

Rockchip Linux UAC App开发指南

文件标识：RK-KF-YF-527

发布版本：V1.1.0

日期：2020-09-03

文件密级：□绝密 □秘密 □内部资料 ■公开

免责声明

本文档按“现状”提供，瑞芯微电子股份有限公司（“本公司”，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

版权所有 © 2020 瑞芯微电子股份有限公司

超越合理使用范畴，非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址：福建省福州市铜盘路软件园A区18号

网址：www.rock-chips.com

客户服务电话：+86-4007-700-590

客户服务传真：+86-591-83951833

客户服务邮箱：fae@rock-chips.com

前言

概述

本文主要描述了UVCApp应用各个模块的使用说明。

产品版本

芯片名称	内核版本
RV1126/RV1109	Linux 4.19

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

修订记录

版本号	作者	修改日期	修改说明
V1.0.0	周弟东	2020-08-24	初始版本
V1.1.0	何华	2020-09-03	添加uevent说明;修改UAC配置描述
V1.1.1	何华	2021-06-29	添加uac数据抓取和测试说明;删除过时的节点说明
V1.1.2	何华	2022-12-26	添加Q&A; 添加节点配置示例

目录

Rockchip Linux UAC App开发指南

- 简介
- 源码说明
- UAC框架流程
 - Slave(从)设备端的uevent事件
 - Master(主)设备开启/关闭放音(播放), Slave(从)设备收到的uevent事件
 - Master(主)设备开启/关闭录音, Slave(从)设备收到的uevent事件
 - 放音/录音设置采样率的uevent事件
 - 放音/录音设置音量大小和静音的uevent事件
 - UAC数据流
 - Master放音, Slave录音并播放
 - Slave mic录音, 数据发往Master流程
- 音频处理节点
 - 内置的音频节点
 - 音频节点的配置
 - 配置参数
 - 内嵌处理节点的配置
 - 录音 /放音节点的配置
 - 3A(skv)节点配置
 - eq/drc节点配置
 - hpf/lpf节点配置
 - resample节点配置
 - filter_volume
 - track_mode节点
 - chn_swap节点
 - 节点间的连接
 - 处理节点序号(id)
 - 外部程序获取节点的输出数据
 - 用户自定义节点获取数据(推荐)
 - 用户设置监听函数获取数据(不推荐)
- UAC json文件配置
- UAC外部参数设置
- uac节点, 数据的抓取
- uac录音/放音测试
 - 录音
 - 放音
- Q&A
 - 第三方算法和处理集成
 - uac断音/杂音问题

简介

uac_app 是基于RK自主研发多媒体播放器rockit，实现UAC功能，其主要作用是：

1. 实现uac驱动相关event事件监听，创建播放器，开启uac功能。
2. 调用rockit完成uac功能。

源码说明

```
├─ CMakeLists.txt
├─ configs
│   ├── file_read_usb_playback.json
│   ├── mic_recode_usb_playback.json
│   ├── usb_recode_speaker_playback.json
│   └── configs_skv.json
├─ doc
│   └── zh-cn
│       ├── resources
│       │   ├── kernel_config.png
│       │   ├── ubuntu_uac_capture.png
│       │   └── ubuntu_uac_playback.png
│       └── Rockchip_Quick_Start_Linux_UAC_CN.md
├─ src
│   ├── main.cpp
│   ├── uac_control.cpp
│   ├── uac_control.h
│   ├── uevent.cpp
│   └── uevent.h
└─ uac.sh
```

- 编译相关：
关：/external/uac_app/CMakeLists.txt、/buildroot/package/rockchip/uac_app/Config.in
uac_app.mk
- 入口：main.c
- uac脚本配置相关：uac.sh
- uac_app代码实现：uac初始化、uac uevent事件监听、rockit播放器开启和控制、音量事件监听和处理、采样率事件监听和处理、反初始化等处理：
 1. event.cpp：uac事件监听线程实现
 2. uac_control.cpp：播放器开启和控制和uac事件处理实现
 3. graph_control.cpp：uac处理节点参数设置。

UAC框架流程

UAC的具体描述和说明可以参考[Rockchip Quick Start Linux UAC CN.md](#)的UAC_APP章节，这里做一个流程梳理和总结。

UAC动作/命令的发起和停止，均是由一个设备发起，这个发起的设备，在本文中称为Master(主)设备，被动执行的设备，在本文中称为Slave(从)设备。以PC和RV1126为例，将RV1126连入PC。PC为Master设备，RV1126为Slave设备，任何的录音和放音动作，都是从PC端开启，RV1126遵照PC端的指令执行相应动作，其数据流如下：

Master端放音：Master设备(PC)写USB 声卡-->UAC驱动-->Slave设备(RV1126)读USB 声卡

Master端放音：Slave设备(RV1126)写USB 声卡-->UAC驱动-->Master设备(PC)读USB 声卡

Slave(从)设备端的uevent事件

由于Slave(从)设备永远都是被动执行Master(主)设备的命令，因此Slave(从)设备必须能够正确获取Master设备的命令和动作，这个是通过uevent事件来完成的。Rockchip的UAC驱动，会将Master端的命令/动作，转换成不同的uevent事件来通知Slave设备做出相应的响应。目前UAC驱动中完成的几种uevent事件：

- 录音/放音的开启和关闭uevent事件
- 录音/放音采样率设置uevent事件
- 录音/放音音量大小和静音的uevent事件

Master(主)设备开启/关闭放音(播放)，Slave(从)设备收到的uevent事件

Master(主)设备端打开USB声卡，并开始往USB声卡写数据。以Master(主)设备为Ubuntu PC为例，在PC端输入如下命令：

```
aplay -Dhw:1,0 -r 48000 -c 2 -f s16_le test.wav
```

其中，-Dhw:1,0表示Ubuntu PC端看到的uac设备的声卡card为1，device为0。

上述命令，PC会打开声卡hw:1,0，并以采样率48K，声道2播放test.wav文件。

Slave(从)设备端的uac_app会从uac驱动收到如下的uevent事件：

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_INTERFACE
strs[4] = STREAM_DIRECTION=OUT
strs[5] = STREAM_STATE=ON
```

说明：

strs[0] = ACTION=change 无特殊意义

strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget UAC1_Gadget表明当前使用的uac1协议，如果使用是的uac2协议，那么该处为UAC2_Gadget.

strs[3] = USB_STATE=SET_INTERFACE 表明当前的动作

strs[4] = STREAM_DIRECTION=OUT OUT表明数据流的方向(对Master设备来说)，OUT表明数据从Master通过UAC驱动发送到Slave设备，因此对于Slave(从)设备来说，需要从USB声卡录音/读取数据。

strs[5] = STREAM_STATE=ON ON表明当前动作为打开，即打开声卡

当uac_app收到如上uevent时，表明Master设备已开启了从usb声卡放音，此时，Slave端应建立对应的数据通路，从usb声卡读取音频数据。

当Master(主)设备关闭写USB声卡时，uac_app会从UAC驱动中获取如下的uevent事件：

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_INTERFACE
strs[4] = STREAM_DIRECTION=OUT
strs[5] = STREAM_STATE=OFF
```

说明：

strs[0]~strs[4]：同3.11的说明。

strs[5] = STREAM_STATE=OFF OFF表示当前动作为关闭，即关闭声卡。

当uac_app收到如上uevent时，表明Master设备已关闭了从usb声卡放音，此时，Slave端销毁对应的数据通路。

Master(主)设备开启/关闭录音，Slave(从)设备收到的uevent事件

Master(主)设备端打开USB声卡，并开始从USB声卡录制数据。以Master(主)设备为Ubuntu PC为例，在PC端输入如下命令：

```
arecord -Dhw:1,0 -f s16_le -r 48000 -c 2
```

其中，-Dhw:1,0表示Ubuntu PC端看到的uac设备的声卡card为1，device为0。

上述命令，PC会打开声卡hw:1,0，并以采样率48K，声道2播放test.wav文件。

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_INTERFACE
strs[4] = STREAM_DIRECTION=IN
strs[5] = STREAM_STATE=ON
```

strs[0]~strs[3]：同3.11的说明。

strs[4] = STREAM_DIRECTION=IN IN表明数据流的方向(相对于Master设备来说)，数据从Slave(从)设备通过USB声卡发往Master(主)设备。

strs[5] = STREAM_STATE=ON ON表示当前动作为打开声卡

当uac_app收到如上uevent时，表明Master设备已开启了从usb声卡录音，此时，Slave端需要建立对应的数据通路，将音频数据写往usb声卡。

当Master设备关闭从USB声卡录音时，uac_app收到的uevent事件如下：

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_INTERFACE
strs[4] = STREAM_DIRECTION=IN
strs[5] = STREAM_STATE=OFF
```

strs[0]~strs[4]：同3.13的说明。

strs[5] = STREAM_STATE=OFF OFF表示当前动作为关闭声卡。

当uac_app收到如上uevent时，表明Master设备已关闭了从usb声卡录音，此时，Slave端销毁对应的数据通路。

放音/录音设置采样率的uevent事件

当设置UAC设备支持多个采样率时(多采样率的配置在uac脚本中，见uac.sh)，需要获取Master设备端录音和放音时的音频数据的采样率，UAC驱动通过如下uevent事件来完成采样率的设置：

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_SAMPLE_RATE
strs[4] = STREAM_DIRECTION=IN
strs[5] = SAMPLE_RATE=48000
```

strs[0]~strs[2] 同3.1.1

strs[3] = USB_STATE=SET_SAMPLE_RATE 表明当前为设置采样率的uevent事件

strs[4] = STREAM_DIRECTION=IN IN表明数据流的方向(对于Master设备)，IN表明数据要从Slave设备发往Master设备，即Master端从usb录音，Slave端从usb放音。

strs[5] = SAMPLE_RATE=48000 48000表明数据的采样率为48K，该数值为Master端需要的音频数据的采样率，该值为Master端打开USB声卡时实际设置的采样率。

收到该uevent事件，说明Master设备已经开启了从usb录音，且所需音频数据的采样率为uevent上报的采样率，因此Slave设备端必须按照对应的采样率准备音频数据，并写往USB声卡。

同理，Master设备放音时，uac_app会收到如下的uevent事件：

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_SAMPLE_RATE
strs[4] = STREAM_DIRECTION=OUT
strs[5] = SAMPLE_RATE=48000
```

strs[0]~strs[3]，strs[5]同上。

strs[4] = STREAM_DIRECTION=OUT OUT表明数据从Master设备写往Slave设备。

需要注意的是：相同的流程，只有当采样率发生变化时，UAC驱动才会上报对应的采样率。

比如：Master进行了2次播放，数据流从Master-->Slave设备。第一次播放音频的采样率为假设为48K，第二次播放音频的采样率如果也为48K，因为2次的采样率相同，那么UAC驱动不会向uac_app上报第二次设置采样率的uevent事件。假如第二次播放设置的采样率为44.1K，因为2次的采样率不相同，uac_app会收到UAC上报2次设置采样率的uevent事件，将第一次的采样率设置为48K，第二次的设置为44.1K。同理Master设备的录音流程。因此，在应用(uac_app)中需要保存最后一次通报的采样率。

放音/录音设置音量大小和静音的uevent事件

当在Master端，调节UAC设备的音量大小或者设置uac设备静音时，UAC驱动会向Slave端会通过发送如下的uevent事件：

设置音量大小的uevent事件：

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadgets 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_VOLUME
strs[4] = STREAM_DIRECTION=OUT
strs[5] = VOLUME=72%
```

strs[0]~strs[2] 同3.1.1

strs[3] = USB_STATE=SET_VOLUME 表明当前动作为设置音量大小

strs[4] = STREAM_DIRECTION=OUT OUT表明设置当前数据流方向为Master发往Slave端。如果是Master从Slave设备录音，则该值为IN。

strs[5] = VOLUME=72% 该数值表示设置的音量大小百分比，合理的值为0~100%，demo中72%表明调整当前音量的72%。

设置/取消静音的uevent事件：

```
strs[0] = ACTION=change
strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
strs[2] = SUBSYSTEM=u_audio
strs[3] = USB_STATE=SET_MUTE
strs[4] = STREAM_DIRECTION=OUT
strs[5] = MUTE=1
```

strs[0]~strs[2] 同上

strs[3] = USB_STATE=SET_MUTE 表明当前动作为设置静音

strs[4] = STREAM_DIRECTION=OUT OUT表明设置当前数据流方向为Master发往Slave端。如果是Master从Slave设备录音，则该值为IN。

strs[5] = MUTE=1 MUTE=1表示Master设置了uac设备的静音，MUTE=0表明Master端取消了uac设备的静音。

由于Master设备端驱动的原因，需要注意如下：

- 只有uac1支持Master设置uac设备静音和音量大小，uac2不支持。
- 只有mac os，window系统才支持Master设置uac设备静音和音量大小，Linux和Android不支持。

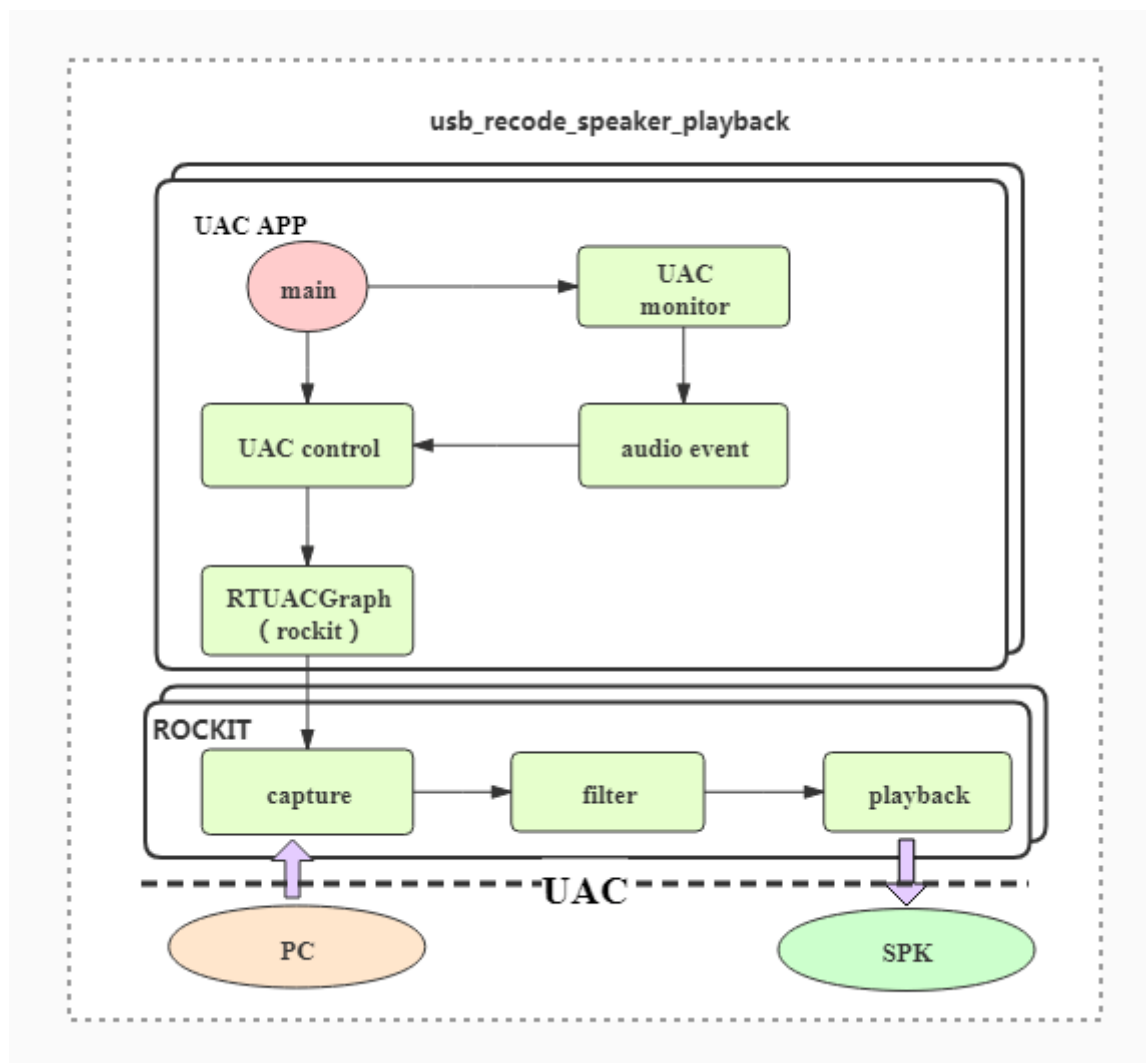
UAC数据流

UAC根据数据流向，可分为相互独立，互不影响的2个流程：

1. Master放音(写usb声卡)-->UAC驱动-->Slave设备(读usb声卡)
2. Slave设备(写usb声卡)-->UAC驱动-->Master设备((读usb声卡)

目前uac_app完成了以上2个流程的实现。其中，流程1在uac_app实现为Master写数据到usb声卡，Slave从usb声卡读取数据，并从speaker输出，记为Master放音Slave录音并播放；流程2在uac_app上实现为Slave设备mic录音，然后写usb声卡，Master端从usb声卡读取数据的流程，记为Slave mic录音，数据发往Master流程。

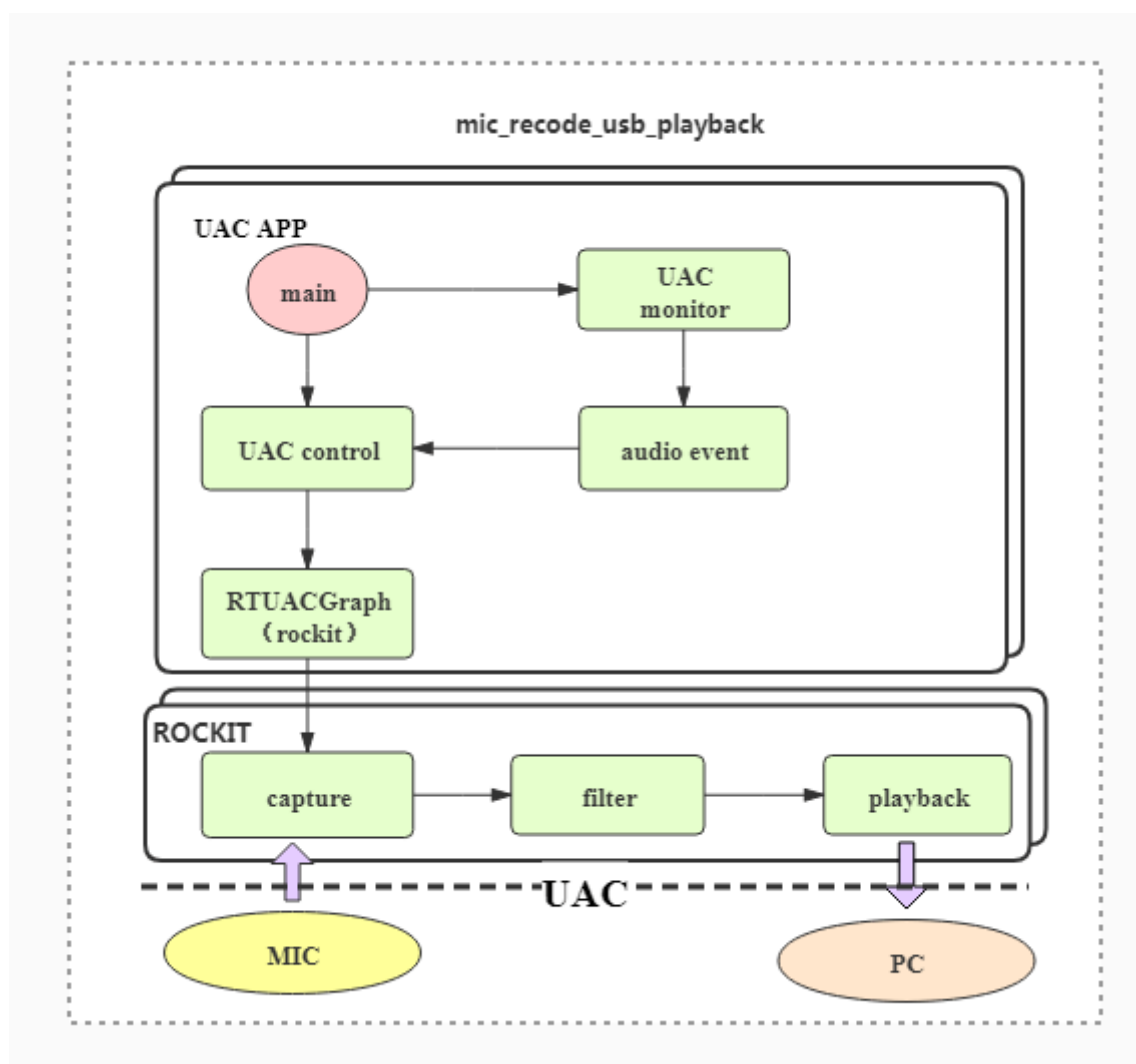
Master放音，Slave录音并播放



其框架图如上，描述如下：

1. Master端打开usb声卡准备放音。
2. UAC驱动发送对应的uevent到Slave端的uac_app。
3. Slave端uac_app收到对应uevent事件，配置usb_recode_speaker_playback.json给rockit，rockit按照usb_recode_speaker_playback.json的描述建立数据通路：usb声卡读取音频数据->各种音频算法处理-->speaker。
4. Master端向usb声卡发送数据(写usb声卡)，UAC驱动将Master端数据传送Slave端，Slave端从usb声卡读取数据。
5. 当Master退出当前放音流程时，UAC驱动发送对应的uevent到Slave端的uac_app，uac_app退出当前流程。

Slave mic录音，数据发往Master流程



其框架图如上，描述如下：

1. Master设备打开usb声卡准备录音。
2. UAC驱动发送对应的uevent到Slave端的uac_app。
3. uac_app收到对应uevent后，配置mic_recode_usb_playback.json文件给rockit，rockit按照mic_recode_usb_playback.json的描述，创建数据通路：Slave mic-->各种音频算法处理-->usb声卡
4. Slave端完成数据的录制和处理，写usb声卡。UAC驱动完成Slave端数据到Master端的传送。Master设备从usb声卡读取音频数据。
5. 当Master退出当前录音流程时，UAC驱动发送对应的uevent到Slave端的uac_app，uac_app退出当前流程。

rockit是Rockchip借鉴MediaPipe的思想，实现的一套支持UAC，UVC，AI和多媒体播放的多媒体库，支持跨Linux，Anroid等平台，这里不做过多描述。

音频处理节点

uac中功能的实现，都是基于处理节点。音频处理节点即用于完成某个功能的处理单元，常见的处理节点包括录音节点，放音节点，重采样节点，3A节点等。

内置的音频节点

node_name	说明
alsa_capture	alsa录音节点。
alsa_playback	alsa放音节点。
skv	3A算法节点。
eq_drc	均衡器(Equalizer)/动态范围控制(Dynamic Range Control)。
ahpf/alpf	高通滤波(High Pass Filter)/低通滤波器(Low Pass Filter)。
resample	音频重采样节点。
filter_volume	软件调节音量/静音节点。
track_mode	2声道(左右声道)音频数据处理。
chn_swap	音频声道数据复制，提取的处理。

当内嵌节点不能满足要求时，客户可自行定义处理节点，添加方式请参考 [第三方算法和处理集成](#) 添加自己的处理节点。

音频节点的配置

配置参数

rockit中定义并实现了节点四种参数类型的解析，描述如下：

参数类型	参数描述
node_opts	定义节点信息。
node_opts_extra	定义节点的扩展信息
stream_opts	定义输入/输出数据信息
stream_opts_extra	定义输入/输出数据扩展信息

```
"node_0": {
  "node_opts": {
    "node_name" : "alsa_capture"
  },
  "node_opts_extra": {
    "node_source_uri" : "hw:0,0",
    .....
  },
  "stream_opts": {
    "stream_fmt_in" : "audio:pcm_16",
    .....
  },
  "stream_opts_extra": {
    "opt_samaple_rate": 48000,
    .....
  },
},
},
```

以上四种类型没有严格的规定，因此客户自行定义参数，可添加到如上的任意类型中。rockit中常见的配置参数如下：

参数类型	配置参数	参数描述
node_opts	node_name	节点名称，rockit会根据节点名称查找并创建插件。目前rockit内部实现的音频插件见下个表格
node_opts_extra	node_source_uri	不同节点的含义不同，请见不同节点配置。
	node_buff_type	节点输出buffer来源： 0：节点内部自行分配； 1：外部代码分配并提供给节点。外部代码需要管理buffer的分配，释放，重新分配等操作。 音频节点配置0(内部分配)。
	node_buff_count	节点输出buffer的个数。 rockit内部是使用内存池(memory pool)方式管理内存，节点在创建时会创建对应数量buffer到内存池。节点运行时会首先从该内存池获取空闲buffer，数据处理完毕后会将该buffer送往与它相连的下级节点；为了避免相连节点串行运行，通常要求该值大于等于2。
	node_buff_size	节点单个输出buffer的大小，单位字节。 当前节点分配buffer总大小 = 输出buffer的个数 * 单个输出buffer的大小。
	node_buff_alloc_type	节点分配buffer的方式。 "malloc"表示使用malloc方式分配内存，即从堆中分配内存。
stream_opts_extra	opt_samaple_rate	音频数据采样率。
	opt_format	音频数据格式/位深度。 audio:pcm_8: 一个采样点8bits。 audio:pcm_16: 一个采样点16bits。 audio:pcm_24：一个采样点24bits。 audio:pcm_32:一个采样点32bits。 内置的3A、EQ/DRC、高通滤波，音量等节点只支持16bit数据，因此推荐整条uac通路设置16bit格式。
	opt_channel	声道数。
	opt_channel_layout	声道布局(layout)。 前缀int64表明该数据为int64类型。该值的每一个bit位表示一个声道，比如对于8声道的数据，其layout的二进制为: 0b' 11111111，对应的十进制为255，因此可将其设置为"int64:255"。注意，其值必须和opt_channel的声道数量相对应。
	opt_ref_channel_layout	回采数据的布局。 前缀int64表明该数据为int64类型。该值为音频aec算法的私有参数，用于标示出回采数据所在的声道。比如其值为"int64:63"，63的二进制可写成0b'111111，标示8channel的音频数据中，channel0~channel5为回采数据。

参数类型	配置参数	参数描述
	opt_rec_channel_layout	录音数据的布局。 前缀int64表明该数据为int64类型。该值为音频aec算法的私有参数，用于标示出录音数据所在的声道。比如其值为"int64:192"，192的二进制可写成0b'11000000，标示8channel的音频数据中，channel6，channel7为录音数据。
	opt_peroid_size	每次硬件中断处理音频数据的帧数。 alsa_capture/alsa_playback节点的私有参数，对应alsa hw paramter中的peroid_size。默认peroid_size=512帧。
	opt_peroid_count	处理完一个buffer数据所需的硬件中断次数。 alsa_capture/alsa_playback节点的私有参数，对应alsa hw paramter中的peroid_count。默认peroid_count=4。
	opt_alsa_mode	打开声卡的模式。 alsa_capture/alsa_playback节点的私有参数，对应alsa pcm_open中的mode。nonblock：非阻塞方式，按照轮休的方式读/写声卡。 interleaved：数据交替存放。 默认为block，unmmap，interleaved方式读/写数据。
	opt_audio_ppm	ppm使能开关。 1：打开ppm使能。 0：禁止ppm使能。 没做配置该参数时，默认为0。ppm的作用，请见 uac杂音/断音问题 usb与mic/spk时钟异源。
	opt_read_size	读取数据的大小，单位字节。 alsa_capture节点的私有参数。该值必须小于等于node_buff_size定义的buffer大小。
	opt_start_delay	启动传输的延时。 对应alsa sw的start_threshold，单位us。默认值： 录音：1(无延时)。 放音：(opt_peroid_size*opt_peroid_count)/2 帧数 对应的延时。
	opt_frames_count	当前节点一次处理需要的数据帧数。
stream_opts	stream_fmt_in	输入流数据格式。如audio:pcm_16。
	stream_fmt_out	输出流数据格式。如audio:pcm_16。
	stream_input	输入流名称。该值用于节点间的连接，见 节点间的连接 。
	stream_output	输出流名称。该值用于节点间的连接，见 节点间的连接 。

内嵌处理节点的配置

录音 / 放音节点的配置

【描述】

rockit内置的放音/录音节点。录音节点是指打开录音声卡读取音频数据的节点；放音节点是指音频通过声卡播放的节点。

【示例】

```
"node_0": {
  "node_opts": {
    "node_name"      : "alsa_capture"
  },
  "node_opts_extra": {
    "node_source_uri" : "default:CARD=rockchiprk809co",
    "node_buff_type"  : 0,
    "node_buff_count" : 4,
    "node_buff_size"  : 2048,
    "node_buff_alloc_type" : "malloc"
  },
  "stream_opts_extra": {
    "opt_audio_ppm"    : 1,
    "opt_samaple_rate" : 16000,
    "opt_format"       : "audio:pcm_16",
    "opt_channel"      : 2,
    "opt_channel_layout" : "int64:3",
    "opt_peroid_size"  : 256,
    "opt_peroid_count" : 4,
    "opt_read_size"    : 1024
  },
  "stream_opts": {
    "stream_fmt_in"    : "audio:pcm_16",
    "stream_fmt_out"   : "audio:pcm_16",
    "stream_output"    : "audio:pcm_0"
  }
},
```

```
"node_2": {
  "node_opts": {
    "node_name"      : "alsa_playback"
  },
  "node_opts_extra": {
    "node_source_uri" : "hw:1,0"
  },
  "stream_opts": {
    "stream_fmt_in"    : "audio:pcm_16",
    "stream_fmt_out"   : "audio:pcm_16",
    "stream_input"     : "audio:pcm_3"
  },
  "stream_opts_extra": {
    "opt_start_delay"  : 16000,
    "opt_alsa_mode"    : "nonblock",
    "opt_samaple_rate" : 16000,
    "opt_format"       : "audio:pcm_16",
    "opt_channel"      : 2,
  },
}
```

```

    "opt_channel_layout" : "int64:3"
}
}

```

【说明】

- "node_name"：定义为"alsa_capture"和"alsa_playback"表示内置的录音/放音节点。
- "opt_alsa_mode"：热插拔设备(比如usb，hdmi)等可由用户操作(比如拔插)而导致音频录音/放音停止的声卡，需要配置nonblock方式，否则容易导致读取/播放数据卡住。当没有该配置，声卡默认为阻塞读写。
- 录音节点只有输入流("stream_output")，没有输出流("stream_input")。示例中定义了录音声卡的参数，录制16K 2chns pcm16的数据。buffer一次读取数据为1024字节(opt_read_size)，即硬件中断一次(peroid_size=256帧)对应的大小。
- 放音节点只有输出流("stream_input")，没有输入流("stream_output")。示例中定义了放音声卡的默认参数(16K 2chns pcm16 非阻塞模式)。
- "opt_start_delay"：定义了放音声卡启动传输时的数据缓冲(单位us)。16K 16000us的数据量为256帧，即放音声卡写满256帧数据后启动传输/播放。
- "opt_peroid_size"：定义硬件中断一次的帧数，对应alsa的peroid_size。当没有该配置时，默认为256帧。
- "opt_peroid_count"：定义驱动中缓存buffer数量。驱动中定义的buffer缓存数据的总大小(帧) = "opt_peroid_size" * "opt_peroid_count"。当没有该配置时，默认缓存buffer数量为4。
- "node_source_uri"：配置录音/放音声卡名。

物理声卡名，指的是直接使用"hw:x,y"作为声卡名，其中x表示声卡序号，y表示该声卡下的设备y。比如hw:0,0表示声卡0设备0。虚拟声卡名，指的是使用"default:CARD=rockchiprk809co"来配置声卡名。通常不能直接使用虚拟声卡名来配置声卡，除非在alsa的相关配置文件中有定义相关的虚拟声卡。以RV1126 SDK板为例，虚拟声卡rockchiprk809co定义在/usr/share/alsa/cards/rockchip_rk809-.conf中。客户如想定义自己的虚拟声卡，可在asound.conf中定义，该文件会在开启启动时，由alsa自动加载并解析。比如在/etc/目录建立asound.conf文件，并填入如下：

```

pcm.my_card {
    .....
    slave {
        pcm "hw:0,0"
    }
}

```

如上示例中，定义了放音声卡"my_card"，并最终映射到物理声卡hw:0,0上。因此当应用使用"my_card"打开声卡时，实际最终alsa会打开声卡"hw:0,0"，"my_card"相当于"hw:0,0"声卡的别名。

使用cat /proc/asound/cards命令，可以看到声卡名和声卡序号的对应关系。以RV1126 SDK板为例，信息如下：

```

[root@RV1126_RV1109:/]# cat /proc/asound/cards
0 [rockchiprk809co]: rockchip_rk809- - rockchip_rk809-codec
                      rockchip_rk809-codec
1 [UAC1Gadget      ]: UAC1_Gadget - UAC1_Gadget
                      UAC1_Gadget 0
7 [Loopback        ]: Loopback - Loopback
                      Loopback 1

```

如有3个声卡信息。声卡0为rockchiprk809co，其声卡下挂载着1个录音声卡pcm0c(mic)和一个放音声卡pcm0p(喇叭)。使用"hw:0,0"放音时，实际使用是声卡0的pcm0p放音。使用"hw:0,0"录音时，实际使用的是声卡0的pcm0c录音。

```
[root@RV1126_RV1109:/proc/asound/card0]# ls -l
total 0
-r--r--r-- 1 root root 0 Aug  4 09:07 id
dr-xr-xr-x 3 root root 0 Aug  4 09:07 pcm0c
dr-xr-xr-x 3 root root 0 Aug  4 09:07 pcm0p
```

可通过如下的命令，查看某声卡x的放音和录音的参数，其中x表示具体的声卡序号：

```
cat /proc/asound/cardx/pcm0p/sub0/hw_params
cat /proc/asound/cardx/pcm0c/sub0/hw_params
```

声卡1为uac声卡，其在uac功能使能后显示。

pcm0c为uac的录音声卡，可通过该声卡读取从Master发送的音频数据。

pcm0p为uac的放音声卡，可通过该声卡将uac设备的音频数据发送给Master端。

```
[root@RV1126_RV1109:/proc/asound/card1]# ls -l
total 0
-r--r--r-- 1 root root 0 Aug  4 09:13 id
dr-xr-xr-x 3 root root 0 Aug  4 09:13 pcm0c
dr-xr-xr-x 3 root root 0 Aug  4 09:13 pcm0p
```

【注意】

uac声卡下并不总是挂载有uac的录音和放音声卡。uac录音和放音声卡的使能，是由uac配置节点的x_chmask值决定(x为c或者p，详情请见《Rockchip_Quick_Start_Linux_UAC_CN.pdf》UAC应用配置章节)。

3A(skv)节点配置

【描述】

rockchip内置的3A处理节点，它包含了AEC(回声消除)，AGC(增益控制)，ANR(噪音消除)，BF(波束)，DOA(声源定位)等功能。

【示例】

```
"node_2": {
  "node_opts": {
    "node_name"      : "skv"
  },
  "node_opts_extra": {
    "node_source_uri" : "/oem/usr/share/uac_app/configs_skv.json",
    .....
  },
  "stream_opts_extra": {
    "opt_sample_rate": 16000,
    "opt_format"      : "audio:pcm_16",
    "opt_channel"     : 4,
    "opt_ref_channel_layout" : "int64:12",    // 回采声道的布局
    "opt_rec_channel_layout" : "int64:3",     // 录音声道的布局
    "opt_channel_layout"  : "int64:15"
```



```

    },
    "stream_opts": {
        .....
    }
},

```

【说明】

- "node_name": "skv" 表示当前节点为skv (3A) 节点。
- "node_source_uri": "/oem/usr/share/uac_app/configs_skv.json"。 定义skv算法的参数文件所在路径和文件名。

```

{
  "skv_configs": {
    "aec": {
      "status" : "enable",
      "drop_ref_channel" : 0,
      "aec_mode" : "delay",
      "delay_len" : 0,
      "look_ahead" : 0
    },
    "bf": {
      "status" : "enable",
      "targ" : 0,
      "drop_ref_channel" : 0
    },
    "fast_aec": {
      "status" : "enable:
    },
    "agc": {
      "status" : "enable",
      "attack_time" : "float:80.0",
      "release_time" : "float:1000.0",
      "attenuate_time" : "float:2000.0",
      "max_gain" : "float:100.0",
      "max_peak" : "float:26000.0"
    },
    .....
  }
}

```

"status": "enable" 表示当前算法开启, "status": "disable" 表示当前算法关闭。

字符串"aec", "bf", "fast_aec", "agc"等: 表明当前音频算法, 注意该值/字符串需和代码中的匹配, 不能修改。

其他参数: 音频算法的私有接口, 以"attack_time": "float:80.0"进行说明, 该设置为agc算法的私有设置。

- "opt_samaple_rate": 16000, 定义skv支持的采样率。skv算法库只支持固定采样率的数据(比如16K)。
- "opt_rec_channel_layout": "int64:3", 定义输入数据录音通道, "int64:3"二进制为0b'0011, 表示chn0和chn1为录音数据, 录音声道数为2。skv算法库在编译时, 固定了其支持的录音声道数(比如声道数=2)。当需要支持其他的录音声道数时, 需要修改算法库中支持的声道数并更新算法库。
- "opt_ref_channel_layout": "int64:12", 定义输入数据回采通道, "int64:12"二进制为0b'1100, 表示chn2和chn3为回采数据, 回采声道数为2。skv算法库支持任意回采声道数。

- "opt_channel" : 4 和 "opt_channel_layout" : "int64:15" 定义skv输出流的声道数和布局。通常情况下，输入流的总声道数和总声道布局等于录音加上回采的声道数和布局。某些特殊条件下，总的声道数大于录音和回采数据的声道数，比如如下配置：

```
"stream_opts_extra": {
  "opt_sample_rate": 16000,
  "opt_format"      : "audio:pcm_16",
  "opt_channel"     : 6,
  "opt_ref_channel_layout" : "int64:12", // 0b' 1100
  "opt_rec_channel_layout" : "int64:3",  // 0b' 0011
  "opt_channel_layout"  : "int64:63"    // 0b' 111111
},
```

"opt_channel" :6 和 "opt_channel_layout" : "int64:63" 定义输入流为16K 6声道数据，其中chn4和chn5为无效数据(或静音数据)。

"opt_rec_channel_layout" : "int64:3" 定义录音数据为chn0，chn1。

"opt_ref_channel_layout" : "int64:12" 定义回采数据为chn2，chn3。

为了降低chn4，chn5数据对3A算法的处理，以及降低3A处理cpu的消耗，skv处理节点会首先按照"opt_ref_channel_layout"和"opt_rec_channel_layout"的定义，将6chns输入数据提取成4chn数据(只包含chn0~chn4的数据)，然后再送3A算法处理。

- 3A算法为纯软件处理，因此回采和录音声道越多，cpu消耗越大。客户需要评估其产品最复杂场景下的cpu使用率。
- 该节点不支持程序运行过程中，通过invoke接口动态修改3A算法参数。

eq/drc节点配置

【描述】

EQ(Equalizer)用于调整各频段信号的增益值，rockchip提供的EQ库支持8 EQ和10 EQ；DRC(Dynamic Range Control)主要用于对音频输出信号进行动态控制。

【示例】

```
"node_1": {
  "node_opts": {
    "node_name"      : "eq_drc"
  },
  "node_opts_extra": {
    "node_source_uri" : "/oem/usr/share/uac_app/eqdrc_configs_44100_2.bin",
    .....
  },
  "stream_opts": {
    .....
  },
  "stream_opts_extra": {
    "opt_frames_count": 256,
    .....
  }
},
```

【说明】

- "node_name" : "eq_drc" 表示当前节点为eq/drc处理节点。

- "node_source_uri" : "/oem/usr/share/uac_app/eqdrc_configs_44100_2.bin" 定义eq/drc算法参数文件所在路径和文件名。该配置文件需要使用rockchip提供的工具生成。注意该参数文件只支持其设定采样率的音频输入，比如如上demo中只支持44.1k音频数据的输入，当输入数据为48k时，需要使用48k对应参数文件。
- "opt_frames_count": 256 表示eq/drc节点一次处理数据的帧数。该帧客户可按照需求设定，通常建议设置成eq/drc前级节点送入数据的帧数。

hpf/lpf节点配置

【描述】

rockit内置的高通/低通滤波节点。

滤波器	支持的采样率	滤波频率(hz)
alpf(低通)	16K, 32K, 44.1K, 48K	100,150,200,250,300,350,400
ahpf(高通)	16K, 32K, 44.1K, 48K	100,150,200,250,300,350,400

【示例】

```
"node_1": {
  "node_opts": {
    "node_name"      : "ahpf"    // 音频高通滤波
  },
  .....
  "stream_opts_extra": {
    "opt_freq_hz": 200,    // 滤波频率
    .....
  }
},
```

【说明】

- "node_name": "ahpf" 表示当前节点为高通滤波节点，其中"ahpf"表示高通滤波，"alpf"表示低通滤波。
- "opt_freq_hz": 200 表示设置的滤波频率，客户可根据自己的需求，选择支持的频率进行设置。当前为高通率器时，表示滤除该值以下频率的数据；当为低通滤波时，表示滤除该值以上频率的数据。
- 该节点不支持程序运行过程中，通过invoke接口设置过来频率和采样率。

resample节点配置

【描述】

rockit内嵌的重采样节点，支持音频任意采样率的转换；支持最多8声道数据的声道转换；不同数据格式的转换(8bit, 16bit, 24bit, 32bit, float等)。

【示例】

```

"node_1": {
  "node_opts": {
    "node_name"      : "resample"
  },
  .....
  "stream_opts_extra": {
    "opt_samaple_rate": 48000,
    "opt_format"      : "audio:pcm_16",
    "opt_channel"     : 2,
    "opt_channel_layout" : "int64:3"
  }
},

```

【说明】

- "node_name": "resample" 表示当前节点为重采样节点。
- "opt_samaple_rate": 48000 配置resample节点输出数据的采样率，示例中resample输出48K采样率的数据。
- "opt_format" : "audio:pcm_16" 配置resample节点输出数据的格式，示例中resample输出16 bit数据(short)。
- "opt_channel" : 2, 配置resample节点输出数据的声道数，示例中resample输出2 声道数据。
- 该节点支持程序运行过程中，通过invoke接口输出数据的采样率，声道数和格式。

filter_volume

【描述】

rockit内部内置的音量调节和静音节点。

	取值
mute	0: 非静音。 1: 静音。
volume	[0.0, 1.0]

【示例】

```

"node_1": {
  "node_opts": {
    "node_name"      : "filter_volume"
  },
  .....
  "stream_opts_extra": {
    "opt_volume" : "float:1.0",
    "opt_mute"   : 0,
    "opt_format" : "audio:pcm_16",
    .....
  },
},

```

【说明】

- "node_name": "filter_volume" 表示当前节点为音量节点。
- "opt_volume": "float:1.0" 表示当前音量为最大值1.0，当音量大小设置为0.0，相当于静音。
- "opt_mute" : 0 0表示当前不静音，1 表示静音。

- 当"opt_volume" 和"opt_mute"不配置时，默认为非静音状态，音量大小为1.0(最大)。
- 该节点音量大小和静音状态，只影响使用该节点的流程，并非设置alsa或者硬件的音量。
- 音量/静音节点只支持16bit数据的处理。
- 该节点支持程序运行过程中，通过invoke接口修改音量大小和静音状态。

track_mode节点

【描述】

rockit内置的2声道数据的处理节点。该节点只支持2声道16bit的数据的输入输出。

模式	说明
normal	不处理，2声道数据正常数据。
both_left	右声道数据赋值为左声道数据，即输出的2声道数据均为左声道数据。
both_right	左声道数据赋值为右声道数据，即输出的2声道数据均为右声道数据。
exchange	左右声道数据交换，即左声道为原来右声道数据，右声道为原来左声道数据。
mix	左右声道数据混音后输出。
left_mute	左声道数据静音，右声道数据保持不变。
right_mute	右声道数据静音，左声道数据保持不变。
both_mute	左右声道数据静音。

【示例】

```
"node_1": {
  "node_opts": {
    "node_name"      : "track_mode"
  },
  .....
  "stream_opts_extra": {
    "opt_track_mode" : "normal",
    "opt_samaple_rate": 48000,
    "opt_format"      : "audio:pcm_16",
    "opt_channel"      : 2,
    "opt_channel_layout" : "int64:3"
  }
},
```

【说明】

- "node_name": "track_mode" 表示当前节点为2声道数据节点。
- 该节点支持2声道，16bit数据的处理。
- 该节点支持程序运行过程中，通过invoke接口动态设置模式。

chn_swap节点

【描述】

rockit内置的用于复制、提取、扩展声道数据的节点。该节点支持任意采样率，16bit音频数据的处理。

【示例1】

```

"node_1": {
  "node_opts": {
    "node_name"      : "chn_swap"
  },
  .....
  "stream_opts_extra": {
    "opt_proc_method" : "chns_drop_swap",
    "opt_in_chns"     : "12 chn;map:1,3,5,7,9,11,2,4,6,8,10,12",
    "opt_out_chns"    : "8 chn;map:1,2,3,4,5,6,9,10",
    "opt_sample_rate" : 16000,
    "opt_format"      : "audio:pcm_16",
    "opt_channel"     : 8,
    "opt_channel_layout" : "int64:255"
  }
},

```

【说明】

- "node_name": "chn_swap" 表示当前节点为chn_swap节点。
- "opt_proc_method": "chns_drop_swap" 定义当前节点的处理方法。目前就定义了"chns_drop_swap"。
- "opt_in_chns" : "12 chn;map:1,3,5,7,9,11,2,4,6,8,10,12" 定义该节点输入数据(流)的声道数和声道布局。示例中输入流为12声道数据。
- "opt_out_chns" : "8 chn;map:1,2,3,4,5,6,9,10" 定义该节点输出数据(流)的声道数和声道数。示例中输出流为8声道(8 chn)数据，丢掉了声道7,8,11,12，且对数据流中的声道进行了重排(见map:1,2,3,4,5,6,9,10)。
- 该节点只支持16bit的数据，因此定义数据格式为"opt_format": "audio:pcm_16"。
- "opt_channel" : 8 定义该节点输出的总声道数，需要与opt_out_chns定义的声道数相等。
- 该节点不支持程序运行过程中，通过invoke接口动态设置参数。

【示例2】

```

"node_1": {
  "node_opts": {
    "node_name"      : "chn_swap"
  },
  .....
  "stream_opts_extra": {
    "opt_proc_method" : "chns_drop_swap",
    "opt_in_chns"     : "1 chn;map:1",
    "opt_out_chns"    : "2 chn;map:1,1",
    "opt_sample_rate" : 48000,
    "opt_format"      : "audio:pcm_16",
    "opt_channel"     : 2,
    "opt_channel_layout" : "int64:3"
  }
},

```

【说明】

- "opt_in_chns" : "1 chn;map:1" 表明当前输入数据为1声道数据。
- "opt_out_chns" : "2 chn;map:1,1" 表达当前输出数据为2声道数据，且该2声道数据均被置位为输入的1声道数据。

节点间的连接

【描述】

节点A输出的数据，送往节点B中处理，即节点A的输出作为节点B的输入，称为节点A和节点B相连接。rockit中不同处理节点，是通过stream_input/stream_output标记(字符串)连接的。stream_input表示当前节点的输入流，除首节点(通常是录音节点)外，其余节点都有输入流；stream_output表示当前节点的输出流，除最后节点(通常为放音节点)外，其余节点都有输出流。

【示例1】

```
{
  "pipe_0": {
    "node_0": {
      "node_opts": {
        "node_name"      : "alsa_capture"
      },
      .....
      "stream_opts": {
        .....
        "stream_output"  : "audio:pcm_0"
      }
    },
    "node_1": {
      "node_opts": {
        "node_name"      : "skv"
      },
      .....
      "stream_opts": {
        .....
        "stream_input"   : "audio:pcm_0",
        "stream_output"  : "audio:pcm_1"
      }
    },
    "node_2": {
      "node_opts": {
        "node_name"      : "alsa_playback"
      },
      .....
      "stream_opts": {
        .....
        "stream_input"   : "audio:pcm_1"
      }
    }
  }
}
```

- 示例中定义了3个节点，分别为录音节点(alsa_capture)，3A处理节点(skV)以及放音节点(alsa_playback)。示例流程为录音节点的数据进3A处理后，从放音节点播放出来。
- alsa_capture作为当前流程数据的输入节点，因此它只有输出流，输出流名为"audio:pcm_0"。
- 3A处理节点的输入流名为"audio:pcm_0"，其名称和 alsa_capture节点的输出流名一致，即 alsa_capture节点数据的输出作为3A节点数据的输入；3A处理节点的输出流名为"audio:pcm_1"，它表示3A节点将数据处理完毕后，将数据送往输出流"audio:pcm_1"。
- alsa_playback为当前流程的最后节点，因此它只有输入流"audio:pcm_1"，没有输出流，且其输入流名和3A输出流名一致，即3A节点的输出作为alsa_playback节点的输入，表示alsa_playback播放3A节点处理后的数据。

【示例2】

```
{
  "pipe_0": {
    "node_0": {
      "node_opts": {
        "node_name"      : "alsa_capture"
      },
      .....
      "stream_opts": {
        .....
        "stream_output"  : "audio:pcm_0"
      }
    },
    "node_1": {
      "node_opts": {
        "node_name"      : "skv"
      },
      .....
      "stream_opts": {
        .....
        "stream_input"   : "audio:pcm_0",
        "stream_output"  : "audio:pcm_1"
      }
    },
    "node_2": {
      "node_opts": {
        "node_name"      : "filter_volume"
      },
      .....
      "stream_opts": {
        .....
        "stream_input"   : "audio:pcm_0"
        "stream_output"  : "audio:pcm_2"
      }
    },
    .....
  }
}
```

- 示例中定义了多个节点，分别为录音节点(alsa_capture)，3A处理节点(skV)以及音量静音/设置节点(filter_volume)，其他节点省略。
- alsa_capture作为当前流程数据的输入节点，因此它只有输出流，输出流名为"audio:pcm_0"。
- 3A节点的输入流名为"audio:pcm_0"，其名称和 alsa_capture节点的输出流名一致，即 alsa_capture节点数据的输出作为3A节点数据的输入；3A处理节点的输出流名为"audio:pcm_1"，它表示3A节点将数据处理完毕后，将数据送往输出流"audio:pcm_1"。
- filter_volume节点的输入流名为"audio:pcm_0"，其名称和 alsa_capture节点的输出流名一致，即 alsa_capture节点数据的输出作为filter_volume节点数据的输入；filter_volume节点的输出流名为"audio:pcm_2"。
- 上述示例中，alsa_capture节点的数据被同时送给了3A节点和音量处理节点。当一份数据被送往多个节点时，rockit中实际只会保留一份音频数据，保留该音频数据的buffer，会通过应用计数的方式，送往多个节点作为处理节点的输入，处理节点在使用完毕该buffer后，引用计数会减1。当引减为1时表示所有处理节点使用完毕，则该buffer会归还给alsa_capture节点，以用于后续的录音。

处理节点序号(id)

rockit中没有对处理节点的序号(id)做特定要求，只要符合如下条件：

- 处理节点的命名为“node_x”，x可以为任意数字，不要求前后相连的节点序号连续。如下示例为一个正确的配置：

```
{
  "pipe_0": {
    "node_1": {
      .....
      "stream_opts": {
        .....
        "stream_input"    : "audio:pcm_0"
      }
    },
    "node_3": {
      .....
      "stream_opts": {
        .....
        "stream_input"    : "audio:pcm_0",
        "stream_output"   : "audio:pcm_1"
      }
    },
    "node_5": {
      .....
      "stream_opts": {
        .....
        "stream_input"    : "audio:pcm_1",
        "stream_output"   : "audio:pcm_2"
      }
    }
  }
}
```

- 不允许一个处理流程中(pipe)的节点重名。比如如下节点配置中node_1重名，运行时会出现问题：

```
{
  "pipe_0": {
    "node_1": {
      .....
    },
    "node_3": {
      .....
    },
    "node_1": {
      .....
    },
    .....
  }
}
```

外部程序获取节点的输出数据

用户自定义节点获取数据(推荐)

- 用户可自定义节点，自定义节点的方法请见 [第三方算法和处理集成](#)。
- 将自定义节点配置到json文件中，放置到想监听节点的后面。
- 自定义的process函数中可访问到前级节点送入的数据，访问方法请见[第三方算法和处理集成](#)中的介绍。需要注意的时，当前自定义节点对于数据的访问，修改等，都可能对下级节点产生影响。因此如用户不像修改当前数据时，最好能对当前数据备份，并及时将原始数据送入下一级节点。

如下为用户自定义 node_10用于访问node_1输出数据的配置。

```
{
  "pipe_0": {
    "node_1": {
      .....
      "stream_opts": {
        .....
        "stream_output" : "audio:pcm_0"
      }
    },
    "node_2": {
      .....
      "stream_opts": {
        .....
        "stream_input" : "audio:pcm_0",
        "stream_output" : "audio:pcm_1"
      }
    },
    .....
  }
}
```

```
{
  "pipe_0": {
    "node_1": {
      .....
      "stream_opts": {
        .....
        "stream_output" : "audio:pcm_0"
      }
    },
    "node_10": { // 监听节点
      .....
      "stream_opts": {
        .....
        "stream_input" : "audio:pcm_0",
        "stream_output" : "audio:pcm_1"
      }
    },
    "node_2": {
      .....
      "stream_opts": {
        .....
        "stream_input" : "audio:pcm_1",
        "stream_output" : "audio:pcm_2"
      }
    }
  }
}
```

```

    }
    },
    .....
}
}

```

用户设置监听函数获取数据(不推荐)

rockit也支持外部程序通过设置监听函数(回调函数)的方式，获取某处理节点的输出数据。但是需要说明的是，因为大部分的处理节点在处理数据时，会修改并设置buffer中有效数据的长度，偏移和音频参数等，因此多个程序并发访问数据，可能会存在问题，因而通常不推荐使用这种方法来获取数据。

监听函数原型：RT_RET (*callback)(RTMediaBuffer *buffer)；

【参数】：保存音频数据的buffer。

【返回值】：0(RT_OK)表示成功。该返回值内部没有使用，可总是返回0。

【示例】：外部程序获取3A节点的输出数据。配置如下：

```

{
    "pipe_0": {
        "node_0": {
            "node_opts": {
                "node_name"      : "alsa_capture"
            },
            .....
            "stream_opts": {
                .....
                "stream_input"   : "audio:pcm_0",
                "stream_output"  : "audio:pcm_1"
            }
        },
        "node_1": {
            "node_opts": {
                "node_name"      : "skv"
            },
            "stream_opts": {
                .....
                "stream_input"   : "audio:pcm_1",
                "stream_output"  : "audio:pcm_2"
            }
        },
        .....
    }
}

```

- skv节点输出流 名称为"audio:pcm_2"。
- 在uac_app代码中，设置对输出流"audio:pcm_2"的监听函数/回调函数。

```

// callback function
RT_RET uac_get_buffer_callback(RTMediaBuffer *buffer) {
    if (buffer == NULL)
        return RT_ERR_NULL_PTR;
}

```

```

// this is a demo to get the data from RTMediaBuffer
int length = buffer->getLength(); // the length of data
int offset = buffer->getOffset(); // the offset length
char *address = (char*)buffer->getData(); // the start address of
buffer
char *data = &address[offset]; // the start address of data
memcpy(mybuffer, data, length); // copy data to my buffer

// this is a demo to get the parameters from RTMediaBuffer
int samplerate = 0;
if (meta->findInt32("opt_samplerate", &samplerate)) {
    ALOGD("find samplerate = %d", samplerate);
}

int channels = 0;
if (meta->findInt32("opt_channel", &channels)) {
    ALOGD("find channels = %d", channels);
}

// release to this buffer
buffer->release();
return RT_OK;
}

int uac_start(int type) {
    if (gUAControl == NULL)
        return -1;

    .....
    uac->autoBuild(config);

    // add callback function to output stream "audio:pcm_2"
    if (type == UAC_STREAM_PLAYBACK) {
        std::string callbackStreamId = std::string("audio:pcm_2");
        uac->observeOutputStream(callbackStreamId, uac_get_buffer_callback);
    }

    .....
    return 0;
}

```

1. 查看想要监听节点的输出流名字，**只有含有输出流的节点才能设置监听。**
2. 使用RTTaskGraph的observeOutputStream函数设置监听/回调函数。
observeOutputStream函数的第一个参数，为待监听的输出流名字，第二个参数为监听函数指针。
3. 程序运行时，处理节点会将处理成功后的数据，同时送给与它相连的节点和监听/回调函数。
因为监听函数和后级节点拿到的都是同一个RTMediaBuffer，因此在回调函数不能修改RTMediaBuffer中的任何数据和参数，否则会导致后级节点拿到的数据和参数也会发生变化；
监听函数获取/访问RTMediaBuffer的数据和参数时，不应长时间占据该buffer；监听函数使用RTMediaBuffer完毕后，需要调用RTMediaBuffer的release函数将该buffer归还回rockit。

UAC json文件配置

uac_app中2个数据流程，由rockit媒体库解析json配置文件，完成对应音频组件的创建和处理。json文件的主要作用是定义整个数据通路的结构、各个音频组件的参数等。客户可按照自己硬件配置和要求，修改json文件中对应的设置，以满足其需求。uac_app定义的2个conf文件位于external\uac_app\configs目录：

- mic_recode_usb_playback.json：Master从uac设备上录音的流程。
- usb_recode_speaker_playback.json：Master的数据通过uac设备播放的流程。

【示例】：Master端从uac设备录音

```
{
  "pipe_0": {
    "node_0": {
      "node_opts": {
        "node_name"      : "alsa_capture"
      },
      "node_opts_extra": {
        "node_source_uri" : "hw:0,0",
        "node_buff_type"  : 0,
        "node_buff_count" : 2,
        "node_buff_size"  : 4096,
        "node_buff_alloc_type" : "malloc"
      },
      "stream_opts_extra": {
        "opt_audio_ppm"    : 1,
        "opt_sample_rate"  : 16000,
        "opt_format"       : "audio:pcm_16",
        "opt_channel"      : 8,
        "opt_channel_layout" : "int64:255",
        "opt_peroid_size"  : 256,
        "opt_peroid_count" : 4,
        "opt_read_size"    : 4096,
      },
      "stream_opts": {
        "stream_fmt_in"    : "audio:pcm_16",
        "stream_fmt_out"   : "audio:pcm_16",
        "stream_output"    : "audio:pcm_0"
      }
    },
    "node_1": {
      "node_opts": {
        "node_name"      : "skv"
      },
      "node_opts_extra": {
        "node_source_uri" : "/oem/usr/share/uac_app/configs_skv.json",
        "node_buff_type"  : 0,
        "node_buff_count" : 2,
        "node_buff_size"  : 2048,
        "node_buff_alloc_type" : "malloc"
      },
      "stream_opts_extra": {
        "opt_sample_rate"  : 16000,
        "opt_format"       : "audio:pcm_16",
        "opt_channel"      : 8,
        "opt_ref_channel_layout" : "int64:63",
      }
    }
  }
}
```

```

        "opt_rec_channel_layout" : "int64:192",
        "opt_channel_layout" : "int64:255"
    },
    "stream_opts": {
        "stream_fmt_in" : "audio:pcm_16",
        "stream_fmt_out" : "audio:pcm_16",
        "stream_input" : "audio:pcm_0",
        "stream_output" : "audio:pcm_1"
    }
},
"node_2": {
    "node_opts": {
        "node_name" : "resample"
    },
    "node_opts_extra": {
        "node_buff_type" : 0,
        "node_buff_count" : 2,
        "node_buff_size" : 2048,
        "node_buff_alloc_type" : "malloc"
    },
    "stream_opts_extra": {
        "opt_sample_rate": 48000,
        "opt_format" : "audio:pcm_16",
        "opt_channel" : 2,
        "opt_channel_layout" : "int64:3"
    },
    "stream_opts": {
        "stream_fmt_in" : "audio:pcm_16",
        "stream_fmt_out" : "audio:pcm_16",
        "stream_input" : "audio:pcm_1",
        "stream_output" : "audio:pcm_2"
    }
},
"node_3": {
    "node_opts": {
        "node_name" : "alsa_playback"
    },
    "node_opts_extra": {
        "node_source_uri" : "hw:1,0"
    },
    "stream_opts": {
        "stream_fmt_in" : "audio:pcm_16",
        "stream_fmt_out" : "audio:pcm_16",
        "stream_input" : "audio:pcm_2"
    },
    "stream_opts_extra": {
        "opt_start_delay" : 16000,
        "opt_alsa_mode" : "nonblock",
        "opt_sample_rate": 48000,
        "opt_format" : "audio:pcm_16",
        "opt_channel" : 2,
        "opt_channel_layout" : "int64:3",
        "opt_peroid_size" : 256,
        "opt_peroid_count" : 4
    }
}
}
}

```

UAC外部参数设置

json文件配置了uac流程中各处理节点的默认参数，但部分节点的参数需要根据Master端实际需求和用户要求来设置。

- Slave端(uac设备端) usb录音、放音声卡的采样率。该采样率由Master端放音、录音时打开声卡的采样率决定。
- Slave端音量大小和静音状态。该音量大小和静音状态，由Master端的设置决定。
- ppm值。该值由Master端和Slave端频率差异决定，uac驱动会自动计算出该值并通过uevent事件上报。
- 其他客户自定义的参数调整。

rockit定义的数据结构(类)RtMetaData，其实现思想是将参数按照key：value的方式保存起来。RtMetaData提供了设置/获取int，float，string，point等类型的接口，如下：

```
virtual RT_BOOL setCString(const char* key, const char *value);
virtual RT_BOOL setInt32(const char* key, INT32 value);
virtual RT_BOOL setInt64(const char* key, INT64 value);
virtual RT_BOOL setFloat(const char* key, float value);
virtual RT_BOOL setPointer(const char* key, RT_PTR value, RTMetaValueFree
freeFunc = RT_NULL);
virtual RT_BOOL findCString(const char* key, const char **value) const;
virtual RT_BOOL findInt32(const char* key, INT32 *value) const;
virtual RT_BOOL findInt64(const char* key, INT64 *value) const;
virtual RT_BOOL findFloat(const char* key, float *value) const;
virtual RT_BOOL findPointer(const char* key, RT_PTR *value) const;
```

其中，key为写入参数的名称，value为设置参数。通过以上接口，RtMetaData可以传输任意类型的数据。

uac_app默认实现了对音频采样率，音量大小和静音，以及ppm值的外部参数设置(代码：graph_control.cpp)。

【示例】：设置usb录音/放音声卡的采样率。

uac_app中将需要修改的参数，写入到RtMetaData内部，然后调用rockit的invoke接口，将RtMetaData传入到对应的音频处理节点，rockit内部的音频处理节点通过find接口，将对应的参数取出，修改节点的参数配置。如下图set_uac_parameter为uac_app中实时设置参数的接口。

```
void graph_set_samplerate(RTUACGraph* uac, int type, UCAudioConfig& config) {
    if (uac == NULL)
        return;

    int sampleRate = config.sampleRate;
    if (sampleRate == 0)
        return;

    RtMetaData meta;
    meta.setInt32("opt_samaple_rate", sampleRate);
    ALOGD("%s: type = %d, sampleRate = %d\n", __FUNCTION__, type, sampleRate);
    /*
     * 1. for usb capture(Master send datas to uac device),
     *    update sampelrate to usb record card.
     * 2. for usb playback(Master record datas from uac device),
     *    update output samplerate of resample before usb playback,
     *    and usb playback will check the parameters of every buffer,
```

```

    *    and will reopen usb sound card if the paramters(samplerate)
    *    is changed.
    */
    if (type == UAC_STREAM_RECORD) {
        // the usb record always the first node
        int usbNodeId = 0;
        meta.setInt32(kKeyTaskNodeId, usbNodeId);
        meta.setCString(kKeyPipeInvokeCmd, OPT_SET_ALSA_CAPTURE);
    } else {
        // find the resample before usb playback, see
        mic_recode_usb_playback.json
        int resampleNodeId = 1;
        meta.setInt32(kKeyTaskNodeId, resampleNodeId);
        meta.setCString(kKeyPipeInvokeCmd, OPT_SET_RESAMPLE);
    }

    uac->invoke(GRAPH_CMD_TASK_NODE_PRIVATE_CMD, &meta);
}

```

以Master端从Slave端录音为例，对上述示例代码进行说明：

- Master端打开录音，从uac设备端录音。录音参数为44.1k，2channels。
- uac设备端从uac驱动收到uevent事件，获取Master需要音频数据的采样率。
- uac设备端通过如上代码将Master需要的采样率设置到rockit中。
 1. 对于Master发送音频数据到uac设备播放，该采样率需要设置到usb录音声卡节点。
 2. 对于Master从uac设备端录音，该采样率需要设置到usb放音声卡前的resample节点。该resample节点用于将uac设备的mic录制的的数据，转换成Master需要数据的采样率。usb放音节点会逐帧检测resample节点的数据，然后按照resample数据的实际采样发开声卡发送数据。
- RtMetaData设置参数时，需要根据当前流程使用的配置文件，找到需要设置的节点的序号，然后将序号和待更新的参数设置到meta中。
 1. usb录音声卡的节点为0(node_0)，因此当Master通过uac设备端播放时，需要设置usb录音声卡的采样率，因此向节点0设置采样率。其数据流程为：Master设备---->usb录音声卡---->resample---->speaker。

```

int usbNodeId = 0;
meta.setInt32(kKeyTaskNodeId, usbNodeId);

```

2. resample节点为节点1(node_1)，因此当Master从uac设备端录制时，需要设置resample节点输出采样率，因此向节点1设置采样率，其数据流程为：mic---->resample---->usb放音声卡---->Master设备。

```

int resampleNodeId = 1;
meta.setInt32(kKeyTaskNodeId, resampleNodeId);

```

```

uac->invoke(GRAPH_CMD_TASK_NODE_PRIVATE_CMD, &meta);

```

invoke函数将meta数据传递到对应的节点中，实现参数的更新。

- 同理其它参数的设置。

【注意】

如果客户修改、删除、增加了配置文件的节点或者节点id，那么一定要在对应函数中(通常是位于graph_control.cpp)，修改代码中节点id，否则会导致参数设置失败，从而引起uac运行的各种问题。

uac节点，数据的抓取

为了方便开发人员/客户获取uac各节点的数据，可在对应的配置文件中(比如mic_recode_usb_playback.json)，配置如下选项来导出某节点的输出数据，配置项："rt_debug_out_file": "path/xxxxx.pcm"。

【示例】

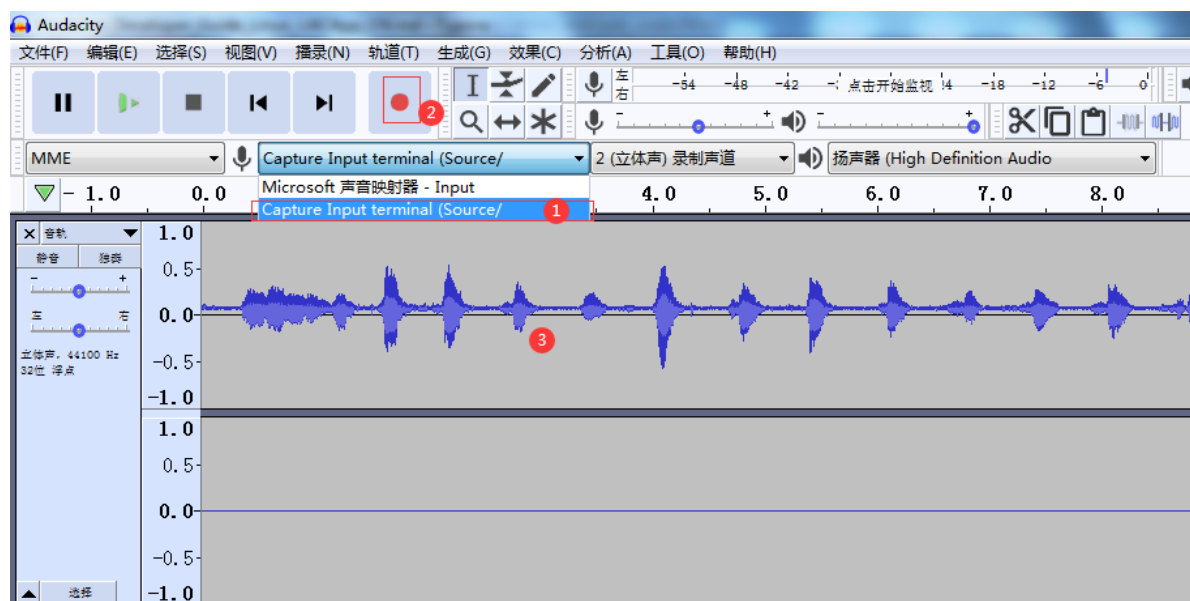
```
{
  "pipe_0": {
    "node_0": {
      "node_opts": {
        "node_name"      : "alsa_capture"
      },
      "stream_opts_extra": {
        "rt_debug_out_file" : "/userdata/cap.pcm",
        .....
      },
      .....
    },
    "node_1": {
      "node_opts": {
        "node_name"      : "skv"
      },
      "stream_opts_extra": {
        "rt_debug_out_file" : "/userdata/skv.pcm",
        .....
      },
      .....
    },
    "node_2": {
      "node_opts": {
        "node_name"      : "resample"
      },
      "stream_opts_extra": {
        "rt_debug_out_file" : "/userdata/resample.pcm",
        .....
      },
      .....
    },
    .....
  }
}
```

node0~node2 分别通过"rt_debug_out_file": "/userdata/xxx.pcm"定义了输出文件。定义了该配置后，当前节点在数据处理完毕输往下一个节点前，会将音频数据写到userdata的xxx.pcm文件中。客户可任意定义文件路径和文件名，只要uac_app有权限读写该目录；uac的每次启动，都会重新覆盖写对应文件；写文件会影响uac数据传递的效率，当当前cpu/ddr较紧张或者写flash很耗时(SPI FLASH)时，写文件有可能引起声卡的overrun和underun。

uac录音/放音测试

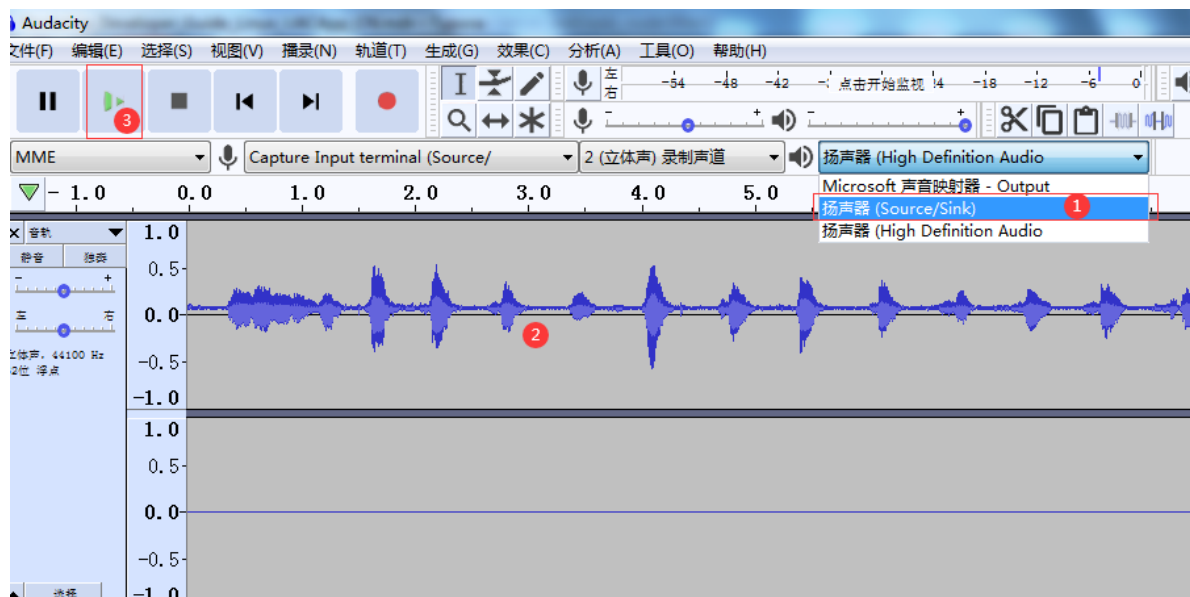
在Window上，可使用任何录音/播放软件进行测试。如下以使用开源免费工具Audacity进行说明。

录音



1. uac设备插入pc，pc端识别到uac设备后，Audacity的录音设备列表中找到/选择uac设备。注意：需要usb脚本中配置了uac的录音功能
2. 点击录音按钮，启动录音。
3. 录制到声音后，Audacity直接将波形显示在界面上。如图，为单mic录制到的数据。

放音



1. uac设备插入pc，pc端识别到uac设备后，Audacity的放音设备列表中找到/选择uac设备。注意：需要usb脚本中配置了uac的放音功能。
2. Audacity中导入要播放的音频文件。对于wav文件，可从文件-->打开直接导入，如果是pcm文件，在需从文件-->导入-->原始数据，并手动填写数据的采样率，声道数，字节序等参数。
3. 点击播放按钮，启动uac放音，观察uac设备上speaker的输出。

Q&A

第三方算法和处理集成

rockit通常是以TaskNode为基本的处理单元。当客户需要在UAC的处理流程中，增加自己的处理时，可按照如下的方法操作：

定义自己的处理节点，该节点必须继承rockchip定义的TaskNode的基类RTTaskNode。例如：

```
class RTRockitDemoNode : public RTTaskNode {
public:
    RTRockitDemoNode();
    virtual ~RTRockitDemoNode();
    virtual RT_RET open(RTTaskNodeContext *context);
    virtual RT_RET process(RTTaskNodeContext *context);
    virtual RT_RET close(RTTaskNodeContext *context);

protected:
    virtual RT_RET my_process(RTMediaBuffer *in, RTMediaBuffer *out);
};
```

如上代码定义了处理节点RTRockitDemoNode，它继承RTTaskNode，其中客户需要实现纯虚拟函数open，process和close函数，函数的简单说明如下：

open: 初始化自定义处理，在函数中完成自定义处理内存的分配，变量的初始化，库的打开和初始化等。参数RTTaskNodeContext *context上当前TaskNode的上下文，配置文件中RtMetaData数据保存在该变量中，可通过context->options()拿到并保存到RtMetaData中的参数。初始化成功时，返回RT_OK(0)，返回其他值时表示失败。其Demo代码如下：

```
RT_RET RTRockitDemoNode::open(RTTaskNodeContext *context) {
    RT_RET err = RT_OK;
    RtMetaData *meta = context->options();
    RTNodeDemoCtx *ctx = reinterpret_cast<RTNodeDemoCtx *>(mCtx);
    if (RT_NULL == ctx) {
        RT_LOGE("context is NULL, please alloc RTNodeDemoCtx in construct");
        return RT_ERR_INIT;
    }
    // check parameters
    INT32 sampleRate = 0;
    if (!meta->findInt32("opt_samaple_rate", &sampleRate)) {
        RT_LOGE("samplerate not find");
        return RT_ERR_INIT;
    }
    ctx->mSampleRate = sampleRate;
    // the other init steps
    err = .....;
    return err;
}
```

RtMetaData中的相关参数，通常是rockit框架从json配置文件中读取到的，其定义可参考uac_app\configs下的相关文件。

close：用于关闭当前节点，释放其占用的资源。

process: 完成相关数据的处理。

```
RT_RET RTRockitDemoNode::process(RTTaskNodeContext *context) {
    RTMediaBuffer *inputBuffer = RT_NULL;
```

```

RTMediaBuffer *outputBuffer = RT_NULL;
// get the first buffer in input queue
input = context->inputHeadBuffer();
if (input != RT_NULL) {
    // get a empty out buffer to store the data after process
    // the fun will be blocked by default if no output left
    output = context->dequeOutputBuffer();
    if (output != RT_NULL) {
        err = my_process(input, output);
        // in no data in input buffer, dequeue it from input queue
        // and release it
        if (input->getLength() <= 0) {
            input = context->dequeInputBuffer();
            input->release();
        }
        // if success, set output to output queue.
        if (err == RT_OK) {
            context->queueOutputBuffer(output);
        } else {
            // if fail, release output buffer
            output->release();
        }
    } else {
        input = context->dequeInputBuffer();
        input->release();
    }
}
return RT_OK;
}

```

RTTaskNode类中通常包含有2个或2个以上的输入/输出队列。输入队列，保存前级RTTaskNode节点送入的待处理数据，输出队列保存有当前节点处理完毕，待送入后级节点的数据(如果有后级RTTaskNode节点的话)。

input = context->inputHeadBuffer();表示获取输入队列中第一个未处理的buffer。注意该操作并未将第一个buffer从输入队列中删除。

output = context->dequeOutputBuffer(); 表示获取输入队列中空闲buffer，用于保存当前节点处理后的数据。如果当前节点是整个通路的最后一级节点，即当前节点无需输出buffer，那么process中可无需这段代码。该操作默认为阻塞操作，即当前节点的输出队列中没有空闲buffer时，该函数会阻塞住，直至退出或者有空闲buffer为止。该队列中默认空闲buffer的大小和数量，通常由配置文件定义：

```

"node_opts_extra": {
    "node_buff_type" : 0,
    "node_buff_count" : 4,
    "node_buff_size" : 2048,
    "node_buff_alloc_type" : "malloc"
},

```

如上为配置文件中某节点buffer大小和数量的配置。

"node_buff_type": 0表示buffer的获取方式，0表示自动分配。

"node_buff_count": 4 表示buffer的数量为4块。

"node_buff_size" : 2048 表示每个buffer的大小为2048个字节。

"node_buff_alloc_type": "malloc" 表示用malloc的方式分配buffer，对于视频数据需要使用到DRM buffer，可配置为对应值。

如上buffer的配置，分配，管理等，由rockit框架中自动完成，客户无需自己实现。

```
RT_RET RTRockitDemoNode::my_process(RTMediaBuffer *in, RTMediaBuffer *out) {
    UINT8 *data = reinterpret_cast<UINT8*>(in->getData());
    UINT32 srcOffset = in->getOffset();
    UINT32 srcLength = in->getLength();
    void *inputData = reinterpret_cast<void*>(&data[srcOffset]);

    INT32 eos = 0;
    INT32 error = 0;
    in->getMetaData()->findInt32("opt_av_eos", &eos);
    in->getMetaData()->findInt32("opt_av_err", &error);
    // if input buffer is eos
    if (eos || error) {
        dstFrame->setRange(0, 0);
        if (eos)
            out->getMetaData()->setInt32("opt_av_eos", 1);
        if (error)
            out->getMetaData()->setInt32("opt_av_err", 1);
        return RT_OK;
    }

    // if output buffer is too small, realloc it
    if (out->getSize() < srcLength) {
        out->signalBufferRealloc(srcLength);
    }
    // process data, here we copy data in demo
    memcpy(out->getData(), inputData, srcLength);
    // set the length of output buffer
    out->setRange(0, srcLength);
    // set parameters to RTMetaData
    out->getMetaData()->setInt32("opt_sample_rate", aaaaa);
    out->getMetaData()->setInt32("opt_channel", bbbbbb);
    out->getMetaData()->setInt32("opt_format", RT_AUDIO_FMT_PCM_S16);

    // set data left in input buffer, this is no data left in demo
    in->setRange(0, 0);
    return RT_OK;
}
```

如上my_process中：

```
UINT8 *data = reinterpret_cast<UINT8*>(in->getData());
UINT32 srcOffset = in->getOffset();
UINT32 srcLength = in->getLength();
void *inputData = reinterpret_cast<void*>(&data[srcOffset]);
```

获取输入buffer未处理数据的起始位置。

```

in->getMetaData()->findInt32("opt_av_eos", &eos);
in->getMetaData()->findInt32("opt_av_err", &error);
if (eos || error) {
    // no valid data in this buffer
    out->setRange(0, 0);
    if (eos)
        out->getMetaData()->setInt32("opt_av_eos", 1);
    if (error)
        out->getMetaData()->setInt32("opt_av_err", 1);
    return RT_OK;
}

```

检测当前buffer是否最后一帧/错误帧。eos通常标记当前buffer为整个数据的最后一帧，如读文件方式测试时，读取到文件末尾时，需要将该buffer标记为eos，后续节点读取到该标记后，需将该标记传递给它的后级节点，当整个通路所有节点都收到了eos标记后，表明当前通路的数据已全部处理完毕。

```

if (out->getSize() < srcLength) {
    out->signalBufferRealloc(srcLength);
}

```

该代码比较检查输出buffer的大小。因为当前demo中的处理是memcpy，因此直接使用输入buffer有效数据的长度以及输出buffer的大小做比较，当输出buffer大小不够时，需要调用signalBufferRealloc重新分配输出buffer。客户需要按照自己添加的处理，来判断当前输出buffer大小是否足够，该输出buffer的默认大小由"node_buff_size" : 2048 这个配置决定。

```

out->setRange(0, srcLength);
// set parameters to RTMetaData
*(out->getMetaData()) = *(in->getMetaData());
out->getMetaData()->setInt32("opt_sample_rate", aaaaa);
out->getMetaData()->setInt32("opt_channel", bbbbb);
out->getMetaData()->setInt32("opt_format", RT_AUDIO_FMT_PCM_S16);
in->setRange(0, 0);

```

void RTMediaBuffer::setRange(UINT32 offset, UINT32 length)，用于设置buffer中有效数据的长度和偏移。

out->setRange(0, srcLength); 表示输出buffer有效数据的偏移为0，有效数据长度为srcLength。

in->setRange(0, 0); 表示输入buffer中有效数据的偏移为0，有效数据长度为0，即没有有效数据剩余。当还有数据剩余时，可使用in->setRange(srcOffset + size, srcLength - size)，其中srcOffset 为处理前有效数据的偏移，srcLength为使用前有效数据的长度，size为当前处理使用掉的数据。

*(out->getMetaData()) = *(in->getMetaData()); 表示将输入buffer中所有的RtMetaData信息复制到输出buffer中。

out->getMetaData()->setInt32(aaaaa, bbbbb); 表示更新RtMetaData中的某些参数，客户需要根据自己节点的功能，更新部分音频参数。比如当前节点为重采样节点，那么通过重采样节点后，需要更新重采后数据的采样率到RtMetaData中，以使得后续节点能够正确识别出当前数据的参数。

```

if (input->getLength() <= 0) {
    input = context->dequeInputBuffer();
    input->release();
}

```

当输入buffer中没有剩余数据时，input = context->dequeInputBuffer()将其从输入队列中删除，input->release()会将该buffer释放。在rockit中，所有节点的buffer都是通过buffer pool管理，这里所谓的释放，会将该buffer放回buffer pool中并标记为可用/空闲。该buffer通常是前级节点处理后送入到当前节点输入队列的buffer，因此input->release()会将该buffer还给前级节点的buffer pool，使得前级节点里的context->dequeOutputBuffer()能够解除阻塞，并拿到空闲的buffer。

```
if (err == RT_OK) {
    context->queueOutputBuffer(output);
} else {
    // if fail, release output buffer
    output->release();
}
```

context->queueOutputBuffer(output);该函数将输出buffer送到当前节点的输出队列中，rockit框架最终会将该buffer送入到它的下一级节点的输入队列中。

自定义节点注册到rockit中。

```
// 用于节点创建， 该函数指针将存于RTNodeStub.mCreateObj中
static RTTaskNode* createRockitDemoNode() {
    return new RTRockitDemoNode();
}

//节点信息存根， 用于完成节点注册
RTNodeStub node_stub_rockit_demo {
    // 节点uid， 节点的唯一标识符 (0~1000)
    .mUid          = kStubRockitDemo,
    // 节点名， 主要用于节点查找、创建
    // corp_role_name, 命名保证唯一
    .mName         = "rockit_demo",
    // 版本号
    .mVersion      = "v1.0",
    // 节点创建方法；改成宏定义
    .mCreateObj    = createRockitDemoNode,
    .mCapsSrc      = { "audio/x-raw", RT_PAD_SRC, {RT_NULL, RT_NULL} },
    .mCapsSink     = { "audio/x-raw", RT_PAD_SINK, {RT_NULL, RT_NULL} },
};

RT_NODE_FACTORY_REGISTER_STUB(node_stub_rockit_demo);
```

结构体RTNodeStub为rockit定义的特定结构体，用于完成节点的定义和注册。mUid和mName用于在不同节点的区分和识别，因此需保证它们值的唯一性。mCreateObj定义了构造/初始化 节点。mCapsSrc和mCapsSink暂无使用。

RT_NODE_FACTORY_REGISTER_STUB(node_stub_rockit_demo); 表示将node_stub_rockit_demo注册到rockit中。

完成自定义节点代码编写后，为了应用(比如uac_app)可调用到自定义节点的代码：

1. 可将自定义节点代码和rockit库编译成一个新库，uac_app代码中链接新库。
2. 将自定义代码放入到应用(比如uac_app中)，uac_app调用rockit库和自定义节点代码实现相关功能。

配置文件中添加自处理节点：

```
"node_0": {
```

```

        .....
        "stream_opts": {
            .....
            "stream_output"    : "node0_out"
        }
    },
    "node_1": {
        "node_opts": {
            "node_name"        : "rockit_demo"
        },
        "stream_opts": {
            .....
            "stream_input"     : "node0_out",
            "stream_output"    : "node1_out"
        }
    },
    .....
},
"node_2": {
    .....
    "stream_opts": {
        .....
        "stream_input"        : "node1_out",
        "stream_output"        : "node2_out"
    }
    .....
},
.....

```

当rockit加载如上配置文件时，会根据节点名和已注册到rockit中的RTNodeStub的mName相比较，找到对应的RTNodeStub，并调用mCreateObj的函数指针，完成节点的构造。如上配置中，node_0的输出stream_output作为node_1的输入，两者的标记完全一样，都是字符串"node0_out"。如果节点A的stream_output 和节点B的stream_input标记完成相同时，表示的是节点A的输出作为节点B的输入，即节点A是节点B的上级节点。因此，如上配置的数据流为node_0--->node_1--->node_2，其中箭头方向为数据的流动方向。

uac断音/杂音问题

导致uac断音/杂音的原因多种多样，归纳有如下几点：

- usb与mic/spk时钟异源

uac设备的usb时钟(uac驱动传递数据的时钟)，通常都由host端提供(uac设备作为slave设备)。uac设备上的mic或者spk的时钟，通常由uac设备上晶振提供。uac经过一段时间后运行后，由于host端和uac设备上时钟的些许差异，会导致uac设备端mic/speaker出现xrun(overflow/underrun)，从而导致的断音/杂音。若uac设备上mic/spk的时钟和usb时钟同源(即都由host提供)，可跳过此节。

现象：在uac使用过程中，如果mic/spk/usb声卡节点，周期性的出现xrun。出现xrun的周期，由host端晶振与uac设备的晶振差异决定，差异越大越容易出现，通常出现xrun的周期由几分钟到几小时不等。

解决方法：uac驱动在uac启动后，会计算uac设备上时钟和host时钟的差异，并将该时钟差异通过uevent事件(即ppm事件)向uac_app上报。uac_app在接收到ppm事件后，会将该ppm值设置到rockit中的mic/spk节点，mic/spk节点最终通过amixer接口通知i2s/pdm/tdm驱动微调对应的时钟(PLL)。

如下为uac_app中，ppm接收、解析并设置的代码(uac_app/src/uevent.cpp中)：

```
/*
```



```

* strs[0] = ACTION=change
* strs[1] = DEVPATH=/devices/virtual/u_audio/UAC1_Gadget 0
* strs[2] = SUBSYSTEM=u_audio
* strs[3] = USB_STATE=SET_AUDIO_CLK
* strs[4] = PPM=-21
* strs[5] = SEQNUM=1573
*/
void audio_set_ppm(const struct _uevent *uevent) {
    char *ppmStr = uevent->strs[UAC_KEY_PPM];
    if (compare(ppmStr, UAC_PPM)) {
        int ppm = 0;
        sscanf(ppmStr, "PPM=%d", &ppm);
        uac_set_ppm(UAC_STREAM_RECORD, ppm);
        uac_set_ppm(UAC_STREAM_PLAYBACK, ppm);
    }
}

```

最终，ppm值最终被设置给mic/spk节点(uac_app/src/graph_control.cpp)。

```

void graph_set_ppm(RTUACGraph* uac, int type, UACAudioConfig& config) {
    if (uac == NULL)
        return;

    RtMetaData meta;
    int ppm = config.ppm;
    ALOGD("type = %d, ppm = %d\n", type, ppm);
    if (type == UAC_STREAM_RECORD) {
        // spk node, see usb_recode_speaker_playback.json
        meta.setInt32(kKeyTaskNodeId, 3);
        meta.setInt32(OPT_PPM, ppm);
        meta.setCString(kKeyPipeInvokeCmd, OPT_SET_PPM);
    } else {
        // mic node, see mic_recode_usb_playback.json
        meta.setInt32(kKeyTaskNodeId, 0);
        meta.setInt32(OPT_PPM, ppm);
        meta.setCString(kKeyPipeInvokeCmd, OPT_SET_PPM);
    }

    uac->invoke(GRAPH_CMD_TASK_NODE_PRIVATE_CMD, &meta);
}

```

注意，需要在uac的json文件中，打开mic/spk节点ppm设置的使能开关(没有配置时，默认不开启设置ppm功能)。如下为mic_recode_usb_playback.json中ppm配置使能：

```

"node_0": {
    "node_opts": {
        "node_name"      : "alsa_capture"
    },
    "stream_opts_extra": {
        "opt_audio_ppm"  : 1,      // enable ppm adjust in mic node
        .....
    },
    .....
},

```

最终时钟的调整，是rockit调用amixer接口，通知i2s/tdm/pdm驱动实现，因此需要在i2s/tdm/pdm驱动中打开对时钟调整的使能。具体请参考docs/Common/AUDIO目录的《Rockchip_Developer_Guide_Audio_CN.pdf》2.7 音频时钟补偿。

驱动成功配置ppm后，可通过amixer contents命令，看到对应节点：

```
[root@RV1126_RV1109:/]# amixer contents
numid=0,iface=PCM,name='PCM Clk Compensation In PPM'
; type=INTEGER,access=rw-----,values=1,min=-1000,max=1000,step=1
: values=-6
[root@RV1126_RV1109:/]#
```

当uac运行时，ppm值最终会写到numid=0,iface=PCM,name='PCM Clk Compensation In PPM'中，可通过amixer contents查看value值是否有被正确设置。ppm值的正负，表示pll调整的方向，负值表示要微调慢当前pll的频率，即host端的频率比uac device mic/spk的频率慢，因此要微调慢mic/spk的录放音速度。

用户可通过如下的命令，查看ppm设置前后相关硬件频率的变化(以i2s为例，如果是其他接口比如pdm，请查看pdm的对应频率)。

```
cat /sys/kernel/debug/clk/clk_summary | grep cp11
cat /sys/kernel/debug/clk/clk_summary | grep i2s
```

注意：音频和其他模块共享 PLL时，产品需要评估时钟微调对共享模块的影响。如该PLL与以太网共享，而以太网需要精确的时钟，那么如上通过ppm方式调整PLL的方法，可能导致以太网无法正常工作。

- ddr带宽的影响。

当uac/uvic设备某些特定场景，比如视频通话，uvic视频分辨率为H264 4K 30fps，或者uvic+ai(比如人脸检测等)等应用教复杂的场景下，ISP/视频编码/NPU模块会占据非常高的ddr带宽，导致音频模块因为无法及时访问ddr而无法及时处理数据，从而导致的断音/杂音。该问题在单独使用uac时，不会出现断音/杂音问题；ddr使用率较高的场景，比如uvic(H264 4K 30fps)+uac+eptz等复合场景下，uac出现不规则的断音/杂音。

可通过对应的带宽统计工具，查看对应使用场景下，ddr带宽的使用率。如下为使用统计工具，统计1126 只有uac情况下(ddr频率为924M)的带宽使用率：

```
[root@RV1126_RV1109]# ./rk-msch-probe -c rv1126 -f 924
v1.09_20201120

update ddr freq: 924Mhz
CH0:
probes statistics:

```

	master	bw(MB/s)	bw prorated(%)	utilization(%)
cpu		172.85	89.40%	2.34%
npu		0.30	0.16%	0.00%
vi		0.00	0.00%	0.00%
vpu		20.20	10.45%	0.27%
total		193.35	100.00%	2.62%
theoretical bw		7392.00		

```

ddr monitor statistics:
ddr load = 217.61MB/s(2.94%) [RD:194.57MB/s(2.63%), WR:23.04MB/s(0.31%),
ACT(access : active): 2.34, srex:42.62%, pdex:10.23%, clkstp:0.00%,
lp:52.85%]
```

- cpu使用率的影响。

音频数据的处理(包括3A算法),都是通过cpu来完成,因此当cpu使用较高时,会导致音频数据处理不及时而导致断音/杂音。

该问题导致的现象,与ddr带宽使用率较高时出现的断音/杂音类似,可通过ddr统计工具或者top命令查看cpu的使用率来定位问题。

需要注意的是,芯片为变频运行时,top命令统计周期内,可能查看到的cpu平均使用率并不高,但在短时间内(比如100ms)cpu使用率高,由于uac通路上每个节点的缓冲buffer的数据很少,也会导致断音/杂音。

将cpu定频到较高的频率。

```
cat /sys/devices/system/cpu/cpufreq/policy0/cpuinfo_cur_freq
// 查看当前cpu频率
cat /sys/devices/system/cpu/cpufreq/policy0/scaling_available_frequencies
// 查看cpu支持的频率
echo userspace > /sys/devices/system/cpu/cpufreq/policy0/scaling_governor
// 使能设置定频开关
echo xxxxx > /sys/devices/system/cpu/cpufreq/policy0/scaling_setspeed
// 设置xxxxx频率
```

如上命令,可能不同芯片稍有差别。如果是因为cpu使用率导致的问题,用户可从设置芯片支持的最高频率开始,逐步降低频率测试,直到找到不出现断音/杂音的合适频率为止。当芯片使用变频策略,可考虑将该合适频率以下的频率全部删除。

- uac节点参数设置错误。

该问题是因为uac_app设置给相关节点参数错误,导致的音频断音/杂音,见[UAC外部参数设置](#)。

以host通过uac从1126/1109录音为例,其处理节点和数据流为:mic-->3A-->volume-->resample-->usb声卡-->host.

mic: 1126/1109 端录音节点

3A: 1126/1109 端3A节点

volume: 1126/1109 端音量节点

resample: 1126/1109 端声道/采样率处理节点

usb : 1126/1109 端usb声卡节点。

以采样率设置为例:如果host端需要的是48K, 2channels数据;resample默认输出的数据为44.1K, 2channels数据。uac驱动将host端需要数据的采样率通过uevent事件通知uac_app, uac_app应将48K设置到resample节点,作为resample节点的输出采样率。当uac_app中设置代码的nodeID与json文件中resample的nodeID不一致时(uac_app\src\graph_control.cpp),会导致采样率无法正确设置到resample中,从而导致resample输出的44.1K的数据,被usb当做48K的数据发送给host,从而导致uac断音/杂音。如果uac流程刚开启,就出现非常频繁的overrun/underrun打印,可重点怀疑此时相关节点的参数是否被正确设置。