技 术 文 件

技术文件名称：

技术文件编号：

版 本：

拟 制

审 核

会 签

标准化

批 准

海宁奕斯伟集成电路设计有限公司

修改记录

| 文件编号 | 版本号 | 拟制人/  修改人 | 拟制/修改日期 | 更改理由 | 主要更改内容  （写要点即可） |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 注1：每次更改归档文件（指归档到事业部或公司档案室的文件）时，需填写此表。  注2：文件第一次归档时，“更改理由”、“主要更改内容”栏写“无”。 | | | | | |

**目 录**

[1. 引言 5](#_Toc104800619)

[1.1 编写目的 5](#_Toc104800620)

[1.2 软件简介 5](#_Toc104800621)

[2. 总体设计方案及构想 5](#_Toc104800622)

[3. Pipeline基本框架 6](#_Toc104800623)

[4. Element模块 6](#_Toc104800624)

[4.1 数据结构 7](#_Toc104800625)

[4.1.1 struct audio\_element\_cfg\_t 7](#_Toc104800626)

[4.1.2 struct audio\_element 7](#_Toc104800627)

[4.1.3 struct audio\_element\_info\_t 8](#_Toc104800628)

[4.1.4 enum audio\_element\_state\_t 8](#_Toc104800629)

[4.1.5 enum audio\_element\_msg\_cmd\_t 9](#_Toc104800630)

[4.1.6 enum audio\_element\_status\_t 9](#_Toc104800631)

[4.2 API接口 10](#_Toc104800632)

[4.2.3 audio\_element\_init 10](#_Toc104800635)

[4.2.4 audio\_element\_deinit 10](#_Toc104800636)

[4.2.5 audio\_element\_set\_tag 10](#_Toc104800637)

[4.2.6 audio\_element\_set\_info 11](#_Toc104800638)

[4.2.7 audio\_element\_set\_uri 11](#_Toc104800639)

[4.2.8 audio\_element\_run 11](#_Toc104800640)

[4.2.9 audio\_element\_terminate 11](#_Toc104800641)

[4.2.10 audio\_element\_stop 12](#_Toc104800642)

[4.2.11 audio\_element\_wait\_for\_stop 12](#_Toc104800643)

[4.2.12 audio\_element\_pause 12](#_Toc104800644)

[4.2.13 audio\_element\_resume 13](#_Toc104800645)

[4.2.14 audio\_element\_msg\_set\_listener 13](#_Toc104800646)

[4.2.15 audio\_element\_set\_event\_callback 13](#_Toc104800647)

[4.2.16 audio\_element\_msg\_remove\_listener 14](#_Toc104800648)

[4.2.17 audio\_element\_set\_input\_ringbuf 14](#_Toc104800649)

[4.2.18 audio\_element\_set\_output\_ringbuf 14](#_Toc104800650)

[4.2.19 audio\_element\_abort\_input\_ringbuf 15](#_Toc104800651)

[4.2.20 audio\_element\_abort\_output\_ringbuf 15](#_Toc104800652)

[4.2.21 audio\_element\_wait\_for\_buffer 15](#_Toc104800653)

[4.2.22 audio\_element\_report\_status 15](#_Toc104800654)

[4.2.23 audio\_element\_report\_info 16](#_Toc104800655)

[4.2.24 audio\_element\_report\_codec\_fmt 16](#_Toc104800656)

[4.2.25 audio\_element\_report\_pos 16](#_Toc104800657)

[4.2.26 audio\_element\_input 16](#_Toc104800658)

[4.2.27 audio\_element\_output 17](#_Toc104800659)

[4.2.28 audio\_element\_set\_read\_cb 17](#_Toc104800660)

[4.2.29 audio\_element\_set\_write\_cb 17](#_Toc104800661)

[4.2.30 audio\_element\_process\_init 18](#_Toc104800662)

[4.2.31 audio\_element\_process\_deinit 18](#_Toc104800663)

[5. Pipeline模块 18](#_Toc104800664)

[5.1 pipeline概念和组成 18](#_Toc104800665)

[5.2 基于pipeline播放步骤 19](#_Toc104800666)

[5.3 数据结构 19](#_Toc104800667)

[5.3.1 struct audio\_pipeline 19](#_Toc104800668)

[5.4 API接口 20](#_Toc104800669)

[5.4.1 audio\_pipeline\_init 20](#_Toc104800670)

[5.4.2 audio\_pipeline\_deinit 20](#_Toc104800671)

[5.4.3 audio\_pipeline\_register 20](#_Toc104800672)

[5.4.4 audio\_pipeline\_unregister 20](#_Toc104800673)

[5.4.5 audio\_pipeline\_run 21](#_Toc104800674)

[5.4.6 audio\_pipeline\_terminate 21](#_Toc104800675)

[5.4.7 audio\_pipeline\_terminate\_with\_ticks 21](#_Toc104800676)

[5.4.8 audio\_pipeline\_resume 21](#_Toc104800677)

[5.4.9 audio\_pipeline\_pause 22](#_Toc104800678)

[5.4.10 audio\_pipeline\_stop 22](#_Toc104800679)

[5.4.11 audio\_pipeline\_wait\_for\_stop 22](#_Toc104800680)

[5.4.12 audio\_pipeline\_wait\_for\_stop\_with\_ticks 22](#_Toc104800681)

[5.4.13 audio\_pipeline\_link 22](#_Toc104800682)

[5.4.14 audio\_pipeline\_unlink 23](#_Toc104800683)

[5.4.15 audio\_pipeline\_get\_el\_by\_tag 23](#_Toc104800684)

[5.4.16 audio\_pipeline\_get\_el\_once 23](#_Toc104800685)

[5.4.17 audio\_pipeline\_remove\_listener 23](#_Toc104800686)

[5.4.18 audio\_pipeline\_set\_listener 24](#_Toc104800687)

[5.4.19 audio\_pipeline\_get\_event\_iface 24](#_Toc104800688)

[5.4.20 audio\_pipeline\_link\_insert 24](#_Toc104800689)

[5.4.21 audio\_pipeline\_register\_more 24](#_Toc104800690)

[5.4.22 audio\_pipeline\_unregister\_more 25](#_Toc104800691)

[5.4.23 audio\_pipeline\_link\_more 25](#_Toc104800692)

[5.4.24 audio\_pipeline\_listen\_more 25](#_Toc104800693)

[5.4.25 audio\_pipeline\_check\_items\_state 25](#_Toc104800694)

[5.4.26 audio\_pipeline\_reset\_items\_state 26](#_Toc104800695)

[5.4.27 audio\_pipeline\_reset\_ringbuffer 26](#_Toc104800696)

[5.4.28 audio\_pipeline\_reset\_elements 26](#_Toc104800697)

[5.4.29 audio\_pipeline\_reset\_kept\_state 26](#_Toc104800698)

[5.4.30 audio\_pipeline\_breakup\_elements 26](#_Toc104800699)

[5.4.31 audio\_pipeline\_relink 27](#_Toc104800700)

[5.4.32 audio\_pipeline\_relink\_more 27](#_Toc104800701)

[5.4.33 audio\_pipeline\_change\_state 27](#_Toc104800702)

[6. Event Interface模块 28](#_Toc104800703)

[6.1 数据结构 28](#_Toc104800704)

[6.1.1 audio\_event\_iface\_msg\_t 28](#_Toc104800705)

[6.1.2 audio\_event\_iface\_cfg\_t 28](#_Toc104800706)

[6.1.3 audio\_event\_iface\_item\_t 29](#_Toc104800707)

[6.2 类型定义 29](#_Toc104800708)

[6.2.1 on\_event\_iface\_func 29](#_Toc104800709)

[6.2.2 audio\_event\_iface\_handle\_t 29](#_Toc104800710)

[6.2.3 API接口 30](#_Toc104800711)

[6.2.4 audio\_event\_iface\_init 30](#_Toc104800712)

[6.2.5 audio\_event\_iface\_destroy 30](#_Toc104800713)

[6.2.6 audio\_event\_iface\_set\_listener 30](#_Toc104800714)

[6.2.7 audio\_event\_iface\_remove\_listener 31](#_Toc104800715)

[6.2.8 audio\_event\_iface\_set\_cmd\_waiting\_timeout 31](#_Toc104800716)

[6.2.9 audio\_event\_iface\_waiting\_cmd\_msg 31](#_Toc104800717)

[6.2.10 audio\_event\_iface\_cmd 31](#_Toc104800718)

[6.2.11 audio\_event\_iface\_cmd\_from\_isr 32](#_Toc104800719)

[6.2.12 audio\_event\_iface\_sendout 32](#_Toc104800720)

[6.2.13 audio\_event\_iface\_discard 32](#_Toc104800721)

[6.2.14 audio\_event\_iface\_listen 32](#_Toc104800722)

[6.2.15 audio\_event\_iface\_get\_queue\_handle 32](#_Toc104800723)

[6.2.16 audio\_event\_iface\_read 33](#_Toc104800724)

[6.2.17 audio\_event\_iface\_get\_msg\_queue\_handle 33](#_Toc104800725)

[6.2.18 audio\_event\_iface\_set\_msg\_listener 33](#_Toc104800726)

[7. pipeline通信流程 33](#_Toc104800727)

[7.1 pipeline通信机制 34](#_Toc104800728)

[7.2 外设监听线程 34](#_Toc104800729)

[7.3 Pipeline监听线程 35](#_Toc104800730)

[7.4 Element监听线程 36](#_Toc104800731)

# 引言

## 编写目的

编写本文档主要是阐述轻智能手表的音频子系统的pipeline基本框架。

## 软件简介

软件名称：ESWIN\_AF

软件任务：ESWIN\_AF提供跨平台的音频子系统。

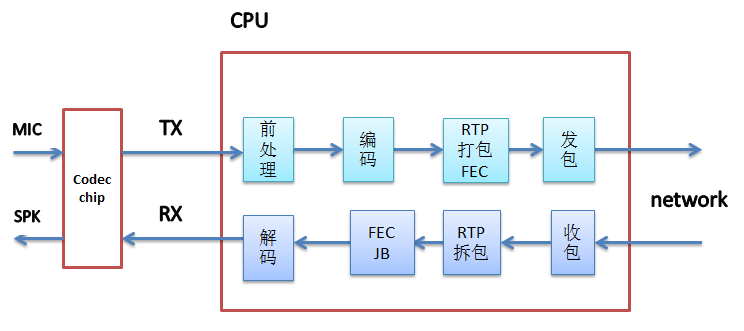
软件主要功能简介：能够在rtos操作系统下播放音频文件和音频流，实现语音通信等。能够进行基本的播放控制（播放，暂停，进度条显示，拖动进度条等）。

# 总体设计方案及构想

本部分介绍轻智能手表的总体设计方案。

嵌入式音频的应用主要分为语音通信（语音采集和编码）和音频播放两种。

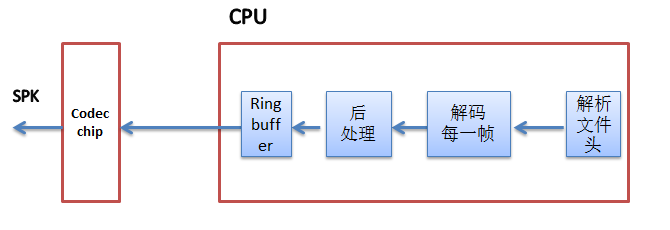
语音通信的软件架构通常如下：



语音发送时由codec芯片采集到语音的模拟信号转换成PCM数据然后通过I2S总线发送给CPU，CPU收到PCM数据后首先做前处理，主要包括回音消除(AEC)/噪声抑制(ANS)/自动增益控制(AGC)/VAD等，然后把PCM数据根据指定的codec编码成码流，再打上RTP包头，根据需要还有可能做FEC/重传语音的RTP包等以补偿网络的丢包。最后把生成的RTP包通过UDP socket发到网络中。

语音接收时首先通过UDP socket收RTP包，根据指定看是否需要做FEC等，然后把RTP包或者RTP包的payload(不同的方案有不同的做法)放进jitter buffer中等待取走。  
从jitter buffer取时取走的是payload，也就是码流。然后对码流解码得到PCM数据，还要看是否需要做PLC/CNG等。最后把PCM数据通过I2S总线送给codec芯片再转换成模拟信号播放出来。这样双方就可以通话了。

音乐播放分播放本地音乐和播放云端音乐。播放本地音乐相对简单。播放云端音乐通常是边下载边播放，一般用DLNA或者Apple的Airplay等协议实现下载。播放本地音乐的软件架构通常如下：



首先解析音频文件头得到相关信息，比如编码类型，采样率，通道数等；据此，进行解码得到PCM数据；必要时做后续处理，比如均衡器，使音乐更美妙动听；并将PCM数据放进ring buffer中。播放线程会定时的从ring buffer中取走需要的PCM，通过I2S总线送给codec芯片，codec芯片把PCM数据转换成模拟信号然后播放出来，即可听到动听的音乐了。

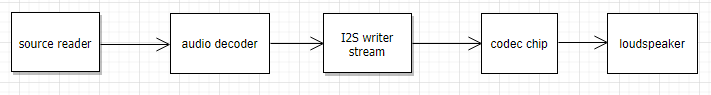
不论是语音通信还是音频播放，如果把数据想象成流水的话，每个模块的功能虽然不同，其处理流程，都是接收上一个模块过来的数据，然后经过处理后传递给下一个模块。将每个模块抽象为一个element，模块之间通过某种方式连接起来，由此可以得到基于一种基于pipeline的音频子系统。

# Pipeline基本框架

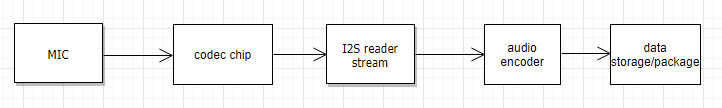
音频子系统设计方案基于pipeline流水线架构，主要职责是控制音频数据流stream以及用ring\_buffer连接各音频元素elements。它按顺序并启动音频元素，负责从前一个元素检索数据然后传递给后面一个元素，还可从每个元素获取事件、处理事件或将其传递到更高层。

可以注册多种Elements组合，在连接的时候选择一种Elements组合连接起来，也可以在断开再组成其他的Elements组合，这使得使用pipeline非常灵活。

以音频播放为例，我们可以将使用如下的pipeline实现：



同理，基于语音编码的应用可用如下pipeline实现:



基于pipeline的音频子系统方案，主要包括了pipeline，element和event interface等关键模块，下面分别描述。

Pipeline官方文档，有各个模块更详细的api接口介绍：  
https://docs.espressif.com/projects/esp-adf/zh\_CN/latest/api-reference/codecs/index.html

# Element模块

Element作为构建pipeline的基础组成部分，其主要作用是获得数据输入，处理数据流，并将其输出到下一个元素。每一个Element都是作为一个独立的task运行.对于每一个element，最关键的是要实现struct audio\_element中的结构体成员函数，将ringbuffer与element挂载。

## 数据结构

### struct audio\_element\_cfg\_t

typedef struct {

    io\_func             open;             /\*!< Open callback function \*/

    io\_func             seek;             /\*!< Seek callback function \*/

    process\_func        process;          /\*!< Process callback function \*/

    io\_func             close;            /\*!< Close callback function \*/

    io\_func             destroy;          /\*!< Destroy callback function \*/

    stream\_func         read;             /\*!< Read callback function \*/

    stream\_func         write;            /\*!< Write callback function \*/

    int                 buffer\_len;       /\*!< Buffer length use for an Element \*/

    int                 task\_stack;       /\*!< Element task stack \*/

    int              task\_prio;        /\*!< Element task priority (based on freeRTOS priority) \*/

    int                 task\_core;        /\*!< Element task running in core (0 or 1) \*/

    int                 out\_rb\_size;      /\*!< Output ringbuffer size \*/

    void                \*data;            /\*!< User context \*/

    const char          \*tag;             /\*!< Element tag \*/

    int                 multi\_in\_rb\_num;  /\*!< The number of multiple input ringbuffer \*/

    int                 multi\_out\_rb\_num; /\*!< The number of multiple output ringbuffer \*/

} audio\_element\_cfg\_t;

### struct audio\_element

struct audio\_element

    /\* Functions/RingBuffers \*/

    io\_func                     open;

    io\_func                     seek;

    process\_func               process;

    io\_func                     close;

    io\_func                     destroy;

    io\_type\_t                   read\_type;

    union {

        ringbuf\_handle\_t        input\_rb;

        io\_callback\_t           read\_cb;

    } in;

    io\_type\_t                   write\_type;

    union {

        ringbuf\_handle\_t        output\_rb;

        io\_callback\_t           write\_cb;

    } out;

    audio\_multi\_rb\_t            multi\_in;

    audio\_multi\_rb\_t            multi\_out;

    /\* Properties \*/

    bool                        is\_open;

    audio\_element\_state\_t       state;

    events\_type\_t               events\_type;

    audio\_event\_iface\_handle\_t  iface\_event;

    audio\_callback\_t            callback\_event;

    int                         buf\_size;

    char                        \*buf;

    char                        \*tag;

    int                         task\_stack;

    int                         task\_prio;

    int                         task\_core;

    pthread\_mutex\_t            lock;

    audio\_element\_info\_t        info;

    audio\_element\_info\_t        \*report\_info;

    /\* PrivateData \*/

    void                        \*data;

    EventGroupHandle\_t          state\_event;

    int                         input\_wait\_time;

    int                         output\_wait\_time;

    int                         out\_buf\_size\_expect;

    int                         out\_rb\_size;

    bool                        is\_running;

    bool                        task\_run;

    bool                        stopping;

};

### struct audio\_element\_info\_t

typedef struct {

    int sample\_rates;                   /\*!< Sample rates in Hz \*/

    int channels;                        /\*!< Number of audio channel, mono is 1, stereo is 2 \*/

    int bits;                              /\*!< Bit wide (8, 16, 24, 32 bits) \*/

    int64\_t byte\_pos;                   /\*!< The current position (in bytes) being processed for an element \*/

    int64\_t total\_bytes;                /\*!< The total bytes for an element \*/

    char \*uri;                            /\*!< URI (optional) \*/

    audio\_codec\_t codec\_fmt;         /\*!< Music format (optional) \*/

    audio\_element\_reserve\_data\_t reserve\_data; /\*< This value is reserved for user use (optional) \*/

} audio\_element\_info\_t;

### enum audio\_element\_state\_t

typedef enum {

    AEL\_STATE\_NONE = 0,

    AEL\_STATE\_INIT,

    AEL\_STATE\_RUNNING,

    AEL\_STATE\_PAUSED,

    AEL\_STATE\_STOPPED,

    AEL\_STATE\_FINISHED,

    AEL\_STATE\_ERROR

} audio\_element\_state\_t;

### enum audio\_element\_msg\_cmd\_t

/\*\*

 \* Audio element action command, process on dispatcher

 \*/

typedef enum {

    AEL\_MSG\_CMD\_NONE                = 0,

    AEL\_MSG\_CMD\_ERROR               = 1,

    AEL\_MSG\_CMD\_FINISH              = 2,

    AEL\_MSG\_CMD\_STOP                = 3,

    AEL\_MSG\_CMD\_PAUSE               = 4,

    AEL\_MSG\_CMD\_RESUME              = 5,

    AEL\_MSG\_CMD\_DESTROY             = 6,

    // AEL\_MSG\_CMD\_CHANGE\_STATE        = 7,

    AEL\_MSG\_CMD\_REPORT\_STATUS       = 8,

    AEL\_MSG\_CMD\_REPORT\_MUSIC\_INFO   = 9,

    AEL\_MSG\_CMD\_REPORT\_CODEC\_FMT    = 10,

    AEL\_MSG\_CMD\_REPORT\_POSITION     = 11,

} audio\_element\_msg\_cmd\_t

### enum audio\_element\_status\_t

/\*\*

 \* Audio element status report

 \*/

typedef enum {

    AEL\_STATUS\_NONE                     = 0,

    AEL\_STATUS\_ERROR\_OPEN               = 1,

    AEL\_STATUS\_ERROR\_INPUT              = 2,

    AEL\_STATUS\_ERROR\_PROCESS            = 3,

    AEL\_STATUS\_ERROR\_OUTPUT             = 4,

    AEL\_STATUS\_ERROR\_CLOSE              = 5,

    AEL\_STATUS\_ERROR\_TIMEOUT            = 6,

    AEL\_STATUS\_ERROR\_UNKNOWN            = 7,

    AEL\_STATUS\_INPUT\_DONE               = 8,

    AEL\_STATUS\_INPUT\_BUFFERING          = 9,

    AEL\_STATUS\_OUTPUT\_DONE              = 10,

    AEL\_STATUS\_OUTPUT\_BUFFERING         = 11,

    AEL\_STATUS\_STATE\_RUNNING            = 12,

    AEL\_STATUS\_STATE\_PAUSED             = 13,

    AEL\_STATUS\_STATE\_STOPPED            = 14,

    AEL\_STATUS\_STATE\_FINISHED           = 15,

    AEL\_STATUS\_MOUNTED                  = 16,

    AEL\_STATUS\_UNMOUNTED                = 17,

} audio\_element\_status\_t;

## API接口



### audio\_element\_init

/\*\*

 \* @brief      Initialize audio element with config.

 \* @param      config  The configuration

 \* @return

 \*     - audio\_elemenent handle object

 \*     - NULL

 \*/

audio\_element\_handle\_t audio\_element\_init(audio\_element\_cfg\_t \*config);

### audio\_element\_deinit

/\*\*

 \* @brief      Destroy audio element handle object, stop, clear, deletel all.

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_deinit(audio\_element\_handle\_t el);

### audio\_element\_set\_tag

/\*\*

 \* @brief      Set elemenet tag name, or clear if tag = NULL.

 \* @param[in]  el    The audio element handle

 \* @param[in]  tag   The tag name pointer

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_set\_tag(audio\_element\_handle\_t el, const char \*tag);

### audio\_element\_set\_info

/\*\*

 \* @brief      Set audio element infomation.

 \* @param[in]  el    The audio element handle

 \* @param      info  The information pointer

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

esp\_err\_t audio\_element\_setinfo(audio\_element\_handle\_t el, audio\_element\_info\_t \*info);

### audio\_element\_set\_uri

/\*\*

 \* @brief      Set audio element URI.

 \* @param[in]  el    The audio element handle

 \* @param[in]  uri   The uri pointer

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_set\_uri(audio\_element\_handle\_t el, const char \*uri);

### audio\_element\_run

/\*\*

 \* @brief      Start Audio Element.

 \*             With this function, audio\_element will start as freeRTOS task,

 \*             and put the task into 'PAUSED' state.

 \*             Note: Element does not actually start when this function returns

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_run(audio\_element\_handle\_t el);

### audio\_element\_terminate

/\*\*

 \* @brief      Terminate Audio Element.

 \*             With this function, audio\_element will exit the task function.

 \*             Note: this API only sends request. It does not actually terminate immediately when this function returns.

 \* @param[in]  el    The audio element handle

 \*

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_terminate(audio\_element\_handle\_t el);

### audio\_element\_stop

/\*\*

 \* @brief      Request stop of the Audio Element.

 \*             After receiving the stop request, the element will ignore the actions being performed

 \*             (read/write, wait for the ringbuffer ...) and close the task, reset the state variables.

 \*             Note: this API only sends requests, Element does not actually stop when this function returns

 \*

 \* @param[in]  el    The audio element handle

 \*

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_stop(audio\_element\_handle\_t el);

### audio\_element\_wait\_for\_stop

/\*\*

 \* @brief      After the `audio\_element\_stop` function is called, the Element task will perform some abort procedures.

 \*             This function will be blocked (Time is DEFAULT\_MAX\_WAIT\_TIME) until Element Task has done and exit.

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_wait\_for\_stop(audio\_element\_handle\_t el);

### audio\_element\_pause

/\*\*

 \* @brief      Request audio Element enter 'PAUSE' state.

 \*             In this state, the task will wait for any event

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_pause(audio\_element\_handle\_t el);

### audio\_element\_resume

/\*\*

 \* @brief      Request audio Element enter 'RUNNING' state.

 \*             In this state, the task listens to events and invokes the callback functions.

 \*             At the same time it will wait until the size/total\_size of the output ringbuffer is greater than or equal to `wait\_for\_rb\_threshold`.

 \*             If the timeout period has been exceeded and ringbuffer output has not yet reached `wait\_for\_rb\_threshold` then the function will return.

 \* @param[in]  el                     The audio element handle

 \* @param[in]  wait\_for\_rb\_threshold  The wait for rb threshold (0 .. 1)

 \* @param[in]  timeout                The timeout

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_resume(audio\_element\_handle\_t el, float wait\_for\_rb\_threshold, TickType\_t timeout);

### audio\_element\_msg\_set\_listener

/\*\*

 \* @brief      This function will add a `listener` to listen to all events from audio element `el`.

 \*             Any event from el->external\_event will be send to the `listener`.

 \* @param      el           The audio element handle

 \* @param      listener     The event will be listen to

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_msg\_set\_listener(audio\_element\_handle\_t el, audio\_event\_iface\_handle\_t listener);

### audio\_element\_set\_event\_callback

/\*\*

 \* @brief      This function will add a `callback` to be called from audio element `el`.

 \*             Any event to caller will cause to call callback function.

 \* @param      el           The audio element handle

 \* @param      cb\_func      The callback function

 \* @param      ctx          Caller context

 \*

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_set\_event\_callback(audio\_element\_handle\_t el, event\_cb\_func cb\_func, void \*ctx);

### audio\_element\_msg\_remove\_listener

/\*\*

 \* @brief      Remove listener out of el.

 \*             No new events will be sent to the listener.

 \* @param[in]  el        The audio element handle

 \* @param      listener  The listener

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_msg\_remove\_listener(audio\_element\_handle\_t el, audio\_event\_iface\_handle\_t listener);

### audio\_element\_set\_input\_ringbuf

/\*\*

 \* @brief      Set Element input ringbuffer

 \*

 \* @param[in]  el    The audio element handle

 \* @param[in]  rb    The ringbuffer handle

 \*

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_set\_input\_ringbuf(audio\_element\_handle\_t el, ringbuf\_handle\_t rb);

### audio\_element\_set\_output\_ringbuf

/\*\*

 \* @brief      Set Element output ringbuffer.

 \* @param[in]  el    The audio element handle

 \* @param[in]  rb    The ringbuffer handle

 \*

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_set\_output\_ringbuf(audio\_element\_handle\_t el, ringbuf\_handle\_t rb);

### audio\_element\_abort\_input\_ringbuf

/\*\*

 \* @brief      If the element is requesting data from the input ringbuffer, this function forces it to abort.

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_abort\_input\_ringbuf(audio\_element\_handle\_t el);

### audio\_element\_abort\_output\_ringbuf

/\*\*

 \* @brief      If the element is waiting to write data to the ringbuffer output, this function forces it to abort.

 \* @param[in]  el   The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_abort\_output\_ringbuf(audio\_element\_handle\_t el);

### audio\_element\_wait\_for\_buffer

/\*\*

 \* @brief      This function will wait until the sizeof the output ringbuffer is greater than or equal to `size\_expect`.

 \*             If the timeout period has been exceeded and ringbuffer output has not yet reached `size\_expect`

 \*             then the function will return `ESP\_FAIL`

 \* @param[in]  el           The audio element handle

 \* @param[in]  size\_expect  The size expect

 \* @param[in]  timeout      The timeout

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_wait\_for\_buffer(audio\_element\_handle\_t el, int size\_expect, TickType\_t timeout);

### audio\_element\_report\_status

/\*\*

 \* @brief      Element will sendout event (status) to event by this function.

 \* @param[in]  el      The audio element handle

 \* @param[in]  status  The status

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_report\_status(audio\_element\_handle\_t el, audio\_element\_status\_t status);

### audio\_element\_report\_info

/\*\*

 \* @brief      Element will sendout event (information) to event by this function.

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_report\_info(audio\_element\_handle\_t el);

### audio\_element\_report\_codec\_fmt

/\*\*

 \* @brief      Element will sendout event (codec format) to event by this function.

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_report\_codec\_fmt(audio\_element\_handle\_t el);

### audio\_element\_report\_pos

/\*\*

 \* @brief      Element will sendout event with a duplicate information by this function.

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*     - ESP\_ERR\_NO\_MEM

 \*/

esp\_err\_t audio\_element\_report\_pos(audio\_element\_handle\_t el);

### audio\_element\_input

/\*\*

 \* @brief      Call this function to provice Element input data.

 \*             Depending on setup using ringbuffer or function callback, Element invokes read ringbuffer, or calls read callback funtion.

 \* @param[in]  el            The audio element handle

 \* @param      buffer        The buffer pointer

 \* @param[in]  wanted\_size   The wanted size

 \* @return

 \*        - > 0 number of bytes produced

 \*        - <=0 audio\_element\_err\_t

 \*/

audio\_element\_err\_t audio\_element\_input(audio\_element\_handle\_t el, char \*buffer, int wanted\_size);

### audio\_element\_output

/\*\*

 \* @brief      Call this function to sendout Element output data.

 \*             Depending on setup using ringbuffer or function callback, Element will invoke write to ringbuffer, or call write callback funtion.

 \* @param[in]  el          The audio element handle

 \* @param      buffer      The buffer pointer

 \* @param[in]  write\_size  The write size

 \* @return

 \*        - > 0 number of bytes written

 \*        - <=0 audio\_element\_err\_t

 \*/

audio\_element\_err\_t audio\_element\_output(audio\_element\_handle\_t el, char \*buffer, int write\_size);

### audio\_element\_set\_read\_cb

/\*\*

 \* @brief     This API allows the application to set a read callback for the first audio\_element in the pipeline for allowing the pipeline to interface with other systems. The callback is invoked every time the audio element requires data to be processed.

 \* @param[in]  el        The audio element handle

 \* @param[in]  fn        Callback read function. The callback function should return number of bytes read or -1

 \*                       in case of error in reading. Note that the callback function may decide to block and

 \*                       that may block the entire pipeline.

 \* @param[in]  context   An optional context which will be passed to callback function on every invocation

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_set\_read\_cb(audio\_element\_handle\_t el, stream\_func fn, void \*context);

### audio\_element\_set\_write\_cb

/\*\*

 \* @brief     This API allows the application to set a write callback for the last audio\_element in the pipeline for allowing the pipeline to interface with other systems.The callback is invoked every time the audio element has a processed data that needs to be passed forward.

 \* @param[in]  el        The audio element

 \* @param[in]  fn        Callback write function

 \*  The callback function should return number of bytes written or -1 in case of error in writing.

 \*  Note that the callback function may decide to block and that may block the entire pipeline.

 \* @param[in]  context   An optional context which will be passed to callback function on every invocation

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_set\_write\_cb(audio\_element\_handle\_t el, stream\_func fn, void \*context);

### audio\_element\_process\_init

/\*\*

 \* @brief      Provides a way to call element's `open`

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_process\_init(audio\_element\_handle\_t el);

### audio\_element\_process\_deinit

/\*\*

 \* @brief      Provides a way to call elements's `close`

 \* @param[in]  el    The audio element handle

 \* @return

 \*     - ESP\_OK

 \*     - ESP\_FAIL

 \*/

esp\_err\_t audio\_element\_process\_deinit(audio\_element\_handle\_t el);

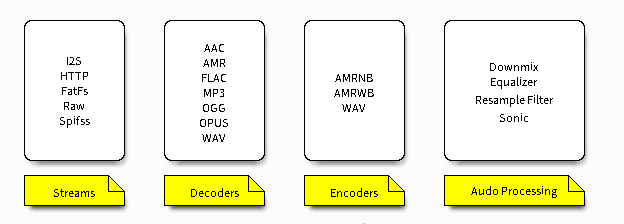
# Pipeline模块

## pipeline概念和组成

将音频的一般流程对象化，如Codecs，Streams或Filters。Pipeline(管道)：通过组合Elements加入到Pipeline来开发应用程序。把每个模块都看成是一个element，然后构建连接和操作这些element的方法，用户可以通过自己的需求把不同elements排列组合，形成一个又一个不同的pipeline，pipeline像是一条完整的解决方案。

pipeline的作用是控制音频数据流以及把音频元素Elements和各自的ringbuffer连接起来。不管是连接还是启动Element都是按顺序来进行的，从前一个Element那里检索数据并把它传给下一个Element。当然也要获取各个Element的事件，处理事件或发送给更高层。通过event来建立管道pipeline中各音频元素Elements之间的通信。

下图是基于pipeline的框架中抽象出来的4类Element：



## 基于pipeline播放步骤

音频播放的一般流程:获取音频流(音频输入流)-->音频流加工处理---->音频输出流。以播放sdcard里面的mp3音频为例，其一般步骤如下：

[ 1 ] 启动codec芯片

[2.0] 创建用于播放的音频管道

[2.1]创建fatfs流以从sdcard读取数据

[2.2]创建i2s流以将数据写入codec芯片

[2.3]创建mp3解码器以解码mp3文件

[2.4]将所有元素注册到音频管道

audio\_pipeline\_register(pipeline, fatfs\_stream\_reader, “file”);

audio\_pipeline\_register(pipeline, mp3\_decoder, “mp3”);

audio\_pipeline\_register(pipeline, i2s\_stream\_writer, “i2s”);

[2.5] 将其链接在一起[sdcard]–>fatfs\_stream–>mp3\_decoder–>i2s\_stream–>[codec\_chip] audio\_pipeline\_link(pipeline, (const char []) {“file”, “mp3”, “i2s”}, 3);

[2.6] Set up uri (file as fatfs\_stream, mp3 as mp3 decoder, and default output is i2s)

[3] Set up event listener

[3.1]来自管道所有元素的Listening event

[3.2]来自外围设备的Listening event

[ 4 ] 启动音频管道

[ 5 ] 监听所有管道事件 while (1) / 当最后一个管道元素（在这种情况下为i2s\_stream\_writer）接收到停止事件时停止\*/

[ 6 ] 停止音频管道

## 数据结构

### struct audio\_pipeline

struct audio\_pipeline {

    audio\_element\_list\_t        el\_list; //audio\_element\_item链表

    ringbuf\_list\_t               rb\_list; //ringbuff链表

    audio\_element\_state\_t       state;

    pthread\_mutex\_t            lock;

    bool                        linked;

    audio\_event\_iface\_handle\_t  listener;

};

typedef struct audio\_pipeline\_cfg {

     int rb\_size;         /\*!< Audio Pipeline ringbuffer size \*/

} audio\_pipeline\_cfg\_t;

## API接口

### audio\_pipeline\_init

[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t)audio\_pipeline\_init( [audio\_pipeline\_cfg\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv420audio_pipeline_cfg_t) \*config)

初始化audio\_pipeline\_handle\_t对象audio\_pipeline负责控制音频数据流并将音频元素与ringbuffer连接起来，它会依次连接并启动音频元素，负责从前一个元素中取出数据并传递给后一个元素。还可以从每个元素中获取事件、处理事件或将其传递给更高层。

**返回**

* audio\_pipeline\_handle\_t 成功
* 出现任何错误时为 NULL

**参数**

* config：配置 - audio\_pipeline\_cfg\_t

### audio\_pipeline\_deinit

esp\_err\_t audio\_pipeline\_deinit( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

该函数删除audio\_pipeline中所有元素的链接，取消所有事件的注册，调用已注册元素的destroy函数，并释放init函数分配的内存。简而言之，释放所有内存。

**返回**

ESP\_OK

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_register

audio\_pipeline\_register（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t)pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)el，const char\* name）

为audio\_pipeline注册一个元素，每个元素可以注册多次，但是name（作为String）在audio\_pipeline中必须是唯一的，用于标识前面提到的创建链接的元素audio\_pipeline\_link.

因为停止流水线或暂停流水线很大程度上取决于寄存器顺序。请严格按照以下顺序注册元素：输入元素在前，过程中间，输出元素在后。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄
* [in] el: 音频元素句柄
* [in] name: 此 audio\_pipeline 中 audio\_element 的名称标识符

### audio\_pipeline\_unregister

esp\_err\_t audio\_pipeline\_unregister（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)el ）

在 audio\_pipeline 中取消注册 audio\_element，将其从列表中删除。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄
* [in] el: 音频元素句柄

### audio\_pipeline\_run

esp\_err\_t audio\_pipeline\_run( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

启动音频管道。

使用此函数，audio\_pipeline 将为已使用链接函数链接的所有元素创建任务。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_terminate

esp\_err\_t audio\_pipeline\_terminate( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

停止音频管道。

使用此函数，audio\_pipeline 将销毁已使用链接函数链接的所有元素的任务。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_terminate\_with\_ticks

esp\_err\_t audio\_pipeline\_terminate\_with\_ticks（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，TickType\_t ticks\_to\_wait ）

停止具有特定滴答声的音频管道超时。

使用此函数，audio\_pipeline 将销毁已使用链接函数链接的所有元素的任务。

**返回**

* ESP\_OK
* ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄
* [in] ticks\_to\_wait: 阻塞等待元素销毁的最长时间

### audio\_pipeline\_resume

esp\_err\_t audio\_pipeline\_resume( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

此函数会将所有元素设置为RUNNING状态，并将音频数据作为 audio\_pipeline 的固有特性进行处理。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_pause

esp\_err\_t audio\_pipeline\_pause( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

此函数会将所有元素设置为PAUSED状态。除了数据处理停止之外，一切都保持不变。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_stop

esp\_err\_t audio\_pipeline\_stop( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

停止所有链接的元素。用于audio\_pipeline\_wait\_for\_stop保持同步。管道中元素的链接状态保持不变，事件仍被注册。停止的 audio\_pipeline 由 重新启动audio\_pipeline\_resume。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_wait\_for\_stop

esp\_err\_t audio\_pipeline\_wait\_for\_stop( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

该audio\_pipeline\_stop函数向元素发送请求并退出。但他们需要时间来摆脱时间阻塞的任务。此函数将等到portMAX\_DELAY管道中的所有元素实际停止。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_wait\_for\_stop\_with\_ticks

esp\_err\_t audio\_pipeline\_wait\_for\_stop\_with\_ticks（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，TickType\_t ticks\_to\_wait ）

该audio\_pipeline\_stop函数向元素发送请求并退出。但他们需要时间来摆脱时间阻塞的任务。此函数将等到ticks\_to\_wait管道中的所有元素实际停止。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄
* [in] ticks\_to\_wait: 阻塞等待停止的最长时间

### audio\_pipeline\_link

esp\_err\_t audio\_pipeline\_link( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline, const char \* link\_tag [], int link\_num )

添加到 audio\_pipeline 的 audio\_element 在被此函数调用之前将被断开连接。根据name已经注册的元素audio\_pipeline\_register，数据的路径将按照link\_tag的顺序链接。索引 0 处的元素是第一个，索引是最终的。以及 audio\_pipeline 将订阅所有元素的事件。link\_num -1

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄
* link\_tag: 元素数组由name注册audio\_pipeline\_register
* [in] link\_numlink\_tag:数组的元素总数

### audio\_pipeline\_unlink

esp\_err\_t audio\_pipeline\_unlink( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

移除元素的连接，以及取消订阅事件。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_get\_el\_by\_tag

[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html#_CPPv422audio_element_handle_t)audio\_pipeline\_get\_el\_by\_tag( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline, const char \*tag）

按标签从已注册的管道中查找未保留的元素。

**返回**

* 出现任何错误时为 NULL
* 其他人谈成功

**参数**

* [in] pipeline: 音频管道句柄
* [in] tag: 一个字符指针

### audio\_pipeline\_get\_el\_once

[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html#_CPPv422audio_element_handle_t)audio\_pipeline\_get\_el\_once( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline, const [audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html#_CPPv422audio_element_handle_t)start\_el ,const char \*tag）

基于开始元素，通过标签从注册的管道中查找未保留的元素。

**返回**

* 出现任何错误时为 NULL
* 其他人谈成功

**参数**

* [in] pipeline: 音频管道句柄
* [in] start\_el: 特定的开始元素
* [in] tag: 一个字符指针

### audio\_pipeline\_remove\_listener

esp\_err\_t audio\_pipeline\_remove\_listener( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline)

从此 audio\_pipeline 中删除事件侦听器。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_set\_listener

esp\_err\_t audio\_pipeline\_set\_listener（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t)evt ）

为这个音频管道设置事件监听器，来自这个管道的任何事件都可以被监听evt

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄
* [in] evt: 事件句柄

### audio\_pipeline\_get\_event\_iface

[audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t)audio\_pipeline\_get\_event\_iface( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline)

获取此管道使用的事件 iface。

**返回**

事件句柄

**参数**

* [in] pipeline: 管道

### audio\_pipeline\_link\_insert

esp\_err\_t audio\_pipeline\_link\_insert（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，bool first，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)prev，[ringbuf\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/abstraction/ringbuf.html" \l "_CPPv416ringbuf_handle_t" \o "ringbuf_handle_t)conect\_rb，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)next ）

将特定的audio\_element插入audio\_pipeline，前一个元素通过环形缓冲区连接到下一个元素。

**返回**

* ESP\_OK
* ESP\_FAIL

**参数**

* [in] pipeline：音频管道句柄
* [in] first: 前一个元素是第一个输入元素，需要设置true
* [in] prev: 上一个元素
* [in] conect\_rb: 连接环形缓冲区
* [in] next: 下一个元素

### audio\_pipeline\_register\_more

esp\_err\_t audio\_pipeline\_register\_more（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)element\_1，... ）

向 audio\_pipeline 注册一个以 NULL 结尾的元素列表。

**返回**

* ESP\_OK
* ESP\_FAIL

**参数**

* [in] pipeline：音频管道句柄
* [in] element\_1：要添加到 audio\_pipeline 的元素。
* [in] ...：要添加到 audio\_pipeline 的附加元素。

### audio\_pipeline\_unregister\_more

esp\_err\_t audio\_pipeline\_unregister\_more（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)element\_1，... ）

取消注册一个以 NULL 结尾的元素列表到 audio\_pipeline。

**返回**

* ESP\_OK
* ESP\_FAIL

**参数**

* [in] pipeline：音频管道句柄
* [in] element\_1：要添加到 audio\_pipeline 的元素。
* [in] ...：要添加到 audio\_pipeline 的附加元素。

### audio\_pipeline\_link\_more

esp\_err\_t audio\_pipeline\_link\_more（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)element\_1，... ）

将一个以 NULL 结尾的元素列表添加到 audio\_pipeline。

**返回**

* ESP\_OK
* ESP\_FAIL

**参数**

* [in] pipeline：音频管道句柄
* [in] element\_1：要添加到 audio\_pipeline 的元素。
* [in] ...：要添加到 audio\_pipeline 的附加元素。

### audio\_pipeline\_listen\_more

esp\_err\_t audio\_pipeline\_listen\_more（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)element\_1，... ）

为 audio\_pipeline 订阅以 NULL 结尾的元素事件列表。

**返回**

* ESP\_OK
* ESP\_FAIL

**参数**

* [in] pipeline：音频管道句柄
* [in] element\_1: 订阅 audio\_pipeline 的元素事件。
* [in] ...: 订阅 audio\_pipeline 的附加元素事件。

### audio\_pipeline\_check\_items\_state

esp\_err\_t audio\_pipeline\_check\_items\_state（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)dest\_el，[audio\_element\_status\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_status_t" \o "audio_element_status_t) status）

更新目标元素状态并检查所有链接元素状态是否相同。

**返回**

* ESP\_OK 所有链接的元素状态都相同。
* ESP\_FAIL 所有链接元素状态不同。

**参数**

* [in] pipeline：音频管道句柄
* [in] dest\_el: 目标元素
* [in] status: 新状态

### audio\_pipeline\_reset\_items\_state

esp\_err\_t audio\_pipeline\_reset\_items\_state( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline)

将管道元素项状态重置为AEL\_STATUS\_NONE

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_reset\_ringbuffer

esp\_err\_t audio\_pipeline\_reset\_ringbuffer( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline)

重置管道元素环形缓冲区。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_reset\_elements

esp\_err\_t audio\_pipeline\_reset\_elements( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline)

重置管道链接元素状态。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄

### audio\_pipeline\_reset\_kept\_state

esp\_err\_t audio\_pipeline\_reset\_kept\_state（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)el ）

重置特定元素保持状态。

**返回**

* ESP\_OK 成功
* 出现任何错误时的 ESP\_FAIL

**参数**

* [in] pipeline: 音频管道句柄
* [in] el：音频元素句柄

### audio\_pipeline\_breakup\_elements

esp\_err\_t audio\_pipeline\_breakup\_elements( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)keep\_ctx\_el )

分解特定的所有链接元素pipeline。包含和kept\_ctx\_el工作前（AEL\_STATE\_RUNNING 或 AEL\_STATE\_PAUSED）元素和连接的环形缓冲区将被保留。

**笔记**

kept\_ctx\_el为 NULL时没有保留元素。此函数将取消订阅所有元素的事件。

**返回**

* ESP\_OK 所有链接的元素状态都相同。
* ESP\_ERR\_INVALID\_ARG 参数无效。

**参数**

* [in] pipeline：音频管道句柄
* [in] kept\_ctx\_el: 目的地保留元素

### audio\_pipeline\_relink

esp\_err\_t audio\_pipeline\_relink( [audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html" \l "_CPPv423audio_pipeline_handle_t" \o "audio_pipeline_handle_t) pipeline, const char \* link\_tag [], int link\_num )

name根据已经注册的元素，按照`link\_tag 中audio\_pipeline\_register的顺序重新链接管道。names

**笔记**

如果 ringbuffer 不足以连接新的管道将创建新的 ringbuffer。

**返回**

* ESP\_OK 所有链接的元素状态都相同。
* ESP\_FAIL 错误。
* ESP\_ERR\_INVALID\_ARG 参数无效。

**参数**

* [in] pipeline: 音频管道句柄
* link\_tagname:被注册的元素数组audio\_pipeline\_register
* [in] link\_numlink\_tag:数组的元素总数

### audio\_pipeline\_relink\_more

esp\_err\_t audio\_pipeline\_relink\_more（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv422audio_element_handle_t" \o "audio_element_handle_t)element\_1，... ）

将一个以 NULL 结尾的元素列表添加到 audio\_pipeline。

**笔记**

如果 ringbuffer 不足以连接新的管道将创建新的 ringbuffer。

**返回**

* ESP\_OK 所有链接的元素状态都相同。
* ESP\_FAIL 错误。
* ESP\_ERR\_INVALID\_ARG 参数无效。

**参数**

* [in] pipeline: 音频管道句柄
* [in] element\_1：要添加到 audio\_pipeline 的元素。
* [in] ...：要添加到 audio\_pipeline 的附加元素。

### audio\_pipeline\_change\_state

esp\_err\_t audio\_pipeline\_change\_state（[audio\_pipeline\_handle\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_pipeline.html#_CPPv423audio_pipeline_handle_t) pipeline，[audio\_element\_state\_t](https://docs.espressif.com/projects/esp-adf/zh_CN/latest/api-reference/framework/audio_element.html" \l "_CPPv421audio_element_state_t" \o "audio_element_state_t)new\_state ）

设置管道状态。

**返回**

* ESP\_OK 所有链接的元素状态都相同。
* ESP\_FAIL 错误。

**参数**

* [in] pipeline: 音频管道句柄
* [in] new\_state：将设置新状态

# Event Interface模块

ADF提供了事件接口API来在管道中的音频元素之间建立通信。该API是围绕FreeRTOS队列构建的。它实现了监听器来监视传入的消息，并使用回调函数通知消息。

应用例子：在包括Get-install / play\_mp3\_control的几个例子中，对该API的实现进行了演示。

## 数据结构

### audio\_event\_iface\_msg\_t

该结构体描述了消息队列中某一任务的详细信息。通过函数audio\_event\_iface\_listen监听消息队列，获取本次pipeline任务中来自pipeline内部event事件（例如来自decoder解码完成后的播放指令）或者一些外设事件（例如点击、长按等）的事件信息。

1）定义：

typedef struct {

    int cmd;                /\*!< Command id \*/

    void \*data;             /\*!< Data pointer \*/

    int data\_len;           /\*!< Data length \*/

    void \*source;           /\*!< Source event \*/

    int source\_type;        /\*!< Source type (To know where it came from) \*/

    bool need\_free\_data;    /\*!< Need to free data pointer after the event has been processed \*/

} audio\_event\_iface\_msg\_t;

2）成员含义

|  |  |  |
| --- | --- | --- |
| 成员 | 描述 | 列子 |
| cmd | 任务指令 | 枚举：audio\_element\_msg\_cmd\_t、periph\_touch\_event\_id\_t、periph\_button\_event\_id\_t、periph\_adc\_button\_event\_id\_t等 |
| data | 按键编号 | int8\_t：BUTTON\_MUTE\_ID、BUTTON\_SET\_ID等 |
| data\_len | 数据长度 | 根据data来自不同的source时，data类型不同，长度也不同。 |
| source | 任务源 | audio\_element\_handle\_t、periph\_service\_handle\_t、esp\_periph\_handle\_t、 |
| source\_type | 任务源类型 | 枚举：audio\_element\_type\_t、esp\_periph\_id\_t等 |
| need\_free\_data | 是否释放data指针 |  |

### audio\_event\_iface\_cfg\_t

该结构体用于设置audio\_event\_iface\_handle\_t参数配置。此处初始化为默认设置AUDIO\_EVENT\_IFACE\_DEFAULT\_CFG。

1. 定义

typedef struct {

int internal\_queue\_size; /\*!< It's optional, Queue size for event `internal\_queue` \*/

int external\_queue\_size; /\*!< It's optional, Queue size for event `external\_queue` \*/

on\_event\_iface\_func on\_cmd; /\*!< Function callback for listener when any event arrived \*/

void \*context; /\*!< Context will pass to callback function \*/

TickType\_t wait\_time; /\*!< Timeout to check for event queue \*/

int type; /\*!< it will pass to audio\_event\_iface\_msg\_t source\_type (To know where it came from) \*/

} audio\_event\_iface\_cfg\_t;

1. 成员含义

|  |  |  |
| --- | --- | --- |
| 成员 | 描述 | AUDIO\_EVENT\_IFACE\_DEFAULT\_CFG |
| internal\_queue\_size | 内部消息队列大小 | 5 |
| external\_queue\_size | 外部消息队列大小 | 5 |
| on\_event\_iface\_func | 回调函数 | Null；类型为函数指针；参数为audio\_event\_iface\_msg\_t \*，void \* |
| context | 回调函数参数 | Null； |
| wait\_time | 检查消息队列时限 | 0xffffffffUL；类型为uint32\_t |
| type | 任务源类型 | 0；同source\_type |

### audio\_event\_iface\_item\_t

该结构体用于描述消息队列句柄和大小，并且结构体成员next，以宏扩展的方式记录了，消息队列链表中下一个audio\_event\_iface\_item类型的结构体。

typedef struct audio\_event\_iface\_item {

STAILQ\_ENTRY(audio\_event\_iface\_item) next;

mqd\_t queue;

int queue\_size;

int mark\_to\_remove;

} audio\_event\_iface\_item\_t;

## 类型定义

### on\_event\_iface\_func

该函数指针类型定义如下：

typedef esp\_err\_t (\*on\_event\_iface\_func)(audio\_event\_iface\_msg\_t \*, void \*);

此函数指针类型为audio\_event\_iface\_cfg\_t和audio\_event\_iface结构体成员类型。该成员用于定义，每个element或者peripherals listener在收到来自内部消息队列信息后的，回调操作。

针对element的回调函数为audio\_element\_on\_cmd；

针对peripherals的回调函数为process\_peripheral\_event，可为空。

### audio\_event\_iface\_handle\_t

1）定义

该结构体指针类型定义如下：

Typedef struct audio\_event\_iface \*audio\_event\_iface\_handle\_t

struct audio\_event\_iface {

mqd\_t internal\_queue;

mqd\_t external\_queue;

char internal\_mqname[16] ;

int internal\_queue\_size;

int external\_queue\_size;

audio\_event\_iface\_list\_t listening\_queues;

void \*context;

on\_event\_iface\_func on\_cmd;

int wait\_time;

int type; };

2）成员描述

该结构体部分成员介绍如下：

|  |  |  |
| --- | --- | --- |
| 成员 | 描述 | 类型 |
| internal\_queue | 内部消息队列句柄 | typedef struct QueueDefinition \*mqd\_t;  为FreeRTOS queue.h中定义的类型 |
| external\_queue | 外部消息队列句柄 |
| audio\_event\_iface\_list\_t | 消息队列链表结构体类型 | 描述了以audio\_event\_iface\_item类型为元素的结构体链表的链表头和链表尾的下一个。 |

### API接口

### audio\_event\_iface\_init

[audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) audio\_event\_iface\_init ([audio\_event\_iface\_cfg\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv423audio_event_iface_cfg_t" \o "audio_event_iface_cfg_t) \*config)

Initialize audio event.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* config: The configurations

### audio\_event\_iface\_destroy

esp\_err\_t audio\_event\_iface\_destroy([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t) evt)

Cleanup event, it doesn’t free evt pointer.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* evt: The event

### audio\_event\_iface\_set\_listener

esp\_err\_t audio\_event\_iface\_set\_listener([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) evt, [audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t) listener)

Add audio event evt to the listener, then we can listen evt event from listen

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* listener: The event can listen another event
* evt: The event to be added to

### audio\_event\_iface\_remove\_listener

esp\_err\_t audio\_event\_iface\_remove\_listener([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) listener,  [audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) evt)

Remove audio event evt from the listener.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* listener: The event listener
* evt: The event to be removed from

### audio\_event\_iface\_set\_cmd\_waiting\_timeout

esp\_err\_t audio\_event\_iface\_set\_cmd\_waiting\_timeout([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) evt, TickType\_t wait\_time)

Set current queue wait time for the event.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* evt: The event
* [in] wait\_time: The wait time

### audio\_event\_iface\_waiting\_cmd\_msg

esp\_err\_t audio\_event\_iface\_waiting\_cmd\_msg([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) evt)

Waiting internal queue message.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* evt: The event

### audio\_event\_iface\_cmd

esp\_err\_t audio\_event\_iface\_cmd([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t) evt, [audio\_event\_iface\_msg\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv423audio_event_iface_msg_t" \o "audio_event_iface_msg_t) \*msg)

Trigger an event for internal queue with a message.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* evt: The event
* msg: The message

### audio\_event\_iface\_cmd\_from\_isr

esp\_err\_t audio\_event\_iface\_cmd\_from\_isr([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) evt, [audio\_event\_iface\_msg\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv423audio_event_iface_msg_t" \o "audio_event_iface_msg_t) \*msg)

It’s same with audio\_event\_iface\_cmd, but can send a message from ISR.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* [in] evt: The event
* msg: The message

### audio\_event\_iface\_sendout

esp\_err\_t audio\_event\_iface\_sendout([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t) evt, [audio\_event\_iface\_msg\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv423audio_event_iface_msg_t" \o "audio_event_iface_msg_t) \*msg)

Trigger and event out with a message.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* evt: The event
* msg: The message

### audio\_event\_iface\_discard

esp\_err\_t audio\_event\_iface\_discard([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t) evt)

Discard all ongoing event message.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* evt: The event

### audio\_event\_iface\_listen

esp\_err\_t audio\_event\_iface\_listen([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t) evt, [audio\_event\_iface\_msg\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv423audio_event_iface_msg_t" \o "audio_event_iface_msg_t) \*msg, TickType\_t wait\_time)

Listening and invoke callback function if there are any event are comming.

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* evt: The event
* msg: The message
* wait\_time: The wait time

### audio\_event\_iface\_get\_queue\_handle

QueueHandle\_t audio\_event\_iface\_get\_queue\_handle([audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) evt)

Get External queue handle of Emmitter.

**Return**

Externalmqd\_t

**Parameters**

* [in] evt: The external queue

### audio\_event\_iface\_read

***esp\_err\_t audio\_event\_iface\_read(****[audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t)* ***evt,****[audio\_event\_iface\_msg\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv423audio_event_iface_msg_t" \o "audio_event_iface_msg_t)****\*msg, TickType\_t wait\_time)***

Read the event from all the registered event emitters in the queue set of the interface.

**Return**

* ESP\_OK On successful receiving of event
* ESP\_FAIL In case of a timeout or invalid parameter passed

**Parameters**

* [in] evt: The event interface
* [out] msg: The pointer to structure in which event is to be received
* [in] wait\_time: Timeout for receiving event

### audio\_event\_iface\_get\_msg\_queue\_handle

***QueueHandle\_t audio\_event\_iface\_get\_msg\_queue\_handle(***[***audio\_event\_iface\_handle\_t***](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t) ***evt)***

Get Internal queue handle of Emmitter.

**Return**

Internalmqd\_t

**Parameters**

* [in] evt: The Internal queue

### audio\_event\_iface\_set\_msg\_listener

***esp\_err\_t audio\_event\_iface\_set\_msg\_listener(***[*audio\_event\_iface\_handle\_t*](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html#_CPPv426audio_event_iface_handle_t)***evt,****[audio\_event\_iface\_handle\_t](https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/framework/audio_event_iface.html" \l "_CPPv426audio_event_iface_handle_t" \o "audio_event_iface_handle_t)* ***listener)***

Add audio internal event evt to the listener, then we can listen evt event from listen

**Return**

* ESP\_OK
* ESP\_FAIL

**Parameters**

* listener: The event can listen another event

evt: The event to be added to

# pipeline通信流程

pipeline和各个element、外设间的通信，都依赖于event interface模块相关的数据结构和api接口实现。每一个Pipeline、element和外设实例，都有一个audio\_event\_iface\_handle\_t成员用于处理内部和外部消息。当移植到nuttx上后，audio\_event\_iface\_handle\_t结构体定义如下：  


其中内部消息队列internal\_queue，用于记录每个实例监听消息的消息队列；外部队列external\_queue，用于记录每个实例发送消息的消息队列。

## pipeline通信机制



以如上pipeline中，各element组成的实例，其通过内部和外部消息队列通信的流程图如下：



## 外设监听线程

外设的监听及处理消息的流程如下：



## Pipeline监听线程

Pipeline作为主线程，通过其内部队列，接受来自外设和element的反馈消息，并做相应的处理就，再发送消息到element或外设的内部消息队列，反馈信息。其工作流程如下：



## Element监听线程

每个element都会启动一个工作线程。该工作线程主要完成两项工作

1. 调用el->process。从前一个element的ringbuffer中读取音频数据流；process处理（解码、编码、重采样）相应的音频数据流；将处理后的数据流写入该element输出ringbuffer。
2. 监听来自本element内部消息队列的信息，完成相应停止、汇报信息、恢复等指令，必要时向pipeline反馈信息。

