



应用指南：音频处理库

桃芯科技（苏州）有限公司

官网：www.ingchips.com

www.ingchips.cn

邮箱：service@ingchips.com

电话：010-85160285

地址：北京市海淀区知春路 49 号紫金数 3 号楼 8 层 803

深圳市南山区科技园曙光大厦 1009

版权申明

本文档以及其所包含的内容为桃芯科技（苏州）有限公司所有，并受中国法律和其他可适用的国际公约的版权保护。未经桃芯科技的事先书面许可，不得在任何其他地方以任何形式（有形或无形的）以任何电子或其他方式复制、分发、传输、展示、出版或广播，不允许从内容的任何副本中更改或删除任何商标、版权或其他通知。违反者将对其违反行为所造成的任何以及全部损害承担责任，桃芯科技保留采取任何法律所允许范围内救济措施的权利。

目录

版本历史	ix
第一章 概览	1
1.1 模块设计原则	1
1.2 缩略语及术语	1
1.3 参考文档	2
第二章 降噪	3
2.1 使用方法	3
2.2 性能	4
2.2.1 ING916XX	4
2.3 应用建议	4
第三章 ADPCM 编解码	5
3.1 使用方法	5
3.1.1 编码	5
3.1.2 解码	6
3.2 性能	7
3.2.1 ING916XX	7
第四章 SBC/mSBC 编解码	9
4.1 帧描述参数	9

4.2	使用方法	10
4.2.1	编码	10
4.2.2	解码	12
4.3	性能	13
4.3.1	ING916XX	14
第五章	Opus 编码	15
5.1	使用方法	15
5.2	参数选择与评估	19
5.2.1	临时内存评估	19
5.2.2	性能评估	20
5.3	性能	21
5.3.1	ING916XX	21
5.4	线程安全性	21
第六章	致谢	23

插图

表格

1.1	缩略语	1
1.2	术语	2
5.1	Opus 采样率与帧长	17

版本历史

版本	信息	日期
0.1	初始版本	2024-09-05

第一章 概览

音频处理库包含一组音频处理模块，开发者可以根据需要选用其中的模块，以获得高品质的音频体验。本音频处理库是免费附送的，以预编译库的形式提供。

本音频处理库只能运行于以下芯片：

- ING916XX

1.1 模块设计原则

为了适配资源紧张的嵌入式系统，本音频处理库在设计时遵循下列原则。

- 内存管理

音频处理往往涉及较大量的数据处理，需要较多的内存。考虑到嵌入式系统的特点，内存由开发者负责分配，库内的模块可以完全不使用堆（`malloc/free`），而且不会从栈上分配大块内存。

- 线程安全性

各模块采用面向对象式的接口。如无特殊说明，多个模块实例可以并发执行。

1.2 缩略语及术语

表 1.1: 缩略语

缩略语	说明
ADC	模数转换器（Analog-to-Digital Converter）

缩略语	说明
ADPCM	自适应脉冲编码调制 (ADaptive Pulse Coded Modulation)
mSBC	改良低复杂度子带编解码器 (modified Low Complexity SubBand Codec)
PCM	脉冲编码调制 (Pulse Coded Modulation)
SBC	低复杂度子带编解码器 (Low Complexity SubBand Codec)

表 1.2: 术语

缩略语	说明
Opus	一个完全开放、免版权、用途广泛的音频编解码器

1.3 参考文档

1. ING916XX 系列芯片数据手册¹
2. SBC 技术规范²
3. Opus 交互式音频编解码器³

¹<http://www.ingchips.com/product/70.html>

²https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=544797

³<https://opus-codec.org/>

第二章 降噪

降噪模块先将音频转换到频域，估计噪声谱，完成降噪，最后再转换到时域。降噪模块每次处理一个音频帧，一个音频帧包含 `AUDIO_DENOISE_BLOCK_LEN` 个采样。音频采样率支持 8 kHz、16 kHz，推荐使用 8 kHz。

2.1 使用方法

1. 初始化对象

```
audio_denoise_context_t *audio_denoise_init(  
    void *buf,  
    uint32_t sample_rate);
```

`buf` 是用来存放对象的内存空间，其大小为 `AUDIO_DENOISE_CONTEXT_MEM_SIZE` 字节。

2. 处理音频

```
void audio_denoise_process(  
    audio_denoise_context_t *ctx,    // 对象  
    const int16_t *in,               // 音频输入  
    int16_t *out,                    // 降噪输出  
    void *scratch);                  // 临时内存
```

`in`、`out` 各自包含 `AUDIO_DENOISE_BLOCK_LEN` 个采样。降噪输出 `out` 可以与 `in` 相同，数据原地处理（`in-place`）。

`scratch` 指向用来存放中间结果的内存空间，其大小为 `AUDIO_DENOISE_SCRATCH_MEM_SIZE` 字节。

一个降噪对象只能处理一个声道的数据。如果需要同时处理多个声道，则需要创建多个对象。如果并发调用多个降噪对象的 `audio_denoise_process` 接口，那么各对象需要使用独立的 `scratch`；如果顺序调用多个降噪对象的 `audio_denoise_process` 接口，那么可以使用同一块 `scratch` 内存，例如：

```
audio_denoise_process(ctx_left_ch, ..., ..., scratch);  
audio_denoise_process(ctx_right_ch, ..., ..., scratch);
```

2.2 性能

以下数据仅供参考。实际表现受 Cache、RTOS、中断等因素影响。

2.2.1 ING916XX

当 CPU 主频为 112 MHz 时，调用一次 `audio_denoise_process` 大约需要 6 ms，或者说 `audio_denoise_process` 消耗的 CPU 频率约为 42 MHz。

2.3 应用建议

此降噪模块仅推荐用于 ADC 采集模拟麦克风信号的场景。

第三章 ADPCM 编解码

ADPCM 编码将 PCM 转换为每采样占用 4-bit 的压缩格式；ADPCM 解码把这种压缩格式转换为 16-bit PCM。

3.1 使用方法

3.1.1 编码

1. 定义回调函数

这个回调函数用来接收编码结果，其签名为：

```
typedef void (*adpcm_encode_output_cb_f)(  
    uint8_t output, // 编码输出, 包含两个 4-bit 数据  
    void *param); // 用户数据
```

2. 初始化编码器对象

```
void adpcm_enc_init(  
    adpcm_enc_t *adpcm, // 编码器对象  
    adpcm_encode_output_cb_f callback, // 回调函数  
    void *param); // 传给回调函数的用户数据
```

3. 编码

```
void adpcm_encode(  
    adpcm_enc_t *adpcm,           // 编码器对象  
    const pcm_sample_t *input,    // 音频数据  
    int input_size);             // 音频采样数
```

采样数 `input_size` 可以是奇数。每产生两个 4-bit 编码输出（拼接为 1 个字节¹），就会调用一次 `callback`。

3.1.2 解码

1. 定义回调函数

这个回调函数用来接收解码结果，其签名为：

```
typedef void (*adpcm_decode_output_cb_f)(  
    pcm_sample_t output, // 解码输出  
    void* param);       // 用户数据
```

2. 初始化解码器

```
void adpcm_dec_init(  
    adpcm_dec_t* adpcm,           // 解码器对象  
    adpcm_decode_output_cb_f callback, // 回调函数  
    void* param);                // 传给回调函数的用户数据
```

3. 解码

```
void adpcm_decode(  
    adpcm_dec_t *adpcm, // 解码器对象  
    uint8_t data);      // ADPCM 编码
```

¹设高 4-bit 对应第 n 个采样，则低 4-bit 对应第 $n + 1$ 个采样。

3.2 性能

以下数据仅供参考。实际表现受 Cache、RTOS、中断等因素影响。

3.2.1 ING916XX

当 CPU 主频为 112 MHz 时，adpcm_encode 处理 1 个采样仅需 $0.7\mu s$ ，或者说处理 1 个采样消耗的 CPU 频率约为 80 Hz，处理 16 kHz 采样消耗的 CPU 频率约为 1.25 MHz。

当 CPU 主频为 112 MHz 时，adpcm_decode 解码 1 个字节输出 2 个采样约需 $1.9\mu s$ ，或者说解码 1 个字节消耗的 CPU 频率约为 213 Hz，解码 16 kHz ADPCM 消耗的 CPU 频率约为 3.4 MHz。

第四章 SBC/mSBC 编解码

SBC 支持多种采样率、多种帧长。一个 SBC/mSBC 音频帧由 4 部分组成：

```
struct
{
    frame_header;
    scale_factors;
    audio_samples;
    padding;
};
```

其中 frame_header 里的第一字节为 sync_word。对于 SBC，sync_word 固定为 0x9C，而 mSBC 则为 0xAD。

```
struct frame_header
{
    uint8_t sync_word;
    ....
};
```

4.1 帧描述参数

帧描述参数见 sbc_frame 结构体：

```

struct sbc_frame
{
    bool msbc;           // 是否为 mSBC
    enum sbc_freq freq;  // 采样频率
    enum sbc_mode mode;  // 声道模式
    enum sbc_bam bam;    // 比特分配方式
    int nblocks, nsubbands; // 分块数, 子带数
    int bitpool;         // bit 池大小
};

```

nblocks 应为 4、8、12 或 16，nsubbands 可为 4 或 8。进行编码时，每个声道上的每一帧需要 (nblocks × nsubbands) 个采样，该值也可通过 `sbc_get_frame_samples()` 获取。

bitpool 是一个块 (nsubbands 个子带) 所能占据的最多比特数。

mSBC 使用一组固定的参数：

```

const struct sbc_frame msbc_frame = {
    .msbc = true,
    .mode = SBC_MODE_MONO,
    .freq = SBC_FREQ_16K,
    .bam = SBC_BAM_LOUDNESS,
    .nsubbands = 8, .nblocks = 15,
    .bitpool = 26
};

```

4.2 使用方法

4.2.1 编码

1. 确定帧描述参数

确定了帧描述参数后，务必使用 `sbc_get_frame_size()` 等函数检查参数是否合法。

2. 检查关键参数

- `sbc_get_frame_size()` 获得编码后每个帧的字节长度;
- `sbc_get_frame_bitrate()` 获得编码后的比特率;
- `sbc_get_frame_samples()` 为一个声道编码一个帧所需要的采样数

如果帧描述参数不合法，这些函数都将返回 0。

3. 初始化对象

```
void sbc_reset(
    sbc_t *sbc); // SBC 对象
```

4. 进行编码

音频处理库里包含两个编码函数，其区别在于 `sbc_encode2` 的临时内存由外部分配，而 `sbc_encode` 的临时内存则在栈上分配。对于栈空间紧张的应用，应该使用 `sbc_encode2`。调用一次 `sbc_encode2` 或者 `sbc_encode` 完成一帧编码，编码成功返回 0 否则返回错误码。

`sbc_encode2` 的函数签名如下：

```
int sbc_encode2(
    sbc_t *sbc,           // SBC 对象
    const int16_t *pcmL, // 左声道 PCM 数据
    int pitchL,           // 左声道 PCM 相邻数据在 pcmL 里的间隔
    const int16_t *pcmR, // 右声道 PCM 数据
    int pitchR,           // 右声道 PCM 相邻数据在 pcmR 里的间隔
    const struct sbc_frame *frame, // 帧描述参数
    void *data,            // 编码输出
    unsigned size,         // 编码输出的内存长度
    void *scratch);        // 临时内存
```

当只编码一个声道时，忽略 `pcmR` 和 `pitchR` 参数。`pitchL` 和 `pitchR` 分别控制如何从 `pcmL` 和 `pcmR` 读取采样： $sample[n] = pcm[n * pitch]$ 。举例说明如下：

- 只有一个声道的数据：

```
sbc_encode2(sbc, pcm, 1, ...);
```

- 要编码两个声道，且两个声道的数据独立存放：

```
sbc_encode2(sbc, pcm_l, 1, pcm_r, 1, ...);
```

- 要编码两个声道，且两个声道的数据交织存放，即 `pcm[] = {左, 右, 左, 右, ...}`：

```
sbc_encode2(sbc, pcm, 2, pcm + 1, 2, ...);
```

`size` 参数至少为 `sbc_get_frame_size(frame)`。

临时内存 `scratch` 应给按 `int` 型对齐，大小至少为 `SBC_ENCODE_SCRATCH_MEM_SIZE`。

`sbc_encode` 比 `sbc_encode2` 缺少 `scratch` 参数，其它参数完全一致，不再赘述。

当使用 `mSBC` 编码时，`frame` 只需要设置 `msbc = true`，不需要完整填写 `mSBC` 帧参数：

```
const struct sbc_frame msbc_frame = {
    .msbc = true,
};
sbc_encode2(...,
    &msbc_frame,
    ...);
```

4.2.2 解码

1. 初始化

```
void sbc_reset(
    sbc_t *sbc); // SBC 对象
```

2. 进行解码

同编码类似，音频处理库里包含两个解码函数，其区别在于 `sbc_decode2` 的临时内存由外部分配，而 `sbc_decode` 的临时内存则在栈上分配。对于栈空间紧张的应用，应该使用

sbc_decode2。调用一次 sbc_decode2 或者 sbc_decode 完成一帧解码，解码成功返回 0 否则返回错误码。

sbc_decode2 的函数签名如下：

```
int sbc_decode2(
    sbc_t *sbc,          // SBC 对象
    const void *data,    // 输入数据（即编码后的一帧）
    unsigned size,       // 输入数据的长度，应不小于该帧的长度
    struct sbc_frame *frame, // 解出的帧描述参数
    int16_t *pcm1,       // 左声道 PCM 解码输出
    int pitch1,          // 左声道 PCM 相邻数据在 pcm1 里的间隔
    int16_t *pcm2,       // 右声道 PCM 解码输出
    int pitch2,          // 右声道 PCM 相邻数据在 pcm2 里的间隔
    void *scratch);      // 临时内存
```

pitch1 和 pitch2 的含义与 sbc_encode2 里相同，区别在于后者用于读取 PCM 数据，而在这里用于写入 PCM 数据。

临时内存 scratch 应给按 int 型对齐，大小至少为 SBC_DECODE_SCRATCH_MEM_SIZE。

调用解码函数时，必须保证 pcm1 和 pcm2 空间足够，即每个声道都足够容纳 SBC_MAX_SAMPLES 个采样。sbc_decode 比 sbc_decode2 缺少 scratch 参数，其它参数完全一致，不再赘述。

4.3 性能

以下数据仅供参考。实际表现受 Cache、RTOS、中断等因素影响。

使用如下帧描述参数：

```
const struct sbc_frame frame_param =
{
    .freq = SBC_FREQ_16K,
    .mode = SBC_MODE_MONO,
    .nsbbs = 4,
```

```
.nblocks = 8,  
.bam = SBC_BAM_LOUDNESS,  
.bitpool = 16  
};
```

4.3.1 ING916XX

当 CPU 主频为 112 MHz 时，sbc_encode2 编码一帧需要约 0.1 ms，或者说消耗的 CPU 频率约为 5.6 MHz。

当 CPU 主频为 112 MHz 时，sbc_decode2 编码一帧需要约 0.09 ms，或者说消耗的 CPU 频率约为 5 MHz。

第五章 Opus 编码

Opus 支持窄带（4 kHz）、中等带宽（6 kHz）、宽带（8 kHz）、超宽带（12 kHz）、全带宽（24 kHz）等多种音频带宽，支持 2.5 ms、5 ms、10 ms、20 ms、40 ms、60 ms、80 ms、100 ms、120 ms 等 9 种帧长。Opus 兼具较好的音质和较高的压缩率，计算复杂度也较高。音频处理库裁剪了 Opus 编码器，使其能运行于嵌入式系统。

5.1 使用方法

1. 初始化

使用 `opus_encoder_init` 初始化编码器对象：

```
int opus_encoder_init(  
    OpusEncoder *st, // 编码器对象  
    opus_int32 Fs,   // 采样率  
    int channels,    // 声道数  
    int application  // 应用类型  
);
```

通过 `opus_encoder_get_size()` 获得编码器对象的大小。采样率只能是 8000，12000，16000，24000 或者 48000。声道数只能是 1 或者 2。应用类型及适用场景如下。

- `OPUS_APPLICATION_VOIP`：适用于大多数 VoIP、视频会议等注重声音质量和易懂性的场景；
- `OPUS_APPLICATION_AUDIO`：适用于广播或 Hi-Fi 等要求解码输出尽量贴近原始输入的场景；
- `OPUS_APPLICATION_RESTRICTED_LOWDELAY`：仅用于需要最低延迟的场景。

下面的代码演示了如何从堆上分配用来存放编码器对象内存，并初始化编码器对象：

```
int size = opus_encoder_get_size(1);
OpusEncoder *enc = malloc(size);
if (NULL == enc)
{
    ... // error handling
}

int error = opus_encoder_init(enc, Fs,
    channels, application);
if (error)
{
    ... // error handling
}
```

2. 设置参数

使用 `opus_encoder_ctl()` 设置编码参数。`opus_defines.h` 里列出了所有可设置的参数。例如，将比特率设为 80 kbps：

```
opus_encoder_ctl(enc, OPUS_SET_BITRATE(80000));
```

3. 设置临时内存

```
void opus_set_scratch_mem(
    const void *buf, // 起始位置
    int size);       // 临时内存的大小（单位：字节）
```

在程序运行过程中，如果发现临时内存空间不足，会调用 `opus_on_run_out_scratch_mem`。音频库里包含了该函数的弱定义，开发者可以重新定义这个函数以自定义处理方法。这个函数的弱定义大致为：

```
void __attribute__((weak)) opus_on_run_of_out_scratch_mem(  
    const char *fn, int line_no)  
{  
    platform_raise_assertion(fn, line_no);  
}
```

请参考“参数选择与评估”了解如何确定临时内存的大小。

4. 编码

调用 `opus_encode` 编码一个音频帧。

```
opus_int32 opus_encode(  
    OpusEncoder *st,           // 编码器对象  
    const opus_int16 *pcm,     // PCM 输入  
    int frame_size,           // 这一帧的每个声道所包含的采样数  
    unsigned char *data,       // 编码输出（载荷）  
    opus_int32 max_data_bytes // 编码输出的最大长度  
);
```

当编码两个声道时，左右声道在 `pcm` 里交织排列。`frame_size` 参数结合采样率可推算出音频帧的时长，这个音频帧的时长必须是合法，否则函数将返回一个错误码。各种采样率所允许的 `frame_size` 如表 5.1 所示。

表 5.1: Opus 采样率与帧长

采样率 (Hz)	2.5 ms	5 ms	10 ms	20 ms	40 ms	60 ms	80 ms	100 ms	120 ms
8 k	20	40	80	160	320	480	640	800	960
12 k	30	60	120	240	480	720	960	1200	1440
16 k	40	80	160	320	640	960	1280	1600	1920
24 k	60	120	240	480	960	1440	1920	2400	2880
48 k	120	240	480	960	1920	2880	3840	4800	5760

`max_data_bytes` 是这一帧所允许的最大编码长度，建议预留足够大的空间，不建议用此参数进行比特率调整或控制。

如果编码成功，这个函数将返回编码输出（载荷）的实际长度，否则返回错误码（负值）。

5. 音频帧打包

opus_encode 所输出的 data 仅为载荷部分，还需要附加帧长信息才能组成可解码的音频流。opus_demo 所使用的帧头结构为：

- 帧长：4 字节，大端模式
- FINAL_RANGE：4 字节，大端模式

test_opus_data 函数演示了如何将编码结果保存为 opus_demo 所支持的帧格式。将 save_bytes() 收到的字节流保存到文件，就可以用 opus_demo 解码，回听效果。

```
void test_opus_data(OpusEncoder *enc,
    const int16_t *in, const int total_samples,
    const int sample_rate, const int samples_per_frame,
    uint8_t *output, const int max_output_bytes)
{
    unsigned char int_field[4];
    uint32_t enc_final_range;
    int i;
    for (i = 0; i < total_samples - samples_per_frame;
        i += samples_per_frame)
    {
        int r = opus_encode(enc, in + i, samples_per_frame,
            output, max_output_bytes);
        if (r < 0) platform_raise_assertion("opus_encode", r);

        big_endian_store_32(int_field, 0, (uint32_t)r);
        save_bytes(int_field, sizeof(int_field));

        opus_encoder_ctl(enc, OPUS_GET_FINAL_RANGE(&enc_final_range));
        big_endian_store_32(int_field, 0, enc_final_range);
        save_bytes(int_field, sizeof(int_field));

        save_bytes(output, r);
    }
}
```

```
}  
}
```

5.2 参数选择与评估

5.2.1 临时内存评估

不同的参数将显著影响所需要的临时内存的大小。音频处理库附带了一个 Windows 测试程序 `opus_demo`，这个程序包含了完整版的解码器和裁剪过的编码器。运行这个编码器，可以得到需要的临时内存。

这里使用 Audacity¹ 辅助转换和播放 PCM 数据。

1. 准备测试数据

准备一个音频文件（比如一首歌曲或一段录音），使用 Audacity 按 Opus 支持的某一采样率（例如 16 kHz）导出²为单声道无格式的 16-bit PCM 文件（例如保存为 `data_16k.raw`）。

2. 编码测试

运行 `opus_demo`，编码测试数据。

```
opus_demo -e audio 16000 1 100000 data_16k.raw result.enc
```

这里以 100 kbps 的比特率转换为 `result.enc`。程序会打印出所需要的临时空间的大小（单位：字节）：

```
stack_max_usage = 12345
```

3. 解码测试

如有必要，可再运行 `opus_demo` 解码 `result.enc`：

¹<https://www.audacityteam.org/>

²https://manual.audacityteam.org/man/other_uncompressed_files_export_options.html

```
opus_demo -d 16000 1 result.enc result.dec
```

在 Audacity 里导入³无格式的 PCM 文件 `result.dec`，采样率 16 kHz，回听编解码效果。

重复上述步骤，确定应用中所要使用的采样率、比特率等关键参数，根据工具报告的 `stack_max_usage` 确定临时空间大小。

5.2.2 性能评估

`test_opus_performance` 函数演示了如何评估编码所消耗的时间。

```
void test_opus_performance(OpusEncoder *enc,
    const int16_t *in, const int total_samples,
    const int sample_rate, const int samples_per_frame,
    uint8_t *output, const int max_output_bytes)
{
    int i = 0;
    int frame_cnt = 0;
    uint32_t total_time = 0;

    for (i = 0; i < total_samples - samples_per_frame;
        i += samples_per_frame, frame_cnt++)
    {
        int64_t t = platform_get_us_time();
        int r = opus_encode(enc, in + i, samples_per_frame,
            output, max_output_bytes);
        uint32_t tt = (uint32_t)(platform_get_us_time() - t);
        platform_printf("%d: len = %d, %u\n", frame_cnt, r, tt);
        total_time += tt;
    }

    platform_printf("average time per frame = %d us\n",
```

³https://manual.audacityteam.org/man/file_menu_import.html#raw_data

```
total_time / frame_cnt);  
platform_printf("scratch max used      = %d bytes\n",  
opus_scratch_get_max_used_size());  
}
```

5.3 性能

以下数据仅供参考。实际表现受 Cache、RTOS、中断、音频数据等因素影响。

使用如下参数：

- 采样率：16 kHz
- 单声道
- 使用 OPUS_APPLICATION_AUDIO
- 比特率：80 kbps
- 帧长：10 ms

5.3.1 ING916XX

当 CPU 主频为 112 MHz 时，sbc_encode2 编码一帧平均约需要 5 ms，或者说消耗的 CPU 频率约为 56 MHz。

需要的临时内存为 8944 字节。opus_encoder_get_size(1) = 7196，所以总计需要约 16 kB 内存。

5.4 线程安全性

编译时定义了 NONTHREADSAFE_PSEUDOSTACK，所以只允许单线程使用。

第六章 致谢

音频处理库包含了几种开源软件（库），详情请见代码仓库¹。

¹<https://github.com/ingchips/libaudio>

