Audio模块HAL重构设计文档

1. 引言

Hal层是介于Audio Framework层和Audio Driver层一段C代码模块， 起一个承上启下的作用，由于安卓的设计思想是对硬件屏蔽，所以Hal层是客户改动最多的地方，也是改动人员最多最杂的地方，由于历史原因，此部分代码风格迥异，代码嵌套复杂，逻辑不清晰，维护起来极大的消耗了音频工程师的精力，在此提出重构计划，解决上述问题，提高代码的质量，降低维护成本。

1.1编写目的

此文档用于提交评审会，做代码重构的评审材料

1.2适用范围

适用于重构评审会评审

1.3参考人员

适用于音频模块相关人员和Fae工程师

2. 需求分析

2.1 Hal层基本功能介绍

Hal层涉及的功能有

1. 录制
2. 播放
3. 重采样
4. 透传
5. 卡拉OK(包含去啸叫)
6. 蓝牙语音
7. 动态插拔检测

2.2 Hal代码要求分析

Hal层作为一个承上启下的中间件，需要兼容上下两个层，需要达到以下基本要求

2.2.1. 代码风格统一而精炼

由于**Hal**会常常多人经手，如果不保持非常严格的风格和修改要求，后续版本差异化会越来越严重，消耗非常多

2.2.2. 代码解耦性好

由于**Hal**常常用于增加或修改客户的需求，所以要良好的耦合性，非常方便添加和删减功能

2.2.3. 详细的文档介绍

很多客户有自主开发的能力，Hal层作为最好入手的一个模块，需要有详实的文档提供给客户，让客户自行分析

2.24. 高效的Debug手段

由于音频常常有偶先的问题，此时就需要有一些高效的Debug手段，用于快速定位问题

2.3Hal重构目的分析

> Hal层重构一是为了满足上述良好的代码要求，二是增加更好的维护手段，对此我们有以下几点

2.3.1. 代码风格紊乱

- 之前
  由于修改人员过杂过多，甚至客户也会自行修改，所以Hal的代码风格非常乱，一眼看过去不清楚哪些可用哪些不可用，以及分别是什么功能，在修改过程中，常常被迫另外开功能函数，或者直接改流程等办法来暴力解决问题
- 修改点
  给出严格的代码风格要求，包括命名规则，函数规模，返回值要求等等

2.3.2. Debug手段落后低效

- 之前
  如果出问题了，常常需要修改Hal的代码，添加文件接口，然后播放数据，保存数据流，这涉及到编写代码，测试，烧录，重启，再测试。特别有些问题非常难复现，如果一直dump，会导致数据量过大，无法继续写入。
- 修改点
  增加动态dump功能，漏出debug节点，往节点写on，则自动在预定目录下生成截取的数据流，当复现问题时，可以立马写on，此时抓取数据，抓取完后写off关闭，然后拉出来分析即可

2.3.3. 动态看各种信息

- 之前
  如果想知道此时音频的状态信息，比方说多少声卡，每个声卡支持什么样的硬件参数设置，哪些声卡是active的，哪些是disable的，是输出设备还是输入设备，是否有重采样动作，是否打开蓝牙语音通路，当前输出输入参数是多少。等等。是非常麻烦的，需要看代码，看打印，cat各个设备节点的信息

- 修改点
  增加信息dump功能，打开后，直接在指定目录下生成信息文件，详细记录了上述的所有信息，并给出分析，比方说重采样有打开

2.3.4. 良好的耦合性

- 之前
  各个结构体嵌套，互相影响，无法解耦
- 修改点
  各个功能完全独立，拥有自己单独的变量和结构体，并有注释分块，非常方便解耦以及添加新的功能

2.3.5. 规范的Debug流程

- 之前
  客户方面出问题了，常常要索取客户的hal代码，对比log信息，看整个流程分析问题
- 修改点
  定义了一套错误码，通过错误码，编写文档，教会客户排查问题，大大减少简单问题的反复

2.3.6. 代码管理的统一

- 之前
  每个系列芯片，都有自己的Hal层代码，要维护多套代码
- 修改点
  维护一套代码

3.基本框架分析(以H6为例)

3.1. 代码目录结构分析

- 之前的Hal层写成了一个文件audio_hw.c，放在了android/hardware/aw下，
- 现在的Hal重构成三个文件，audio_hw.c里面是基本的框架回调函数，function.c是回调函数将会调用的，用于真正处理逻辑的工作函数，debug.c里面是用于处理debug信息的文件

3.2 关键流程基本分析

3.2.1 第一次加载流程分析

- audioserver进程打开对应的/system/lib/hw下的.so库，加载对应的Hal层，然后调用adev_open（ ）函数，初始化基本参数，并注册基本回调函数
- 调用adev_open_output_stream和adev_open_input_stream，获取对应的stream流信息，并注册对应的输入输出回调函数，其中最重要的两个分别是负责输出的out_write和负责输入的in_read

3.2.2. 输出流程分析

- 进程调用上个流程注册的out_write函数，直接下放函数，通过原生框架的tinyalsa接口中的pcm_write将数据写给对应的声卡

3.2.3. 热插拔流程分析(以HDMI输出为例)

- 插上USB声卡设备，框架层获取信息后，通过adev_set_parameter函数传递类似”audio_active_out_device=AUDIO_HDMI”这样的字符串，通过解析字符串，获取USB插上的信息，并将HDMI设置为输出对象

3.2.4. 卡拉OK流程分析

- 插上USB设备后，启动一个一直loop的线程，一直从USB设备获取数据，将获取的数据和系统本身的数据混合后，往音频输出口输出

3.2.5. 重采样流程分析

- 当应用试图播放一个16k采样率的文件，但是驱动只支持48k的时候，Hal层通过out_get_rate()告诉框架，下面的采样率是48k，此时框架会将数据从16k转为48k然后再下放

3.2.6. 透传流程分析

- 框架层通过adev_open将透传的flag标志位传下来，Hal层检测到后，当在out_write中检测到stream流下放的是透传数据时，将这个数据送给透传设备

3.2.7. 蓝牙语音流程分析

- 框架通过adev_set_mode下放IN_CALL的标志位，Hal层检测到此标志位后，将当前的输入输出通通切换到蓝牙设备上

4. 重构代码分析

4.1 关键数据定义

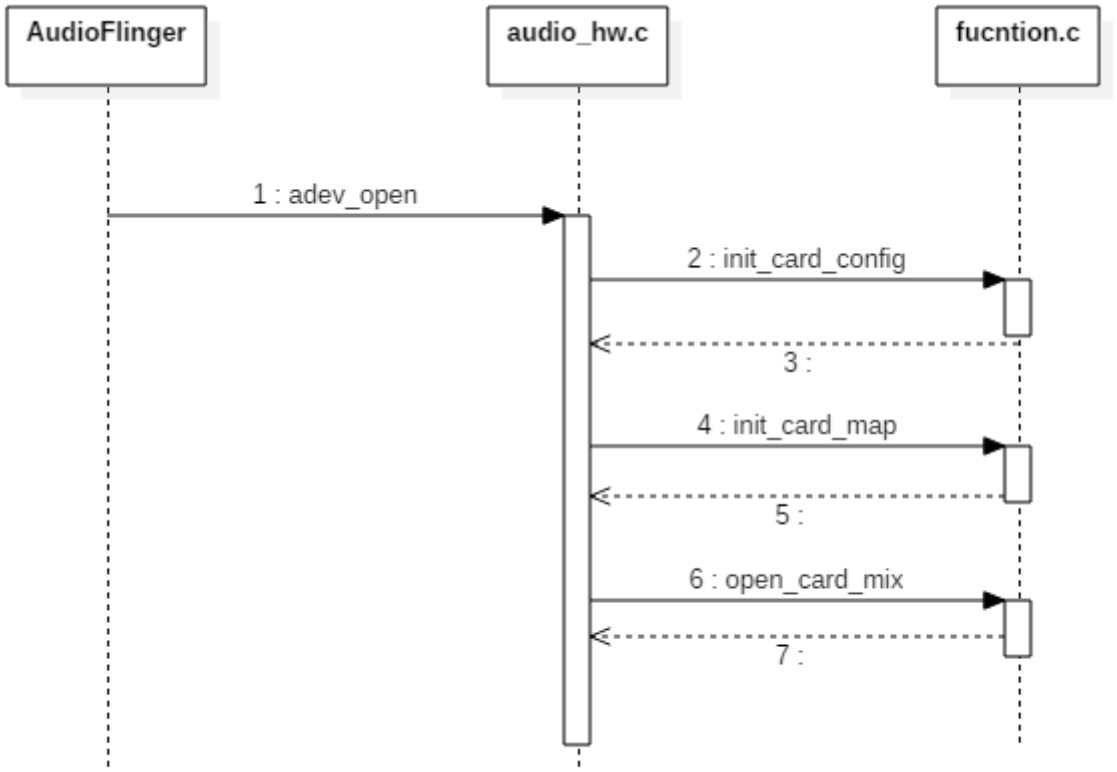
在代码中定义了一个声卡结构体

```
/* card strcut */
typedef struct map
{
    char name[NAME_LEN];        /* the name of card */
    char nickname[NAME_LEN];    /* the nickname of card */
    union
    {
        int device_write;        /* write card num for H6 ahub only(fixed)*/
        int device_read;        /* read card num for H6 ahub only(fixed)*/
    };
    int card_num;                /* support card num for H6 ahub only */
    int active;                    /* active flag */
    int hub_flag;                /* if device is link to hub true for yes false for no*/
}card_str;
```

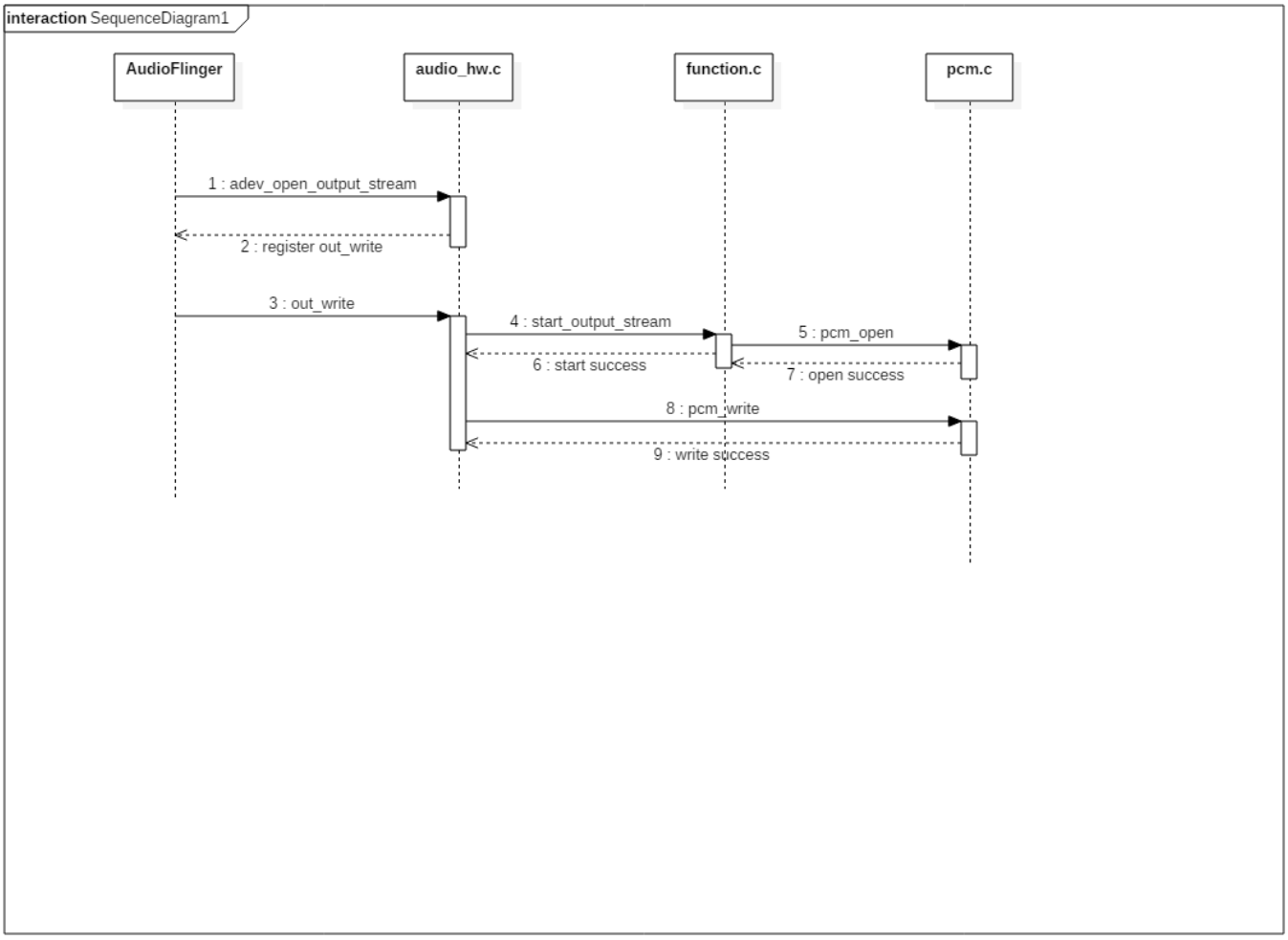
此结构体是重构中最关键的点，用于存储声卡的信息，包括真实卡名，昵称(比方说USB HDMI)，用于读写的真实卡节点，是否使能，是否属于H6 hub节点(由于hub节点和其他标准节点操作不一样)

4.2 关键流程代码框架图

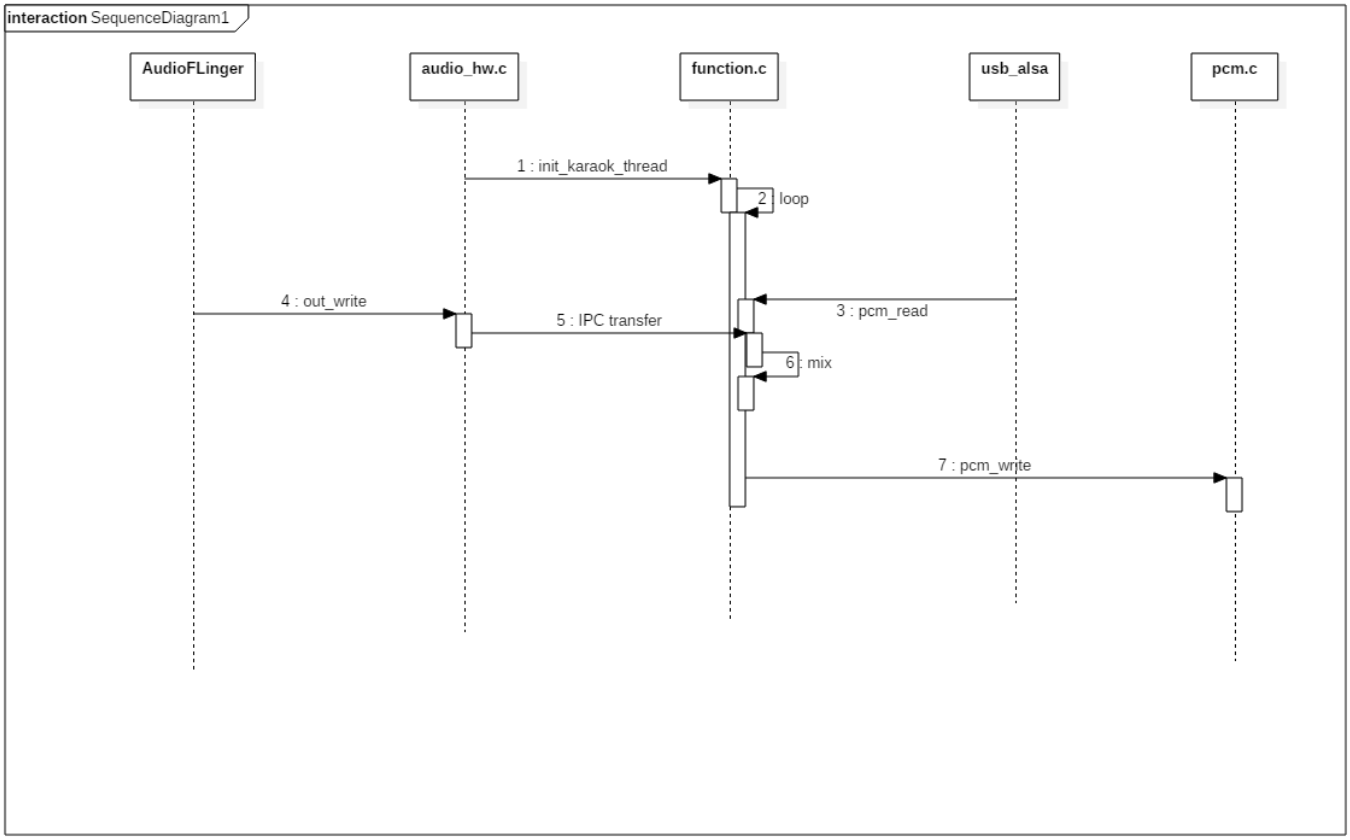
4.2.1初始化流程


AudioFlinger
audio_hw.c
fucntion.c
1 : adev_open
2 : init_card_config
3 :
4 : init_card_map
5 :
6 : open_card_mix
7 :

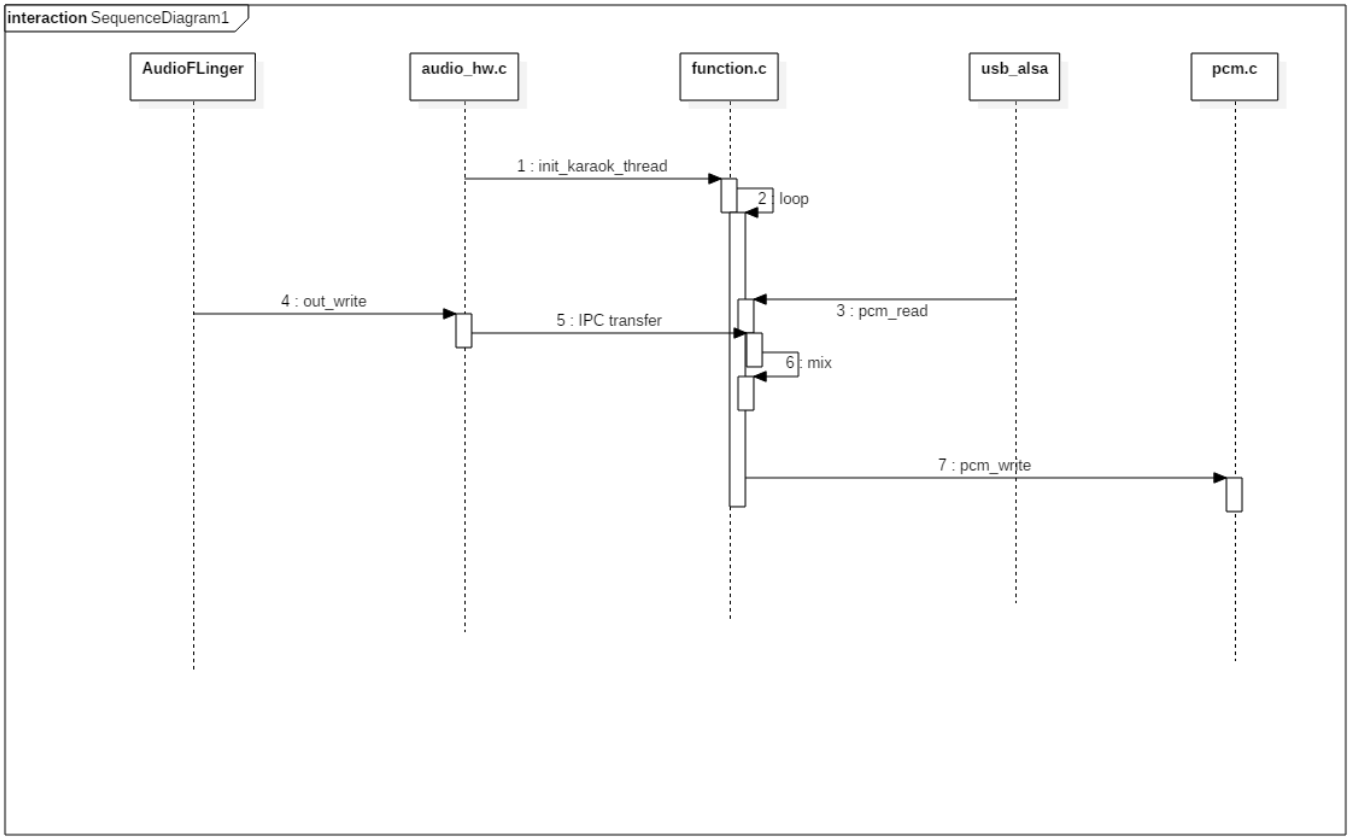
4.2.2输出流程


interaction SequenceDiagram1
AudioFlinger
audio_hw.c
function.c
pcm.c
1 : adev_open_output_stream
2 : register out_write
3 : out_write
4 : start_output_stream
5 : pcm_open
6 : start success
7 : open success
8 : pcm_write
9 : write success

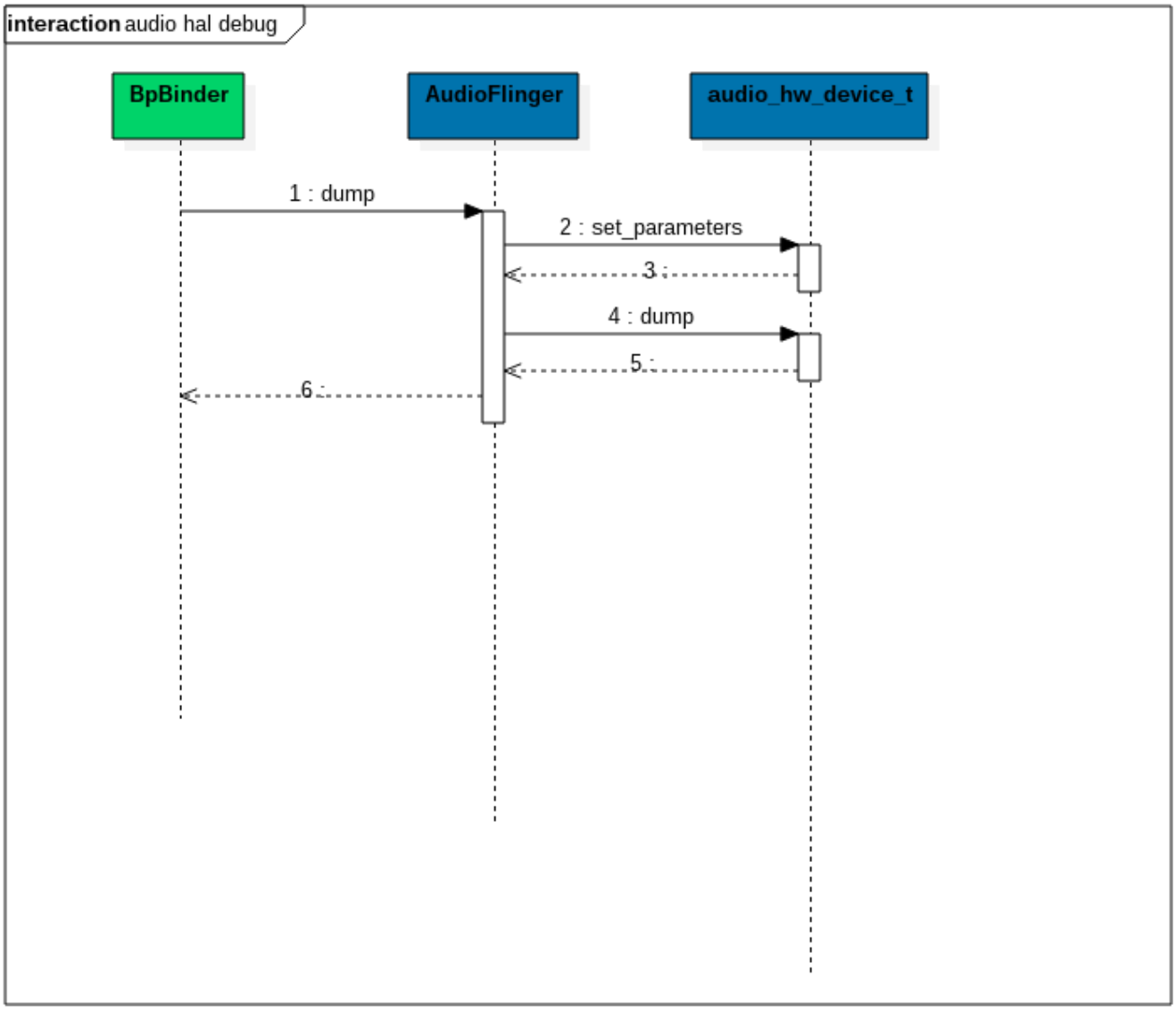
4.2.3. 卡拉OK



4.2.4. 热插拔(以插入usb音响为例)



4.2.5 dump功能



总结

从源头上完成代码的严格控制，包括风格，错误处理，文档书写规范等等，可以极大的减少代码的不稳定性，很大程度上减少问题的重复