpAbsVolSupported) {

index = getAbsoluteVolumeIndex((getIndex(device) + 5)/10);

} else if (isFullVolumeDevice(device)) {

index = (mIndexMax + 5)/10;

} else if (device == AudioSystem.DEVICE\_OUT\_HEARING\_AID) {

index = (mIndexMax + 5)/10;

} else {

index = (getIndex(device) + 5)/10;

}

setStreamVolumeIndex(index, device);

}

复制代码

private void setStreamVolumeIndex(int index, int device) {

// Only set audio policy BT SCO stream volume to 0 when the stream is actually muted.

// This allows RX path muting by the audio HAL only when explicitly muted but not when

// index is just set to 0 to repect BT requirements

if (mStreamType == AudioSystem.STREAM\_BLUETOOTH\_SCO && index == 0

&& !isFullyMuted()) {

index = 1;

}

int streamType = mStreamType;

if (mStreamType >= AudioSystem.STREAM\_VOICE\_CALL\_GS\_MIN && mStreamType <= AudioSystem.STREAM\_VOICE\_CALL\_GS\_MAX) {

streamType = AudioSystem.STREAM\_VOICE\_CALL;

}

if (DEBUG\_VOL) Log.i(TAG, "setStreamVolumeIndex mStreamType " + mStreamType + ", streamType " + streamType + ", index " + index);

AudioSystem.setStreamVolumeIndexAS(streamType, index, device);

}

复制代码

## 滑动条调节

private void setStreamVolume() {

ensureValidStreamType(streamType);

int streamTypeAlias = mStreamVolumeAlias[streamType];

VolumeStreamState streamState = mStreamStates[streamTypeAlias];

// lyh 拿device

final int device = getDeviceForStream(streamType);

int oldIndex;

synchronized (mSafeMediaVolumeStateLock) {

// reset any pending volume command

mPendingVolumeCommand = null;

oldIndex = streamState.getIndex(device);

index = rescaleIndex(index \* 10, streamType, streamTypeAlias);

if (streamTypeAlias == AudioSystem.STREAM\_MUSIC) {

setSystemAudioVolume(oldIndex, index, getStreamMaxVolume(streamType), flags);

}

// lyh 检查是否安全音量

if (!checkSafeMediaVolume(streamTypeAlias, index, device)) {

mVolumeController.postDisplaySafeVolumeWarning(flags);

mPendingVolumeCommand = new StreamVolumeCommand(

streamType, index, flags, device);

} else {

// lyh 调节音量

onSetStreamVolume(streamType, index, flags, device, caller, hasModifyAudioSettings);

index = mStreamStates[streamType].getIndex(device);

}

}

// lyh 音量更新

sendVolumeUpdate(streamType, oldIndex, index, flags, device);

}

复制代码

private void onSetStreamVolume() {

final int stream = mStreamVolumeAlias[streamType];

// lyh

setStreamVolumeInt(stream, index, device, false, caller, hasModifyAudioSettings);

// setting volume on ui sounds stream type also controls silent mode

if (((flags & AudioManager.FLAG\_ALLOW\_RINGER\_MODES) != 0) ||

(stream == getUiSoundsStreamType())) {

setRingerMode(getNewRingerMode(stream, index, flags),

TAG + ".onSetStreamVolume", false /\*external\*/);

}

}

复制代码

private void setStreamVolumeInt() {

VolumeStreamState streamState = mStreamStates[streamType];

// 发送Handler消息,与按键调节流程一致了

sendMsg(mAudioHandler,

MSG\_SET\_DEVICE\_VOLUME,

SENDMSG\_QUEUE,

device,

0,

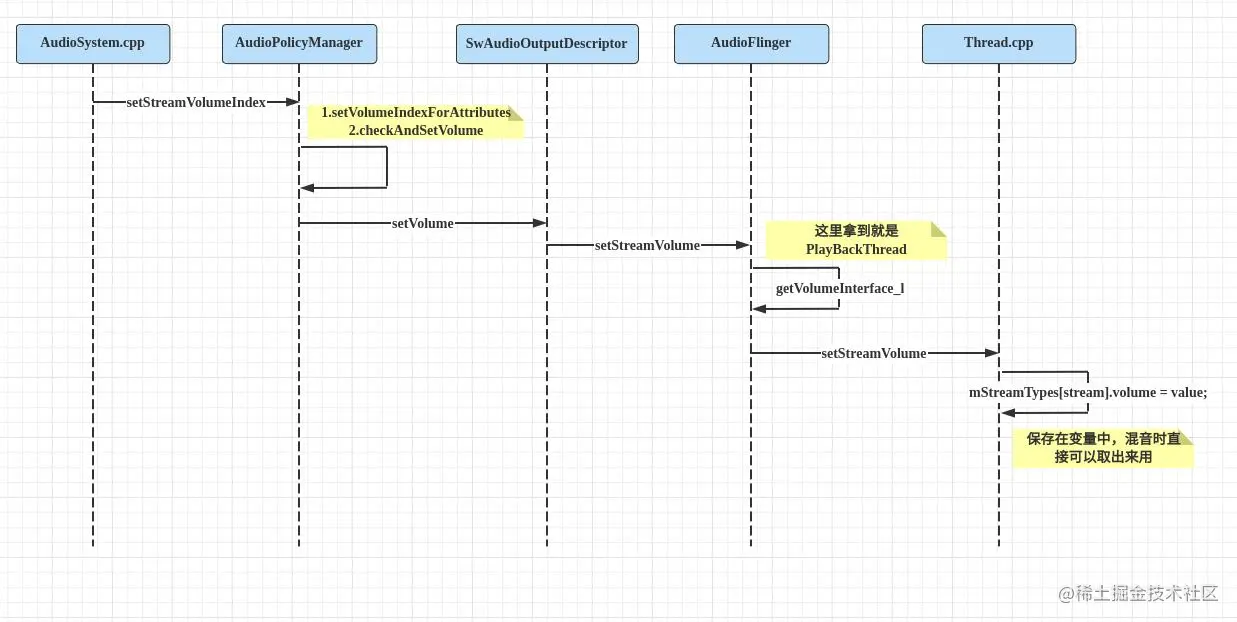
streamState,

0);

}

复制代码

# 音量调节Native层



static jintandroid\_media\_AudioSystem\_setStreamVolumeIndex()

{

return (jint) check\_AudioSystem\_Command(

AudioSystem::setStreamVolumeIndex()

）;

}

复制代码

status\_t AudioSystem::setStreamVolumeIndex()

{

const sp<IAudioPolicyService>& aps = AudioSystem::get\_audio\_policy\_service();

return aps->setStreamVolumeIndex(stream, index, device);

}

复制代码

status\_t AudioPolicyManager::setStreamVolumeIndex(){

return setVolumeIndexForAttributes(attributes, index, device);

}

复制代码

status\_t AudioPolicyManager::setVolumeIndexForAttributes()

{

// lyh 遍历所有设备

for (size\_t i = 0; i < mOutputs.size(); i++) {

// lyh 设备描述符

sp<SwAudioOutputDescriptor> desc = mOutputs.valueAt(i);

DeviceTypeSet curDevices = desc->devices().types();

// lyh 检查并设置音量

status\_t volStatus = checkAndSetVolume(

curves, vs, index, desc, curDevices,

((vs == toVolumeSource(AUDIO\_STREAM\_SYSTEM))?

TOUCH\_SOUND\_FIXED\_DELAY\_MS : 0));

}

mpClientInterface->onAudioVolumeGroupChanged(group, 0 /\*flags\*/);

return status;

}

复制代码

status\_t AudioPolicyManager::checkAndSetVolume()

{

// lyh 计算音量

float volumeDb = computeVolume(curves, volumeSource, index, deviceTypes);

// lyh 设置音量 SwAudioOutputDescriptor::setVolume

outputDesc->setVolume(

volumeDb, volumeSource, curves.getStreamTypes(), deviceTypes, delayMs, force);

return NO\_ERROR;

}

复制代码

bool SwAudioOutputDescriptor::setVolume(){

for (const auto& devicePort : devices()) {

// lyh 遍历设置音量

for (const auto &stream : streams) {

// lyh mClientInterface == AudioPolicyService

mClientInterface->setStreamVolume(stream, volumeAmpl, mIoHandle, delayMs);

}

}

}

return true;

}

复制代码

int AudioPolicyService::setStreamVolume(){

return (int)mAudioCommandThread->volumeCommand(stream, volume,

output, delayMs);

}

复制代码

status\_t AudioPolicyService::AudioCommandThread::volumeCommand()

{

sp<AudioCommand> command = new AudioCommand();

command->mCommand = SET\_VOLUME;

sp<VolumeData> data = new VolumeData();

data->mStream = stream;

data->mVolume = volume;

data->mIO = output;

command->mParam = data;

command->mWaitStatus = true;

// lyh 发送指令[SET\_VOLUME]去调节音量

return sendCommand(command, delayMs);

}

复制代码

status\_t AudioPolicyService::AudioCommandThread::sendCommand()

{

{

Mutex::Autolock \_l(mLock);

insertCommand\_l(command, delayMs);

// lyh 唤醒AudioCommandThread

mWaitWorkCV.signal();

}

Mutex::Autolock \_l(command->mLock);

while (command->mWaitStatus) {

nsecs\_t timeOutNs = kAudioCommandTimeoutNs + milliseconds(delayMs);

if (command->mCond.waitRelative(command->mLock, timeOutNs) != NO\_ERROR) {

command->mStatus = TIMED\_OUT;

command->mWaitStatus = false;

}

}

return command->mStatus;

}

复制代码

bool AudioPolicyService::AudioCommandThread::threadLoop()

{

switch (command->mCommand) {

case SET\_VOLUME: {

// lyh 又回到AudioSystem跨进程调用AudioFlinger

command->mStatus = AudioSystem::setStreamVolume(data->mStream,

data->mVolume,

data->mIO);

mLock.lock();

}

break;

}

}

复制代码

status\_t AudioFlinger::setStreamVolume(audio\_stream\_type\_t stream, float value,

audio\_io\_handle\_t output)

{

// lyh 通过继承关系知道【VolumeInterface = PlaybackThread】

VolumeInterface \*volumeInterface = getVolumeInterface\_l(output);

if (volumeInterface == NULL) {

return BAD\_VALUE;

}

volumeInterface->setStreamVolume(stream, value);

return NO\_ERROR;

}

复制代码

void AudioFlinger::PlaybackThread::setStreamVolume(audio\_stream\_type\_t stream, float value)

{

Mutex::Autolock \_l(mLock);

// 赋值

mStreamTypes[stream].volume = value;

// lyh 通知，PlaybackThread线程(例MixerThread)

broadcast\_l();

}

作者：liyihuanx  
链接：https://juejin.cn/post/7111975599430139917  
来源：稀土掘金  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

**Android Framework 音频子系统（13）音量调节之基础**

**Android Framework 音频子系统（13）音量调节之基础**

**本章关键点总结 & 说明：**

**本章节主要关注➕ 以上思维导图左上 音量调节 部分 即可。说明了音量的基础知识和AudioFlinger调节音量流程，主要包括：**

**AudioFlinger对master volume, stream volume的初始化设置流程**

**AudioFlinger的setMasterVolume 主音量设置流程**

**AudioFlinger 的setStreamVolume 流音量设置流程**

**播放线程 加载音量设置 流程**

**1 音量基础知识**

**@1 四大类Volume音量**

**master volume：设置它等于设置所有的stream volume和track volume。它可以写到声卡里面去，控制所有声音的音量。也可以不写到声卡里面去，而是作为一个乘数因子来影响所有的音量。换句话说：master volume 可以设置所有的AudioTrack volume和stream volume。**

**stream volume：设置某一stream的音量，Android系统中支持10种stream。各种stream的音量也可以单独设置、互不影响。比如"音乐音量"不应该影响到"来电振铃"、"闹钟"、"通话"的音量。**

**stream volume alias：设置的是同一组stream音量，分组在Android源码中称之为"别名"，即alias。比如在电话中，5种stream（STREAM\_SYSTEM、STREAM\_RING、STREAM\_NOTIFICATION、STREAM\_SYSTEM\_ENFORCED、STREAM\_DTMF）的alias都是STREAM\_RING，那么对应的滑动条即可控制这5种stream的音量。**

**AudioTrack  volume： 单个App设置音量时设置的是这个，它只影响本App的音量。**

**@2 十种stream**

**Android系统中有10种stream，在system/core/include/system/audio.h中定义，但把这10种stream分成组，属于同一组的stream具有相同的别名(alias)。在我们设置音量时，一个音量调节滑动条具有一个alias，具有相同alias的stream都会受到这个滑动条的影响。stream与alias的关系（参考）如下所示：**

**@3 声音播放的两种路径**

**MixerThread：APP对音量的设置不会影响到声卡的硬件音量,而只会影响APP的音频数据的幅值(变小或放大),这些音频数据最终被混合后传给声卡。**

**DirectOutputThread(比如HDMI，单个音频应用程序单独使用一个声卡)：同一时间里只有一个APP、只有一个AudioTrack使用它，所以该AudioTrack的音量可以被DirectOutputThread直接用来设置硬件音量，这种声卡使用的不多。若配置文件中参数信息包含"flags AUDIO\_OUTPUT\_FLAG\_DIRECT"，则表示这个声卡可以被某个App独占。App就能以DirectOutputThread的形式来使用这个声卡。**

**@4 混音的逻辑**

**app1：混音数据1 =  音频数据1 \* master\_volume \* stream1\_volume \* AudioTrack1\_volume**

**app2：混音数据2 = 音频数据2 \* master\_volume \* stream2\_volume \* AudioTrack2\_volume**

**app3：混音数据3 = 音频数据3 \* master\_volume \* stream3\_volume \* AudioTrack3\_volume**

**混合在一起: 最终混音 =混音数据1+混音数据2+混音数据3，然后把混合后的数据写给硬件。**

**@5 音频系统中的一些关键变量说明：**

**AudioFlinger类中有关成员：**

**stream\_type\_t mStreamTypes[AUDIO\_STREAM\_CNT];**

**float mMasterVolume; //存储master volume**

**bool mMasterMute; //存储是否静音**

**playbackThread类中：**

**//为DuplicatingThread的OutputTrack多出一项， DuplicatingThread可以用于在两个声卡上播放出同样的声音**

**stream\_type\_t mStreamTypes[AUDIO\_STREAM\_CNT + 1];**

**bool mMasterMute;**

**float mMasterVolume; //来源于AudioFlinger中的同名的变量**

**AudioTrack类中(App端)：**

**float mVolume[2]; //两项，分别表示App设置的左右声道的音量**

**1.**

**说明：stream volume和audioTreack中的volume只是软件上的处理，masterVolue中保存的值若HAL提供了相应的写函数就会写给硬件。**

**2 AudioFlinger调节音量流程**

**2.1 AudioFlinger音量设置流程说明**

**音量设置是通过逻辑运算将音量值存放在变量中，之后再播放中重新进行一轮逻辑运算，最终和声音数据一起写入到声卡中，进而播放出合理的声音。**

**@1 AudioFlinger对master volume, stream volume的初始化设置流程**

**最开始MasterVolume,、MasterMute、StreamVolume、StreamMute的初始化是在AudioFlinger对象创建时初始化的，MasterVolume,、MasterMute是在构造器中直接初始化，代码如下：**

**AudioFlinger::AudioFlinger()**

**: BnAudioFlinger(),**

**mPrimaryHardwareDev(NULL),**

**mAudioHwDevs(NULL),**

**mHardwareStatus(AUDIO\_HW\_IDLE),**

**mMasterVolume(1.0f),//初值1.0f**

**mMasterMute(false),//静音初值**

**mNextUniqueId(1),**

**mMode(AUDIO\_MODE\_INVALID),**

**mBtNrecIsOff(false),**

**mIsLowRamDevice(true),**

**mIsDeviceTypeKnown(false),**

**mGlobalEffectEnableTime(0),**

**mPrimaryOutputSampleRate(0)**

**{**

**//...**

**}**

**而StreamVolume、StreamMute的初始化则是在成员变量结构体中初始化的，代码如下：**

**//在创建结构体的时候 直接初始化**

**struct stream\_type\_t {**

**stream\_type\_t()**

**: volume(1.0f),**

**mute(false)**

**{**

**}**

**float volume;**

**bool mute;**

**};**

**这一阶段初始化后，在加载音频库的时候执行loadHwModule，代码如下：**

**audio\_module\_handle\_t AudioFlinger::loadHwModule(const char \*name)**

**{**

**if (name == NULL) {**

**return 0;**

**}**

**if (!settingsAllowed()) {**

**return 0;**

**}**

**Mutex::Autolock \_l(mLock);**

**return loadHwModule\_l(name);**

**}**

**继续分析loadHwModule\_l，代码实现如下：**

**// loadHwModule\_l() must be called with AudioFlinger::mLock held**

**audio\_module\_handle\_t AudioFlinger::loadHwModule\_l(const char \*name)**

**{**

**for (size\_t i = 0; i < mAudioHwDevs.size(); i++) {**

**if (strncmp(mAudioHwDevs.valueAt(i)->moduleName(), name, strlen(name)) == 0) {**

**ALOGW("loadHwModule() module %s already loaded", name);**

**return mAudioHwDevs.keyAt(i);**

**}**

**}**

**audio\_hw\_device\_t \*dev;**

**//获取audio\_hw\_device\_t类型设备dev，可以直接操作HAL层**

**int rc = load\_audio\_interface(name, &dev);**

**mHardwareStatus = AUDIO\_HW\_INIT;**

**rc = dev->init\_check(dev);**

**mHardwareStatus = AUDIO\_HW\_IDLE;**

**AudioHwDevice::Flags flags = static\_cast<AudioHwDevice::Flags>(0);**

**{ // scope for auto-lock pattern**

**AutoMutex lock(mHardwareLock);**

**if (0 == mAudioHwDevs.size()) {**

**mHardwareStatus = AUDIO\_HW\_GET\_MASTER\_VOLUME;**

**//只要dev中含有get\_master\_volume，表明可以从库中获取master\_volume的初值**

**if (NULL != dev->get\_master\_volume) {**

**float mv;**

**if (OK == dev->get\_master\_volume(dev, &mv)) {**

**mMasterVolume = mv;**

**}**

**}**

**mHardwareStatus = AUDIO\_HW\_GET\_MASTER\_MUTE;**

**//只要dev中含有get\_master\_mute，表明可以从库中获取master\_mute的初值**

**if (NULL != dev->get\_master\_mute) {**

**bool mm;**

**if (OK == dev->get\_master\_mute(dev, &mm)) {**

**mMasterMute = mm;**

**}**

**}**

**}**

**mHardwareStatus = AUDIO\_HW\_SET\_MASTER\_VOLUME;**

**//设置master\_volume的初值 到硬件中**

**if ((NULL != dev->set\_master\_volume) &&**

**(OK == dev->set\_master\_volume(dev, mMasterVolume))) {**

**flags = static\_cast<AudioHwDevice::Flags>(flags |**

**AudioHwDevice::AHWD\_CAN\_SET\_MASTER\_VOLUME);**

**}**

**mHardwareStatus = AUDIO\_HW\_SET\_MASTER\_MUTE;**

**//设置master\_mute的初值 到硬件中**

**if ((NULL != dev->set\_master\_mute) &&**

**(OK == dev->set\_master\_mute(dev, mMasterMute))) {**

**flags = static\_cast<AudioHwDevice::Flags>(flags |**

**AudioHwDevice::AHWD\_CAN\_SET\_MASTER\_MUTE);**

**}**

**mHardwareStatus = AUDIO\_HW\_IDLE;**

**}**

**audio\_module\_handle\_t handle = nextUniqueId();**

**mAudioHwDevs.add(handle, new AudioHwDevice(handle, name, dev, flags));**

**return handle;**

**}**

**这里对MasterVolume,、MasterMute进行二次初始化，即如果音频库是支持初值设置的，则以音频库中的值为主，否则就是AudioFlinger创建时的初始值。**

**@2 AudioFlinger::setMasterVolume 主音量设置流程**

**AudioFlinger::setMasterVolume的代码实现如下：**

**status\_t AudioFlinger::setMasterVolume(float value)**

**{**

**status\_t ret = initCheck();**

**//...**

**Mutex::Autolock \_l(mLock);**

**mMasterVolume = value;**

**for (size\_t i = 0; i < mAudioHwDevs.size(); i++) {**

**AutoMutex lock(mHardwareLock);**

**AudioHwDevice \*dev = mAudioHwDevs.valueAt(i);**

**mHardwareStatus = AUDIO\_HW\_SET\_MASTER\_VOLUME;**

**//直接将master\_volume的值设置到硬件中**

**if (dev->canSetMasterVolume()) {**

**dev->hwDevice()->set\_master\_volume(dev->hwDevice(), value);**

**}**

**mHardwareStatus = AUDIO\_HW\_IDLE;**

**}**

**//将master\_volume的值设置到各个播放线程中**

**for (size\_t i = 0; i < mPlaybackThreads.size(); i++)**

**mPlaybackThreads.valueAt(i)->setMasterVolume(value);**

**return NO\_ERROR;**

**}**

**这里PlaybackThread::setMasterVolume的代码实现如下：**

**void AudioFlinger::PlaybackThread::setMasterVolume(float value)**

**{**

**Mutex::Autolock \_l(mLock);**

**// Don't apply master volume in SW if our HAL can do it for us.**

**if (mOutput && mOutput->audioHwDev &&**

**mOutput->audioHwDev->canSetMasterVolume()) {**

**mMasterVolume = 1.0;**

**} else {**

**mMasterVolume = value;**

**}**

**}**

**可以看到都是直接操作HAL层的接口进行参数设置。**

**@3 AudioFlinger::setStreamVolume 流音量设置流程**

**AudioFlinger::setStreamVolume的 代码实现如下：**

**status\_t AudioFlinger::setStreamVolume(audio\_stream\_type\_t stream, float value,**

**audio\_io\_handle\_t output)**

**{**

**status\_t status = checkStreamType(stream);**

**AutoMutex lock(mLock);**

**PlaybackThread \*thread = NULL;**

**if (output != AUDIO\_IO\_HANDLE\_NONE) {**

**thread = checkPlaybackThread\_l(output);**

**if (thread == NULL) {**

**return BAD\_VALUE;**

**}**

**}**

**mStreamTypes[stream].volume = value;**

**if (thread == NULL) {**

**//未指定线程则全部播放线程 均设置**

**for (size\_t i = 0; i < mPlaybackThreads.size(); i++) {**

**mPlaybackThreads.valueAt(i)->setStreamVolume(stream, value);**

**}**

**} else {**

**//指定线程则直接设置**

**thread->setStreamVolume(stream, value);**

**}**

**return NO\_ERROR;**

**}**

**继续分析播放线程的setStreamVolume方法，代码实现如下：**

**void AudioFlinger::PlaybackThread::setStreamVolume(audio\_stream\_type\_t stream, float value)**

**{**

**Mutex::Autolock \_l(mLock);**

**mStreamTypes[stream].volume = value;//赋值**

**broadcast\_l();**

**}**

**实际上 每个播放线程中都有 mStreamTypes[stream].volume，和 AudioFlinger的mStreamTypes[stream].volume是一致的。**

**@4 AudioTrack volume的设置**

**AudioTrack::setVolume的代码实现如下：**

**status\_t AudioTrack::setVolume(float volume)**

**{**

**return setVolume(volume, volume);**

**}**

**status\_t AudioTrack::setVolume(float left, float right)**

**{**

**//...**

**AutoMutex lock(mLock);**

**mVolume[AUDIO\_INTERLEAVE\_LEFT] = left;**

**mVolume[AUDIO\_INTERLEAVE\_RIGHT] = right;**

**//这里会通过ClientProxy将音量参数设置到共享内存中**

**//这里的mProxy =**

**//new AudioTrackClientProxy(cblk, buffers, frameCount, mFrameSizeAF);**

**mProxy->setVolumeLR(gain\_minifloat\_pack(gain\_from\_float(left), gain\_from\_float(right)));**

**if (isOffloaded\_l()) {**

**mAudioTrack->signal();**

**}**

**return NO\_ERROR;**

**}**

**这里是把这个数据记录在mVolumeLR域中，创建Proxy时传递的Cblk参数就是共享内存的头部。**

**2.2 播放线程 加载音量设置 流程**

**@1 源码流程分析说明**

**这里分析时主要针对音量部分相关代码进行分析，代码实现如下：**

**// prepareTracks\_l() must be called with ThreadBase::mLock held**

**AudioFlinger::PlaybackThread::mixer\_state AudioFlinger::MixerThread::prepareTracks\_l(**

**Vector< sp<Track> > \*tracksToRemove)**

**{**

**//...**

**float masterVolume = mMasterVolume;**

**bool masterMute = mMasterMute;**

**if (masterMute) {//如果静音条件为真，则设置masterVolume=0**

**masterVolume = 0;**

**}**

**//...**

**mMixerBufferValid = false; // mMixerBuffer has no valid data until appropriate tracks found.**

**mEffectBufferValid = false; // mEffectBuffer has no valid data until tracks found.**

**for (size\_t i=0 ; i<count ; i++) {**

**const sp<Track> t = mActiveTracks[i].promote();**

**if (t == 0) {**

**continue;**

**}**

**// this const just means the local variable doesn't change**

**Track\* const track = t.get();**

**//...**

**{ // local variable scope to avoid goto warning**

**audio\_track\_cblk\_t\* cblk = track->cblk();**

**int name = track->name();**

**size\_t desiredFrames;**

**uint32\_t sr = track->sampleRate();**

**if (sr == mSampleRate) {**

**desiredFrames = mNormalFrameCount;**

**} else {**

**// +1 for rounding and +1 for additional sample needed for interpolation**

**desiredFrames = (mNormalFrameCount \* sr) / mSampleRate + 1 + 1;**

**desiredFrames += mAudioMixer->getUnreleasedFrames(track->name());**

**}**

**uint32\_t minFrames = 1;**

**if ((track->sharedBuffer() == 0) && !track->isStopped() && !track->isPausing() &&**

**(mMixerStatusIgnoringFastTracks == MIXER\_TRACKS\_READY)) {**

**minFrames = desiredFrames;**

**}**

**size\_t framesReady = track->framesReady();**

**if ((framesReady >= minFrames) && track->isReady() &&**

**!track->isPaused() && !track->isTerminated())**

**{**

**mixedTracks++;**

**//...**

**// compute volume for this track**

**uint32\_t vl, vr; // in U8.24 integer format**

**float vlf, vrf, vaf; // in [0.0, 1.0] float format**

**if (track->isPausing() || mStreamTypes[track->streamType()].mute) {**

**vl = vr = 0;**

**vlf = vrf = vaf = 0.;**

**if (track->isPausing()) {**

**track->setPaused();**

**}**

**} else {**

**// read original volumes with volume control**

**//获取 StreamType Volume**

**float typeVolume = mStreamTypes[track->streamType()].volume;**

**float v = masterVolume \* typeVolume;**

**//获取共享内存代理**

**AudioTrackServerProxy \*proxy = track->mAudioTrackServerProxy;**

**//从共享内存中获得左右声道**

**gain\_minifloat\_packed\_t vlr = proxy->getVolumeLR();**

**vlf = float\_from\_gain(gain\_minifloat\_unpack\_left(vlr));**

**vrf = float\_from\_gain(gain\_minifloat\_unpack\_right(vlr));**

**// track volumes come from shared memory, so can't be trusted and must be clamped**

**//边界判断**

**if (vlf > GAIN\_FLOAT\_UNITY) {**

**ALOGV("Track left volume out of range: %.3g", vlf);**

**vlf = GAIN\_FLOAT\_UNITY;**

**}**

**if (vrf > GAIN\_FLOAT\_UNITY) {**

**ALOGV("Track right volume out of range: %.3g", vrf);**

**vrf = GAIN\_FLOAT\_UNITY;**

**}**

**// now apply the master volume and stream type volume**

**//放大系数：master\_volume \* stream\_volume \* AudioTrack\_volume**

**vlf \*= v;**

**vrf \*= v;**

**// assuming master volume and stream type volume each go up to 1.0,**

**// then derive vl and vr as U8.24 versions for the effect chain**

**//下面主要是左右声道转换成AUX单声道的一些逻辑运算**

**const float scaleto8\_24 = MAX\_GAIN\_INT \* MAX\_GAIN\_INT;**

**vl = (uint32\_t) (scaleto8\_24 \* vlf);**

**vr = (uint32\_t) (scaleto8\_24 \* vrf);**

**// vl and vr are now in U8.24 format**

**uint16\_t sendLevel = proxy->getSendLevel\_U4\_12();**

**// send level comes from shared memory and so may be corrupt**

**if (sendLevel > MAX\_GAIN\_INT) {**

**ALOGV("Track send level out of range: %04X", sendLevel);**

**sendLevel = MAX\_GAIN\_INT;**

**}**

**// vaf is represented as [0.0, 1.0] float by rescaling sendLevel**

**vaf = v \* sendLevel \* (1. / MAX\_GAIN\_INT);**

**}**

**// Delegate volume control to effect in track effect chain if needed**

**if (chain != 0 && chain->setVolume\_l(&vl, &vr)) {**

**// Do not ramp volume if volume is controlled by effect**

**param = AudioMixer::VOLUME;**

**// Update remaining floating point volume levels**

**vlf = (float)vl / (1 << 24);**

**vrf = (float)vr / (1 << 24);**

**track->mHasVolumeController = true;**

**} else {**

**// force no volume ramp when volume controller was just disabled or removed**

**// from effect chain to avoid volume spike**

**if (track->mHasVolumeController) {**

**param = AudioMixer::VOLUME;**

**}**

**track->mHasVolumeController = false;**

**}**

**// XXX: these things DON'T need to be done each time**

**mAudioMixer->setBufferProvider(name, track);**

**mAudioMixer->enable(name);**

**//关键点：通过参数设置将音量信息传递出去**

**mAudioMixer->setParameter(name, param, AudioMixer::VOLUME0, &vlf);**

**mAudioMixer->setParameter(name, param, AudioMixer::VOLUME1, &vrf);**

**mAudioMixer->setParameter(name, param, AudioMixer::AUXLEVEL, &vaf);**

**//设置其他参数**

**//...**

**// reset retry count**

**track->mRetryCount = kMaxTrackRetries;**

**if (mMixerStatusIgnoringFastTracks != MIXER\_TRACKS\_READY ||**

**mixerStatus != MIXER\_TRACKS\_ENABLED) {**

**mixerStatus = MIXER\_TRACKS\_READY;**

**}**

**} else {**

**//...**

**}**

**} // local variable scope to avoid goto warning**

**track\_is\_ready: ;**

**}**

**//...**

**return mixerStatus;**

**}**

**这里专注分析AudioMixer的 参数设置setParameter方法，代码实现如下：**

**void AudioMixer::setParameter(int name, int target, int param, void \*value)**

**{**

**name -= TRACK0;**

**track\_t& track = mState.tracks[name];**

**int valueInt = static\_cast<int>(reinterpret\_cast<uintptr\_t>(value));**

**int32\_t \*valueBuf = reinterpret\_cast<int32\_t\*>(value);**

**switch (target) {**

**//...**

**case RAMP\_VOLUME:**

**case VOLUME:**

**switch (param) {**

**case AUXLEVEL:**

**if (setVolumeRampVariables(\*reinterpret\_cast<float\*>(value),**

**target == RAMP\_VOLUME ? mState.frameCount : 0,**

**&track.auxLevel, &track.prevAuxLevel, &track.auxInc,**

**&track.mAuxLevel, &track.mPrevAuxLevel, &track.mAuxInc)) {**

**invalidateState(1 << name);**

**}**

**break;**

**default:**

**if ((unsigned)param >= VOLUME0 && (unsigned)param < VOLUME0 + MAX\_NUM\_VOLUMES) {**

**//setVolumeRampVariables主要是 float和int类型之间的转换的一些逻辑操作**

**if (setVolumeRampVariables(\*reinterpret\_cast<float\*>(value),**

**target == RAMP\_VOLUME ? mState.frameCount : 0,**

**&track.volume[param - VOLUME0], &track.prevVolume[param - VOLUME0],**

**&track.volumeInc[param - VOLUME0],**

**&track.mVolume[param - VOLUME0], &track.mPrevVolume[param - VOLUME0],**

**&track.mVolumeInc[param - VOLUME0])) {**

**invalidateState(1 << name);**

**}**

**} else {**

**LOG\_ALWAYS\_FATAL("setParameter volume: bad param %d", param);**

**}**

**}**

**break;**

**default:**

**LOG\_ALWAYS\_FATAL("setParameter: bad target %d", target);**

**}**

**}**

**这里专注 invalidateState的实现，代码如下：**

**void AudioMixer::invalidateState(uint32\_t mask)**

**{**

**if (mask != 0) {**

**mState.needsChanged |= mask;**

**mState.hook = process\_\_validate;**

**}**

**}**

**process\_\_validate的实现如下：**

**void AudioMixer::process\_\_validate(state\_t\* state, int64\_t pts)**

**{**

**//...**

**// compute everything we need...**

**while (en) {**

**//...**

**if (n & NEEDS\_MUTE) {**

**t.hook = track\_\_nop;**

**} else {**

**if (n & NEEDS\_AUX) {**

**all16BitsStereoNoResample = false;**

**}**

**if (n & NEEDS\_RESAMPLE) {**

**all16BitsStereoNoResample = false;**

**resampling = true;**

**t.hook = getTrackHook(TRACKTYPE\_RESAMPLE, t.mMixerChannelCount,**

**t.mMixerInFormat, t.mMixerFormat);**

**} else {**

**if ((n & NEEDS\_CHANNEL\_COUNT\_\_MASK) == NEEDS\_CHANNEL\_1){**

**t.hook = getTrackHook(**

**t.mMixerChannelCount == 2 // TODO: MONO\_HACK.**

**? TRACKTYPE\_NORESAMPLEMONO : TRACKTYPE\_NORESAMPLE,**

**t.mMixerChannelCount,**

**t.mMixerInFormat, t.mMixerFormat);**

**all16BitsStereoNoResample = false;**

**}**

**if ((n & NEEDS\_CHANNEL\_COUNT\_\_MASK) >= NEEDS\_CHANNEL\_2){**

**t.hook = getTrackHook(TRACKTYPE\_NORESAMPLE, t.mMixerChannelCount,**

**t.mMixerInFormat, t.mMixerFormat);**

**}**

**}**

**}**

**}**

**//...**

**}**

**这里主要关注getTrackHook函数，代码实现如下：**

**AudioMixer::hook\_t AudioMixer::getTrackHook(int trackType, uint32\_t channelCount,**

**audio\_format\_t mixerInFormat, audio\_format\_t mixerOutFormat \_\_unused)**

**{**

**if (!kUseNewMixer && channelCount == FCC\_2 && mixerInFormat == AUDIO\_FORMAT\_PCM\_16\_BIT) {**

**switch (trackType) {**

**//...**

**case TRACKTYPE\_NORESAMPLE:**

**return track\_\_16BitsStereo;**

**default:**

**break;**

**}**

**}**

**//...**

**return NULL;**

**}**

**这里以关注TRACKTYPE\_NORESAMPLE为例，最终会调用到track\_\_16BitsStereo，代码实现如下：**

**void AudioMixer::track\_\_16BitsStereo(track\_t\* t, int32\_t\* out, size\_t frameCount,**

**int32\_t\* temp \_\_unused, int32\_t\* aux)**

**{**

**ALOGVV("track\_\_16BitsStereo\n");**

**const int16\_t \*in = static\_cast<const int16\_t \*>(t->in);**

**if (CC\_UNLIKELY(aux != NULL)) {**

**//忽略AUX相关处理**

**} else {**

**// ramp gain**

**if (CC\_UNLIKELY(t->volumeInc[0]|t->volumeInc[1])) {**

**int32\_t vl = t->prevVolume[0];**

**int32\_t vr = t->prevVolume[1];**

**const int32\_t vlInc = t->volumeInc[0];左声道音量**

**const int32\_t vrInc = t->volumeInc[1];右声道音量**

**do {**

**\*out++ += (vl >> 16) \* (int32\_t) \*in++;**

**\*out++ += (vr >> 16) \* (int32\_t) \*in++;**

**vl += vlInc;**

**vr += vrInc;**

**} while (--frameCount);**

**t->prevVolume[0] = vl;**

**t->prevVolume[1] = vr;**

**t->adjustVolumeRamp(false);**

**}**

**// constant gain**

**else {**

**const uint32\_t vrl = t->volumeRL;**

**do {**

**uint32\_t rl = \*reinterpret\_cast<const uint32\_t \*>(in);**

**in += 2;**

**out[0] = mulAddRL(1, rl, vrl, out[0]);**

**out[1] = mulAddRL(0, rl, vrl, out[1]);**

**out += 2;**

**} while (--frameCount);**

**}**

**}**

**t->in = in;**

**}**

**最终我们把数据存储到out中，这里的mulAddRL有三个，这里以下面的这个实现为例，代码如下：**

**//另外2种实现模式类似**

**static inline**

**int32\_t mulAddRL(int left, uint32\_t inRL, int32\_t v, int32\_t a)**

**{**

**#if USE\_INLINE\_ASSEMBLY**

**int32\_t out;**

**if (left) {**

**asm( "smlawb %[out], %[v], %[inRL], %[a] \n"**

**: [out]"=r"(out)**

**: [inRL]"%r"(inRL), [v]"r"(v), [a]"r"(a)**

**: );**

**} else {**

**asm( "smlawt %[out], %[v], %[inRL], %[a] \n"**

**: [out]"=r"(out)**

**: [inRL]"%r"(inRL), [v]"r"(v), [a]"r"(a)**

**: );**

**}**

**return out;**

**#else**

**int16\_t s = left ? int16\_t(inRL) : int16\_t(inRL>>16);**

**return a + int32\_t((int64\_t(v) \* s) >> 16);**

**#endif**

**}**

**虽然有可能会使用汇编语句来优化，但实际上逻辑是一致的（这里用outL表示左声道，outR表示右声道）：**

**左声道：outL = (inRL&0xffff \* v) + outL //前16位**

**右声道：outR = (inRL>>16    \* v) + outR //后16位**

**最后outL 和 outR 合并成一个值（低16bit是左声道数据，高16bit是右声道数据）并返回。这里实际上是属于播放音频中MixerThread::prepareTracks\_l中 tracks[x].hook中的一个操作，通过这操作有了prepareTrack\_l设置的参数，在threadLoop\_mix中进行混音。最后通过threadLoop\_write用于混音后的音频输出，最后将填充好的Buffer写入到硬件中。**

**@2 关于MixerThread::prepareTracks\_l涉及的track\_t结构体的说明**

**代码实现如下：**

**struct track\_t {**

**//...**

**// TODO: Eventually remove legacy integer volume settings**

**//int类型 普通声音**

**union {**

**int16\_t volume[MAX\_NUM\_VOLUMES]; // U4.12 fixed point (top bit should be zero)**

**int32\_t volumeRL;**

**};**

**int32\_t prevVolume[MAX\_NUM\_VOLUMES];**

**int32\_t volumeInc[MAX\_NUM\_VOLUMES];**

**//...**

**//int类型 aux声音**

**int32\_t auxInc;**

**int32\_t prevAuxLevel;**

**int16\_t auxLevel; // 0 <= auxLevel <= MAX\_GAIN\_INT, but signed for mul performance**

**//...**

**//float类型 普通声音**

**float mVolume[MAX\_NUM\_VOLUMES]; // floating point set volume**

**float mPrevVolume[MAX\_NUM\_VOLUMES]; // floating point previous volume**

**float mVolumeInc[MAX\_NUM\_VOLUMES]; // floating point volume increment**

**//...**

**//float类型 aux声音**

**float mAuxLevel; // floating point set aux level**

**float mPrevAuxLevel; // floating point prev aux level**

**float mAuxInc; // floating point aux increment**

**//...**

**};**

**这里aux的数据实际上就是 左右声道叠加在一起，通过特定处理后转换成 特定声道的方式。**

**这里我们可以发现，track\_t结构体中共有4组音量变量。都是PreVolume，VolumeInc，Volume这种模式，这三种变量的意义如下图所示：**

**解读如下：**

**PreVolume：之前的音量初始值**

**VolumeInc：表示每次调节的步长**

**Volume：当前的音量 = master\_volume \* stream\_volume \* AudioTrack\_volume**

**-----------------------------------**

**©著作权归作者所有：来自51CTO博客作者AGS\_wangdsh的原创作品，请联系作者获取转载授权，否则将追究法律责任**

**Android Framework 音频子系统（13）音量调节之基础**

**<https://blog.51cto.com/u_14344871/3369898>**

# **Android平台音量调节（二）Native的流程处理**

# Native的流程处理

前面只是说了AudioService中的逻辑，最终是通过AudioSystem.setStreamVolumeIndex(mStreamType, index, device);设置到native层的。那么我们就来看看native是怎么处理的。

## 音频device

setStreamVolumeIndex根据device和流类型来设置，换言之，每种设备的每种流类型的音量是分开的，可以不一样。比如蓝牙耳机的Music音量可以是15，而Speaker的音量可以是5。那么问题来了，上层根本没有 这些逻辑？这个是怎么实现的。我们先来看，怎么获取到的设备！

\* frameworks/base/services/core/java/com/android/server/audio/AudioService.java::VolumeStreamState

public int observeDevicesForStream\_syncVSS(boolean checkOthers) {

final int devices = AudioSystem.getDevicesForStream(mStreamType);

if (devices == mObservedDevices) {

return devices;

}

final int prevDevices = mObservedDevices;

mObservedDevices = devices;

if (checkOthers) {

// one stream's devices have changed, check the others

observeDevicesForStreams(mStreamType);

}

// log base stream changes to the event log

if (mStreamVolumeAlias[mStreamType] == mStreamType) {

EventLogTags.writeStreamDevicesChanged(mStreamType, prevDevices, devices);

}

sendBroadcastToAll(mStreamDevicesChanged

.putExtra(AudioManager.EXTRA\_PREV\_VOLUME\_STREAM\_DEVICES, prevDevices)

.putExtra(AudioManager.EXTRA\_VOLUME\_STREAM\_DEVICES, devices));

return devices;

}

device是通过AudioSystem.getDevicesForStream(mStreamType)获取到的，mStreamType是流类型，不是映射的别名。

native的getDevicesForStream函数如下：

audio\_devices\_t AudioSystem::getDevicesForStream(audio\_stream\_type\_t stream){

const sp<IAudioPolicyService>& aps = AudioSystem::get\_audio\_policy\_service();

if (aps == 0) return AUDIO\_DEVICE\_NONE;

return aps->getDevicesForStream(stream);}

最后都是通过AudioPolicy来决定的。

\* frameworks/av/services/audiopolicy/service/AudioPolicyInterfaceImpl.cpp

audio\_devices\_t AudioPolicyService::getDevicesForStream(audio\_stream\_type\_t stream){

if (uint32\_t(stream) >= AUDIO\_STREAM\_PUBLIC\_CNT) {

return AUDIO\_DEVICE\_NONE;

}

if (mAudioPolicyManager == NULL) {

return AUDIO\_DEVICE\_NONE;

}

Mutex::Autolock \_l(mLock);

return mAudioPolicyManager->getDevicesForStream(stream);}

mAudioPolicyManagers是AudioPolicy的核心，AudioPolicyService起来的时候就创建了mAudioPolicyManager以及相应的一些线程。

frameworks/av/services/audiopolicy/service/AudioPolicyService.cpp

void AudioPolicyService::onFirstRef(){

{

Mutex::Autolock \_l(mLock);

// start tone playback thread

mTonePlaybackThread = new AudioCommandThread(String8("ApmTone"), this);

// start audio commands thread

mAudioCommandThread = new AudioCommandThread(String8("ApmAudio"), this);

// start output activity command thread

mOutputCommandThread = new AudioCommandThread(String8("ApmOutput"), this);

mAudioPolicyClient = new AudioPolicyClient(this);

mAudioPolicyManager = createAudioPolicyManager(mAudioPolicyClient);

}

// load audio processing modules

sp<AudioPolicyEffects>audioPolicyEffects = new AudioPolicyEffects();

{

Mutex::Autolock \_l(mLock);

mAudioPolicyEffects = audioPolicyEffects;

}}

mAudioPolicyManager通过createAudioPolicyManager函数创建的，AOSP的createAudioPolicyManager如下：

\* frameworks/av/services/audiopolicy/manager/AudioPolicyFactory.cpp

extern "C" AudioPolicyInterface\* createAudioPolicyManager(

AudioPolicyClientInterface \*clientInterface){

return new AudioPolicyManager(clientInterface);}

Vendor也可以自己实现自己的AudioPolicyManager，比如高通的：

\* hardware/qcom/audio/policy\_hal/AudioPolicyManager.cpp

extern "C" AudioPolicyInterface\* createAudioPolicyManager(

AudioPolicyClientInterface \*clientInterface){

return new AudioPolicyManagerCustom(clientInterface);}

我们继续来看getDevicesForStream函数：

\* frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp

audio\_devices\_t AudioPolicyManager::getDevicesForStream(audio\_stream\_type\_t stream) {

// By checking the range of stream before calling getStrategy, we avoid

// getStrategy's behavior for invalid streams. getStrategy would do a ALOGE

// and then return STRATEGY\_MEDIA, but we want to return the empty set.

if (stream < (audio\_stream\_type\_t) 0 || stream >= AUDIO\_STREAM\_PUBLIC\_CNT) {

return AUDIO\_DEVICE\_NONE;

}

audio\_devices\_t devices = AUDIO\_DEVICE\_NONE;

for (int curStream = 0; curStream < AUDIO\_STREAM\_FOR\_POLICY\_CNT; curStream++) {

if (!streamsMatchForvolume(stream, (audio\_stream\_type\_t)curStream)) {

continue;

}

routing\_strategy curStrategy = getStrategy((audio\_stream\_type\_t)curStream);

audio\_devices\_t curDevices =

getDeviceForStrategy((routing\_strategy)curStrategy, false /\*fromCache\*/);

SortedVector<audio\_io\_handle\_t> outputs = getOutputsForDevice(curDevices, mOutputs);

for (size\_t i = 0; i < outputs.size(); i++) {

sp<AudioOutputDescriptor> outputDesc = mOutputs.valueFor(outputs[i]);

if (outputDesc->isStreamActive((audio\_stream\_type\_t)curStream)) {

curDevices |= outputDesc->device();

}

}

devices |= curDevices;

}

/\*Filter SPEAKER\_SAFE out of results, as AudioService doesn't know about it

and doesn't really need to.\*/

if (devices & AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE) {

devices |= AUDIO\_DEVICE\_OUT\_SPEAKER;

devices &= ~AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE;

}

return devices;}

* streamsMatchForvolume 就是判断两个是否相等
* getStrategy函数，将stream转换为Strategy
* getDeviceForStrategy 根据 Strategy 获取device
* getOutputsForDevice 获取device相关的output
* output相关的device，outputDesc->device()

## 音频流到音频策略的映射

在native会将流转换为策略，根据策略选择设备等。

\* frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp

routing\_strategy AudioPolicyManager::getStrategy(audio\_stream\_type\_t stream) const{

ALOG\_ASSERT(stream != AUDIO\_STREAM\_PATCH,"getStrategy() called for AUDIO\_STREAM\_PATCH");

return mEngine->getStrategyForStream(stream);}

mEngine是在AudioPolicyManager的构造函数中生成的。

AudioPolicyManager::AudioPolicyManager(AudioPolicyClientInterface \*clientInterface)

:... ...{

... ...

// Once policy config has been parsed, retrieve an instance of the engine and initialize it.

audio\_policy::EngineInstance \*engineInstance = audio\_policy::EngineInstance::getInstance();

if (!engineInstance) {

ALOGE("%s: Could not get an instance of policy engine", \_\_FUNCTION\_\_);

return;

}

// Retrieve the Policy Manager Interface

mEngine = engineInstance->queryInterface<AudioPolicyManagerInterface>();

if (mEngine == NULL) {

ALOGE("%s: Failed to get Policy Engine Interface", \_\_FUNCTION\_\_);

return;

}

mEngine->setObserver(this);

status\_t status = mEngine->initCheck();

Engine可以可以配置的，根据 USE\_CONFIGURABLE\_AUDIO\_POLICY定义，我们这里分析Default的enginedefault。

\* frameworks/av/services/audiopolicy/enginedefault/src/Engine.cpp

routing\_strategy Engine::getStrategyForStream(audio\_stream\_type\_t stream){

// stream to strategy mapping

switch (stream) {

case AUDIO\_STREAM\_VOICE\_CALL:

case AUDIO\_STREAM\_BLUETOOTH\_SCO:

return STRATEGY\_PHONE;

case AUDIO\_STREAM\_RING:

case AUDIO\_STREAM\_ALARM:

return STRATEGY\_SONIFICATION;

case AUDIO\_STREAM\_NOTIFICATION:

return STRATEGY\_SONIFICATION\_RESPECTFUL;

case AUDIO\_STREAM\_DTMF:

return STRATEGY\_DTMF;

default:

ALOGE("unknown stream type %d", stream);

case AUDIO\_STREAM\_SYSTEM:

// NOTE: SYSTEM stream uses MEDIA strategy because muting music and switching outputs

// while key clicks are played produces a poor result

case AUDIO\_STREAM\_MUSIC:

return STRATEGY\_MEDIA;

case AUDIO\_STREAM\_ENFORCED\_AUDIBLE:

return STRATEGY\_ENFORCED\_AUDIBLE;

case AUDIO\_STREAM\_TTS:

return STRATEGY\_TRANSMITTED\_THROUGH\_SPEAKER;

case AUDIO\_STREAM\_ACCESSIBILITY:

return STRATEGY\_ACCESSIBILITY;

case AUDIO\_STREAM\_REROUTING:

return STRATEGY\_REROUTING;

}}

native声音流的定义：

\* system/media/audio/include/system/audio-base.h

typedef enum {

AUDIO\_STREAM\_DEFAULT = -1, // (-1)

AUDIO\_STREAM\_MIN = 0,

AUDIO\_STREAM\_VOICE\_CALL = 0,

AUDIO\_STREAM\_SYSTEM = 1,

AUDIO\_STREAM\_RING = 2,

AUDIO\_STREAM\_MUSIC = 3,

AUDIO\_STREAM\_ALARM = 4,

AUDIO\_STREAM\_NOTIFICATION = 5,

AUDIO\_STREAM\_BLUETOOTH\_SCO = 6,

AUDIO\_STREAM\_ENFORCED\_AUDIBLE = 7,

AUDIO\_STREAM\_DTMF = 8,

AUDIO\_STREAM\_TTS = 9,

AUDIO\_STREAM\_ACCESSIBILITY = 10,

AUDIO\_STREAM\_REROUTING = 11,

AUDIO\_STREAM\_PATCH = 12,

AUDIO\_STREAM\_PUBLIC\_CNT = 11, // (ACCESSIBILITY + 1)

AUDIO\_STREAM\_FOR\_POLICY\_CNT = 12, // PATCH

AUDIO\_STREAM\_CNT = 13, // (PATCH + 1)} audio\_stream\_type\_t;

策略定义：

\* frameworks/av/services/audiopolicy/common/include/RoutingStrategy.h

enum routing\_strategy {

STRATEGY\_MEDIA,

STRATEGY\_PHONE,

STRATEGY\_SONIFICATION,

STRATEGY\_SONIFICATION\_RESPECTFUL,

STRATEGY\_DTMF,

STRATEGY\_ENFORCED\_AUDIBLE,

STRATEGY\_TRANSMITTED\_THROUGH\_SPEAKER,

STRATEGY\_ACCESSIBILITY,

STRATEGY\_REROUTING,

NUM\_STRATEGIES};

相互将的对应关系如下：

| **流代号** | **流类型** | **Strategy映射** | **描述** |
| --- | --- | --- | --- |
| 0 | STREAM\_VOICE\_CALL | STRATEGY\_PHONE | 通话相关音频 |
| 6 | AUDIO\_STREAM\_BLUETOOTH\_SCO | STRATEGY\_PHONE | 蓝牙SCO通话相关音频 |
| 2 | AUDIO\_STREAM\_RING | STRATEGY\_SONIFICATION | 铃声 |
| 4 | AUDIO\_STREAM\_ALARM | STRATEGY\_SONIFICATION | 闹钟 |
| 5 | AUDIO\_STREAM\_NOTIFICATION | STRATEGY\_SONIFICATION\_RESPECTFUL | 通知音 |
| 8 | AUDIO\_STREAM\_DTMF | STRATEGY\_DTMF | DTMF音 |
| 1 | AUDIO\_STREAM\_SYSTEM | STRATEGY\_MEDIA | 系统音 |
| 3 | AUDIO\_STREAM\_MUSIC | STRATEGY\_MEDIA | 媒体音 |
| 7 | AUDIO\_STREAM\_ENFORCED\_AUDIBLE | STRATEGY\_ENFORCED\_AUDIBLE | 强制为Speaker出声 |
| 9 | AUDIO\_STREAM\_TTS | STRATEGY\_TRANSMITTED\_THROUGH\_SPEAKER | TTS 播报 |
| 10 | AUDIO\_STREAM\_ACCESSIBILITY | STRATEGY\_ACCESSIBILITY | 辅助音 |
| 11 | AUDIO\_STREAM\_REROUTING | STRATEGY\_REROUTING | 动态输出混音 |

AUDIO\_STREAM\_ENFORCED\_AUDIBLE 这个流是不让用户静音的，强制为Speaker出声。比如拍照音，拍照音是必须Speaker出声的，防偷拍。

回到getDevicesForStream函数～

## 获取设备

根据流类型，获取到策略后，再根据策略获取相关的设备。

\* frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp

audio\_devices\_t AudioPolicyManager::getDeviceForStrategy(routing\_strategy strategy,

bool fromCache){

// Routing

// see if we have an explicit route

// scan the whole RouteMap, for each entry, convert the stream type to a strategy

// (getStrategy(stream)).

// if the strategy from the stream type in the RouteMap is the same as the argument above,

// and activity count is non-zero and the device in the route descriptor is available

// then select this device.

for (size\_t routeIndex = 0; routeIndex < mOutputRoutes.size(); routeIndex++) {

sp<SessionRoute> route = mOutputRoutes.valueAt(routeIndex);

routing\_strategy routeStrategy = getStrategy(route->mStreamType);

if ((routeStrategy == strategy) && route->isActiveOrChanged() &&

(mAvailableOutputDevices.indexOf(route->mDeviceDescriptor) >= 0)) {

return route->mDeviceDescriptor->type();

}

}

if (fromCache) {

ALOGVV("getDeviceForStrategy() from cache strategy %d, device %x",

strategy, mDeviceForStrategy[strategy]);

return mDeviceForStrategy[strategy];

}

return mEngine->getDeviceForStrategy(strategy);}

最终通过mEngine来实现的。STRATEGY\_TRANSMITTED\_THROUGH\_SPEAKER

\* frameworks/av/services/audiopolicy/enginedefault/src/Engine.cpp

audio\_devices\_t Engine::getDeviceForStrategy(routing\_strategy strategy) const{

DeviceVector availableOutputDevices = mApmObserver->getAvailableOutputDevices();

DeviceVector availableInputDevices = mApmObserver->getAvailableInputDevices();

const SwAudioOutputCollection &outputs = mApmObserver->getOutputs();

return getDeviceForStrategyInt(strategy, availableOutputDevices,

availableInputDevices, outputs);}

主要实现在 getDeviceForStrategyInt 函数中。getDeviceForStrategyInt函数非常长。这里就不贴全部的代码。起主要有以下几个方面决定：

1. 是否在通话中
2. 当前是什么流出于Active状态
3. 有没有ForceUse

我们来看看STRATEGY\_SONIFICATION\_RESPECTFUL，AUDIO\_STREAM\_NOTIFICATION场景的：

case STRATEGY\_SONIFICATION\_RESPECTFUL:

if (isInCall()) {

device = getDeviceForStrategyInt(

STRATEGY\_SONIFICATION, availableOutputDevices, availableInputDevices, outputs);

} else if (outputs.isStreamActiveRemotely(AUDIO\_STREAM\_MUSIC,

SONIFICATION\_RESPECTFUL\_AFTER\_MUSIC\_DELAY)) {

// while media is playing on a remote device, use the the sonification behavior.

// Note that we test this usecase before testing if media is playing because

// the isStreamActive() method only informs about the activity of a stream, not

// if it's for local playback. Note also that we use the same delay between both tests

device = getDeviceForStrategyInt(

STRATEGY\_SONIFICATION, availableOutputDevices, availableInputDevices, outputs);

//user "safe" speaker if available instead of normal speaker to avoid triggering

//other acoustic safety mechanisms for notification

if ((device & AUDIO\_DEVICE\_OUT\_SPEAKER) &&

(availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE)) {

device |= AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE;

device &= ~AUDIO\_DEVICE\_OUT\_SPEAKER;

}

} else if (outputs.isStreamActive(

AUDIO\_STREAM\_MUSIC, SONIFICATION\_RESPECTFUL\_AFTER\_MUSIC\_DELAY)

|| outputs.isStreamActive(

AUDIO\_STREAM\_ACCESSIBILITY, SONIFICATION\_RESPECTFUL\_AFTER\_MUSIC\_DELAY))

{

// while media/a11y is playing (or has recently played), use the same device

device = getDeviceForStrategyInt(

STRATEGY\_MEDIA, availableOutputDevices, availableInputDevices, outputs);

} else {

// when media is not playing anymore, fall back on the sonification behavior

device = getDeviceForStrategyInt(

STRATEGY\_SONIFICATION, availableOutputDevices, availableInputDevices, outputs);

//user "safe" speaker if available instead of normal speaker to avoid triggering

//other acoustic safety mechanisms for notification

if ((device & AUDIO\_DEVICE\_OUT\_SPEAKER) &&

(availableOutputDevicesType & AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE)) {

device |= AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE;

device &= ~AUDIO\_DEVICE\_OUT\_SPEAKER;

}

}

break;

1. 如果是通话中，转换为STRATEGY\_SONIFICATION
2. 如果Remote Music Active，转换为STRATEGY\_SONIFICATION，但是如果支持SPEAKER\_SAFE，优先使用SPEAKER\_SAFE
3. 如果AUDIO\_STREAM\_MUSIC或AUDIO\_STREAM\_ACCESSIBILITY，转换为STRATEGY\_MEDIA
4. 其余情况，采用STRATEGY\_SONIFICATION，但是如果支持SPEAKER\_SAFE，优先使用SPEAKER\_SAFE

具体情况，具体看，比较复杂的。Anyway，到此，我们的device算是获取到了。

## Native设置音量

音量都是通过AudioSystem的setStreamVolumeIndex函数，设置到native的：

public void applyDeviceVolume\_syncVSS(int device) {

int index;

if (mIsMuted) {

index = 0;

} else if ((device & AudioSystem.DEVICE\_OUT\_ALL\_A2DP) != 0 && mAvrcpAbsVolSupported) {

index = getAbsoluteVolumeIndex((getIndex(device) + 5)/10);

} else if ((device & mFullVolumeDevices) != 0) {

index = (mIndexMax + 5)/10;

} else {

index = (getIndex(device) + 5)/10;

}

AudioSystem.setStreamVolumeIndex(mStreamType, index, device);

}

中间的过程省略，我们直接AudioPolicyManager的实现：

\* frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp

status\_t AudioPolicyManager::setStreamVolumeIndex(audio\_stream\_type\_t stream,

int index,

audio\_devices\_t device){

... ...

// Force max volume if stream cannot be muted

if (!mVolumeCurves->canBeMuted(stream)) index = mVolumeCurves->getVolumeIndexMax(stream);

ALOGV("setStreamVolumeIndex() stream %d, device %08x, index %d",

stream, device, index);

// update other private stream volumes which follow this one

for (int curStream = 0; curStream < AUDIO\_STREAM\_FOR\_POLICY\_CNT; curStream++) {

if (!streamsMatchForvolume(stream, (audio\_stream\_type\_t)curStream)) {

continue;

}

mVolumeCurves->addCurrentVolumeIndex((audio\_stream\_type\_t)curStream, device, index);

}

status\_t status = NO\_ERROR;

for (size\_t i = 0; i < mOutputs.size(); i++) {

sp<SwAudioOutputDescriptor> desc = mOutputs.valueAt(i);

audio\_devices\_t curDevice = Volume::getDeviceForVolume(desc->device());

for (int curStream = 0; curStream < AUDIO\_STREAM\_FOR\_POLICY\_CNT; curStream++) {

if (!streamsMatchForvolume(stream, (audio\_stream\_type\_t)curStream)) {

continue;

}

if (!(desc->isStreamActive((audio\_stream\_type\_t)curStream) ||

(isInCall() && (curStream == AUDIO\_STREAM\_VOICE\_CALL)))) {

continue;

}

routing\_strategy curStrategy = getStrategy((audio\_stream\_type\_t)curStream);

audio\_devices\_t curStreamDevice = Volume::getDeviceForVolume(getDeviceForStrategy(

curStrategy, false /\*fromCache\*/));

if ((device != AUDIO\_DEVICE\_OUT\_DEFAULT\_FOR\_VOLUME) &&

((curStreamDevice & device) == 0)) {

continue;

}

bool applyVolume;

if (device != AUDIO\_DEVICE\_OUT\_DEFAULT\_FOR\_VOLUME) {

curStreamDevice |= device;

applyVolume = (curDevice & curStreamDevice) != 0;

} else {

applyVolume = !mVolumeCurves->hasVolumeIndexForDevice(

stream, curStreamDevice);

}

if (applyVolume) {

//FIXME: workaround for truncated touch sounds

// delayed volume change for system stream to be removed when the problem is

// handled by system UI

status\_t volStatus =

checkAndSetVolume((audio\_stream\_type\_t)curStream, index, desc, curDevice,

(stream == AUDIO\_STREAM\_SYSTEM) ? TOUCH\_SOUND\_FIXED\_DELAY\_MS : 0);

if (volStatus != NO\_ERROR) {

status = volStatus;

}

}

}

}

return status;}

* mVolumeCurves, 流的描述，是一个集合，用StreamDescriptorCollection表示。具体的流用StreamDescriptor描述。前面的流都是用一个int类型来描述的，现在用StreamDescriptor来描述了。

frameworks/av/services/audiopolicy/common/managerdefinitions/src/StreamDescriptor.cpp

* Volume的index值，通过addCurrentVolumeIndex，保存到了StreamDescriptor中：

\* frameworks/av/services/audiopolicy/common/managerdefinitions/src/StreamDescriptor.cpp

void StreamDescriptor::addCurrentVolumeIndex(audio\_devices\_t device, int index){

mIndexCur.add(device, index);}

Volume按照device保存到mIndexCur中：

KeyedVector<audio\_devices\_t, int> mIndexCur;

* mOutputs 当前的输出设备  
  每一个输出设备用SwAudioOutputDescriptor描述:

frameworks/av/services/audiopolicy/common/managerdefinitions/src/AudioOutputDescriptor.cpp

* 获取设备，再次确认

Volume::getDeviceForVolume

static audio\_devices\_t getDeviceForVolume(audio\_devices\_t device)

{

if (device == AUDIO\_DEVICE\_NONE) {

device = AUDIO\_DEVICE\_OUT\_SPEAKER;

} else if (popcount(device) > 1) {

if (device & AUDIO\_DEVICE\_OUT\_SPEAKER) {

device = AUDIO\_DEVICE\_OUT\_SPEAKER;

} else if (device & AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE) {

device = AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE;

} else if (device & AUDIO\_DEVICE\_OUT\_HDMI\_ARC) {

device = AUDIO\_DEVICE\_OUT\_HDMI\_ARC;

} else if (device & AUDIO\_DEVICE\_OUT\_AUX\_LINE) {

device = AUDIO\_DEVICE\_OUT\_AUX\_LINE;

} else if (device & AUDIO\_DEVICE\_OUT\_SPDIF) {

device = AUDIO\_DEVICE\_OUT\_SPDIF;

} else {

device = (audio\_devices\_t)(device & AUDIO\_DEVICE\_OUT\_ALL\_A2DP);

}

}

/\*SPEAKER\_SAFE is an alias of SPEAKER for purposes of volume control\*/

if (device == AUDIO\_DEVICE\_OUT\_SPEAKER\_SAFE)

device = AUDIO\_DEVICE\_OUT\_SPEAKER;

return device;

}

* 根据流获取策略，根据测试获取device，再根据device判断是否需要应用音量，音量是通过checkAndSetVolume来生效的。

如果当前没有任何流处在active，音量设置时是没有生效的，这根据前面的applyVolume来决定。如果这个时候没有生效，后续在startSource时，也会checkAndSetVolume。

\* frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp

status\_t AudioPolicyManager::checkAndSetVolume(audio\_stream\_type\_t stream,

int index,

const sp<AudioOutputDescriptor>& outputDesc,

audio\_devices\_t device,

int delayMs,

bool force){

// 静音，不修改音量值

if (outputDesc->mMuteCount[stream] != 0) {

ALOGVV("checkAndSetVolume() stream %d muted count %d",

stream, outputDesc->mMuteCount[stream]);

return NO\_ERROR;

}

audio\_policy\_forced\_cfg\_t forceUseForComm =

mEngine->getForceUse(AUDIO\_POLICY\_FORCE\_FOR\_COMMUNICATION);

// 如果是蓝牙通话，不修改音量值，反之亦然

if ((stream == AUDIO\_STREAM\_VOICE\_CALL && forceUseForComm == AUDIO\_POLICY\_FORCE\_BT\_SCO) ||

(stream == AUDIO\_STREAM\_BLUETOOTH\_SCO && forceUseForComm != AUDIO\_POLICY\_FORCE\_BT\_SCO)) {

ALOGV("checkAndSetVolume() cannot set stream %d volume with force use = %d for comm",

stream, forceUseForComm);

return INVALID\_OPERATION;

}

if (device == AUDIO\_DEVICE\_NONE) {

device = outputDesc->device();

}

// 关键代码computeVolume，计算音量值，将index值转换为Db值。

float volumeDb = computeVolume(stream, index, device);

if (outputDesc->isFixedVolume(device)) {

volumeDb = 0.0f;

}

// 设置音量

outputDesc->setVolume(volumeDb, stream, device, delayMs, force);

// 通话音量，SCO设置为最大，headset自己管理。

if (stream == AUDIO\_STREAM\_VOICE\_CALL ||

stream == AUDIO\_STREAM\_BLUETOOTH\_SCO) {

float voiceVolume;

if (stream == AUDIO\_STREAM\_VOICE\_CALL) {

voiceVolume = (float)index/(float)mVolumeCurves->getVolumeIndexMax(stream);

} else {

voiceVolume = 1.0;

}

if (voiceVolume != mLastVoiceVolume) {

mpClientInterface->setVoiceVolume(voiceVolume, delayMs);

mLastVoiceVolume = voiceVolume;

}

}

return NO\_ERROR;}

## 计算音量Db值

\* frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp

float AudioPolicyManager::computeVolume(audio\_stream\_type\_t stream,

int index,

audio\_devices\_t device){

float volumeDB = mVolumeCurves->volIndexToDb(stream, Volume::getDeviceCategory(device), index);

// 处理辅助功能开启，响铃的场景

if ((stream == AUDIO\_STREAM\_ACCESSIBILITY)

&& (AUDIO\_MODE\_RINGTONE == mEngine->getPhoneState())

&& isStreamActive(AUDIO\_STREAM\_RING, 0)) {

const float ringVolumeDB = computeVolume(AUDIO\_STREAM\_RING, index, device);

return ringVolumeDB - 4 > volumeDB ? ringVolumeDB - 4 : volumeDB;

}

// 通话中的场景

if ((stream != AUDIO\_STREAM\_VOICE\_CALL) && (device & AUDIO\_DEVICE\_OUT\_EARPIECE) && isInCall()) {

switch (stream) {

case AUDIO\_STREAM\_SYSTEM:

case AUDIO\_STREAM\_RING:

case AUDIO\_STREAM\_MUSIC:

case AUDIO\_STREAM\_ALARM:

case AUDIO\_STREAM\_NOTIFICATION:

case AUDIO\_STREAM\_ENFORCED\_AUDIBLE:

case AUDIO\_STREAM\_DTMF:

case AUDIO\_STREAM\_ACCESSIBILITY: {

const float maxVoiceVolDb = computeVolume(AUDIO\_STREAM\_VOICE\_CALL, index, device)

+ IN\_CALL\_EARPIECE\_HEADROOM\_DB;

if (volumeDB > maxVoiceVolDb) {

ALOGV("computeVolume() stream %d at vol=%f overriden by stream %d at vol=%f",

stream, volumeDB, AUDIO\_STREAM\_VOICE\_CALL, maxVoiceVolDb);

volumeDB = maxVoiceVolDb;

}

} break;

default:

break;

}

}

// 插耳机的场景，防止响铃等声音过大

const routing\_strategy stream\_strategy = getStrategy(stream);

if ((device & (AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES |

AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET |

AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE |

AUDIO\_DEVICE\_OUT\_USB\_HEADSET)) &&

((stream\_strategy == STRATEGY\_SONIFICATION)

|| (stream\_strategy == STRATEGY\_SONIFICATION\_RESPECTFUL)

|| (stream == AUDIO\_STREAM\_SYSTEM)

|| ((stream\_strategy == STRATEGY\_ENFORCED\_AUDIBLE) &&

(mEngine->getForceUse(AUDIO\_POLICY\_FORCE\_FOR\_SYSTEM) == AUDIO\_POLICY\_FORCE\_NONE))) &&

mVolumeCurves->canBeMuted(stream)) {

// when the phone is ringing we must consider that music could have been paused just before

// by the music application and behave as if music was active if the last music track was

// just stopped

if (isStreamActive(AUDIO\_STREAM\_MUSIC, SONIFICATION\_HEADSET\_MUSIC\_DELAY) ||

mLimitRingtoneVolume) {

volumeDB += SONIFICATION\_HEADSET\_VOLUME\_FACTOR\_DB;

audio\_devices\_t musicDevice = getDeviceForStrategy(STRATEGY\_MEDIA, true /\*fromCache\*/);

float musicVolDB = computeVolume(AUDIO\_STREAM\_MUSIC,

mVolumeCurves->getVolumeIndex(AUDIO\_STREAM\_MUSIC,

musicDevice),

musicDevice);

float minVolDB = (musicVolDB > SONIFICATION\_HEADSET\_VOLUME\_MIN\_DB) ?

musicVolDB : SONIFICATION\_HEADSET\_VOLUME\_MIN\_DB;

if (volumeDB > minVolDB) {

volumeDB = minVolDB;

ALOGV("computeVolume limiting volume to %f musicVol %f", minVolDB, musicVolDB);

}

if (device & (AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP |

AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES)) {

// on A2DP, also ensure notification volume is not too low compared to media when

// intended to be played

if ((volumeDB > -96.0f) &&

(musicVolDB - SONIFICATION\_A2DP\_MAX\_MEDIA\_DIFF\_DB > volumeDB)) {

ALOGV("computeVolume increasing volume for stream=%d device=0x%X from %f to %f",

stream, device,

volumeDB, musicVolDB - SONIFICATION\_A2DP\_MAX\_MEDIA\_DIFF\_DB);

volumeDB = musicVolDB - SONIFICATION\_A2DP\_MAX\_MEDIA\_DIFF\_DB;

}

}

} else if ((Volume::getDeviceForVolume(device) != AUDIO\_DEVICE\_OUT\_SPEAKER) ||

stream\_strategy != STRATEGY\_SONIFICATION) {

volumeDB += SONIFICATION\_HEADSET\_VOLUME\_FACTOR\_DB;

}

}

return volumeDB;}

Db值通过mVolumeCurves->volIndexToDb，进行转换，转换后，再根据实际的场景进行调整。

我们先来看看Volume::getDeviceCategory，Audio这边就死麻烦，各种概念。device\_category将设备进行分类。

\* frameworks/av/services/audiopolicy/common/include/Volume.h

static device\_category getDeviceCategory(audio\_devices\_t device)

{

switch(getDeviceForVolume(device)) {

case AUDIO\_DEVICE\_OUT\_EARPIECE:

return DEVICE\_CATEGORY\_EARPIECE;

case AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET:

case AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE:

case AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO:

case AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET:

case AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP:

case AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_HEADPHONES:

case AUDIO\_DEVICE\_OUT\_USB\_HEADSET:

return DEVICE\_CATEGORY\_HEADSET;

case AUDIO\_DEVICE\_OUT\_LINE:

case AUDIO\_DEVICE\_OUT\_AUX\_DIGITAL:

case AUDIO\_DEVICE\_OUT\_USB\_DEVICE:

return DEVICE\_CATEGORY\_EXT\_MEDIA;

case AUDIO\_DEVICE\_OUT\_SPEAKER:

case AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT:

case AUDIO\_DEVICE\_OUT\_BLUETOOTH\_A2DP\_SPEAKER:

case AUDIO\_DEVICE\_OUT\_USB\_ACCESSORY:

case AUDIO\_DEVICE\_OUT\_REMOTE\_SUBMIX:

default:

return DEVICE\_CATEGORY\_SPEAKER;

}

}

volIndexToDb函数 如下：

float StreamDescriptorCollection::volIndexToDb(audio\_stream\_type\_t stream, device\_category category,

int indexInUi) const{

const StreamDescriptor &streamDesc = valueAt(stream);

return Gains::volIndexToDb(streamDesc.getVolumeCurvePoint(category),

streamDesc.getVolumeIndexMin(), streamDesc.getVolumeIndexMax(),

indexInUi);}

float Gains::volIndexToDb(const VolumeCurvePoint \*curve, int indexMin, int indexMax, int indexInUi){

// the volume index in the UI is relative to the min and max volume indices for this stream type

int nbSteps = 1 + curve[Volume::VOLMAX].mIndex - curve[Volume::VOLMIN].mIndex;

int volIdx = (nbSteps \* (indexInUi - indexMin)) / (indexMax - indexMin);

// find what part of the curve this index volume belongs to, or if it's out of bounds

int segment = 0;

if (volIdx < curve[Volume::VOLMIN].mIndex) { // out of bounds

return VOLUME\_MIN\_DB;

} else if (volIdx < curve[Volume::VOLKNEE1].mIndex) {

segment = 0;

} else if (volIdx < curve[Volume::VOLKNEE2].mIndex) {

segment = 1;

} else if (volIdx <= curve[Volume::VOLMAX].mIndex) {

segment = 2;

} else { // out of bounds

return 0.0f;

}

// linear interpolation in the attenuation table in dB

float decibels = curve[segment].mDBAttenuation +

((float)(volIdx - curve[segment].mIndex)) \*

( (curve[segment+1].mDBAttenuation -

curve[segment].mDBAttenuation) /

((float)(curve[segment+1].mIndex -

curve[segment].mIndex)) );

ALOGVV("VOLUME vol index=[%d %d %d], dB=[%.1f %.1f %.1f]",

curve[segment].mIndex, volIdx,

curve[segment+1].mIndex,

curve[segment].mDBAttenuation,

decibels,

curve[segment+1].mDBAttenuation);

return decibels;}

Db值都是配置在audio\_policy\_volumes.xml中的。AOSP的

frameworks/av/services/audiopolicy/config/audio\_policy\_volumes.xml

安装流和设备分类来区分，比如，耳机的通话音量：

<volume stream="AUDIO\_STREAM\_VOICE\_CALL" deviceCategory="DEVICE\_CATEGORY\_HEADSET">

<point>0,-4200</point>

<point>33,-2800</point>

<point>66,-1400</point>

<point>100,0</point>

</volume>

除了直接定义值，还可以引用其他的，比如耳机的媒体音量，DEFAULT\_MEDIA\_VOLUME\_CURVE。

<volume stream="AUDIO\_STREAM\_MUSIC" deviceCategory="DEVICE\_CATEGORY\_HEADSET"

ref="DEFAULT\_MEDIA\_VOLUME\_CURVE"/>

DEFAULT\_MEDIA\_VOLUME\_CURVE在另外已给表中：

\* frameworks/av/services/audiopolicy/config/default\_volume\_tables.xml

<reference name="DEFAULT\_MEDIA\_VOLUME\_CURVE">

<!-- Default Media reference Volume Curve -->

<point>1,-5800</point>

<point>20,-4000</point>

<point>60,-1700</point>

<point>100,0</point>

</reference>

所以，我们调节音量时，根据音量的index，来转为对应的db，最终生效的是db。很多用户反馈上面音量调的很大了，但是实际的声音还是小，可能就调调整一下这里的db配置了。

## 音量生效

计算后的Db值通过setVolume函数设置给output

outputDesc->setVolume(volumeDb, stream, device, delayMs, force);

函数如下：

bool SwAudioOutputDescriptor::setVolume(float volume,

audio\_stream\_type\_t stream,

audio\_devices\_t device,

uint32\_t delayMs,

bool force){

bool changed = AudioOutputDescriptor::setVolume(volume, stream, device, delayMs, force);

if (changed) {

// Force VOICE\_CALL to track BLUETOOTH\_SCO stream volume when bluetooth audio is

// enabled

float volume = Volume::DbToAmpl(mCurVolume[stream]);

if (stream == AUDIO\_STREAM\_BLUETOOTH\_SCO) {

mClientInterface->setStreamVolume(

AUDIO\_STREAM\_VOICE\_CALL, volume, mIoHandle, delayMs);

}

mClientInterface->setStreamVolume(stream, volume, mIoHandle, delayMs);

}

return changed;}

* 先将音量值保存到Descriptor中

bool AudioOutputDescriptor::setVolume(float volume,

audio\_stream\_type\_t stream,

audio\_devices\_t device \_\_unused,

uint32\_t delayMs,

bool force){

// We actually change the volume if:

// - the float value returned by computeVolume() changed

// - the force flag is set

if (volume != mCurVolume[stream] || force) {

ALOGV("setVolume() for stream %d, volume %f, delay %d", stream, volume, delayMs);

mCurVolume[stream] = volume;

return true;

}

return false;}

每个output都是安装流类型来保存的mCurVolume[stream]

* 通过mClientInterface的setStreamVolume接口

\* frameworks/av/services/audiopolicy/service/AudioPolicyClientImpl.cpp

status\_t AudioPolicyService::AudioPolicyClient::setStreamVolume(audio\_stream\_type\_t stream,

float volume, audio\_io\_handle\_t output,

int delay\_ms){

return mAudioPolicyService->setStreamVolume(stream, volume, output,

delay\_ms);}

AudioPolicyService通过AudioCommandThread，传给AudioManager

\* frameworks/av/services/audiopolicy/service/AudioPolicyService.cpp

int AudioPolicyService::setStreamVolume(audio\_stream\_type\_t stream,

float volume,

audio\_io\_handle\_t output,

int delayMs){

return (int)mAudioCommandThread->volumeCommand(stream, volume,

output, delayMs);}

对应的命令：SET\_VOLUME

bool AudioPolicyService::AudioCommandThread::threadLoop(){

nsecs\_t waitTime = -1;

mLock.lock();

while (!exitPending())

{

sp<AudioPolicyService> svc;

while (!mAudioCommands.isEmpty() && !exitPending()) {... ...

case SET\_VOLUME: {

VolumeData \*data = (VolumeData \*)command->mParam.get();

ALOGV("AudioCommandThread() processing set volume stream %d, \

volume %f, output %d", data->mStream, data->mVolume, data->mIO);

command->mStatus = AudioSystem::setStreamVolume(data->mStream,

data->mVolume,

data->mIO);

}break;

最终还是通过AudioSystem的接口来完成：

status\_t AudioSystem::setStreamVolume(audio\_stream\_type\_t stream, float value,

audio\_io\_handle\_t output){

if (uint32\_t(stream) >= AUDIO\_STREAM\_CNT) return BAD\_VALUE;

const sp<IAudioFlinger>& af = AudioSystem::get\_audio\_flinger();

if (af == 0) return PERMISSION\_DENIED;

af->setStreamVolume(stream, value, output);

return NO\_ERROR;}

AudioFlinger的setStreamVolume函数如下：

status\_t AudioFlinger::setStreamVolume(audio\_stream\_type\_t stream, float value,

audio\_io\_handle\_t output){

ALOGI("setStreamVolume: stream %d, value %f, output %d", stream, value, output);

// 权限

if (!settingsAllowed()) {

return PERMISSION\_DENIED;

}

// 流类型

status\_t status = checkStreamType(stream);

if (status != NO\_ERROR) {

return status;

}

ALOG\_ASSERT(stream != AUDIO\_STREAM\_PATCH, "attempt to change AUDIO\_STREAM\_PATCH volume");

AutoMutex lock(mLock);

Vector<VolumeInterface \*> volumeInterfaces;

//获取volumeInterface

if (output != AUDIO\_IO\_HANDLE\_NONE) {

VolumeInterface \*volumeInterface = getVolumeInterface\_l(output);

if (volumeInterface == NULL) {

return BAD\_VALUE;

}

volumeInterfaces.add(volumeInterface);

}

// 保存音量Db值

mStreamTypes[stream].volume = value;

if (volumeInterfaces.size() == 0) {

volumeInterfaces = getAllVolumeInterfaces\_l();

}

// 设置音量值

for (size\_t i = 0; i < volumeInterfaces.size(); i++) {

volumeInterfaces[i]->setStreamVolume(stream, value);

}

return NO\_ERROR;}

VolumeInterface 主要用以设置音量，PlaybackThread和MmapPlaybackThread实现VolumeInterface具体的接口：

class VolumeInterface {

public:

virtual ~VolumeInterface() {}

virtual void setMasterVolume(float value) = 0;

virtual void setMasterMute(bool muted) = 0;

virtual void setStreamVolume(audio\_stream\_type\_t stream, float value) = 0;

virtual void setStreamMute(audio\_stream\_type\_t stream, bool muted) = 0;

virtual float streamVolume(audio\_stream\_type\_t stream) const = 0;

};

我们来看看PlaybackThread的setStreamVolume函数：

void AudioFlinger::PlaybackThread::setStreamVolume(audio\_stream\_type\_t stream, float value){

Mutex::Autolock \_l(mLock);

size\_t size = mEffectChains.size();

mStreamTypes[stream].volume = value;

for (size\_t i = 0; i < size; i++) {

mEffectChains[i]->setStreamVolume\_l(stream, value);

}

broadcast\_l();}

PlaybackThread将音量值保存下来了，并设置到音效中。

OK，到此，都全是设置音量的过程，那么是在什么地方生效的呢？

对于MixerThread来说，是在这里生效的，音量值最终会被混音，设置到数据流中。

AudioFlinger::PlaybackThread::mixer\_state AudioFlinger::MixerThread::prepareTracks\_l(

Vector< sp<Track> > \*tracksToRemove){... ...

// compute volume for this track

uint32\_t vl, vr; // in U8.24 integer format

float vlf, vrf, vaf; // in [0.0, 1.0] float format

// read original volumes with volume control

float typeVolume = mStreamTypes[track->streamType()].volume;

float v = masterVolume \* typeVolume;

if (track->isPausing() || mStreamTypes[track->streamType()].mute) {

vl = vr = 0;

vlf = vrf = vaf = 0.;

if (track->isPausing()) {

track->setPaused();

}

} else {

sp<AudioTrackServerProxy> proxy = track->mAudioTrackServerProxy;

gain\_minifloat\_packed\_t vlr = proxy->getVolumeLR();

vlf = float\_from\_gain(gain\_minifloat\_unpack\_left(vlr));

vrf = float\_from\_gain(gain\_minifloat\_unpack\_right(vlr));

// track volumes come from shared memory, so can't be trusted and must be clamped

if (vlf > GAIN\_FLOAT\_UNITY) {

ALOGV("Track left volume out of range: %.3g", vlf);

vlf = GAIN\_FLOAT\_UNITY;

}

if (vrf > GAIN\_FLOAT\_UNITY) {

ALOGV("Track right volume out of range: %.3g", vrf);

vrf = GAIN\_FLOAT\_UNITY;

}

const float vh = track->getVolumeHandler()->getVolume(

track->mAudioTrackServerProxy->framesReleased()).first;

// now apply the master volume and stream type volume and shaper volume

vlf \*= v \* vh;

vrf \*= v \* vh;

// assuming master volume and stream type volume each go up to 1.0,

// then derive vl and vr as U8.24 versions for the effect chain

const float scaleto8\_24 = MAX\_GAIN\_INT \* MAX\_GAIN\_INT;

vl = (uint32\_t) (scaleto8\_24 \* vlf);

vr = (uint32\_t) (scaleto8\_24 \* vrf);

// vl and vr are now in U8.24 format

uint16\_t sendLevel = proxy->getSendLevel\_U4\_12();

// send level comes from shared memory and so may be corrupt

if (sendLevel > MAX\_GAIN\_INT) {

ALOGV("Track send level out of range: %04X", sendLevel);

sendLevel = MAX\_GAIN\_INT;

}

// vaf is represented as [0.0, 1.0] float by rescaling sendLevel

vaf = v \* sendLevel \* (1. / MAX\_GAIN\_INT);

}

我们可以通过adb命令，将thread的各个流的音量都dump出来：

$ adb shell dumpsys media.audio\_flinger | grep "Stream volumes in dB"

Stream volumes in dB: 0:-24, 1:-27, 2:-25, 3:-52, 4:0, 5:-27, 6:0, 7:-6, 8:-27, 9:0, 10:-47, 11:0, 12:0

Stream volumes in dB: 0:-24, 1:-27, 2:-25, 3:-52, 4:0, 5:-27, 6:0, 7:-6, 8:-27, 9:0, 10:-47, 11:0, 12:0

Stream volumes in dB: 0:-24, 1:-27, 2:-25, 3:-52, 4:0, 5:-27, 6:0, 7:-6, 8:-27, 9:0, 10:-47,

作者：夕月风  
链接：https://www.jianshu.com/p/892761d689fc  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。