# WM8978音频模块

## 1.WM8978介绍

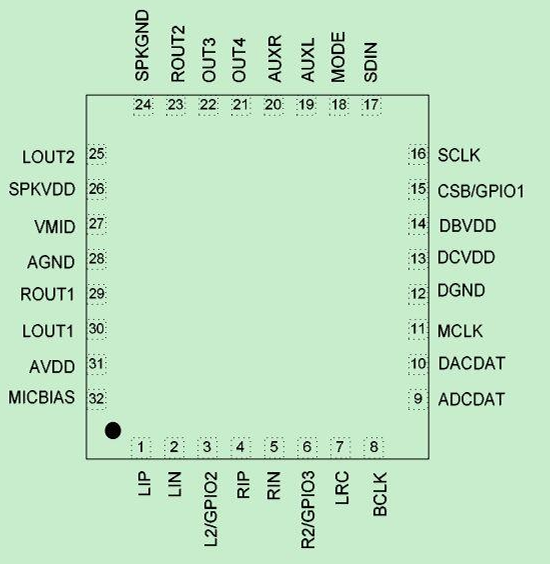
WM8978是一个低功耗的多媒体数字信号译码器，适用于便携式音频设备，可通过I2C通过配置寄存器来控制芯片。

WM8978的LOUT1和ROUT1可用于耳机驱动，LOUT2和ROUT2用于扬声器驱动，在8ΩBTL扬声器/3.3V SPKVDD的条件下输出功率为0.9W。OUT3/OUT4，可以配置成立体声输出（OUT3左边输出，OUT4右输出）

WM8978提供许多麦克风输入配置，包含差分和单端输入。单端输入中，麦克风信号应该输入到LIN或RIN。

WM8978有两个通信接口，一个是数字音频数据通信接口，另一个是控制接口。音频接口使用I2S协议，支持左对齐、右对齐和I2S标准模式（飞利浦标准模式）以及DSP模式A和模拟B。控制接口使用I2C协议，通过控制接口可以访问内部寄存器。WM8978芯片地址为0x1A，共有58个寄存器，表示为R0至R57，每个寄存器功能可参考芯片手册。需要注意的是，WM8978的寄存器是16bit长度，高7位[15:9]bit用于表示寄存器地址，低9位[8:0]用于表示寄存器数据。控制器向芯片发送控制命令时，必须传输16bit指令，芯片会根据接收命令高7位寻址。

引脚功能如下：





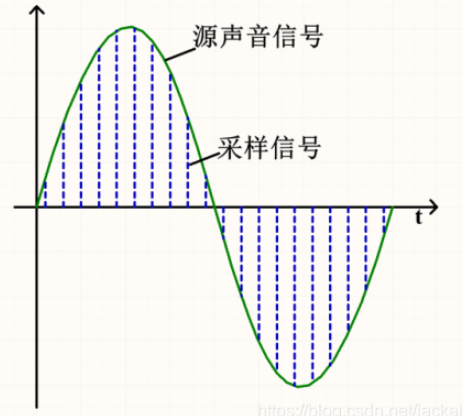


## 2.I2S协议介绍

现实中的声音是一种连续的波，是一种模拟量，这并不适合声音的保存和长距离传输，这时一般的做法就是把模拟量转换成数字量保存，播放时再解码把数字量转换成模拟量输出音频信号。

模拟量转换成数字量一般分为采样、量化、编码这三个过程。用一个比源声音频率高的采样信号去量化原声音，记录每个采样点的值，最后如果把每个采样点连接起来，就可以得到近似源声音的曲线。可以看出采样频率越高最后得到的结果与源声音越吻合，但采样数据也会越大，一般使用44.1KHz采样率就可得到高保真的声音。

采样的结果还需要量化，确定用什么值来表示采样结果，与之相关的就是量化位数。量化位数表示采样点用多少位表示数据范围，常用的有8bit、16bit、24bit或32bit，位数越高最后还原得到的音质越好，但数据量也会越大。



I2S协议是一种广泛用于数字音频传输的串行接口标准。它最初是由飞利浦公司开发，用于解决集成电路之间传输音频数据的问题。

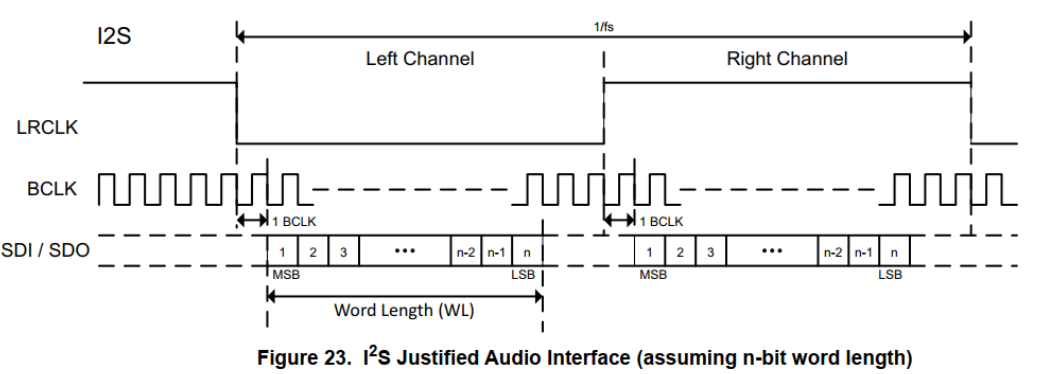
I2S总线接口有3个主要信号，但只能实现数据半双工传输（有些设备增加了拓展引脚，可以实现全双工传输，这里不详细介绍）

1. SD（Serial Data）：串行数据线，用于发送或接收两个时分复用的数据通道上的数字音频数据。
2. WS(Word Select):字段选择线，也称为帧时钟线(LRCLK)，表明当前传输数据的声道（左右声道）。WS线的频率等于采样频率（Fs）
3. CK(Serial Clock):串行时钟线，也称位时钟（BCLK）,数字音频的每一位数据都对应有一个CK脉冲，频率为：2\*采样率\*量化位数（2代表左右两个通道）

另外，有时为了使系统间更好的同步，还要传输一个主时钟（MCK）。

在统一的I2S接口下，出现了多种不同的数据格式，有I2S Philips标准、左对齐（MSB）、右对齐（LSB）、PCM，以下只介绍Philips标准。

**I2S Philips标准时序图如下所示**：

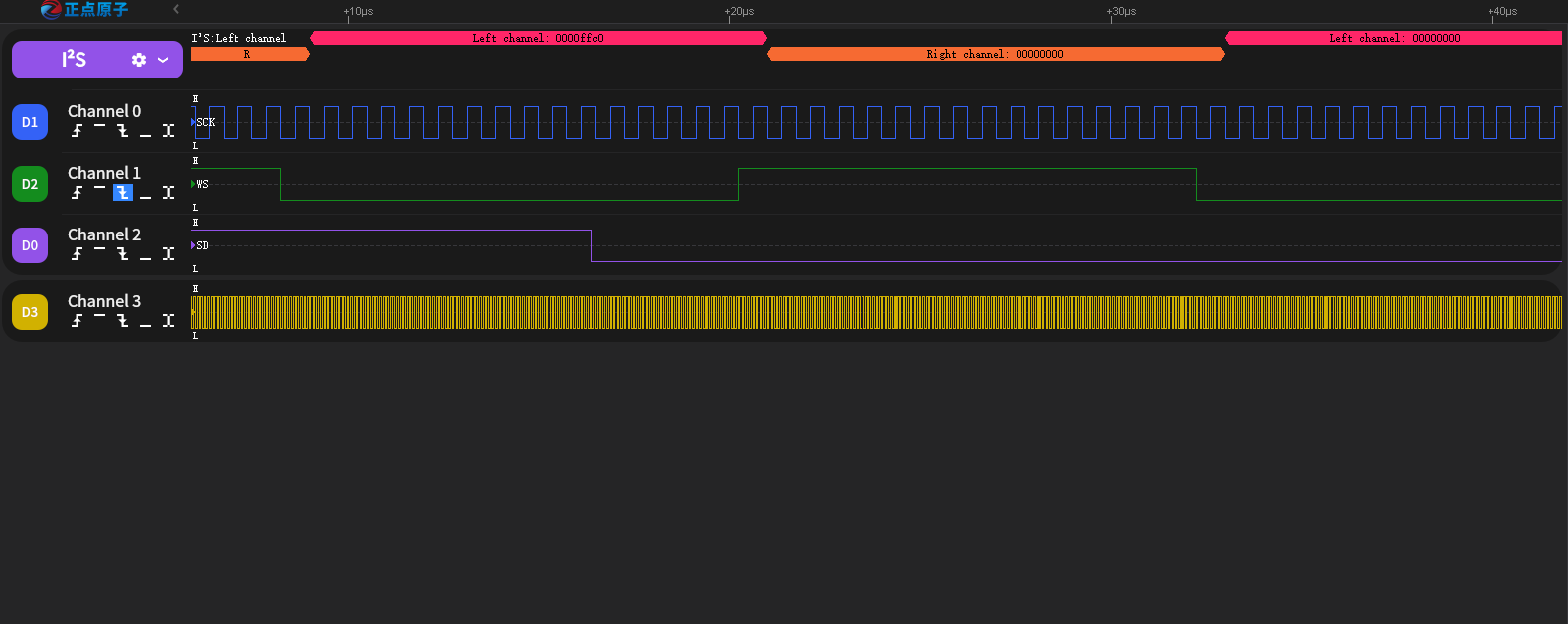


LRCLK(WS)信号用来指示当前正在发送的数据所属声道，低电平表示左声道，高电平表示右声道，它一个周期就表示一帧，而一帧的数据里包含左声道和右声道的数据，所以WS的频率就等于音频的采样率当WS下降沿时表示一帧新的数据开始传输。在标准I2S模式下，WS信号在CK时钟信号的下降沿变化，WS跳变后的第一个bit无效，从第二个bit开始读取。（也就是WS跳变后，等待一个CK时钟周期才读取串行数据线数据。

CK时钟线下降沿的时候，发送端控制数据线SD切换数据bit,当CK时钟线上升沿的时候，接收端从SD读取数据。

另外，根据音频数据的量化位数的不同，需要发送的音频数据的位数也不同，以本次使用的控制器CH32V307VCT6为例，CH32V307VCT6控制器的I2S的数据寄存器只有16bit,因此在传输24bit和32bit数据是需要发送两个。需要注意的是如果接收端能处理的有效位数少于发送端，可以放弃数据帧中多余的低位数据；如果接收端能处理的有效位数多于发送端，可以自行补足剩余的位。本次实验只涉及16bit数据。

使用逻辑分析仪采集I2S波形如下图所示，采样率44.1KHz,16bit

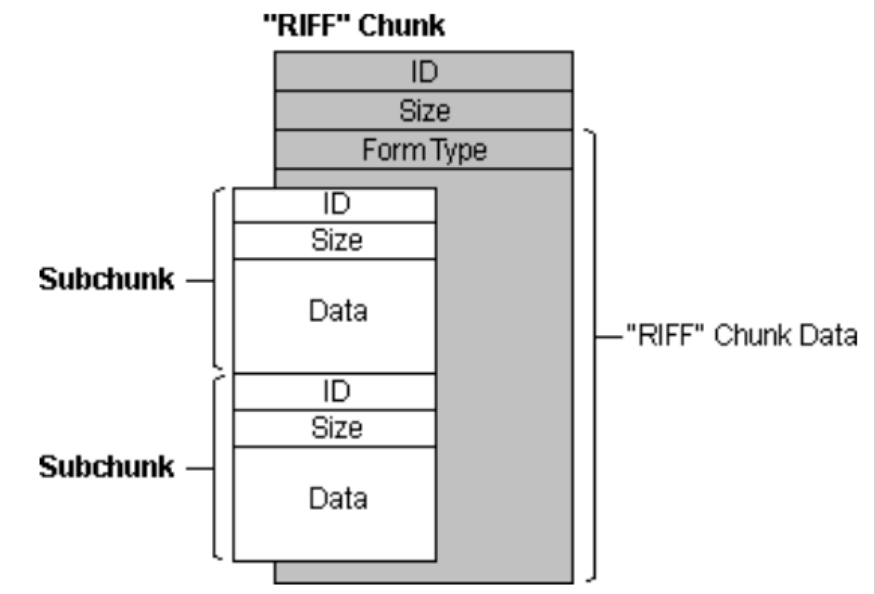


可以看出在一个WS周期内，分别发送了16bit左声道音频和16bit右声道音频。由此也可以计算出CK频率为2\*16\*44.1K = 14.112Mhz。

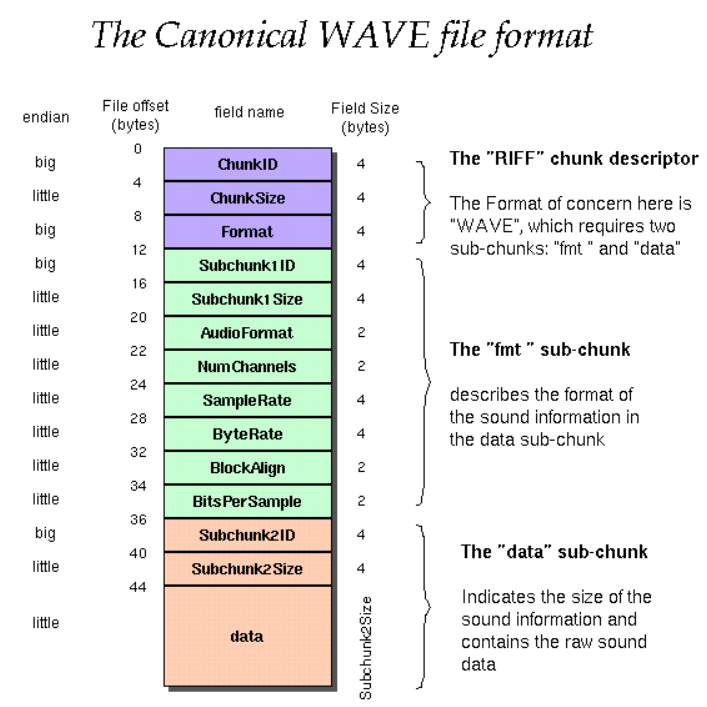
## 3.WAV文件格式

了解完音频传输方式后，我们还需要了解下音频文件。WAV是一种存储声音波形的数字音频格式，是由微软公司和IBM联合设计的，符合资源互换文件格式（RIFF）格式。WAV文件编码支持非压缩的PCM脉冲编码调制格式，还支持压缩型的编码格式，MP3编码同样也可以运用在WAV中。WAV文件最主要的还是使用PCM编码格式，PCM编码是直接存储声波采样被量化后所产生的非压缩数据,故被视为单纯的无损耗编码格式,其优点是可获得高质量的音频信号。  
基于PCM编码的WAV格式是最基本的WAV格式,被声卡直接支持,能直接存储采样的声音数据,所存储的数据能直接通过声卡播放,还原的波形曲线与原始声音波形十分接近。

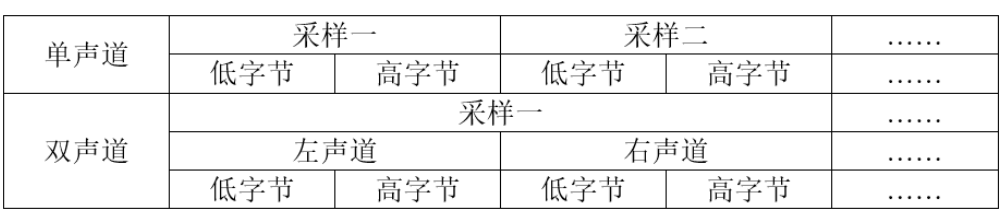
RIFF文件有不同数量的chunk(区块)组成，每个chunk由“标识符”、“数据大小”和“数据”三个部分组成，“标识符”和“数据大小”都占4个字节空间，所以总文件大小为“数据大小”+8。一般来说chunk内部不允许再包含chunk,但ID为“RIFF”和“LIST”的chunk却是允许。对此“RIFF”在其“数据”首4个字节用来存放“格式标识码”。 简单RIFF格式文件结构参考图RIFF文件格式



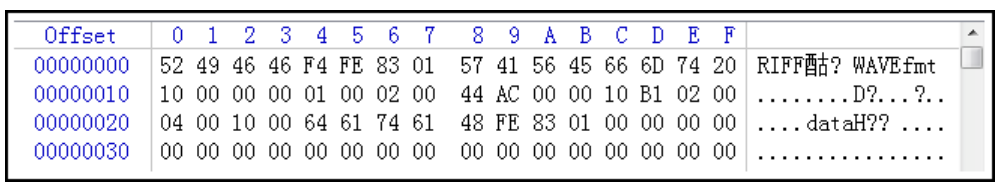
WAV文件是一种结构简单的RIFF文件，其“格式标识码”定义为WAVE.RIFF chunk包括两个子chunk,ID分别fmt和data,还有一个可选的fact chunk。Fmt chunk用于表示音频数据的属性，包括编码方式、声道数目、采样频率、量化位数等。Factchunk是一个可选chunk,一般当wav文件由某些软件转换而成就包含fact chunk。Data chunk包含wav文件的数字化波形数据。Wav文件结构如下图所示：



需要注意的是，wav文件以小端形式来进行数据存储。所谓大端形式，是指低位保存在内存的高地址中，而数据的高位，保存在内存的低地址中；所谓小端模式，是指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中。以量化位数为16bit时不同声道数据在data chunk数据排列为例，如下图所示：



使用winhex查看wav文件，如下图所示：



对44字节文件头进行解读



## 4.FatFs文件系统

了解完wav音频文件，我们还需要考虑如何存储wav文件。我们可以直接将wav文件写入flash或sd卡的指定地址上，在需要的时候从该地址把数据读取出来。但是这样直接存储数据会带来极大的不便，存储数据量多的话不方便查找，也不好以文件的形式分类管理，因此我们需要一种更有效的方式来管理存储内容。这种管理方式就是文件系统。

使用文件系统时，数据都以文件的形式存储。写入新文件时，先在目录中创建一个文件索引，指示文件存放的物理地址，再把数据存储到该地址中，需要读取数据时，可以从目录中找到该文件的索引，进而在相应地址中读取出数据，当然，具体操作还涉及到一些列更复杂的操作。

FatFs是面向小型嵌入式系统的一种通用FAT文件系统，全独立于底层I/O介质，支持FAT12、FAT16、FAT32等格式，可以很容易地不加修改地移植到其他处理器中。接下来以CH32V30为例子移植FatFs文件系统，控制SD卡。

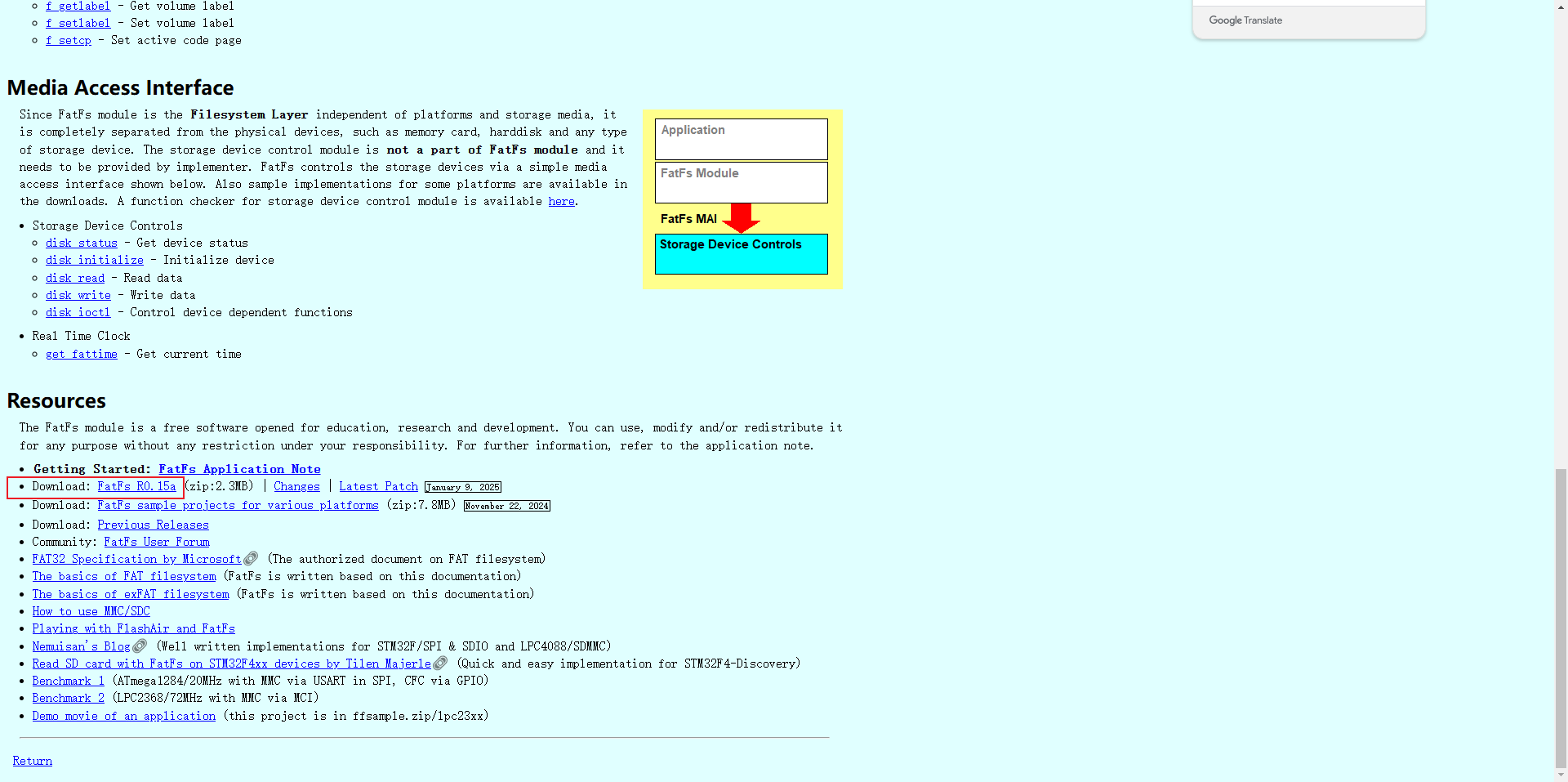
SD卡操作例程可以使用沁恒地官方例程

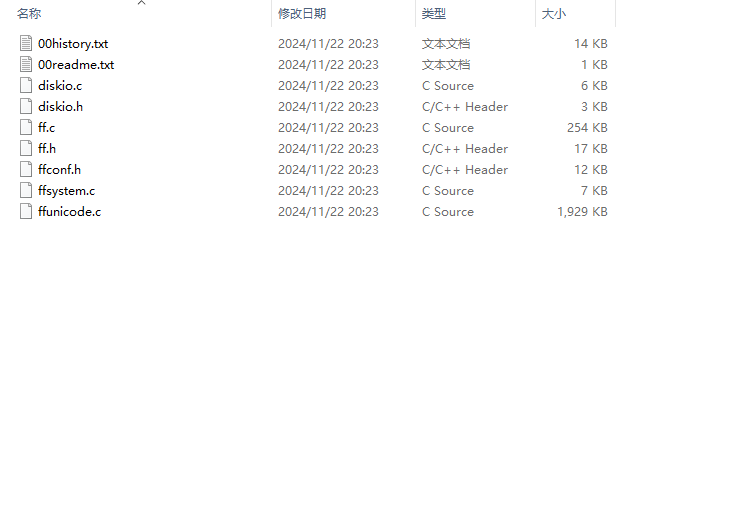
<https://www.wch.cn/downloads/CH32V307EVT_ZIP.html>

打开例程中的EVT-EXAM-SDIO-User文件夹，复制sdio.c和sdio.h文件到自己地工程，这是SD卡的驱动。

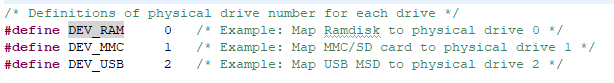
然后到Fatfs官网下载最新的文件

网址：<http://elm-chan.org/fsw/ff/00index_e.html>

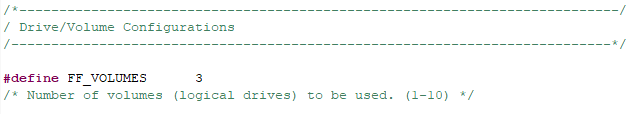




将文件一同复制进工程中。 FatFs的移植主要在diskio.c中，我们将SD卡驱动头文件sdio.h包含进diskio.h中。在sdio.c中，可以三个对设备类型的定义。



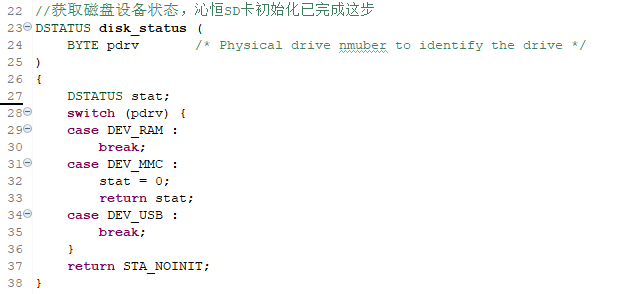
本次移植的是SD卡，属于MMC类型。这里使用了默认的三个类型，最好去ff\_conf.h中修改定义



接着查看sdio.c，可以看到5个函数，disk\_status、disk\_initialize、disk\_read、disk\_write、disk\_ioctl。

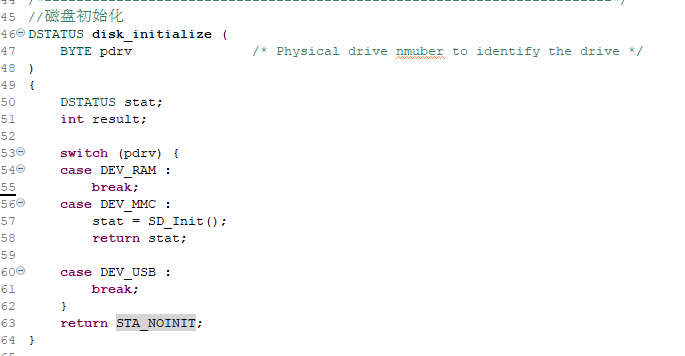
**1.disk\_status**

这是获取磁盘状态的函数，沁恒的SD卡驱动初始化已经对SD的状态作出判断，所以我们不需要这个函数。Switch中除了DEV\_MMC的选型都直接break,DEV\_MMC选型直接返回0，如下所示：



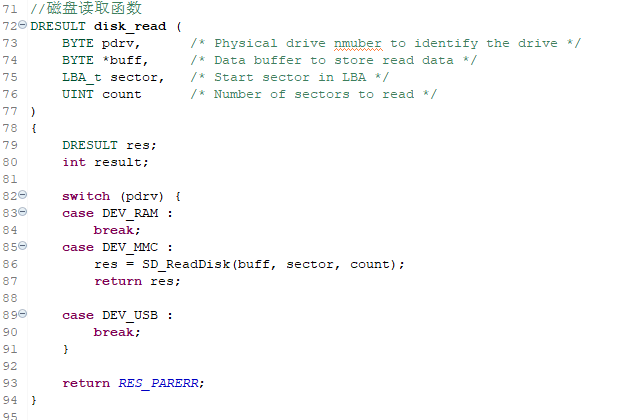
**2.disk\_initialize**

这个函数是磁盘初始化函数，我们同样将除了DEV\_MMC的选型改成break，，然后在DEV\_MMC的选项中引用SD卡初始化函数SD\_Init()。返回值为0表示成功。如下所示：



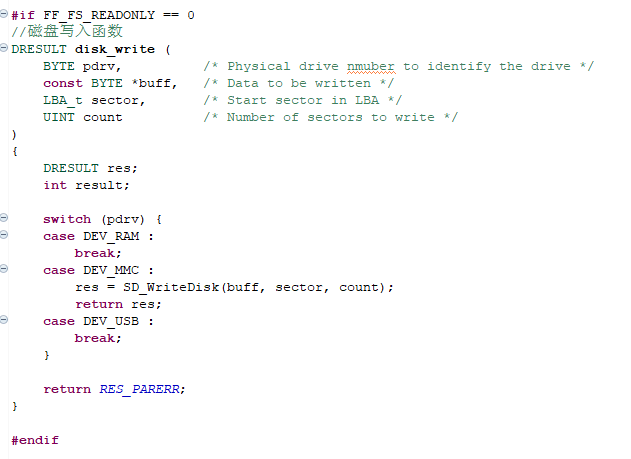
**3.disk\_read**

这是磁盘读取函数，以同样的方式将SD卡读取函数SD\_ReadDisk()复制到DEV\_MMC选项下，然后看SD\_ReadDisk函数需要的参数（数据地址，扇区，数量），与disk\_read的传参相符，程序如下所示：



**4.disk\_write**

这个是磁盘写入函数，同样找到sdio.c中的磁盘写入函数SD\_WriteDisk(),复制进DEV\_MMC选项中，程序如下所示：

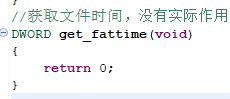


**5.disk\_ioctl**

这个函数主要是用来获取磁盘的各种信息，GET\_SECTOR\_SIZE是获取磁盘大小，GET\_BLOCK\_SIZE是用来指定擦除的最小单位，GET\_SECTOR\_COUNT是获取扇区个数，程序如下所示：



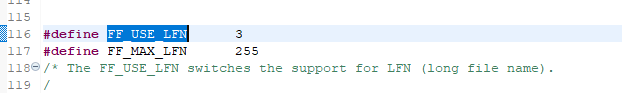
最后在diskio.c文件末尾添加get\_fatting()函数，作用是获取文件时间。用不到的话这个函数功能可以不实现，直接返回0.



完成diskio.c文件的修改后，还需要调整一些配置。首先在ffconf.h文件中，FF\_USE\_MKFS值修改为1，启用格式化磁盘功能



FF\_USE\_LFN值修改为3，启用长文件名



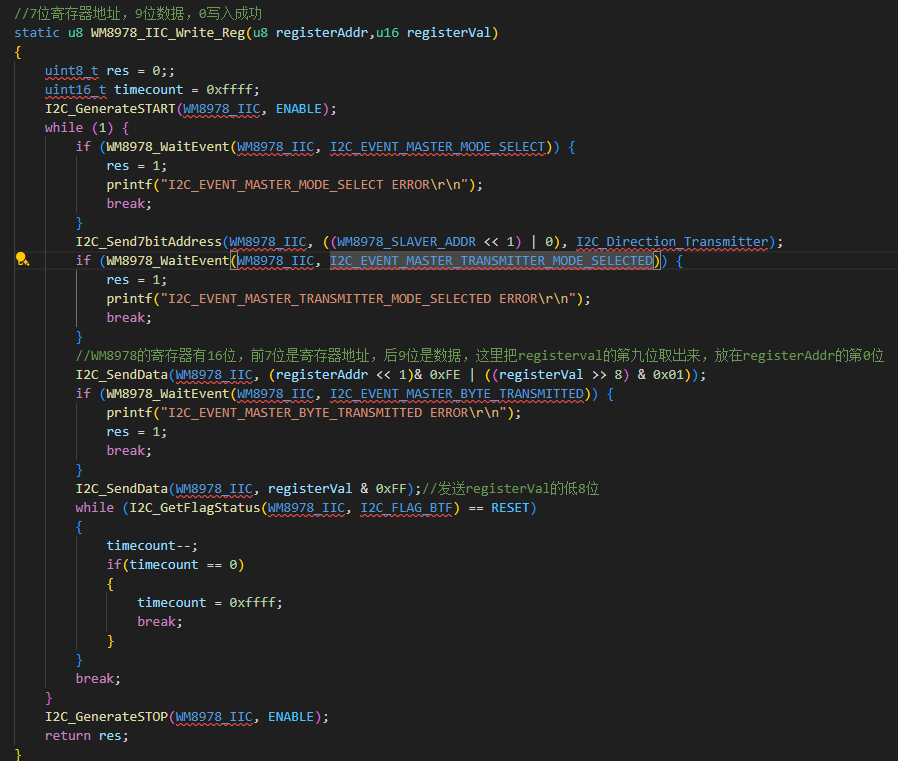
到此就完成了FatFs文件系统的移植。通过f\_mkfs（）格式化SD卡或将SD卡放入读卡器连接电脑格式化为FAT32格式后，就可以使用f\_mount()函数挂载SD卡。

## 5.程序实现

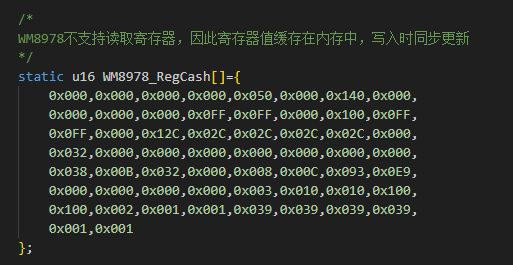
音乐播放功能的实现流程如下：使用读卡器连接电脑，拷贝44.1Khz 16bit WAV音频文件到SD卡，程序先初始化WM8978,然后挂载SD卡，根据音频文件参数配置I2S，然后读取WAV文件音频数据，通过I2S传输给WM8978，就可以在耳机口播放音乐。

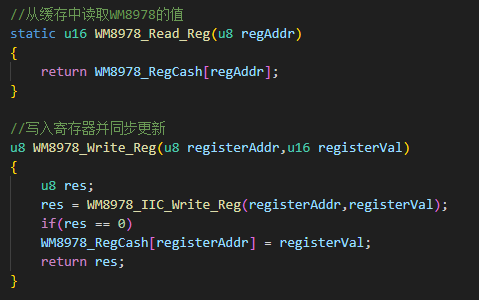
### 5.1 WM8978配置

WM8978\_IIC\_Write\_Reg()函数用于向WM8978寄存器写入数据，函数有两个形参，一个是寄存器地址registerAddr (范围0-57)，一个是寄存器的值registerVal（uint16\_t）。WM8978的寄存器有16位，前7位是寄存器地址，后9位是数据，因为I2C数据一次只能发送8bit数据，所以需要把registerVal变量的第9bit整合到registerVal的第0bit先发送，接着再发送registerVal的低8位数据，程序如下所示：



WM8978的I2C通信模式是只写的，我们无法直接读取WM8978的寄存器内容，但可以创建一个存放WM8978所有寄存器值的数组，系统复位时设置为默认值，每次修改寄存器内容时同步更新该数组。程序如下所示：





基于此，我们需要一些配置WM8978功能的函数

#### 设置耳机音量

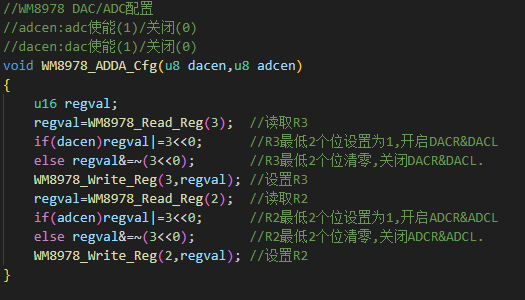


该函数用于改变OUT1通道音量，范围0-63,具体寄存器功能参考芯片手册。

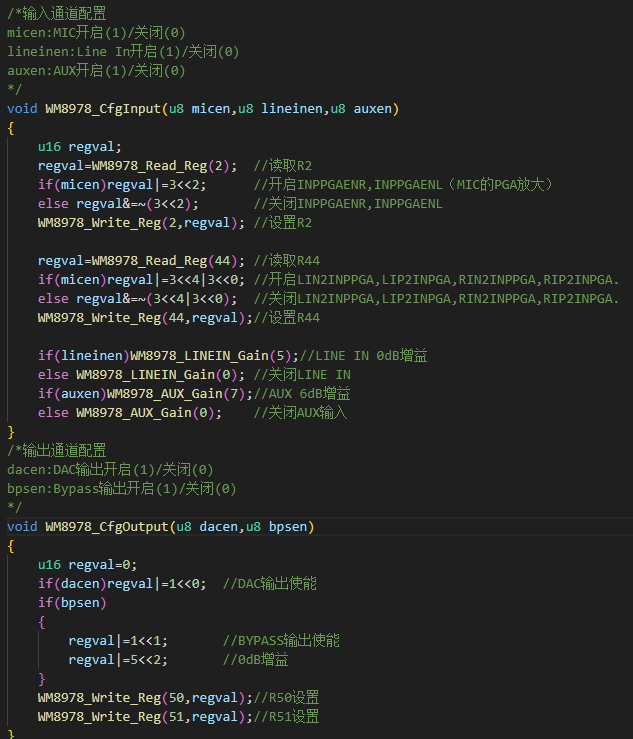
#### 设置音频接口



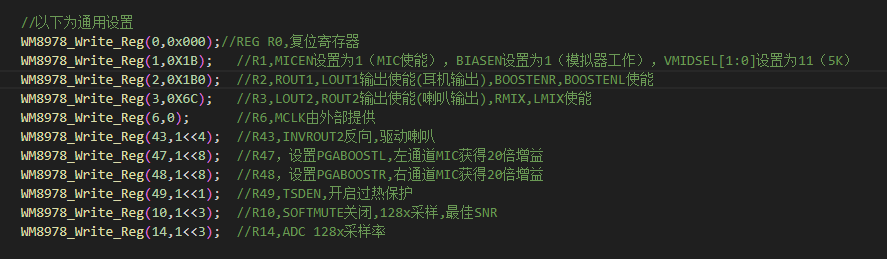
#### DAC/ADC配置



#### 输入输出通道配置



#### 通用初始化配置



### 5.2 I2S配置



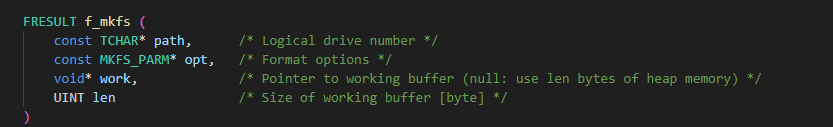
USER\_I2S\_Init用于配置CH32V307的I2S接口工作模式，形参用于配置I2S接口标准、主从模式、有效电平、数据长度和采样频率。本次不使用DMA传输音频数据，因为CH32V307不支持DMA双缓冲传输，使用DMA可以节省CPU资源，但对音质提升不大。



### 5.3 文件操作

我们使用FatFs文件系统，就可以很方便的操作文件。首先需要了解FatFs的几个基本文件操作函数。

#### f\_mkfs



格式化存储空间。

参数：

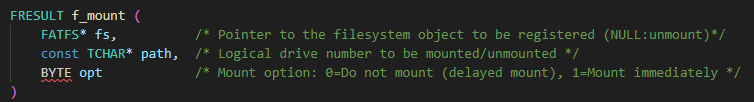
Const TCHAR\* path:逻辑设备编号

Const MKFS\_PARM\* opt：系统的格式，一般写0，格式化为FAT32。

Void\* work:执行格式化时需要的缓存，FAT32时填512

UINT len：表示work缓存的长度

#### f\_mount



在注册或注销一个工作区（文件系统对象），在使用任何其他文件函数之前，必须使用该函数为每个卷注册一个工作区。要注销一个工作区，只要指定\*fs为NULL即可，然后该工作区可以被丢弃。

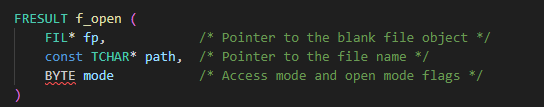
参数：

FATFS\* FS:工作区（文件系统对象）指针

Const TCHAR\* path:工作区的逻辑驱动器号（0-9）

BYTE opt:可选0或1，0表示延迟挂载，1表示立即挂载

#### f\_open



创建/打开一个用于访问文件的文件对象。

参数：

FIL\* fp:空白文件对象结构指针

Const TCHAR\* path：NULL结尾的

BYTE mode: 指定文件的访问类型和打开方法。它是由下列标志的一个组合指定的。

**FA\_READ** 指定读访问对象。可以从文件中读取数据。与FA\_WRITE结合可以进行读写访问

**FA\_WRITE** 指定写访问对象。可以向文件中写入数据。

与FA\_READ结合可以进行读写访问。

**FA\_OPEN\_EXISTING** 打开文件。如果文件不存在，则打开失败。(默认)

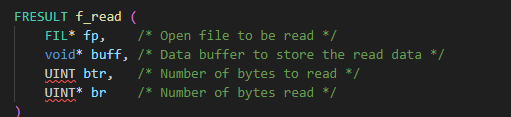
**FA\_OPEN\_ALWAYS** 如果文件存在，则打开；否则，创建一个新文件。

**FA\_CREATE\_NEW** 创建一个新文件。如果文件已存在，则创建失败。

**FA\_CREATE\_ALWAYS** 创建一个新文件。如果文件已存在，则它将被截断并覆盖。

注意：当\_FS\_READONLY==1时，模式标志 FA\_WRITE, FA\_CREATE\_ALWAYS, FA\_CREATE\_NEW, FA\_OPEN\_ALWAYS 是无效的。

#### f\_read



从一个文件中读取数据，可通过反复调用该函数，直至读取完文件。

参数：

FIL\* fp：指向将被读取的已打开文件对象结构的指针

Void\* buff：指向存储读取数据的缓冲区指针

UINT btr：要读取的字节数

UINT\* br：返回已读取字节数变量的指针

### 5.2 WAV文件解析

依据WAV文件格式，我们可以写出WAV解析程序，提取WAV文件的大小，采样频率，量化位数等参数。

//WAV解析初始化

//fname:文件路径+文件名

//wavx:wav 信息存放结构体指针

//返回值:0,成功;1,打开文件失败;2,非WAV文件;3,DATA区域未找到.

u8 wav\_decode\_init(u8\* fname,\_\_wavctrl\* wavx)

{

    FIL\*ftemp;

    u8 \*buf;

    u32 br=0;

    u8 res=0;

    ChunkRIFF \*riff;

    ChunkFMT \*fmt;

    ChunkFACT \*fact;

    ChunkDATA \*data;

    ftemp=(FIL\*)malloc(sizeof(FIL));

    buf=malloc(512);

    if(ftemp&&buf)  //内存申请成功

    {

        res=f\_open(ftemp,(TCHAR\*)fname,FA\_READ);//打开文件

        if(res==FR\_OK)

        {

            f\_read(ftemp,buf,512,&br);  //读取512字节在数据

            riff=(ChunkRIFF \*)buf;      //获取RIFF块

            if(riff->Format==0X45564157)//是WAV文件

            {

                printf("判断是wav文件\r\n");

                fmt=(ChunkFMT \*)(buf+12);   //获取FMT块

                fact=(ChunkFACT \*)(buf+12+8+fmt->ChunkSize);//读取FACT块

                if(fact->ChunkID==0X74636166||fact->ChunkID==0X5453494C)wavx->datastart=12+8+fmt->ChunkSize+8+fact->ChunkSize;//具有fact/LIST块的时候(未测试)

                else wavx->datastart=12+8+fmt->ChunkSize;

                data=(ChunkDATA \*)(buf+wavx->datastart);    //读取DATA块

                if(data->ChunkID==0X61746164)//解析成功!

                {

                    printf("ChunkID解析成功\r\n");

                    wavx->audioformat=fmt->AudioFormat;     //音频格式

                    wavx->nchannels=fmt->NumOfChannels;     //通道数

                    wavx->samplerate=fmt->SampleRate;       //采样率

                    wavx->bitrate=fmt->ByteRate\*8;          //得到位速

                    wavx->blockalign=fmt->BlockAlign;       //块对齐

                    wavx->bps=fmt->BitsPerSample;           //位数,16/24/32位

                    wavx->datasize=data->ChunkSize;         //数据块大小

                    wavx->datastart=wavx->datastart+8;      //数据流开始的地方.

                    printf("wavx->audioformat:%d\r\n",wavx->audioformat);

                    printf("wavx->nchannels:%d\r\n",wavx->nchannels);

                    printf("wavx->samplerate:%d\r\n",wavx->samplerate);

                    printf("wavx->bitrate:%d\r\n",wavx->bitrate);

                    printf("wavx->blockalign:%d\r\n",wavx->blockalign);

                    printf("wavx->bps:%d\r\n",wavx->bps);

                    printf("wavx->datasize:%d\r\n",wavx->datasize);

                    printf("wavx->datastart:%d\r\n",wavx->datastart);

                }else res=3;//data区域未找到.

            }else res=2;//非wav文件

        }else res=1;//打开文件错误\*/

    }

    f\_close(ftemp);

    free(ftemp);//释放内存

    return 0;

}

同样也可以写填充函数，将WAV的音频部分数据填充到一个8192字节的数组中，然后通过I2S发送

//填充buf

//buf:数据区

//size:填充数据量

//bits:位数(16/24)

//返回值:读到的数据个数

u32 wav\_buffill(u8 \*buf,u16 size,u8 bits)

{

    u16 readlen=0;

    u32 bread;

    u16 i;

    u8 \*p;

    u8 res;

    //printf("开始填充\r\n");

    if(bits==24)//24bit音频,需要处理一下

    {

        readlen=(size/4)\*3;                         //此次要读取的字节数

        f\_read(audiodev.file,audiodev.tbuf,readlen,(UINT\*)&bread);  //读取数据

        p=audiodev.tbuf;

        for(i=0;i<size;)

        {

            buf[i++]=p[1];

            buf[i]=p[2];

            i+=2;

            buf[i++]=p[0];

            p+=3;

        }

        bread=(bread\*4)/3;      //填充后的大小.

    }else

    {

        f\_read(audiodev.file,buf,size,(UINT\*)&bread);//16bit音频,直接读取数据

        //printf("实际读取数据量为 = %d\r\n",bread);

        if(bread<size)//不够数据了,补充0

        {

            for(i=bread;i<size-bread;i++)buf[i]=0;

        }

    }

    return bread;

}

播放函数如下所示：

//播放某个WAV文件

//fname:wav文件路径.

//返回值:

u8 wav\_play\_song(u8\* fname)

{

    u8 key;

    u8 count = 0;

    u8 t=0;

    u8 res;

    u32 fillnum;

    u16 i;

    audiodev.file=(FIL\*)malloc(sizeof(FIL));

    audiodev.i2sbuf=malloc(WAV\_I2S\_TX\_DMA\_BUFSIZE);

    audiodev.tbuf=malloc(WAV\_I2S\_TX\_DMA\_BUFSIZE);

    if(audiodev.file&&audiodev.i2sbuf&&audiodev.tbuf)

    {

        res=wav\_decode\_init(fname,&wavctrl);//得到文件的信息

        if(res==0)//解析文件成功

        {

            printf("解析文件成功\r\n");

            if(wavctrl.bps==16)

            {

                //WM8978\_CfgAudioIF(I2S\_Standard\_Phillips,I2S\_DataFormat\_16b);    //飞利浦标准,16位数据长度

                USER\_I2S\_Init(I2S\_Standard\_Phillips,I2S\_Mode\_MasterTx,I2S\_CPOL\_Low,I2S\_DataFormat\_16b,I2S\_AudioFreq\_44k);     //飞利浦标准,主机发送,时钟低电平有效,16位扩展帧长度

            }else if(wavctrl.bps==24)

            {

                //WM8978\_CfgAudioIF(I2S\_Standard\_Phillips,I2S\_DataFormat\_24b);    //飞利浦标准,24位数据长度

                USER\_I2S\_Init(I2S\_Standard\_Phillips,I2S\_Mode\_MasterTx,I2S\_CPOL\_Low,I2S\_DataFormat\_24b,I2S\_AudioFreq\_44k);     //飞利浦标准,主机发送,时钟低电平有效,24位扩展帧长度

            }

            res=f\_open(audiodev.file,(TCHAR\*)fname,FA\_READ);    //打开文件

            if(res==0)

            {

                printf("打开文件成功\r\n");

                f\_lseek(audiodev.file, wavctrl.datastart);      //跳过文件头

                printf("开始播放\r\n");

                while(res==0)

                {

                   fillnum=wav\_buffill(audiodev.i2sbuf,WAV\_I2S\_TX\_DMA\_BUFSIZE,wavctrl.bps);//填充播放数据

                   for(i=0;i<WAV\_I2S\_TX\_DMA\_BUFSIZE/2;i++)

                   {

                    while(SPI\_I2S\_GetFlagStatus(SPI2, SPI\_I2S\_FLAG\_TXE)==RESET);

                    SPI\_I2S\_SendData(SPI2,((u16)audiodev.i2sbuf[i\*2+1]<<8)|audiodev.i2sbuf[i\*2]);//WAV文件数据低端在前

                   }

                    if(fillnum!=WAV\_I2S\_TX\_DMA\_BUFSIZE)//播放结束?

                    {

                        printf("播放结束\r\n");

                        res=1;

                        break;

                    }

                }

            }else res=0XFF;

        }else res=0XFF;

    }else res=0XFF;

    free(audiodev.tbuf);   //释放内存

    free(audiodev.i2sbuf);//释放内存

    free(audiodev.file);   //释放内存

    return res;

}

需要注意的时WAV文件数据低端在前，所以使用I2S发送时需要处理。