



## TXW81x 音频方案开发指南



珠海泰芯半导体有限公司  
Zhuhai Taixin Semiconductor Co., Ltd

珠海市高新区港湾一号科创园港 11 栋 3 楼

保密等级	A	TXW81x 音频方案开发指南	文件编号	TX-0000
发行日期	2023-11-15		文件版本	V1.0

# 责任与版权

## 责任限制

由于产品版本升级或者其他原因，本文档会不定期更新。除非另行约定，泰芯半导体有限公司对本文档所有内容不提供任何担保或授权。

客户应在遵守法律、法规和安全要求的前提下进行产品设计，并做充分验证。泰芯半导体有限公司对应用帮助或客户产品设计不承担任何义务。客户应对其使用泰芯半导体有限公司的产品和应用自行负责。

在适用法律允许的范围内，泰芯半导体有限公司在任何情况下，都不对因使用本文档相关内容及本文档描述的产品而产生的损失和损害进行超过购买支付价款的赔偿（除在涉及人身伤害的情况中根据适用的法律规定的损害赔偿外）。

## 版权申明

泰芯半导体有限公司保留随时修改本文档中任何信息的权利，无需提前通知且不承担任何责任。

未经泰芯半导体有限公司书面同意，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。除非获得相关权利人的许可，否则，任何人不能以任何形式对前述软件进行复制、分发、修改、摘录、反编译、反汇编、解密、反向工程、出租、转让、分许可等侵犯本文档描述的享有版权的软件版权的行为，但是适用法禁止此类限制的除外。



珠海泰芯半导体有限公司  
Zhuhai Taixin Semiconductor Co., Ltd

珠海市高新区港湾一号科技园港 11 栋 3 楼

版权所有 侵权必究  
Copyright © 2023 by Tai Xin All rights reserved

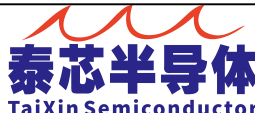
保密等级	A	TXW81x 音频方案开发指南	文件编号	TX-0000
发行日期	2023-11-15		文件版本	V1.0

修订记录

日期	版本	描 述	修订人
2023-11-15	V1.0	初始版本	TX

 <b>泰芯半导体</b> TaiXin Semiconductor	珠海泰芯半导体有限公司 Zhuhai Taixin Semiconductor Co.,Ltd	珠海市高新区港湾一号科创园港 11 栋 3 楼
---	--	-------------------------

版权所有 侵权必究 Copyright © 2023 by Tai Xin All rights reserved
--

保密等级	A	TXW81x 音频方案开发指南	文件编号	TX-0000
发行日期	2023-11-15		文件版本	V1.0
<div>目录</div> <div>TXW81x 音频方案开发指南..... 1</div> <div>1. 概述..... 1</div> <div>2. 硬件开发板..... 2</div> <div>    2.1. 音视频开发板..... 2</div> <div>        2.1.1. 音视频开发板接口介绍..... 2</div> <div>3. 音频开发配置流程..... 3</div> <div>    3.1. 音频方案相关配置..... 3</div> <div>    3.2. audio_adc 麦克风..... 3</div> <div>        3.2.1. audio_adc 音频相关参数配置..... 3</div> <div>        3.2.2. audio_adc 音频接口说明..... 4</div> <div>        3.2.3. audio_adc 配置流程以及应用层初始化说明 (audio_ad.c) ..... 4</div> <div>        3.2.4. audio_adc 音频数据流接收流程 (参考 AT_save_audio.c)..... 6</div>				
		珠海泰芯半导体有限公司 Zhuhai Taixin Semiconductor Co.,Ltd	珠海市高新区港湾一号科技园港 11 栋 3 楼	
版权所有 侵权必究 Copyright © 2023 by Tai Xin All rights reserved				

## 1. 概述

本文主要描述视频开发流程。

本文档主要适用于以下工程师：

- 技术支持工程师
- 方案软件开发工程师

本文档适用的产品范围：

型号	封装	包装
TXW81x		

## 2. 硬件开发板

为了快速入门和方案评估，我们提供各种应用场景的开发板。

### 2.1. 音视频开发板

#### 2.1.1. 音视频开发板接口介绍

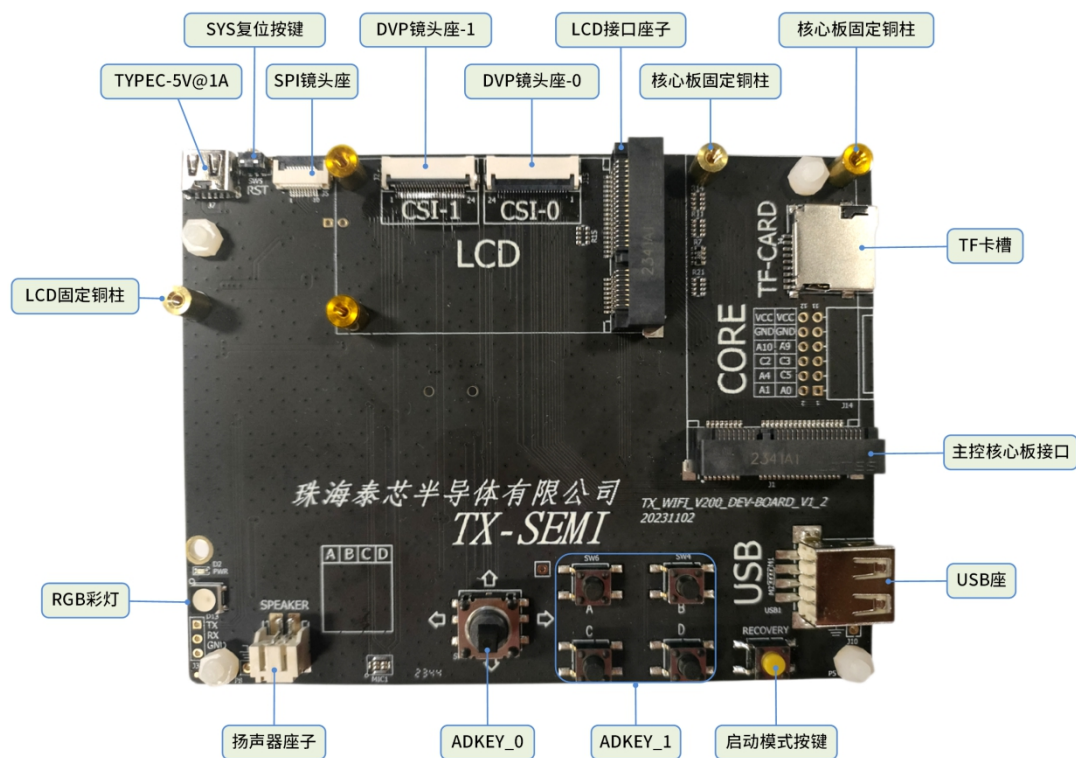


图 2.1.1.1 音视频开发板主视图

特殊说明

**模式启动按键：**此按键可以一键拯救系统，在芯片上电即跑死，cklink 烧写和其他升级都失效的情况下使用。

## 3. 音频开发配置流程

### 3.1. 音频方案相关配置

方案配置主要看 project\_config.h, 打开相关功能就可以打开 sdk 的某个功能, 与音频相关的分别是 AUDIO\_EN 和 AUDIO\_DAC\_EN

```
#define USB_HOST_EN          1
#define MACBUS_USB
#define BLE_PAIR_NET        0
#define WIRELESS_PAIR_CODE  0

#define PIN_SPI_PDN          0
#define PRC_EN               0
#define OF_EN                1
#define USB_EN               1
#define DVP_EN               1
#define VPP_EN               DVP_EN
#define JPG_EN               (1 && DVP_EN)
#define LCD_EN               1
#define SCALE_EN             1
#define SDH_EN               1
#define FS_EN                1
#define SD_SAVE              (0&&SDH_EN&&FS_EN&&JPG_EN)

#define VCAM_EN              (0 || DVP_EN)

#define OPENDML_EN           1
#define UART_FLY_CTRL_EN     0
#define PWM_EN               0

#define AUDIO_EN              1
#define AUDIO_DAC_EN          1
```

### 3.2. audio\_adc 麦克风

#### 3.2.1. audio\_adc 音频相关参数配置

在 sdk 中有自带音频的 demo, 主要是通过宏#define AUDIO\_EN 打开, 默认用的是 audio\_adc。

### 3.2.2. audio\_adc 音频接口说明

在 sdk 中 audio\_adc 的底层接口如下, 但 sdk 中已经将这部分音频数据流配置好, 如果没有特殊处理, 可以直接去接收音频数据。

```
int32 auadc_open(struct auadc_device *auadc, enum auadc_sample_rate sample_rate);  
//sample_rate:音频采样率
```

```
int32 auadc_close(struct auadc_device *auadc);
```

```
int32 auadc_read(struct auadc_device *auadc, void* buf, uint32 bytes);  
//buf:保存音频的地址, 要求 4byte 对齐  
//bytes:接收音频的 buf 长度
```

```
int32 auadc_ioctl(struct auadc_device *auadc, enum auadc_ioctl_cmd ioctl_cmd,  
uint32 param1, uint32 param2);
```

```
int32 auadc_request_irq(struct auadc_device *auadc, enum auadc_irq_flag irq_flag,  
auadc_irq_hdl irq_hdl, uint32 irq_data);  
int32 auadc_release_irq(struct auadc_device *auadc, enum auadc_irq_flag irq_flag);
```

### 3.2.3. audio\_adc 配置流程以及应用层初始化说明(audio\_ad.c)

- 1) audio\_adc\_init初始化, 创建发送流, 然后将音频硬件初始化, auadc\_open打开并且配置采样率, auadc\_request\_irq注册中断, audio\_adc\_start(自定义)是启动音频采集, audio\_adc\_start主要是调用auadc\_read来启动音频采集, audio\_adc\_register注册是应用层的代码, 是为了数据流而注册的一系列函数。



```

int audio_adc_init()
{
    int res = 0;
    struct auadc_device *adc = (struct auadc_device *)dev_get(HG_AUADC_DEVID);

    stream *s = NULL;
    s = open_stream_available(S_ADC_AUDIO, AUDIONUM, 0, opcode_func, NULL);
    if(!s)
    {
        res = -1;
        goto audio_adc_init_err;
    }

    struct audio_adc *audio_priv = (struct audio_adc *)s->priv;
    if(audio_priv)
    {
        struct audio_ad_config *ad_config = (struct audio_ad_config*)os_malloc(sizeof(struct audio_ad_config));
        memset(ad_config, 0, sizeof(struct audio_ad_config));
        ad_config->adc = adc;
        ad_config->priv_el = s;
        audio_priv->audio_hardware_hdl = ad_config;
        audio_adc_register(ad_config, s, AUDIOLEN, audio_set_buf, audio_get_buf);
        ad_config->irq_func = global_audio_ad_read;
        auadc_open(adc, AUADC_SAMPLE_RATE_8K);
        auadc_request_irq(adc, AUADC_IRQ_FLAG_HALF | AUADC_IRQ_FLAG_FULL, (auadc_irq_hdl)audio_adc_irq, (uint32)ad_config);
        audio_adc_start(ad_config);
    }

    audio_adc_init_err:
    return res;
}

```

- 2) opcode\_func是发送流的回调函数, 里面实现了数据流的释放以及空间申请操作, 最后流创建完成后, 会创建一个专门音频处理的任务, 如果有需要, 可以对音频进行滤波等功能, 也可以在接受流终端自己处理音频(尽量不要修改原数据)。
- 3) audio\_deal\_task(void \*arg)是音频处理任务, 可以在发送音频数据之前先将原数据处理(滤波去噪), 处理完后再发送到接收流。

```

static int opcode_func(stream *s, void *priv, int opcode)
{
    static uint8_t *adc_audio_buf = NULL;
    int res = 0;
    switch(opcode)
    {
        case STREAM_OPEN_ENTER:
            break;
        case STREAM_OPEN_EXIT:
            {
                s->priv = (void*)os_malloc(sizeof(struct audio_adc_s));
                if(s->priv)
                {
                    struct audio_adc_s *self_priv = (struct audio_adc_s*)s->priv;
                    self_priv->adc_msgq = (void*)csi_kernel_msgq_new(1, sizeof(uint8_t*));
                    OS_TASK_INIT("adc_audio_deal", &self_priv->thread_hdl, audio_deal_task, s, OS_TASK_PRIORITY_ABOVE_NORMAL, 1024);
                }
                //两个sample点
                adc_audio_buf = os_malloc(AUDIONUM * (AUDIOLEN + FILTER_SAMPLE_LEN*2));
                if(adc_audio_buf)
                {
                    stream_data_dis_mem(s, AUDIONUM);
                }
                //绑定到对应的流
                streamSrc_bind_streamDest(s, R_RECORD_AUDIO);
                streamSrc_bind_streamDest(s, R_RTP_AUDIO);
                streamSrc_bind_streamDest(s, R_AUDIO_TEST);
                streamSrc_bind_streamDest(s, R_SPEAKER);
                streamSrc_bind_streamDest(s, R_AT_SAVE_AUDIO);
                streamSrc_bind_streamDest(s, R_AT_AVI_AUDIO);
            }
            break;
        case STREAM_OPEN_FAIL:
            break;
        case STREAM_DATA_DIS:
            {
                struct data_structure *data = (struct data_structure *)priv;
                int data_num = (int)data->priv;
                data->type = DATA_TYPE_AUDIO_ADC; //设置声音的类型
                data->priv = (void*)AUDIOLEN;
                //注册对应函数
                data->ops = &stream_sound_ops;
                data->data = adc_audio_buf + (data_num)*(AUDIOLEN + FILTER_SAMPLE_LEN*2);
            }
            break;
        case STREAM_DATA_FREE:
            break;
        case STREAM_DATA_FREE_END:
            break;
        //数据发送完成, 可以选择唤醒对应的任务
        case STREAM_SEND_DATA_FINISH:
            break;
        default:
            //默认都返回成功
            break;
    }
    return res;
}

```

### 3.2.4. audio\_adc 音频数据流接收流程(参考 AT\_save\_audio.c)

- 1) 要先保证音频被打开audio\_adc\_init(), 默认sdk如果打开该功能, 默认会调用audio\_adc\_init(), audio\_ad.c中, 里面会创建音频发送流, 设置音频的采样率, 同时要bind对应的接收流。

```

static int opcode_func(stream *s, void *priv, int opcode)
{
    static uint8_t *adc_audio_buf = NULL;
    int res = 0;
    switch(opcode)
    {
        case STREAM_OPEN_ENTER:
            break;
        case STREAM_OPEN_EXIT:
        {
            s->priv = (void*)os_malloc(sizeof(struct audio_adc_s));
            if(s->priv)
            {
                struct audio_adc_s *self_priv = (struct audio_adc_s*)s->priv;
                self_priv->adc_msgq = (void*)csi_kernel_msgq_new(1, sizeof(uint8_t*));
                OS_TASK_INIT("adc_audio_deal", &self_priv->thread_hdl, audio_deal_task, s, OS_TASK_PRIORITY_ABOVE_NORMAL, 1024);
            }
            //两个sample点
            adc_audio_buf = os_malloc(AUDIONUM * (AUDIOLEN + FILTER_SAMPLE_LEN*2));
            if(adc_audio_buf)
            {
                stream_data_dis_mem(s, AUDIONUM);
            }
            //绑定到对应的流
            streamSrc_bind_streamDest(s, R_RECORD_AUDIO);
            streamSrc_bind_streamDest(s, R_RTP_AUDIO);
            streamSrc_bind_streamDest(s, R_AUDIO_TEST);
            streamSrc_bind_streamDest(s, R_SPEAKER);
            streamSrc_bind_streamDest(s, R_AT_SAVE_AUDIO);
            streamSrc_bind_streamDest(s, R_AT_AVI_AUDIO);
        }
    }
}

```

2) 创建对应接收流, 要保证音频发送流已经bind了改接收流。

伪代码:

```

stream* s = NULL;
uint8_t *buf;
uint32_t flen;
//创建接收音频流
s = open_stream_available(R_AT_SAVE_AUDIO, 0, 8, opcode_func, NULL);
//打开文件
fp = osal_fopen(filename, "wb+");
//接收音频
get_f = recv_real_data(s);
//获取音频的buf
buf = get_stream_real_data(get_f);
//获取当前音频帧长度
flen = get_stream_real_data_len(get_f);
//写卡
w_len = osal_fwrite(buf, flen, 1, fp);
//释放get_f
free_data(get_f);
osal_fclose(fp);

```