ADT 模块用户手册

目　　录

1. 概述

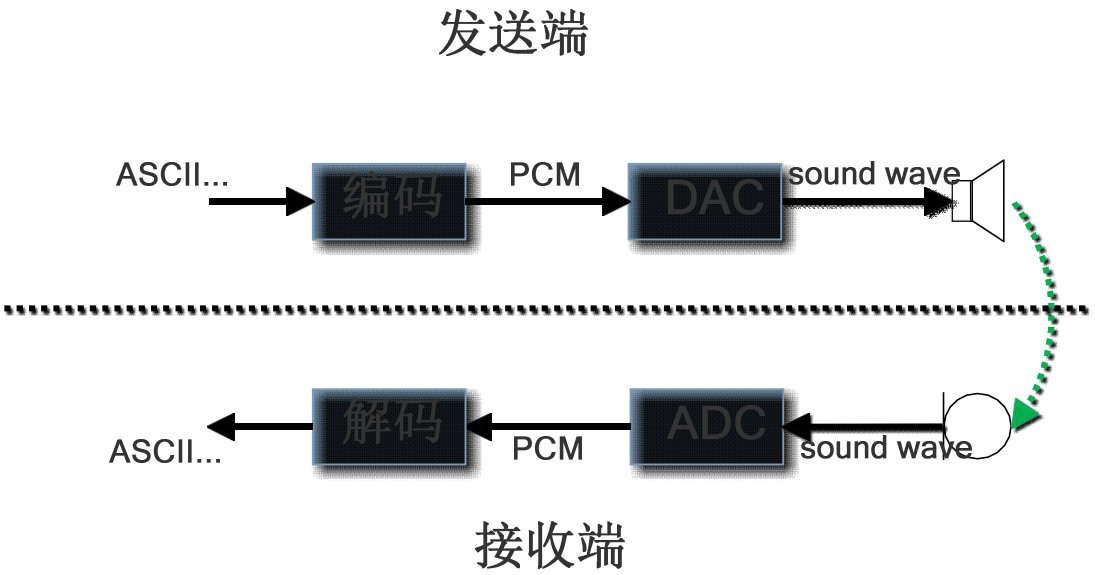

图 1 SA 系统框图

本文所述的 ADT 算法是利用声波来传送数据的一种方法。。如图 1 所示，通信系统应该包含上半部表示的发射端以及下半部表示的接收端。发送端对数据按字节进行编码调制成PCM 信号，经DAC 转换成模拟信号后经扬声器播放出来，同时接收端将声场中的信号采集后经ADC 转换成数字PCM 信号，解码模块对 PCM 信号进行解调和解码之后得到传输的 ASCII 字符信息。通信过程中，DAC 及 ADC 的采样率都需配置为**同一采样频率**。

1.1. 技术 Feature

- 信噪比大于等于 25dB 时，符号错误概率 <0.044%；
- 每个字符传输时间 64~93ms；
- 支持用户自定义纠错编码，保证通信可靠性；

1.2. 资源需求（典型场景,含纠错编码）

		Transmitter(KB)	Receiver(KB)
Memory	Code	3.7	14.6
	ROData	0.1	0.1
	Heap	4~8	60~240
	Stack	-	-
MCPS（MHZ）		19~55 MHz	20~80 MHz[1]

1. 参考下一节 1.3。

1.3. 不同采样率解码最大运算开销（ARM cortex A7 单线程）

频率	MCPS（MHZ）
11025	20
16000	30
22050	37
32000	44
44100	60
48000	74

1.4. 注意事项

1. 通信过程中需要保证发送端和接收端的采样率保持一致；
2. 发送设备的扬声器以及接收设备的麦克风的频响范围应包含**选择的频率范围**的区间；
3. 通信过程中需要采用必要措施使信噪比尽可能大，如，扬声器与麦克风尽量靠近等；
4. 通信过程中同一串字符串编码调制出的 PCM 信号应该连续地被播放和采集，中间**不能出现卡滞断音**；

2. API 描述

2.1. 枚举

2.1.1. freq_type_t

名称	取值	描述	合法的采样频率取值
FREQ_TYPE_LOW	0	低频（2000 Hz - 5000 Hz）	11025 16000 22050 32000 44100 48000
FREQ_TYPE_MIDDLE	1	中频（8000 Hz - 12000 Hz）	32000 44100 48000
FREQ_TYPE_HIGH	2	高频（16000 Hz - 20000 Hz）	44100 48000

说明：

1. 频率范围越高，声音会越尖锐，对手机喇叭的频响范围要求更大，在手机频响支持的情况下抗噪性能会较好。但是所需的采样频率也会更高，运算量更大
2. 频率范围越低，则抗噪性能会更差，但对手机的兼容性更好，并且声音会更悦耳。
3. 一般建议高信噪比条件（小于 5m 以内的传输），采用 FREQ_TYPE_LOW 和 16000Hz 的采样频率。

2.2. 发送端

2.2.1. 结构体

2.2.1.1. config_encoder_t

域	描述	备注
max_strlen	最大支持的字节数	
sample_rate	采样频率，单位 HZ	取值依赖于 freq_type,参见 2.1.1
freq_type	频率范围，参见	决定了 sample_rate 取值是否合法，参见 2.1.1
group_symbol_num	每个分组传输的字节数	合法取值需满足：group_symbol_num+error_correct_num*2<=252
error_correct	是否采用纠错码	
error_correct_num	纠错码的纠错能力(字节数)	

2.2.2. 常量和枚举

2.2.2.1. 宏

名称	取值	描述
RET_ENC_NORMAL	0	编码正常返回
RET_ENC_END	1	编码结束

RET_ENC_ERROR	-1	编码过程出现错误

2.2.3. API 函数

2.2.3.1. encoder_create

void* encoder_create(config_encoder_t* config)			
描述	创建编码器句柄		
	参数类型	含义	备注
调用参数	config_encoder_t * config	编码器的配置参数	编码器的配置参数应与对应解码器的配置参数完全一致
返回值	void*	创建好的编码器句柄	

2.2.3.2. encoder_reset

void* encoder_reset(void* handle)			
描述	创建编码器句柄		
	参数类型	含义	备注
调用参数	void* handle	编码器的句柄	本函数可以在每次编码前调用，这样不用每次编码都调用 encoder_create 函数和 encoder_destroy 函数
返回值	无		

2.2.3.3. encoder_destroy

void encoder_destroy (void* handle)			
描述	释放编码器句柄		
	参数类型	含义	备注
调用参数	void* handle	编码器句柄	
返回值	无		

2.2.3.4. encoder_getoutsize

int encoder_getoutsize (void* handle)			
描述	获取输出所需的最小 buffer 长度		
	参数类型	含义	备注

调用参数	void* handle	编码器句柄	需要创建句柄后调用
返回值	int	输出 PCM buffer 所需最小长度，单位Byte	

2.2.3.5. encoder_setinput

int encoder_setinput(void* handle, unsigned char* input)			
描述	设置需要编码的字符串		
	参数类型	含义	备注
调用参数	void* handle	编码器句柄	
	unsigned char* input	需要编码的字符串，以空字符’\0’结尾	长度不能大于调用函数 *encoder_create* 使用的 max_strlen 参数,否则将会返回错误
返回值	int	参考 2.2.2.1 小节	

2.2.3.6. encoder_getpcm

int encoder_getpcm(void* handle, short* outpcm)			
描述	编码函数,将 ASCII 字符编码调制后输出音频 PCM 信号		
	参数类型	含义	备注
调用参数	void* handle	编码器句柄	
	short* outpcm	编码调制后输出的 PCM 数据，格式为 16 bit，mono，little endian，48KHz 采样率	调用者分配，最小长度可通过 encoder_getoutsize 函数预先获得，亦可预先分配足够大的空间
返回值	int	参考 2.2.2.1 小节	

2.3. 接收端

2.3.1. 结构体

2.3.1.1. config_encoder_t

域	描述	备注
max_strlen	最大支持的字节数	
sample_rate	采样频率，单位 HZ	取值依赖于 freq_type,参见 2.1.1
freq_type	频率范围，参见 2.1.1	决定了 sample_rate 取值是否合法，参见 2.1.1

group_symbol_num	每个分组传输的字节数	合法取值需满足： group_symbol_num+error_correct_num*2<=252
error_correct	是否采用纠错码	
error_correct_num	纠错码的纠错能力(字节数)	

2.3.2. 常量和枚举

2.3.2.1. 宏

名称	取值	描述
RET_DEC_ERROR	-1	解码出错
RET_DEC_NORMAL	0	解码正常返回，需要更多的PCM 输入继续解码
RET_DEC_NOTREADY	1	解码未结束
RET_DEC_END	2	解码结束

2.3.3. API 函数

2.3.3.1. decoder_create

void* decoder_create(config_decoder_t* config)			
描述	创建解码器		
	参数类型	含义	备注
调用参数	config_decoder_t* config	解码器参数配置	解码器的配置参数应与对应的编码器配置参数完全一致，否则解码会出错
返回值	void*	创建好的解码器句柄	

2.3.3.2. decoder_reset

void* decoder_reset(void* handle)			
描述	创建解码器		
	参数类型	含义	备注
调用参数	void* handle	解码器句柄	这个函数可以在每次解码前调用，这样可以不用每次解码都调用decoder_create 函数和 decoder_destroy 函数
返回值	无		

2.3.3.3. decoder_getbsize

int decoder_getbsize(void* handle)			
描述	获取 blocksize，用于每次调用 PCM 的输入长度		
	参数类型	含义	备注
调用参数	void* handle	解码器句柄	
返回值	int	Block 长度，单位样本点	

2.3.3.4. decoder_destroy

void decoder_destroy(void* handle)			
描述	释放解码器		
	参数类型	含义	备注
调用参数	void* handle	解码器句柄	
返回值	无		

2.3.3.5. decoder_fedpcm

int decoder_fedpcm(void* handle, short* pcm)			
描述	对接收到的 PCM 数据进行解码的函数		
	参数类型	含义	备注
调用参数	void* handle	解码器句柄	
	short* pcm	接收到的PCM 数据，16bit，mono	Buffer 包含的样本点数需预先通过调用 decoder_getbsize 确定
返回值	int	参考 2.3.2.1 小节	

2.3.3.6. decoder_getstr

int decoder_getstr(void* handle, unsigned char* str)			
描述	获取解码结果		
	参数类型	含义	备注
调用参数	void* handle	解码器句柄	
	unsigned char* str	解码输出的字符串,以空字符’\0’结尾	调用者分配
返回值	int	参考 2.3.2.1 小节	

2.4. 调用示例

2.4.1. Transmitter

```
#include " adt.h"


int main(int argc, char** argv)
{
    short*  buffer;
    void*  encoder;
    int bsize, ret;
    config_encoder_t config_encoder;
    config_encoder.max_strlen = 128;
    config_encoder.freq_type = FREQ_TYPE_LOW;
    config_encoder.sample_rate = 16000;
    config_encoder.error_correct = 1;
    config_encoder.error_correct_num = 4;
    config_encoder.group_symbol_num = 10
  /* create encoder handle */
    encoder = encoder_create(&config_encoder);
    if(encoder == NULL)
    {
        printf("create encoder handle error !\n");
        return -1;
    }


  /* get size of the buffer to be malloced */
    bsize = encoder_getoutsize(encoder);
    buffer = (short*)malloc(bsize);
    if(buffer == NULL)
    {
        printf("alloc buffer error!\n");
        return -1;
    }

  /* set the input string to be transmitted */
    ret = encoder_setinput(encoder, argv[1]);
    if(ret == RET_ENC_ERROR)
    {
        printf("encoder set input error!\n");
        return -1;
    }

  /* encoding loop */
    ret = RET_ENC_NORMAL;
```

```
    while(ret != RET_ENC_ERROR && ret != RET_ENC_END)
    {

        /* get the encoded pcm data */
        ret = encoder_getpcm(encoder, buffer);
        /* transfer pcm data to DAC */
        ....

    }
    if(ret == RET_ENC_ERROR)
    {
        printf("encoder error occured!\n");
        return -1;
    }
    /* free encoder handle */
    encoder_destroy(encoder);
    return 0;
}
```

2.4.2. Receiver

```
#include "decode_rs.h"



int main(int argc, char** argv)
{
    int ret_dec;
    int items, bsize,i;
    void* decoder;
    short* buffer;
    char out[129];
    ret_dec = RET_DEC_NORMAL;
    config_decoder_t config_decoder;
    config_decoder.max_strlen = 128;
    config_decoder.freq_type = FREQ_TYPE_LOW;
    config_decoder.sample_rate = 16000;
    config_decoder.error_correct = 1;
    config_decoder.error_correct_num = 4;
    config_decoder.group_symbol_num = 10

    /* create decoder handle */
    decoder = decoder_create(& config_decoder);

    if(decoder == NULL)
```

```
{
    printf("allocate handle error !\n");
    return -1;
}

/* get buffer size and allocate buffer */
bsize = decoder_getbsize(decoder);
buffer = malloc(sizeof(short)*bsize);
if(buffer == NULL)
{
    printf("allocate buffer error !\n");
    goto finish;
}
/* decoding loop */
while(ret_dec == RET_DEC_NORMAL)
{
    /*samples get from ADC */
    items = …
    /*padding with zeros if items is less than bsize samples*/
    for(i = items; i< bsize; i++)
    {
        buffer[i] = 0;
    }
    /* input the pcm data to decoder */
    ret_dec = decoder_fedpcm(decoder, buffer);
}
/* check if we can get the output string */
if(ret_dec != RET_DEC_ERROR)
{
    /* get the decoder output */
    ret_dec = decoder_getstr(decoder, out);
    if(ret_dec == RET_DEC_NORMAL)
    {
        /* this is the final decoding output */
        printf("outchar: %s \n", out);
    }else
    {
        /* decoding have not done, so nothing output */
        printf("decoder output nothing! \n");
    }
}else
{
    printf("decoder error !\n");
}
```

```
finish:
    /* free handle */
    decoder_destroy(decoder);
    return 0;
```