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

目录

插图

表格

版本历史

版本	信息	日期
0.1	初始版本	2024-09-05
1.0	增加 AMR-WB、TTS 等功能	2024-10-24

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。

  各模块采用面向对象式的接口，需要销毁对象时，直接释放为对象分配的内存即可，无需其它操作，因此皆未提供专门的销毁接口。

- 线程安全性

  如无特殊说明，一个模块的多个实例可以并发执行。

  一个实例只能在一个线程内使用。如果需要在多个线程中操作某一个实例，必须使用同步机制，避免对同一个实例的并发操作。

1.2 依赖关系

音频处理库依赖于 CMSIS-DSP[1] v1.15.0 及以上。以 Keil µVision 为例，打开 “Manage Run-time Environment”，将 CMSIS-DSP 添加到项目。

音频库支持多种编译器。每种编译器提供两种版本，一为开启硬件单精度浮点运算，库文件带有 `_f` 后缀；一为不开启硬件单精度浮点运算，库文件不带有 `_f` 后缀。

- ING916XX

  开启硬件单精度浮点运算时，GCC 版本使用的编译选项为：

  ```
  -mthumb -mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=hard
  ```

  不开启硬件单精度浮点运算时，GCC 版本使用的编译选项为：

  ```
  -mthumb -mcpu=cortex-m4
  ```

1.3 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）
ADPCM	自适应脉冲编码调制（ADaptive Pulse Coded Modulation）
AMR-WB	自适应多速率-宽带（Adaptive Multi-Rate WideBand）
CMSIS	微控制器软件接口标准（Common Microcontroller Software Interface Standard）
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）
TTS	文本转语音（Text-To-Speech）

[1] https://github.com/ARM-software/CMSIS-DSP

表 1.2: 术语

术语	说明
Opus	一个完全开放、免版税、用途广泛的音频编解码器

1.4 参考文档

1. ING916XX 系列芯片数据手册[2]
2. SBC 技术规范[3]
3. Opus 交互式音频编解码器[4]
4. 汉典[5]

[2]http://www.ingchips.com/product/70.html

[3]https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=544797

[4]https://opus-codec.org

[5]https://www.zdic.net

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 资源消耗

以下数据仅供参考。实际表现受编译器、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 资源消耗

以下数据仅供参考。实际表现受编译器、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 0.7μs，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 1.9μs，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：

```
scb_encode2(sbc, pcm, 1, ...);
```

- 要编码两个声道，且两个声道的数据独立存放：

```
scb_encode2(sbc, pcml, 1, pcmr, 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，...}:

```
scb_encode2(sbc, pcm, 2, pcm + 1, 2, ...);
```

size 参数至少为 sbc_get_frame_size(frame)。

临时内存 scrach 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 *pcml,      // 左声道 PCM 解码输出
    int pitchl,         // 左声道 PCM 相邻数据在 pcml 里的间隔
    int16_t *pcmr,      // 右声道 PCM 解码输出
    int pitchr,         // 右声道 PCM 相邻数据在 pcmr 里的间隔
    void *scratch);     // 临时内存
```

pitchl 和 pitchr 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 scrach 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 pcml 和 pcmr 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 资源消耗

以下数据仅供参考。实际表现受编译器、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,
    .nsubbands = 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。

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。

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可懂性的场景；

- 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_of_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

采样率 (Hz)	2.5 ms	5 ms	10 ms	20 ms	40 ms	60 ms	80 ms	100 ms	120 ms
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。运行 `opus_demo`，可以得出需要的临时内存的大小。

这里使用 Audacity[1] 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 kpbs 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）：

[1] https://www.audacityteam.org/

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

```
stack_max_usage = 12345
```

3. 解码测试

   如有必要，可再运行 opus_demo 解码 result.enc：

```
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);
```

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

```
        platform_printf("%d: len = %d, %u\n", frame_cnt, r, tt);
        total_time += tt;
    }

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

5.3 资源消耗

以下数据仅供参考。实际表现受编译器、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 kpbs
- 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　AMR-WB 编解码

AMR-WB 由 3GPP（第三代合作伙伴计划）制定[1]，是一种用于语音编码的音频压缩标准，广泛应用于移动通信领域，特别是在 3G 和 4G 网络中。AMR-WB 支持的音频带宽范围为 50-7000 Hz，比 AMR-NB 更宽，语音更清晰、自然。具备从 6.6 kbps 到 23.85 kbps 等多种码率，支持动态调整、优化语音质量和带宽占用。

AMR-WB 音频采样率为 16kHz，以 20ms 为一帧，即 320 个采样（`AMR_WB_PCM_FRAME_16k`）。

6.1　使用方法

6.1.1　编码

1. 初始化

   使用 `amr_wb_encoder_init` 初始化编码器对象：

```c
struct amr_wb_encoder *amr_wb_encoder_init(
    int bit_stream_format,  // 比特流格式
    int allow_dtx,          // 是否启用 DTX (1 或 0)
    int mode,               // 默认模式（即码率）
    void *buf);             // 用于存放上下文的内存
```

   支持三种比特流格式，如表 6.1 所示。当用于语音数据压缩时，应该使用 MIME 格式（`AMR_WB_BIT_STREAM_FORMAT_MIME_IETF`）。

[1] 参见 TS 26.171 等

表 6.1: AMR-WB 比特流格式

格式	帧头长度（字节）	载荷格式
ETS	6	每 2 个字节表示一个比特
ITU	4	每 2 个字节表示一个比特
MIME	1	详见 RFC 3267 (5.1，5.3)

帧头长度也可以通过 `amr_wb_get_frame_header_size(mode)` 获得。

`allow_dtx` 参数详见 AMR-WB 规范，可以填 `0`。

模式参数 `mode` 与实际码率对应关系见表 6.2。

表 6.2: AMR-WB 码率模式与帧长（MIME 格式）

模式	码率（kb/s）	每帧比特数	每帧载荷长度（字节）
0	6.6	132	17
1	8.85	177	23
2	12.65	253	32
3	14.25	285	36
4	15.85	317	40
5	18.25	365	40
6	19.85	397	50
7	23.05	461	58
8	23.85	477	60

调用 `amr_wb_encoder_get_context_size()` 可获取 `buf` 所需大小。

2. 编码

调用 `amr_wb_encoder_encode_frame` 编码一个音频帧，其返回值为音频帧的帧头和载荷的总长度。

```
int amr_wb_encoder_encode_frame(
    struct amr_wb_encoder *ctx, // 编码器对象
    const int16_t *pcm_samples, // PCM 输入
    uint8_t *output,            // 编码输出
    void *scratch);             // 临时内存
```

编码输出 output 的大小根据选择的码率确定（见表 6.2），例如，若码率为 6.6 kb/s，则 output 的长度应该至少是 1 + 17 = 18 字节。

调用 amr_wb_encoder_get_context_size() 可获取临时内存 scratch 所需大小。

需要切换模式时，调用 amr_wb_encoder_encode_frame2，先切换模式，再编码。

下面的代码演示了如何编码一整段数据。

```
void *context = malloc(amr_wb_encoder_get_context_size());
void *scratch = malloc(amr_wb_encoder_get_scratch_mem_size());

static uint8_t output[模式所对应的载荷长度 + 1];
const uint16_t *in = ...        // 音频采样
const int NUM_OF_SAMPLES = .... // 总采样数

struct amr_wb_encoder *enc =
    amr_wb_encoder_init(AMR_WB_BIT_STREAM_FORMAT_MIME_IETF,
        0, 模式, context);

for (i = 0;
     i < NUM_OF_SAMPLES - AMR_WB_PCM_FRAME_16k;
     i += AMR_WB_PCM_FRAME_16k)
{
    int r = amr_wb_encoder_encode_frame(
        enc, in + i, output, scratch);

    // 处理 pcm_output
    ....
}
```

6.1.2 解码

1. 初始化

使用 amr_wb_decoder_init 初始化解码器对象：

```
struct amr_wb_decoder *amr_wb_decoder_init(
    int bit_stream_format,      // 比特流格式
    void *buf);                 // 用于存放上下文的内存
```

调用 amr_wb_decoder_get_context_size() 可获取 buf 所需大小。

比特流格式仅支持 AMR_WB_BIT_STREAM_FORMAT_MIME_IETF。

2. 解码帧头

使用 amr_wb_decoder_probe 解码帧头，获得音频帧基本参数：

```
int amr_wb_decoder_probe(
    struct amr_wb_decoder *ctx, // 解码器对象
    const uint8_t *stream);     // 音频流
```

音频流 stream 应该指向一个音频帧的帧头。这个函数将返回这个音频帧的载荷的长度。如果出现错误，将返回负值错误码。

3. 解码载荷

使用 amr_wb_decoder_decode_frame 解码载荷：

```
int amr_wb_decoder_decode_frame(
    struct amr_wb_decoder *ctx, // 解码器对象
    const uint8_t *payload,     // 音频帧载荷
    int16_t *synth_pcm,         // PCM 输出
    void *scratch);             // 临时内存
```

这个函数返回输出的 PCM 采样的个数，即 AMR_WB_PCM_FRAME_16k，synth_pcm 的长度也应为 AMR_WB_PCM_FRAME_16k。

调用 amr_wb_decoder_get_scratch_mem_size() 可获取临时内存 scratch 所需大小。

下面的代码演示了如何解码一段数据。

```c
static uint8_t pcm_output[AMR_WB_PCM_FRAME_16k];

const uint8_t *in = ...; // 音频数据
int data_size = ...;     // 音频数据总长度
void *context = malloc(amr_wb_decoder_get_context_size());
void *scratch = malloc(amr_wb_decoder_get_scratch_mem_size());

struct amr_wb_decoder *dec = amr_wb_decoder_init(
    AMR_WB_BIT_STREAM_FORMAT_MIME_IETF, context);
const int header_size = amr_wb_get_frame_header_size(
    AMR_WB_BIT_STREAM_FORMAT_MIME_IETF);

while (data_size > header_size)
{
    int payload_len = amr_wb_decoder_probe(dec, in);
    in += header_size;
    data_size -= header_size;

    if (payload_len > data_size) break;
    if (payload_len < 0) break;

    int r = amr_wb_decoder_decode_frame(
        dec, in, pcm_output, scratch);

    in        += payload_len;
    data_size -= payload_len;

    // 处理 pcm_output
    ....
}
```

6.2 资源消耗

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

6.2.1 ING916XX

表 6.3: AMR-WB 编解码器的内存需求

功能	context（字节）	scratch（字节）
编码	2788	11898
解码	1552	3826

当 CPU 主频为 112 MHz 时，AMR-WB 编解码器的处理一个音频帧所消耗的时间见表 6.4。从表中可以看出，需要进行实时编码时，只能使用 6.6 kbps，不可使用更高速率。

表 6.4: AMR-WB 编解码器处理一个音频帧所消耗的平均时间

模式	编码（ms）	解码（ms）
0	16.7	7.9
1	20.0	6.6
2	22.2	5.9
3	23.9	5.9
4	23.9	6.0
5	24.6	6.1
6	25.6	6.2
7	25.3	6.2
8	24.8	6.6

第七章　语音合成

请联系我们以获取相关开发资料。

7.1　特性

音频处理库包含一个为嵌入式系统设计的轻量化语音合成引擎，具有如下主要特性：

- 仅支持中文
  - 覆盖 Unicode CJK 统一表意符号基础区字符 20992 个[1]（4E00 - 9FFF），
  - 支持少量不在 Unicode 基础区范围内的汉字（例如：〇）
  - 遇到无法识别的内容时输出短暂的静音
  - 输入文本采用 UTF-8 编码
- 支持拼音
- 英文逐个字母发音
- 整数、小数优化播报
- 流式输出
- 多音字发音自动识别
- 支持自定义语音库

[1]下列 77 个字符除外：乭，迄，岦，尒，吾，毋，恚，垚，鼏，曦，氓，昆，怋，郩，杢，杵，柈，棆，烗，猭，砢，砫，鴐，緧，莶，蘧，禷，迠，闟，蔪，鋼，騝，銲，ㄚ，ㄓ，ㄚ，墡，燸，姵，萓，齾，諴，鵪，褫，衭，伭，伋，伍，莊，禧，佥，佘，皘，皞，皜，皛，称，稱，皾，聡，聧，皡，皢，皛，遙，廓，黎，鰫，喱，郹，棗。这些字符的读音暂时无法依据汉典确定。

当前版本基于波形拼接法，主要组成部分包括（图 7.1）：

- **解析器**（Parser）：根据词典与语法规则，将输入的内容片断（中文、英文、数字等）按顺序转换为音节列表。
- **合成器**（Synthesizer）：根据解析器输出的音节列表，结合预定义的语音库，合成波形。输出格式为单声道 16 bit，16kHz 采样。

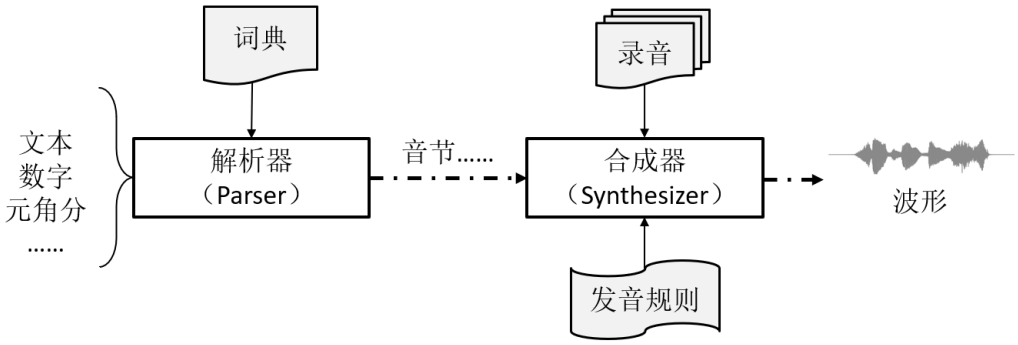

图 7.1: TTS 引擎

7.1.1 工具软件

音频处理库附带了几个程序和 Python 脚本：

- `tts_demo`：在 Windows 上快速测试、演示合成效果
- `tts_firmware` 和 `tts_demo.py`：演示用固件程序，及测试脚本（通过串口发送文本、接收合成结果并回放）
- `tts_gen`：由于自定义语音库

本章文档用到的文本文件，如无特殊说明，都必须使用**无 BOM** UTF-8 编码格式。

7.2 使用方法

1. 初始化

   使用 tts_init 初始化 TTS 引擎对象：

```
struct tts_context *tts_init(
    const struct voice_definition *voice, // 语音库
    int max_syllables,  // 最大音节数
    void *buf);         // 用于存放上下文的内存
```

   语音库以.bin 文件提供，详见 “语音库”。开发者需要预估一次合成可能存在的最大音节数 max_syllables，可以文本字数为参考，并保留一定的余量（如 50%）。buf 的大小与 max_syllables 有关，可通过 tts_get_context_size 获得。

```
int tts_get_context_size(
    int max_syllables); // 最大音节数
```

本章文档里的 “音节” 是广义的，对应语音库里的一段录音。一段录音可能只有一个音节，也可能包含多个音节。

2. 添加内容

   TTS 引擎支持添加多种内容：文本、数字、元角分。分别通过不同的接口添加。

   - 添加文本

     使用 tts_push_utf8_str 添加一段文本。

```
int tts_push_utf8_str(
    struct tts_context *ctx,    // TTS 引擎对象
    const char *utf8_str);      // UTF-8 编码的字符串
```

文本支持中文、拼音、数字，以及个别符号。对于中文，解析器会尝试自动分词并判断多音字。开发者可以通过两种方式干预中文解析，一为填加空格辅助分词[2]，如例 1[3]；一为使用 “[]” 手动组词，如例 2。

例 1：您有新的美团外卖订单，请及时 处理。

例 2：介绍 [锺书] 做 [这份工作] 的是清华同学 [乔冠华] 同志。

空格不但可用来干预分词，还可以用来添加停顿。

如果存在英文，则按照字母逐个发音。

拼音使用 ASCII 码，用一对 “[]” 包围，表示一个词语。四种声调分别使用 1（阴平）、2（阳平）、3（上声）、4（去声）表示。几个示例：

- mǎn: man3（调号放在末尾）
- le: le （轻声无调号）
- lǜ: lv4（以 v 代替 ü）
- jú: ju2（j/q/x 后的 ü 仍写作 u）
- 完整示例：迎着朝阳区的 [zhao1 yang2] 去上学。

另外可以识别 `+`、`-`、`=`、`*`、`#`、`@` 等半角符号。其它无法识别的内容（如标点符号）一律做停顿处理。

文本中的数字（整数或小数）默认按照繁读法合成。如果数字前面是 “\”，则使用简读法，如 “在`\2024` 年”。对于需要使用简读法的数字，也可以替换为对应的汉字再合成，如 “在二零二四年”。

- 添加整数

  使用 `tts_push_integer` 添加一个整数。`tts_push_integer(ctx, 123)` 等效于 `tts_push_utf8_str(ctx, "123")`。

```
int tts_push_integer(
    struct tts_context *ctx,    // TTS 引擎对象
    int64_t value);             // UTF-8 编码的字符串
```

- 添加元角分

  使用 `tts_push_yuan_jiao_fen` 添加元角分。`tts_push_yuan_jiao_fen(ctx, x, y, z)` 等效于 `tts_push_utf8_str(ctx, "x 元 y 角 z 分")`。

[2] 只有紧跟中文字符的空格才被视作分词辅助用途。

[3] 当使用完整版语音库时，“及时处理” 被识别为一个词语，并且多音字 “处” 的声调错误。添加空格后，听感流畅，而且 “处” 的读音也被正确判别。

```
int tts_push_yuan_jiao_fen(
    struct tts_context *ctx,    // TTS 引擎对象
    int64_t yuan,               // 元
    uint8_t jiao,               // 角
    uint8_t fen);               // 分
```

以上几个接口如果成功，都将返回 0。当音节数超过 max_syllables 时，返回负值。

3. 合成

提供两套 API，以两种不同的方式合成语音。

- 方式 1：被动接收波形数据

  使用 tts_synthesize 将所有添加的内容按顺序一并合成语音。

```
int tts_synthesize(
    struct tts_context *ctx,    // TTS 引擎对象
    f_tts_receive_pcm_samples rx_samples,
    void *user_data,            // 传递到回调函数的用户数据
    void *scratch1,             // 临时内存 1
    void *scratch2);            // 临时内存 2
```

  临时内存 scratch1 和 scratch2 的大小分别通过 tts_get_scratch_mem1_size() 和 tts_get_scratch_mem2_size() 获得。scratch1 在 tts_synthesize 结束前不做他用，而 scratch2 可在 rx_samples 里任意使用。回调函数 rx_samples 用以接收合成出的波形数据，其签名为：

```
typedef int (*f_tts_receive_pcm_samples)(
    struct tts_context *ctx,    // TTS 引擎对象
    const int16_t *pcm_samples, // 本次合成出的 PCM 采样
    int number,                 // 本次合成出的 PCM 采样的个数
    int acc_number,             // 已经合成出的 PCM 采样总数（不含本次）
    void *user_data);           // 用户数据
```

  tts_synthesize 每次合成少量 PCM 数据，调用 rx_samples，循环往复，直到合成完成。如果 rx_samples 返回非 0 值，则中止。其伪代码如下：

```
tts_synthesize()
{
    acc_number = 0
    while (!aborted && !done)
    {
        合成 n 个 samples;
        int r = rx_samples(samples, n, acc_number);
        if (r != 0) return r;
        acc_number += n;
    }
}
```

rx_samples 可以阻塞。由于合成需要一定的处理时间，当用 rx_samples 回放音频时，为了保证回放连续，必须缓存一定的数据量，待缓存数据低于某门限时，函数返回，解除阻塞。请参考 “资源消耗” 并结合时延确定缓存数据门限。

abort 标志可通过 tts_abort 异步设置。

```
void tts_abort(
    struct tts_context *ctx);   // TTS 引擎对象
```

- 方式 2：主动调用获取波形数据

  使用 tts_synthesizer_run 将所有添加的内容按顺序一并合成语音。

```
int tts_synthesizer_run(
    struct tts_context *ctx,    // TTS 引擎对象
    f_tts_synthesizer_callback callback,
    void *user_data,            // 传递到回调函数的用户数据
    void *scratch1,             // 临时内存 1
    void *scratch2);            // 临时内存 2
```

临时内存 scratch1 和 scratch2 的大小分别通过 tts_get_scratch_mem1_size() 和 tts_get_scratch_mem2_size() 获得。scratch1 在 tts_synthesize 结束前不做他用，而 scratch2 可在 callback 里任意使用。这个函数返回 callback 的返回值。

回调函数 callback 的签名为：

```
typedef int (*f_tts_synthesizer_callback)(
    struct tts_synthesizer *synthesizer, // 合成引擎对象
    void *user_data);                    // 用户数据
```

回调函数返回值的含义由开发者定义。在这个回调函数里，可调用合成引擎对象的接口来主动获取波形数据。合成引擎对象的接口有两个，一为继续合成少量数据（tts_synthesizer_continue）；一为重新开始（tts_synthesizer_restart）。

```
const int16_t *tts_synthesizer_continue(
    struct tts_synthesizer *synthesizer, // 合成引擎对象
    int *number);                        // 本次合成出的 PCM 采样的个数
```

tts_synthesizer_continue 返回的指针指向本次合成出的 PCM 采样。当合成结束时，返回空指针，同时 number 为 0。下面的回调函数演示了如何多次调用 tts_synthesizer_continue 获得所有数据。

```
int tts_synthesizer_callback(
    struct tts_synthesizer *synthesizer,
    void *user_data)
{
    int number = 0;
    const int16_t *pcm = NULL;
    while (1)
    {
        pcm = tts_synthesizer_continue(synthesizer, &number);
        if (NULL == pcm) break;

        处理数据;
    }
    return 0;
}
```

这个回调函数可以阻塞，也可以随时返回、中断合成。由于合成需要一定的处理时间，当在这个回调里回放音频时，为了保证回放连续，必须缓存一定的数据量，待缓

存数据低于某门限时，调用 tts_synthesizer_continue 补充数据。请参考“资源消耗”并结合时延确定缓存数据门限。

如果需要多次重复播放合成的内容，那么可调用 tts_synthesizer_restart 重新开始合成：

```
void tts_synthesizer_restart(
    struct tts_synthesizer *synthesizer); // 合成引擎对象
```

注意，合成引擎对象 synthesizer 只存在于回调函数 callback 内。callback 一旦返回，这个对象就被销毁。

4. 复位

使用 tts_reset 清空已添加的内容。

```
int tts_reset(
    struct tts_context *ctx);   // TTS 引擎对象
```

tts_synthesize 或 tts_synthesizer_run 正在执行时，不可调用 tts_reset。

另外，可通过 tts_tune 微调合成效果：

```
void tts_tune(
    struct tts_context *ctx, // TTS 引擎对象
    uint8_t tune);           // 微调值（默认：8）
```

微调值越大，则音节与音节之间的间隔越大。

7.3 语音库

语音库由语音音源（录音）、汉语字典、词典等组成，由 tts_gen 工具转换为单一的.bin 文件。这种.bin 文件可以烧录到任意 4 字节对齐的地址。假设烧录到 0x04000000，初始化时将此地址传入 tts_init：

```
struct tts_context *tts_init(
    (const struct voice_definition *)0x04000000,
    ...);
```

7.3.1 内置语音库

音频处理库附带了两种音色的语音库，每种语音库又提供了词典缩减、音质不同的版本。各语音库的大小见表 7.1。

表 7.1: 内置的各语音库

名称	音色	词典	音质	大小（字节）
xiaotao_full_h	女声	完整	高	4823217
xiaotao_full_m	女声	完整	中	4150083
xiaotao_full_l	女声	完整	低	2755734
xiaotao_lite_h	女声	缩减	高	3162253
xiaotao_lite_m	女声	缩减	中	2489119
xiaotao_lite_l	女声	缩减	低	1094770
xiaoxin_full_h	男声	完整	高	4128000
xiaoxin_full_m	男声	完整	中	3614424
xiaoxin_full_l	男声	完整	低	2550588
xiaoxin_lite_h	男声	缩减	高	2467036
xiaoxin_lite_m	男声	缩减	中	1953460
xiaoxin_lite_l	男声	缩减	低	889624

每种语音库推荐的微调值见表 7.2。

表 7.2: 内置语音库推荐的微调值

名称	音色	微调值
xiaotao	女声	8
xiaoxin	男声	200

7.3.2 自定义语音库

利于工具 tts_gen 可以自定义语音库，定制专属音色、优化合成效果。定制步骤如下。

1. 准备两个词库，分别称为 dict1 和 dict2。

   需要单独录音的词汇放到 dict1 里，只需要识别、不需要单独录音的词汇放到 dict2 里。dict1 文件保存以 json 格式保存，如：

```
[
    " 一下",
    " 一个"
]
```

   dict2 是一个文本文件，每行一个词语，如：

```
一一列举
一一对应
```

   这两个词典都可以为空。当解析器识别出 dict1 里的词语，合成时会直接使用录音，效果较佳；当识别出 dict2 里的词语，合成器输出的词语发音将比单字拼接更自然一些。

   dict1、dict2 里的每个词最多可以包含 255 个汉字，dict1 里每个词的录音经压缩后长度不可超过 65535 字节。

2. 使用 tts_gen 导出录音计划

```
tts_gen tts_plan /path/to/dict1.json path/to/tts_plan.json
```

   运行这个命令，就会生成录音计划 tts_plan.json。

3. 进行录音

   按照录音计划完成录音。譬如将拼音 “a (啊)” 的录音保存为 “00000.raw” 文件。

```
{
    "a (啊)": "00000.raw",
    "a1 (锕)": "00001.raw",
    "a2 (嗄)": "00002.raw",
    "ai1 (哀)": "00003.raw",
    ...
}
```

录音时使用单声道，保证音色、音量一致，音质清晰、无杂音，语气平和。

这里的 raw 格式是指无格式的 16kHz 采样，每个采样 16 比特，小端模式。如果录音时采用了其它格式，可以用 ffmpeg[4] 等工具批量转换为这种 raw 格式，如将 mp3 转换为 raw：

```
ffmpeg -i 00000.mp3 -f s16le -ar 16000 -acodec pcm_s16le 00000.raw
```

4. 生成.bin 文件

   假设录音已保存在 `path/to/recordings` 目录下，用以下命令生成.bin 文件。

```
tts_gen data path/to/tts_plan.json path/to/recordings output.bin 5 \
            path/to/dict2
```

这里的参数 5 为编码等级，范围为 0~8，0 表示最低码率，音质最差；8 表示最高码率，音质最佳。如果不定义 `dict2`，`path/to/dict2` 参数可省略。省略 `dict2` 参数后，编码等级也可省略（默认值 5）。`output.bin` 即为生成的.bin 文件。

7.4 资源消耗

TTS 需要较大的 Flash 存储空间，各内置语音库的大小见表 7.1，必须配备一定容量的外置 Flash[5]。

当前版本音频数据使用 AMR-WB 压缩，`tts_synthesize` 函数的主要动作是 AMR-WB 解码，其性能请参考 “AMR-WB 的解码性能”。

[4] https://ffmpeg.org/

[5] https://ingchips.github.io/blog/2024-02-05-external-flash/

7.5 演示

7.5.1 Windows

tts_demo 可以用来快速测试语音库和合成效果。这个程序接受两个必选参数和几个可选参数：

```
tts_demo /path/to/voice/bin /path/to/text/file [--speed X] [--tune V] [--save FN]
```

第一个参数是语音库.bin 文件，第二个参数是一个文本文件名，演示程序将合成这个文件第一行的内容，并播放出来。可选参数：--speed X 将语音速度设为 X，默认值 1.0，即不变速，详见 “语音变速”；--tune V 将微调值设为 V；--save FN 将合成的波形数据按 “raw 格式” 保存到文件 FN。当指定了 --save 参数时，只保存文件，不自动播放。

7.5.2 ING916XX

tts_firmware 是一个可以直接烧录到 ING916XX 开发板的固件。语音库需要单独烧录，请参考关于 “语音库” 的说明。test_tts 是 tts_firmware 程序里的主要函数，演示了从串口读入文本、合成，再通过串口输出波形采样的全过程。

```
void test_tts(UART_TypeDef *port)
{
    static char input[...];
    platform_printf("TTS Demo\n");

    #define MAX_SYLLABLES ...

    const struct voice_definition * voice_def = ...;

    void *context  = malloc(tts_get_context_size(MAX_SYLLABLES));
    void *scratch1 = malloc(tts_get_scratch_mem1_size());
    void *scratch2 = malloc(tts_get_scratch_mem2_size());
```

```
    struct tts_context *ctx = tts_init(voice_def,
        MAX_SYLLABLES, context);

    while (true)
    {
        // 从串口输入字符串（代码从略）
        input_from_uart(port, input, sizeof(input));

        tts_reset(ctx);
        tts_push_utf8_str(ctx, input);

        // save_pcm_samples 将 PCM 从串口输出（代码从略）
        tts_synthesize(ctx, save_pcm_samples, NULL,
            scratch1, scratch2);
    }
}
```

PC 端运行 `tts_demo.py`，允许用户输入不同的文本，收听合成效果。这个脚步依赖若干软件包，如有缺失，运行时会给出提示信息，请按提示信息安装相应的软件包。这个脚本接受一个必选参数串口号，假设开发板串口为 COM9，则如下调用该脚本：

```
python tts_demo.py COM9
```

脚本还接受可选的波特率参数，默认 115200 波特。通过 `-b` 设置波特率参数：

```
python tts_demo.py COM9 -b 921600
```

7.6 局限与建议

当前版本存在以下局限：

- 与当前最先进的 TTS 系统相比，连贯性、语气、韵律等方面存在不足，不够自然

分析：受制于嵌入式系统的计算资源，无法使用当前最先进的 TTS 技术方案。

缓解措施：对于较封闭（待合成的文本相对固定）的应用场景，可以通过自定义语音库的方法录制常见短语，改善效果。

- 多音字识别准确性有限

  分析：当前版本基于词典识别多音字。基于词典分词存在一定的错误概率，而且存在多音词（如朝阳、大夫），所以无法做到完全准确。

  缓解措施：1）使用拼音合成；2）将多音字替换为与正确读音同音的单音字；3）通过自定义语音库为多音词提供正确读音。

- 不支持儿化音、轻声变调

  缓解措施：通过自定义语音库为需要儿化、轻声的词语和短语提供正确读音。

综上，为提高合成效果，请参考以下方法或建议：

- 针对应用场景提炼词语、短语，对关键词语、短语甚至句子直接录音，创建**自定义语音库**；
- 使用空格**辅助分词**，减少分词错误；
- 使用 “[]” **手动组词**；
- 使用**拼音合成**弥补多音字识别的不足；
- 避免使用的、了、吗、呢等语气词。

第八章　语音变速

音频处理库提供了语音变速功能，可把语音在时域上拉长或则缩短，而语音的采样率、基频以及共振峰保持不变。

8.1　使用方法

1. 确定参数

   确定采样率（如 16kHz）和要关注的基频范围（例如 `STRETCH_DEF_FREQ_RANGE`：55 ~ 333 Hz）。本模块要求：

   - 采样率 / 基频下限 < 2400
   - 采样率 / 基频上限 > 24

   或者说，基频不能超出 [采样率 / 2400, 采样率 / 24]。

2. 初始化变速器对象

```c
struct stretch_ctx_t *stretch_init(
    int sampling_rate,    // 采样率
    int lower_freq,       // 基频下限
    int upper_freq,       // 基频上限
    int flags,            // 标志位
    void *buf);           // 上下文内存
```

   标志位 `flags` 是 `STRETCH_..._FLAG` 各种常量的组合，可以为 0。`buf` 是用来存放语音变速器对象上下文的内存，其大小通过 `stretch_get_context_size` 获得。

3. 分配用来接收变速输出的内存

   本模块采用流式处理，假设每次最多输入 max_num_samples 个采样，则变速器每次输出的采样数至多为：

```
int stretch_get_output_capacity(
    struct stretch_ctx_t *ctx,  // 变速器对象
    int max_num_samples,
    float min_speed);
```

   其中 min_speed 是最小速度。速度近似为原始长度与处理后的语音长度的比值，1.0 为无变化，大于 1.0 为压缩（加速），小于 1.0 为拉伸（减速）。

   根据 stretch_get_output_capacity 返回的最大采样数可为输出分配足够的内存（每个采样为一个 int16_t）。

4. 处理语音

   通过 stretch_samples 处理一小段语音数据。这个函数返回的是本次处理所输出的采样的个数。

```
int stretch_samples(
    struct stretch_ctx_t *ctx,  // 变速器对象
    const int16_t *samples,     // 输入的采样
    int num_samples,            // 输入的采样的个数
    int16_t *output,            // 输出
    float speed);               // 本次处理使用的速度
```

5. 刷新数据

   使用 stretch_flush 输出剩余数据。这个函数返回的是所输出的采样的个数。

```
int stretch_flush(
    struct stretch_ctx_t *ctx,  // 变速器对象
    int16_t *output);           // 输出
```

6. 复位

   使用 stretch_reset 复位变速器，为处理下一段语音做好准备。

```
void stretch_reset(
    struct stretch_ctx_t *ctx); // 变速器对象
```

借助变速功能还可实现语音变调（只改变音调，时域长度不变），例如：

1. 将 16kHz 采样的音频直接以 8kHz 播放，音调降低，但时长也被拉长到 2 倍；
2. 使用变速功能将速度变为 2 倍，则时长与原音频保持一致。

8.2 资源消耗

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

内存占用与参数有关，例如：

```
stretch_get_context_size(16000, STRETCH_DEF_FREQ_RANGE, 0) == 1788 (字节)
```

使用上述参数，并且 max_num_samples 取做 AMR_WB_PCM_FRAME_16k，不同 min_speed 设置下的 stretch_get_output_capacity() 返回的内存大小如表 8.1

表 8.1: 为输出预留的内存空间的大小

最小速度	为输出预留的内存（字节）
0.6	3026
0.8	2706
1.2	2386
1.4	2386

提示：用于 TTS 变速时，请留意 `tts_get_scratch_mem2_size()` 的大小，如果足以容纳变速器的输出，则可考虑复用。

8.2.1 ING916XX

在上述参数下，生成 `AMR_WB_PCM_FRAME_16k` 个采样，需要的平均时间可参考表 8.2。

减小基频范围可缩短处理时间。

表 8.2: 特定参数下语音变速时间消耗参考值

速度	时间消耗 (ms)
0.6	5.9
0.8	5.8
1.2	8.8
1.4	10.1
1.6	11.5

第九章　致谢

音频处理库使用了几种开源软件（库），针对嵌入式处理器做了修改或者剪裁，详情请见代码仓库[1]。

[1]https://github.com/ingchips/libaudio