**AG35 Voice Over USB**

---



## 一. AG35 Voice Over USB 简单介绍

USB 是通用串行总线的意思，本质上并不是专门用来传输音频数据的。这里先简单列举 USB2.0 几个特性：双绞线、带电源、数据速率与传输频率无关、自同步（不需要单独传输时钟）、Token（令牌）轮询特性[1]。双绞线的特性使得 USB 天生具有抗共模干扰的能力，带供电使得它带的设备可以不需要电源，令牌轮询特性用于"交通管制"。

USB 是主从模式的总线，Host 控制器决定它下面所有设备一切事务的发送/接收时机。全速下，Host 每 1ms±500μs 生成一个“帧”(frame)。高速下，每 125us±0.0625μs 生成一个“帧”[2]。一个“帧”以 SOF 包打头，包含发送给同一个 Host 控制器下的不同设备的若干个 USB 数据包。因此同一个控制器下的 USB 设备带宽共享： 在同一个 USB 控制器上的全速设备(FullSpeed/FS)、低速设备(LowSpeed/LS)设备之间共享带宽。高速设备(HighSpeed/HS)之间共享带宽，不与全速/低速设备共享，甚至很多南桥会单独设计高速 USB2.0 控制器。这种共享带宽，在下行中是这样实现的：Host 设备发给高速设备 H1 的数据包，高速设备 H2 同样会收到相同的数据包，但是因为接收地址与自身地址不符，H2 设备会无视该数据包。同样的，Host 发给 F1 全速设备的数据包，F2 全速设备和 L1/L2 慢速设备也会收到同样的包，但是会将其无视上行带宽共享则是由主机控制：除非主机在“帧”中发送了 IN 包告诉这个设备准许发送数据，否则设备不允许发送数据。这样就避免了冲突产生。

USB 有四种传输通道类型（Endpoint）：控制(Control)、中断(Interrupt)、批量(Bulk)、同步(Isochronous)。

Control 用于 USB 总线控制等（所有设备都有），Interrupt 多用于鼠标键盘的数据传输（其实在 USB 令牌轮询的特性下这是个伪 Interrupt），Bulk 多用于 U 盘数据传输。

前三种类型都有重传，各有特点这里略过。但 USB 音频类（USB Audio Class，简称 UAC）采用的音频数据传输是第四种端口，也就是 Isochronous，拥有错误检测机制，但是没有重传。为何没有重传呢。

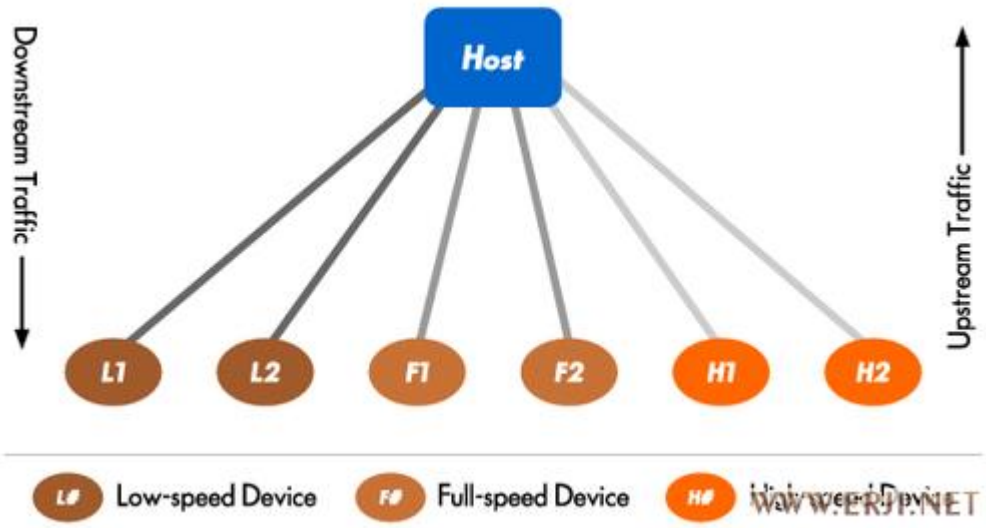
一是音频/视频是很讲究“实时性”的，否则会音画不同步，带来的恶果是很蛋疼的。因此音频/视频设备本质上都需要与主机同步，重传会打乱同步。

二是同步(Isochronous)类型因为要求低延迟，不像别的传输类型有握手过程，同一个"传输端口"内数据包是单向移动的（要么入要么出），Host 不会给一个 OUT 方向的同步传输通道发送 IN 包，没有办法引入重传机制（发生错误时无法通知主机）。[4]

另外还有一点，为了“同步”，每个“帧”中的每个同步(Isochronous)通道只会传送一次数据，这里有人要问了 用一个 IN 和一个 OUT 的 Isochronous 通道不就可以解决错误处理了？（ASYNC 异步下也确实有一对 IN 和 OUT 通道，具体作用下文会解释，反正不是用来“错误检测”）事实上，不可能。Host 在一个 Frame 中轮询各个 Endpoint 的顺序是不可知的，很可能 Host 在一个 Frame 中先轮询了你的 IN 通道，但此时 Host 还没给你的 OUT 通道传送数据。这样就产生了：数据还没发送 怎么能确认数据收到了呢。

但这是否值得担忧呢？USB 自身抗干扰特性(双绞线/屏蔽层)使得正常/正确使用的情况下，产生错误的概率非常非常低。真产生错误了，设备因为 CRC 校验失败，会将错误的 USB 数据包丢弃，而这会导致 usb 界面缓冲区欠载，导致短暂静音/爆音

USB 是主从模式的总线，Host 控制器决定它下面所有设备一切事务的发送/接收时机。全速下，Host 每 1ms±500μs 生成一个“帧”(frame)。高速下，每 125us±0.0625μs 生成一个“帧”[2]。一个“帧”以 SOF 包打头，包含发送给同一个 Host 控制器下的不同设备的若干个 USB 数据包。因此同一个控制器下的 USB 设备带宽共享：




USB 是通用串行总线的意思，本质上并不是专门用来传输音频数据的。这里先简单列举 USB2.0 几个特性：双绞线、带电源、数据速率与传输频率无关、自同步（不需要单独传输时钟）、Token（令牌）轮询特性[1]。双绞线的特性使得 USB 天生具有抗共模干扰的能力，带供电使得它带的设备可以不需要电源，令牌轮询特性用于“交通管制”。

USB 是主从模式的总线，Host 控制器决定它下面所有设备一切事务的发送/接收时机。全速下，Host 每 1ms±500μs 生成一个“帧”(frame)。高速下，每 125us±0.0625μs 生成一个“帧”[2]。一个“帧”以 SOF 包打头，包含发送给同一个 Host 控制器下的不同设备的若干个 USB 数据包。因此同一个控制器下的 USB 设备带宽共享：

在同一个 USB 控制器上的全速设备(FullSpeed/FS)、低速设备(LowSpeed/LS)设备之间共享带宽。高速设备(HighSpeed/HS)之间共享带宽，不与全速/低速设备共享，甚至很多南桥会单独设计高速 USB2.0 控制器。这种共享带宽，在下行中是这样实现的：Host 设备发给高速设备 H1 的数据包，高速设备 H2 同样会收到相同的数据包，但是因为接收地址与自身地址不符，H2 设备会无视该数据包。同样的，Host 发给 F1 全速设备的数据包，F2 全速设备和 L1/L2 慢速设备也会收到同样的包，但是会将其无视。[3]

上行带宽共享则是由主机控制：除非主机在“帧”中发送了 IN 包告诉这个设备准许发送数据，否则设备不允许发送数据。这样就避免了冲突产生。

USB 有四种传输通道类型（Endpoint）：控制(Control)、中断(Interrupt)、批量(Bulk)、同步(Isochronous)。
Control 用于 USB 总线控制等（所有设备都有），Interrupt 多用于鼠标键盘的数据传输（其实在 USB 令牌轮询的特性下这是个伪 Interrupt），Bulk 多用于 U 盘数据传输。
前三种类型都有重传，各有特点这里略过。但 USB 音频类（USB Audio Class，简称 UAC）采用的音频数据传输是第四种端口，也就是 Isochronous，拥有错误检测机制，但是没有重传。

为何没有重传呢。
一是音频/视频是很讲究“实时性”的，否则会音画不同步，带来的恶果是很蛋疼的。因此音频/视频设备本质上都需要与主机同步，重传会打乱同步。

二是同步(Isochronous)类型因为要求低延迟，不像别的传输类型有握手过程，同一个"传输端口"内数据包是单向移动的（要么入要么出），Host 不会给一个 OUT 方向的同步传输通道发送 IN 包，没有办法引入重传机制（发生错误时无法通知主机）。[4]
另外还有一点，为了“同步”，每个“帧”中的每个同步(Isochronous)通道只会传送一次数据。

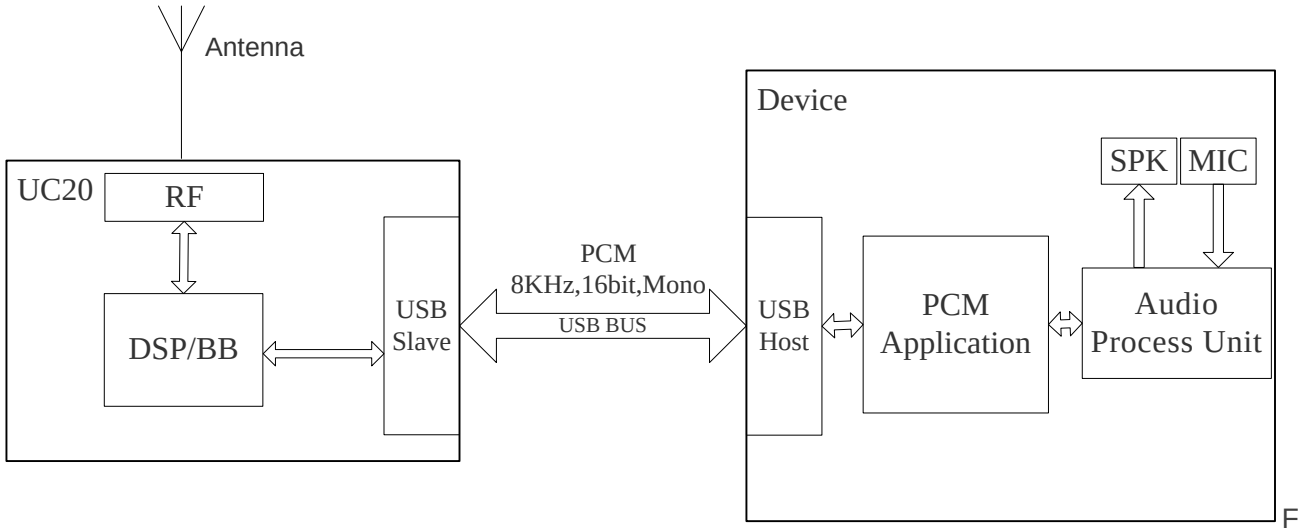
这里有人要问了 用一个 IN 和一个 OUT 的 Isochronous 通道不就可以解决错误处理了？（ASYNC 异步下也确实有一对 IN 和 OUT 通道，具体作用下文会解释，反正不是用来“错误检测”）

事实上，不可能。Host 在一个 Frame 中轮询各个 Endpoint 的顺序是不可知的，很可能 Host 在一个 Frame 中先轮询了你的 IN 通道，但此时 Host 还没给你的 OUT 通道传送数据。这样就产生了：数据还没发送 怎么能确认数据收到了呢。

但这是否值得担忧呢？USB 自身抗干扰特性(双绞线/屏蔽层)使得正常/正确使用的情况下，产生错误的概率非常非常低。真产生错误了，设备因为 CRC 校验失败，会将错误的 USB 数据包丢弃，而这会导致 USB 界面缓冲区欠载，导致短暂静音/爆音

In voice call status, the module obtains voice data through air interface, and DSP decodes the data to voice PCM (Pulse Code Modulation) stream, then the stream will be transferred to device by USB BUS. Finally, PCM application on the device will send PCM stream to audio process unit to broadcast locally. Similarly, after the voice being recorded by MIC on the device, it will be transferred in an opposite direction. Please note that this Voice over USB functionality of UC20 only supports Mono, and the PCM data must be at sample rate of 8KHz and in 16bit linear format.

PCM application on the device should control the sending speed of PCM data strictly to ensure the continuity of voice data. In the course of use, the device will receive 640 bytes of PCM data from the module in every 40ms; PCM application should receive these data and process them as soon as possible.
Meanwhile, PCM application obtains MIC recording data from audio process unit. It is required that PCMapplication should send 1600 bytes of voice data to the module at an interval of 100ms.




igure 1: Voice Path of Voice over USB Functionality

## 二.打开 USB 声卡的接口

（a）UAC（USB Audio class ）功能默认是不开放的，如果需要使能需要配置 USB 配置文件。

修改 usb 的配置文件

Vi /sbin/usb/compositons/9025/ 加入下面这一行（因为之前文件系统，这个文件是可读可写在 AG35 的 openlinux 的版本是只读在模块启动后无法修改，所以需要修改源文件这个文件 ）

```
echo 0 > /sys/class/android_usb/android$num/enable
echo $QUEC_USB_PID > /sys/class/android_usb/android$num/idProduct
echo $QUEC_USB_VID > /sys/class/android_usb/android$num/idVendor
echo diag > /sys/class/android_usb/android0/f_diag/clients
echo 239 > /sys/class/android_usb/android$num/bDeviceClass
echo 2 > /sys/class/android_usb/android$num/bDeviceSubClass
echo 1 > /sys/class/android_usb/android$num/bDeviceProtocol
echo $SERIAL_FUNC > /sys/class/android_usb/android0/f_serial/transports
echo QTI,BAM_DMUX > /sys/class/android_usb/android0/f_rmnet/transports
echo BAM_DMUX > /sys/class/android_usb/android0/f_usb_mbim/mbim_transports
echo $USB_FUNC,ffs,audio > /sys/class/android_usb/android$num/functions
echo 1 > /sys/class/android_usb/android$num/remote_wakeup
echo 1 > /sys/class/android_usb/android$num/f_rndis/wceis
```

（b）发送 AT 指令，打开开启 usbvoice

AG35 模块提供 AT+QPCMV 的命令来打开和关闭 voice over USB

| **AT+ QPCMV　Enable/Disable Voice over USB Functionality** | |
|---|---|
| Test Command<br>**AT+QPCMV=?** | Response<br>**+QPCMV: (0,1),(0,2)**<br><br>**OK** |
| Read Command<br>**AT+QPCMV?** | Response<br>**+QPCMV: <enable>[,<port>]**<br><br>**OK** |
| Write Command<br>**AT+QPCMV=<enable>[,<port>]** | Response<br>**OK**<br>**ERROR** |
| URC | **+QPCMV: 0**<br>**+QPCMV: 1** |
| Reference | |

AT+QPCMV=<enable>,<port>

enable: 1 打开 USB VOICE，0 关闭 USB VOICE

port: 0: USB NEMA 口,1：UART 口，2：USB AUDIO Class

发送下面命令打开 UAC，打开 UAC 之后，会关闭 AG35 的 codec，切换到 USB Voice。

AT+QPCMV=1,2
OK

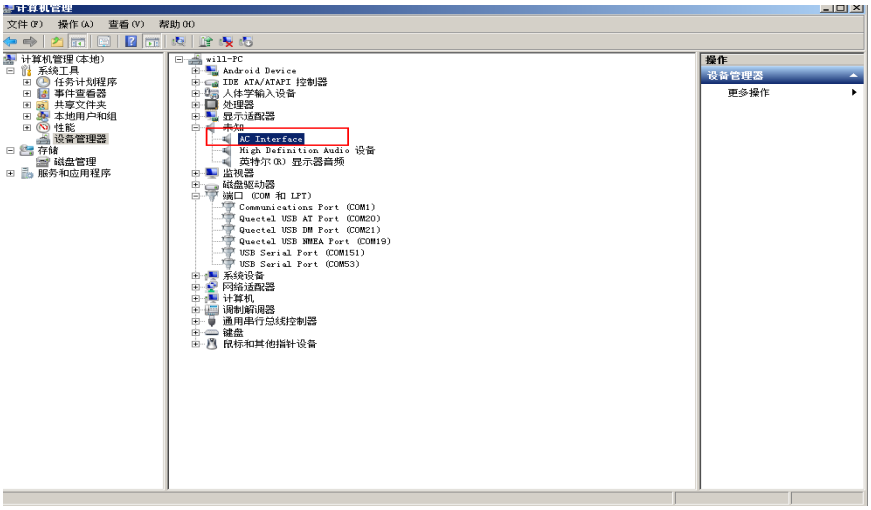
发送下面命令关闭 UAC，关闭 UAC 之后，会切换回 AG35 的 codec。

AT+QPCMV=0,2

OK

Windows 测试

at+qpcmv=1,2 操作之后，Windows 设备管理声卡里面会出现 uac 的声卡设备。




将此声卡设置问默认声卡，在控制面板-》声音去配置。

通过 AG35 的 AT 端口拨打手机，然后在 pc 上播放音乐，在手机能够听到音乐声

在 pc 上面打开录音程序（附件->录音机），可以录取手机端发送过来的声音。

(c)可以通过 tinyplay 播放音乐，但是由 AG35 UAC 支持 8K 采样率，16bit 数据长度，单声道，所以音频文件也必须是 8K 采样率，16bit 数据长度，单声道才能播放。（音源文件可以用格式工厂来进行转换）

tinyplay music.wav -D x -d 0

其中 x 是 AG35 UAC 的声卡是设备号，要根据系统中的声卡实际的设备号来设置

通过 tinycap 录音

tinycap rec.wav -D x -d 0 -c 1 -r 16000

## 三．UAC 的实现

### 1.USB audio class 代码流程分析

```
#define DECLARE_USB_FUNCTION_INIT(_name, _inst_alloc, _func_alloc)      \
    DECLARE_USB_FUNCTION(_name, _inst_alloc, _func_alloc)               \
    static int __init _name ## mod_init(void)                          \
    {                                                                   \
        return usb_function_register(&_name ## usb_func);              \
    }                                                                   \
    static void __exit _name ## mod_exit(void)                         \
    {                                                                   \
        usb_function_unregister(&_name ## usb_func);                    \
    }                                                                   \
    module_init(_name ## mod_init);                                    \
    module_exit(_name ## mod_exit)
```

通过这个宏变量先把 uac 那个重要函数注册 f_audio_alloc_inst, 和 f_audio_alloc,在 f_audio_alloc_inst 主要是指 uac 设备的相关的参数。

```
opts->req_playback_buf_size = UAC1_OUT_EP_MAX_PACKET_SIZE;
opts->req_capture_buf_size = UAC1_IN_EP_MAX_PACKET_SIZE;
opts->req_playback_count = UAC1_OUT_REQ_COUNT;
opts->req_capture_count = UAC1_IN_REQ_COUNT;
opts->audio_playback_buf_size = UAC1_AUDIO_PLAYBACK_BUF_SIZE;
opts->audio_capture_buf_size = UAC1_AUDIO_CAPTURE_BUF_SIZE;
opts->audio_playback_realtime = 1;
opts->sample_rate = UAC1_SAMPLE_RATE;
opts->fn_play = FILE_PCM_PLAYBACK;
opts->fn_cap = FILE_PCM_CAPTURE;
opts->fn_cntl = FILE_CONTROL;
return &opts->func_inst;
« end f_audio_alloc_inst »
```

这些参数 UAC1_SAMPLE_RATE ，采样率一般设置为 8K，16K，UAC1_AUDIO_PLAYBACK_BUF_SiZE,和 UAC1_AUDIO_CAPTURE_BUF_SIZE 一般设置为 16 的倍数，因为数据一般是 16bit 进行搬运的。

f_audio_alloc 这个函数作用非常重要的，audio->card.func.name = “g_audio” ,f_audio_set_alt,当 usb 插入的时候检测 usb 的设备主要是 UAC 设备，INIT_WORK 这个宏主要初始化 playback_work 和 capture_work 这两个重要的工作队列。

```
audio->card.func.bind = f_audio_bind;
audio->card.func.unbind = f_audio_unbind;
audio->card.func.get_alt = f_audio_get_alt;
audio->card.func.set_alt = f_audio_set_alt;
audio->card.func.setup = f_audio_setup;
audio->card.func.disable = f_audio_disable;
audio->card.func.free_func = f_audio_free;

control_selector_init(audio);

INIT_WORK(&audio->playback_work, f_audio_playback_work);
INIT_WORK(&audio->capture_work, f_audio_capture_work);
INIT_WORK(&audio->close_work, f_audio_close_work);
mutex_init(&audio->mutex);
```

F_audio_set_alt 函数中，req_playback_buf_size 正好在 f_audio_alloc_inst 初始化的时候赋给了这些值，HOST 端发过来的命令，模块收到这个命令就要打开 USB 设备，然后通过 HOST 发送过来的命令，来 schedule cpture_work 还是 playback_work.

```
req_playback_buf_size = opts->req_playback_buf_size;
req_capture_buf_size = opts->req_capture_buf_size;
req_playback_count = opts->req_playback_count;
req_capture_count = opts->req_capture_count;
audio_playback_buf_size = opts->audio_playback_buf_size;


atomic_set(&audio->online, 1);
if (intf == uac1_header_desc.bInterfaceNr[0]) {
    if (audio->alt_intf[0] == alt) {
        pr_debug("Alt interface is already set to %d. Do nothing.\n",
            alt);

        return 0;
    }

    if (alt == 1) {
        err = config_ep_by_speed(cdev->gadget, f, in_ep);
        if (err)
            return err;

        err = usb_ep_enable(in_ep);
        if (err) {
            pr_err("Failed to enable capture ep");
            return err;
        }
        in_ep->driver_data = audio;
        audio->capture_copy_buf = 0;
```

现在以 playback_work 被 schedule 来加下 播放的的大致流程。

在 f_audio_alloc 函数中 INIT_WORK(&audio->palyback_work,f_audio_palyback_work); 现在 playback_work 这个对工作队列被调度，执行 f_audio_palyback_work,在 f_audio_playback_work 中 会调用 gaudio_setup 这个函数来设置 ASLA 接口和准备 usb 传输这些设置包括 PCM，MIXer Or MIDI ALSA 这些设备在 USB gadget 使用 ,会调用 gaudio_open_snd_dev 来打开 ALSA PCM 和控制设备文件，并完成 pcm 和控制文件的初始化，在 gaudio_open_snd_dev 中调用 filp_open 来打开 pcm 的 playback 设备和设置 playback substream ， 同时也会调用 playback_default_hw_params，来设置 playback HW 参数比如采样率，channel 等，在 palyback_default_hw_pararm 其实就是调用类似 ioctl 方法.在这些都设置完成之后， 会调用 u_audio_playback_work 来通过 ALSA PCM 设备来播放 音频数据， 在 u_audio_playback_work 调用 gaudio_open_playback_streams 来打开 pcm playback /dev/snd/pcmC0D5p 和设置 substream 接下来就是调试 playback_prepare_params 这些 pcm 的接口这里就不分析了。以上就是 playback 的一个流程，capture 的流程跟 playback 的流程差不多这里就不分析了。

```
static void f_audio_playback_work(struct work_struct *data)
{
	struct f_audio *audio = container_of(data, struct f_audio,
					playback_work);
	struct f_audio_buf *play_buf;
	struct f_uac1_opts *opts = container_of(audio->card.func.fi,
					struct f_uac1_opts, func_inst);
	int audio_playback_buf_size = opts->audio_playback_buf_size;
	unsigned long flags;
	int res = 0;

	pr_debug("%s: started\n", __func__);
	if (! atomic_read(&audio->online)) {
		pr_debug("%s offline\n", __func__);
		return;
	}
	/* set up ASLA audio devices if not already done */
	mutex_lock(&audio->mutex);
	res = gaudio_setup(&audio->card);
	if (res < 0) {
		mutex_unlock(&audio->mutex);
		return;
	}
	mutex_unlock(&audio->mutex);

	spin_lock_irqsave(&audio->playback_lock, flags);
	if (list_empty(&audio->play_queue)) {
		pr_err("playback_buf is empty");
		spin_unlock_irqrestore(&audio->playback_lock, flags);
		return;
	}
	play_buf = list_first_entry(&audio->play_queue,
			struct f_audio_buf, list);
	list_del(&play_buf->list);
	spin_unlock_irqrestore(&audio->playback_lock, flags);

	pr_debug("play_buf->actual = %d", play_buf->actual);

	res = u_audio_playback(&audio->card, play_buf->buf,
				audio_playback_buf_size);
	if (res)
		pr_err("copying failed");
}
```

```
	return -ENOMEM;

_snd_pcm_hw_params_any(params);
_snd_pcm_hw_param_set(params, SNDRV_PCM_HW_PARAM_ACCESS,
		snd->access, 0);
_snd_pcm_hw_param_set(params, SNDRV_PCM_HW_PARAM_FORMAT,
		snd->format, 0);
_snd_pcm_hw_param_set(params, SNDRV_PCM_HW_PARAM_CHANNELS,
		snd->channels, 0);
_snd_pcm_hw_param_set(params, SNDRV_PCM_HW_PARAM_RATE,
		snd->rate, 0);
```

```
size_t u_audio_playback(struct gaudio *card, void *buf, size_t count)
{

#ifndef QUECTEL_UAC_FEATURE
        struct gaudio_snd_dev   *snd = &card->playback;
        struct snd_pcm_substream *substream = snd->substream;
        struct snd_pcm_runtime *runtime = substream->runtime;
        mm_segment_t old_fs;
        ssize_t result;
        snd_pcm_sframes_t frames;
        int err = 0;
#else
        struct gaudio_snd_dev   *snd ;
        struct snd_pcm_substream *substream;
        struct snd_pcm_runtime *runtime;
        mm_segment_t old_fs;
        ssize_t result;
        snd_pcm_sframes_t frames;
        int err = 0;
        if (! card->usb_snd_opened)
                return 0;

        snd = &card->playback;
        substream = snd->substream;
        runtime = substream->runtime;
#endif
        if (! count) {
                pr_err("Buffer is empty, no data to play");
                return 0;
        }
        if (! card->audio_reinit_playback) {
                err = gaudio_open_playback_streams(card);
                if (err) {
                        pr_err("Failed to init audio streams");
                        return 0;
                }
                card->audio_reinit_playback = 1;
        }
```

## 2.usb audio class 驱动的修改

a.在 kernel/driver/usb/gadget/android.c,文件中添加两个文件系统 show 和 store 接口，供应用层 enable 和 disable uac 接口。

```
static ssize_t audio_enable_show(struct device *dev,
                struct device_attribute *attr, char *buf)
{
        return snprintf(buf, PAGE_SIZE, "%d\n", audio_enable);
}

static ssize_t audio_enable_store(struct device *dev,
                struct device_attribute *attr, const char *buf, size_t size)
{
        int value;
        struct android_usb_function *f = dev_get_drvdata(dev);
        struct audio_function_config *config = f->config;

        if (sscanf(buf, "%d", &value) == 1) {
                if (value < 0 || value > 1) {
                        pr_err("audio_enable_store err!\n");
                        return -EINVAL;
                }

                audio_enable = value;

                quec_audio_enable (config->func, value);

                return size;
        }
```

在 quectel_audio_services.c 中对 生成的 sys/class/android_usb/f_audio/audio_enable 进行读写操作，当 audio_enbale 使能操作的时候，就是调用 audio_enable_store 函数中 quec_audio_enble

```
static void connect_current_voice_path(int connect) {
        connect_voice_path(cur_voice_path, connect);

        if (cur_voice_path == VOICE_PATH_AFE_PROXY) {
                if (connect)
                        system("echo 1 > /sys/class/android_usb/f_audio/audio_enable");
                else
                        system("echo 0 > /sys/class/android_usb/f_audio/audio_enable");
        }
}
```

在 f_uac1.c 源文件中实现了 quec_audio_enable 函数，在这个函数主要调用 u_uac1.c

quec_uac_open_snd_dev,在 quec_uac_open_snd_dev ,调用 gaudio_open_snd_dev(),这个函数主要 是打开 ALSA pcm 和控制一些设备文件。

```
int quec_uac_open_snd_dev(struct gaudio *card, int open);

int quec_audio_enable(struct usb_function *func, int enable)
{
        struct f_audio              *audio;

        if (! func) {
                return - 1;
        }

        audio = func_to_audio(func);

        return quec_uac_open_snd_dev(&audio- >card, enable);
}
#endif
```

在 u_uac1.h 中 主 要 修 改 UAC1_AUDIO_PLAYBACK_BUF_SIZE ， UAC1_AUDIO_CAPTURE_BUF_SIZE ， UAC1_SAMPLE_RAE 这几个参数， UAC1_SAMPLE_RARE 只能修改 8k 和 16k。BUF_SIZE 一般修改为 256 。 pcmCoD5P 是对应模块内的 pcm 设备。

```
#define QUECTEL_UAC_FEATURE //add yang.yang 2018-01-05 uac

#define FILE_PCM_PLAYBACK    "/dev/snd/pcmC0D5p"
#define FILE_PCM_CAPTURE     "/dev/snd/pcmC0D6c"
#define FILE_CONTROL         "/dev/snd/controlC0"

//#define FILE_PCM_PLAYBACK "/dev/snd/pcmC0D0p"
//#define FILE_PCM_CAPTURE  "/dev/snd/pcmC0D1c"
//#define FILE_CONTROL      "/dev/snd/controlC0"


#define UAC1_IN_EP_MAX_PACKET_SIZE  32
#define UAC1_OUT_EP_MAX_PACKET_SIZE 32
#define UAC1_OUT_REQ_COUNT       48
#define UAC1_IN_REQ_COUNT        4
#define UAC1_AUDIO_PLAYBACK_BUF_SIZE  320// 256 /* Matches
#define UAC1_AUDIO_CAPTURE_BUF_SIZE    320 //256    /* Mat
#define UAC1_SAMPLE_RATE         8000
/*
```

在 f_uac1.c 中 f_audio_playback_ep_complete，控制 audio_palyback_buf_size 和 copy_buf，当他们之间的差值小于 16bit 时候，就会把 play_queue 加入播放队中，会调用 f_audio_playback_work()函数，在这个函数中会调用 u_audio_playback(&audio->card, play_buf->buf,audio_playback_buf_size);最终完成播放的流程。

```
static int
f_audio_playback_ep_complete(struct usb_ep *ep, struct usb_request *req)
{
	struct f_audio *audio = req->context;
	struct usb_composite_dev *cdev = audio->card.func.config->cdev;
	struct f_audio_buf *copy_buf = audio->playback_copy_buf;
	struct f_uac1_opts *opts;
	int audio_playback_buf_size;
	unsigned long flags;
	int err;

	opts = container_of(audio->card.func.fi, struct f_uac1_opts,
			    func_inst);
	audio_playback_buf_size = opts->audio_playback_buf_size;

	if (! copy_buf)
		return - EINVAL;

	/* Copy buffer is full, add it to the play_queue */
	if (audio_playback_buf_size -  copy_buf->actual < req->actual) {
		pr_debug("audio_playback_buf_size %d - copy_buf->actual %d, req->actual %d",
			audio_playback_buf_size, copy_buf->actual, req->actual);
		spin_lock_irqsave(&audio->playback_lock, flags);
		if (! list_empty(&audio->play_queue) &&
				opts->audio_playback_realtime) {
			pr_debug("over-runs, audio write slow.. drop the packet\n");
			f_audio_buffer_free(copy_buf);
		} else {
			list_add_tail(&copy_buf->list, &audio->play_queue);
		}
```

对于 capture 的流程差不多，

```
static int
f_audio_capture_ep_complete(struct usb_ep *ep, struct usb_request *req)
{
	struct f_audio *audio = req->context;
	struct f_audio_buf *copy_buf = audio->capture_copy_buf;
	struct f_uac1_opts *opts = container_of(audio->card.func.fi,
				struct f_uac1_opts, func_inst);
	int audio_capture_buf_size = opts->audio_capture_buf_size;
	unsigned long flags;
	int err = 0;

	if (copy_buf == 0) {
		pr_debug("copy_buf == 0");
		spin_lock_irqsave(&audio->capture_lock, flags);
		if (list_empty(&audio->capture_queue)) {
			spin_unlock_irqrestore(&audio->capture_lock, flags);
			pr_debug("%s no data from Audio to send\n", __func__);
			schedule_work(&audio->capture_work);
			memset(req->buf, 0, opts->req_capture_buf_size);
			goto done;
		}
```

```
spin_lock_irqsave(&audio->capture_lock, flags);
if (!list_empty(&audio->capture_queue)) {
        spin_unlock_irqrestore(&audio->capture_lock, flags);
        pr_debug("%s !! buffer already filled\n", __func__);
        return;
}
spin_unlock_irqrestore(&audio->capture_lock, flags);

capture_buf = f_audio_buffer_alloc(audio_capture_buf_size);
if (capture_buf <= 0) {
        pr_err("%s: buffer alloc failed\n", __func__);
        return;
}

res = u_audio_capture(&audio->card, capture_buf->buf,
                audio_capture_buf_size);
if (res)
```

## 2.应用层的修改

主要在 quectel_record_play.c 的代码中，主要填了 quec_pcmv_rx_thread, 和 quec_pcmv_tx_thread,

在 quec_pcmv_rx_thread ,主要调用 quec_pcm_open ,quec_pcm_read，在 quec_pcmv_start 函数去创建线程去

```
static void *quec_pcmv_rx_thread(void *arg)
{
    char *buffer;
    struct pcm *pcm;
    unsigned int bufsize          = 0;

    if(! quectel_clt_set_mixer_value("MultiMedia1 Mixer VOC_REC_DL", 1, "1"))
    {
        printf("%s: quec pcmv enabl rx path fail.\n", __FUNCTION__);
        goto end_pcm_rx_thread;
    }

    pcm = quec_pcm_open("hw:0,0", PCM_NMMAP | PCM_IN | PCM_MONO, 8000, 1, SNDRV_PCM_FORMAT_S16_LE, I
    if(pcm == NULL)
    {
        fprintf(stderr, "open tx pcm device failed\n");
        goto close_pcm_rx_path;
    }

    bufsize = quec_get_pem_buffer_len(pcm);
    buffer = calloc(1, bufsize);
    if (! buffer)
    {
        fprintf(stderr, "could not allocate %d bytes\n", buffer);
        goto close_pcm_rx_device;
    }
    memset(buffer, 0, bufsize);

    pthread_detach(pthread_self());
    usleep(20000);

    while(quec_pcmv_started)
    {
        if(! quec_read_pcm(pcm, buffer, bufsize))
        {
```

```
int quec_pcmv_start(void)
{
    pthread_t thread_id;

    if(quec_pcmv_started)
    {
        printf("%s: pcmv has started.\n", __FUNCTION__);
        return TRUE;
    }

    quec_pcmv_started = TRUE;
    if(pthread_create(&thread_id, NULL, quec_pcmv_rx_thread, NULL) ! = 0)
    {
        printf("%s: create qpcmv_rx_thread fail.\n", __FUNCTION__);
        goto rx_thread_err;
    }

    if(pthread_create(&thread_id, NULL, quec_pcmv_tx_thread, NULL) ! = 0)
    {
        printf("%s: create qpcmv_tx_thread fail.\n", __FUNCTION__);
        goto tx_thread_err;
    }

    printf("%s: start pcmv sucess.\n", __FUNCTION__);
    return TRUE;
```