PZ-WM8978 MP3 模块开发手册

本手册将教大家如何在普中 T100&T200 开发板上使用 PZ-WM8978 MP3 模块。本章分为如下几部分内容：

1 PZ-WM8978 模块简介

2 模块使用说明

3 硬件设计

4 软件设计

5 实验现象

1 PZ-WM8978 模块简介

1.1 特性参数

PZ-WM8978 MP3 模块是深圳普中科技推出的一款全功能音频处理模块。它带有一个 HI-FI 级数字信号处理内核，支持增强 3D 硬件环绕音效，以及 5 频段的硬件均衡器，可以有效改善音质。并且具有一个可编程的陷波滤波器，用以去除屏幕开、切换等噪音。

PZ-WM8978 模块同样集成了对麦克风的支持，以及用于一个强悍的扬声器功放，可提供高达 900mW 的高质量音响效果扬声器功率。WM8978 进一步提升了耳机放大器输出功率，在推动 16 欧耳机的时候，每声道最大输出功率高达 40mW，可以连接市面上绝大多数适合随声听的高端 HI-FI 耳机。

WM8988 的主要特性有：

●I2S 接口，支持最高 192K,24bit 音频播放

●DAC 信噪比 98dB；ADC 信噪比 90dB

●支持无电容耳机驱动（提供 40mW@16Ω的输出能力）

●支持扬声器输出（提供 0.9W@8Ω的驱动能力）

●支持立体声差分输入/麦克风输入

●支持左右声道音量独立调节

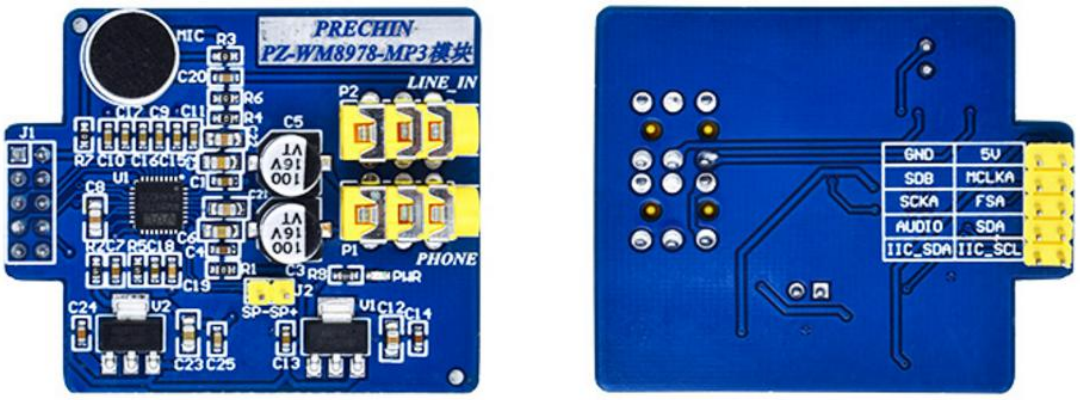
●支持 3D 效果，支持 5 路 EQ 调节

2 模块使用说明

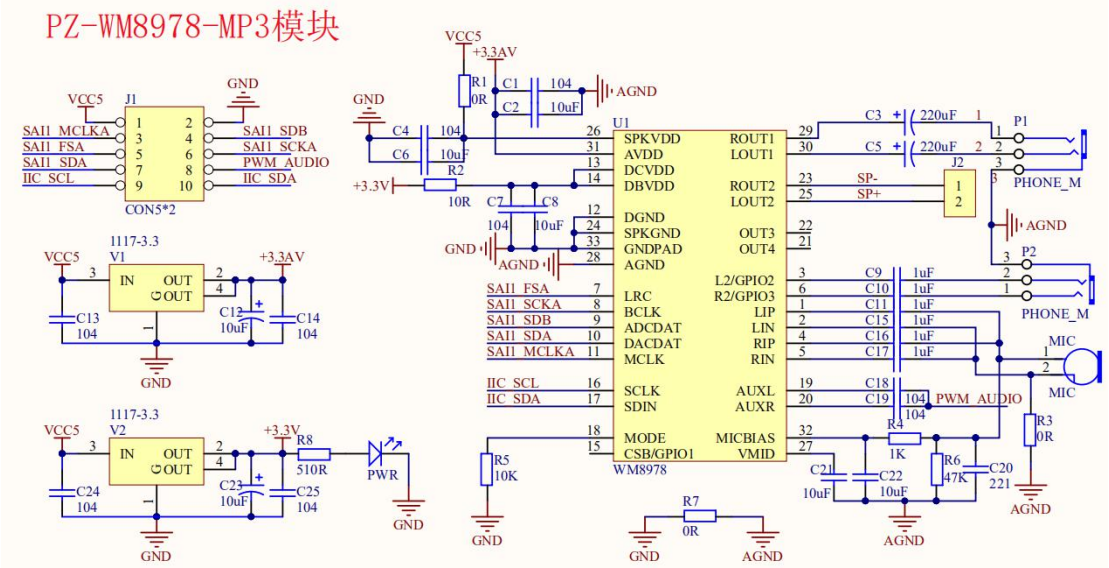
2.1 模块引脚说明

PZ-WM8979 MP3 模块通过 2*5 的排针（2.54mm 间距）同外部连接。模块可以使用杜邦线与普中 T100&T200 等开发板连接进行测试。对应开发板都提供有相应例程，用户可以直接在这些开发板上，对模块进行测试。

PZ-WM8979 MP3 模块外观如下图所示：



PZ-WM8979 MP3 模块原理图如下图所示：



从上图可以看出，通过 J1 排针的 VCC5 输入，经过 V1/V2 稳压芯片，转换为 3.3V 供 WM8978 芯片电源。为了让音质效果更好，这里使用了 2 路稳压，分别是 3.3AV 和 3.3V。3.3AV 给 WM8978 的模拟电源供电，3.3V 给 WM8978 数字电源供电。

MIC 咪头是直接连接在 WM8978 芯片上的，录音的时候，我们可以直接通过咪头实现声音采集。当不采用咪头拾音，而采用外部线路输入的时候， 插入麦克风 P2 即可使用。

P1 接口可以用来插耳机，J2 接口可以外接喇叭（1W@8Ω，需自备）。

PZ-WM8979 MP3 模块通过一个 2*5 的排针（J1）同外部电路连接，各引脚的详细描述如下图所示：

引脚名称	作用说明
5V	5V供电口
GND	地
MCLKA	是主时钟输出脚，固定输出频率为 256×fs，fs 即音频信号采样频率（fs）。
SDB	连接 I2S 的 ADCDAT 脚，是数据输出脚，用于发送数据。
FSA	连接 I2S 的 LRC 脚，用于切换左右声道的数据，它的频率等于音频信号采样率（fs）。
SCKA	连接 I2S 的 BLCK 脚，用作位时钟，是 I2S 主模式下的串行时钟输出以及从模式下的串行时钟输入
SDA	连接 I2S 的 DACDAT 脚，是数据输入脚，用于接收数据。
AUDIO	PWM软件解码
IIC_SCL	IIC时钟线
IIC_SDA	IIC数据线

2.2 WM8978 简介

WM8978 的控制通过 I2S 接口（即数字音频接口）同 MCU 进行音频数据传输（支持音频接收和发送），通过两线（MODE=0，即 IIC 接口）或三线（MODE=1）接口进行配置。WM8978 的 I2S 接口，由 4 个引脚组成：

1，ADCDAT:ADC 数据输出

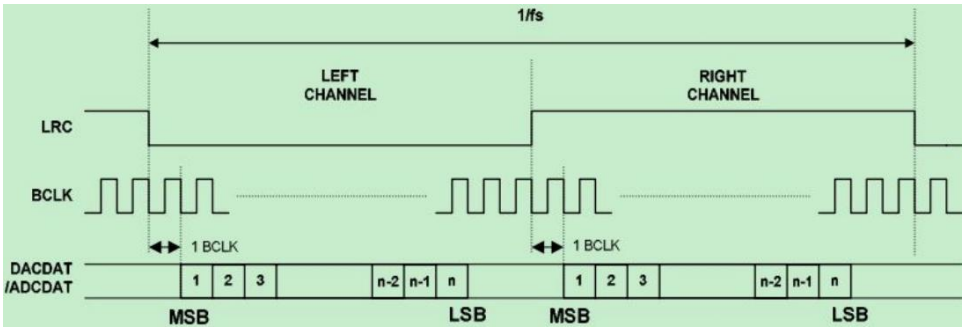
2，DACDAT：DAC 数据输入

3，LRC：数据左/右对齐时钟

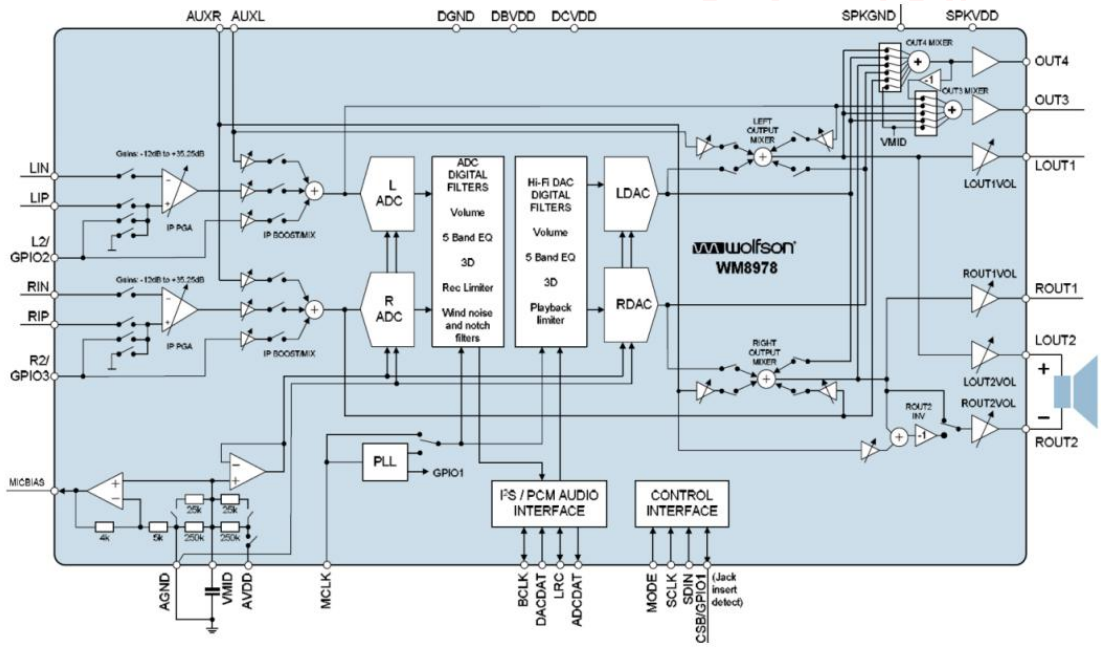
4，BCLK：位时钟，用于同步

WM8978 可作为 I2S 主机，输出 LRC 和 BLCK 时钟，不过我们一般使用 WM8978 作为从机，接收 LRC 和 BLCK。另外，WM8978 的 I2S 接口支持 5 中不同的音频数据模式：左（MSB）对齐标准、右（LSB）对齐标准、飞利浦（I2S）标准、DSP 模式 A 和 DSP 模式 B。本章，我们用飞利浦标准来传输 I2S 数据。

飞利浦（I2S）标准模式，数据在跟随 LRC 传输的 BCLK 的第二个上升沿时传输 MSB，其他位一直到 LSB 按顺序传输。传输依赖于字长、BCLK 频率和采样率，在每个采样的 LSB 和下一个采样的 MSB 之间都应该有未用的 BCLK 周期。飞利浦标准模式的 I2S 数据传输协议如下图所示：



图中，fs 即音频信号的采样率，比如 44.1Khz，因此可以知道，LRC 的频率就是音频信号的采样率。另外，WM8978 还需要一个 MCLK，本章我们采用 STM32 为其提供 MCLK 时钟，MCLK 的频率必须等于 256fs，也就是音频采样率的 256 倍。WM8978 的框图如下图所示：



从上图可以看出，WM8978 内部有很多的模拟开关，用来选择通道，同时还有很多调节器，用来设置增益和音量。

本章，我们通过 IIC 接口（MODE=0）连接 WM8978，不过 WM8978 的 IIC 接口比较特殊：1，只支持写，不支持读数据；2，寄存器长度为 7 位，数据长度为 9 位。3，寄存器字节的最低位用于传输数据的最高位（也就是 9 位数据的最高位，7 位寄存器的最低位）。WM8978 的 IIC 地址固定为：0X1A。关于 WM8978 的 IIC 详细介绍，请看其数据手册第 77 页。

2.3 WM8978 寄存器配置

这里我们简单介绍一下要正常使用 WM8978 来播放音乐，应该执行哪些配置。

1，寄存器 R0（00h），该寄存器用于控制 WM8978 的软复位，写任意值到该寄存器地址，即可实现软复位 WM8978。

2，寄存器 R1（01h），该寄存器主要要设置 BIASEN（bit3），该位设置为 1，模拟部分的放大器才会工作，才可以听到声音。

3，寄存器 R2（02h），该寄存器要设置 ROUT1EN(bit8)，LOUT1EN(bit7)和 SLEEP(bit6)等三个位，ROUT1EN 和 LOUT1EN，设置为 1，使能耳机输出，SLEEP 设置为 0，进入正常工作模式。

4，寄存器 R3（03h），该寄存器要设置 LOUT2EN(bit6)，ROUT2EN(bit5)，RMIXER(bit3)，LMIXER(bit2)，DACENR(bit1)和 DACENL(bit0)等 6 个位。LOUT2EN 和 ROUT2EN，设置为 1，使能喇叭输出；LMIXER 和 RMIXER 设置为 1，使能左右声道混合器；DACENL 和 DACENR 则是使能左右声道的 DAC 了，必须设置为 1。

5，寄存器 R4（04h），该寄存器要设置 WL(bit6:5)和 FMT(bit4:3)等 4 个位。WL(bit6:5)用于设置字长（即设置音频数据有效位数），00 表示 16 位音频，10 表示 24 位音频；FMT(bit4:3)用于设置 I2S 音频数据格式（模式），我们一般设置为 10，表示 I2S 格式，即飞利浦模式。

6，寄存器 R6（06h），该寄存器我们直接全部设置为 0 即可，设置 MCLK 和 BCLK 都来自外部，即由 STM32F4 提供。

7，寄存器 R10（0Ah），该寄存器我们要设置 SOFTMUTE(bit6)和 DACOSR128(bit3)等两个位，SOFTMUTE 设置为 0，关闭软件静音；DACOSR128 设置为 1，DAC 得到最好的 SNR。

8，寄存器 R43（2Bh），该寄存器我们只需要设置 INVROUT2 为 1 即可，反转 ROUT2 输出，更好的驱动喇叭。

9，寄存器 R49（31h），该寄存器我们要设置 SPKBOOST(bit2)和 TSDEN(bit1)这两个位。SPKBOOST 用于设置喇叭的增益，我们默认设置为 0 就好了（gain=-1），如想获得更大的声音，设置为 1（gain=+1.5）即可；TSDEN 用于

设置过热保护，设置为 1（开启）即可。

10，寄存器 R50（32h）和 R51（33h），这两个寄存器设置类似，一个用于设置左声道（R50），另外一个用于设置右声道（R51）。我们只需要设置这两个寄存器的最低位为 1 即可，将左右声道的 DAC 输出接入左右声道混合器里面，才能在耳机/喇叭听到音乐。

11，寄存器 R52（34h）和 R53（35h），这两个寄存器用于设置耳机音量，同样一个用于设置左声道（R52），另外一个用于设置右声道（R53）。这两个寄存器的最高位（HPVU）用于设置是否更新左右声道的音量，最低 6 位用于设置左右声道的音量，我们可以先设置好两个寄存器的音量值，最后设置其中一个寄存器最高位为 1，即可更新音量设置。

12，寄存器 R54（36h）和 R55（37h），这两个寄存器用于设置喇叭音量，同 R52，R53 设置一模一样，这里就不细说了。

以上，就是我们用 WM8978 播放音乐时的设置，按照以上所述，对各个寄存器进行相应的配置,即可使用 WM8978 正常播放音乐了。还有其他一些 3D 设置，EQ 设置等，我们这里就不再介绍了，大家参考 WM8978 的数据手册自行研究下即可。

2.4 WAV 简介

WAV 即 WAVE 文件，WAV 是计算机领域最常用的数字化声音文件格式之一，它是微软专门为 Windows 系统定义的波形文件格式（Waveform Audio），由于其扩展名为"*.wav"。它符合 RIFF(Resource Interchange File Format)文件规范，用于保存 Windows 平台的音频信息资源，被 Windows 平台及其应用程序所广泛支持，该格式也支持 MSADPCM，CCITT A LAW 等多种压缩运算法，支持多种音频数字，取样频率和声道，标准格式化的 WAV 文件和 CD 格式一样，也是 44.1K 的取样频率，16 位量化数字，因此在声音文件质量和 CD 相差无几！

WAV 一般采用线性 PCM（脉冲编码调制）编码，本章，我们也主要讨论 PCM 的播放，因为这个最简单。

WAV 是由若干个 Chunk 组成的。按照在文件中的出现位置包括：RIFF WAVE Chunk、Format Chunk、 Fact Chunk(可选)和 Data Chunk。每个 Chunk 由块标

识符、数据大小和数据三部分组成，如下图所示：

块的标志符（4BYTES）
数据大小 （4BYTES）
数据

其中块标识符由 4 个 ASCII 码构成，数据大小则标出紧跟其后的数据的长度（单位为字节），注意这个长度不包含块标识符和数据大小的长度，即不包含最前面的 8 个字节。所以实际 Chunk 的大小为数据大小加 8。

首先，我们来看看 RIFF 块（RIFF WAVE Chunk），该块以“RIFF”作为标示，紧跟 wav 文件大小（该大小是 wav 文件的总大小-8），然后数据段为“WAVE”，表示是 wav 文件。RIFF 块的 Chunk 结构如下：

```
1.  //RIFF 块
2.  typedef __packed struct
3.  {
4.      u32 ChunkID; //chunk id;这里固定为"RIFF",即 0X46464952
5.      u32 ChunkSize ; //集合大小;文件总大小-8
6.      u32 Format; //格式;WAVE,即 0X45564157
7.  }ChunkRIFF ;
```

接着，我们看看 Format 块（Format Chunk），该块以“fmt ”作为标示（注意有个空格！），一般情况下，该段的大小为 16 个字节，但是有些软件生成的 wav 格式，该部分可能有 18 个字节，含有 2 个字节的附加信息。Format 块的 Chunk 结构如下：

```
1.  //fmt 块
2.  typedef __packed struct
3.  {
4.      u32 ChunkID; //chunk id;这里固定为"fmt ",即 0X20746D66
5.      u32 ChunkSize ; //子集合大小(不包括 ID 和 Size);这里为:20.
6.      u16 AudioFormat; //音频格式;0X10,表示线性 PCM;0X11 表示 IMA ADPCM
7.      u16 NumOfChannels; //通道数量;1,表示单声道;2,表示双声道;
8.      u32 SampleRate; //采样率;0X1F40,表示 8Khz
9.      u32 ByteRate; //字节速率;
10.     u16 BlockAlign; //块对齐(字节);
11.     u16 BitsPerSample; //单个采样数据大小;4 位 ADPCM,设置为 4
12. }ChunkFMT;
```

接下来，我们再看看 Fact 块（Fact Chunk），该块为可选块，以“fact”作为标示，不是每个 WAV 文件都有，在非 PCM 格式的文件中，一般会在 Format 结构后面加入一个 Fact 块，该块 Chunk 结构如下：

```
//fact 块
typedef __packed struct
{
    u32 ChunkID; //chunk id;这里固定为"fact",即 0X74636166;
    u32 ChunkSize ; //子集合大小(不包括 ID 和 Size);这里为:4.
    u32 DataFactSize; //数据转换为 PCM 格式后的大小
}ChunkFACT;
```

DataFactSize 是这个 Chunk 中最重要的数据，如果这是某种压缩格式的声音文件，那么从这里就可以知道他解压缩后的大小。对于解压时的计算会有很大的好处！不过本章我们使用的是 PCM 格式，所以不存在这个块。

最后，我们来看看数据块（Data Chunk），该块是真正保存 wav 数据的地方，以“data”’作为该 Chunk 的标示，然后是数据的大小。数据块的 Chunk 结构如下：

```
//data 块
typedef __packed struct
{
    u32 ChunkID; //chunk id;这里固定为"data",即 0X61746164
    u32 ChunkSize ; //子集合大小(不包括 ID 和 Size);文件大小-60.
}ChunkDATA;
```

ChunkSize 后紧接着就是 wav 数据。根据 Format Chunk 中的声道数以及采样 bit 数，wav 数据的 bit 位置可以分成如表 48.1.1.1 所示的几种形式：

单声道	取样 1	取样 2	取样 3	取样 4	取样 5	取样 6
8 位量化	声道 0	声道 0	声道 0	声道 0	声道 0	声道 0
双声道	取样 1		取样 2		取样 3	
8 位量化	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)	声道 0(左)	声道 1(右)
单声道	取样 1		取样 2		取样 3	
16 位量化	声道 0 (低字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (高字节)
双声道	取样 1				取样 2	
16 位量化	声道 0 (低字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (高字节)	声道 0 (低字节)	声道 0 (高字节)
单声道	取样 1			取样 2		
24 位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)
双声道	取样 1					
24 位量化	声道 0 (低字节)	声道 0 (中字节)	声道 0 (高字节)	声道 1 (低字节)	声道 1 (中字节)	声道 1 (高字节)

本章，我们播放的音频支持：16 位和 24 位，立体声，所以每个取样为 4/6 个字节，低字节在前，高字节在后。在得到这些 wav 数据以后，通过 I2S 丢给

WM8978，就可以欣赏音乐了。

2.5 I2S 简介

I2S(Inter IC Sound)总线，又称集成电路内置音频总线，是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专责于音频设备之间的数据传输，广泛应用于各种多媒体系统。它采用了沿独立的导线传输时钟与数据信号的设计，通过将数据和时钟信号分离，避免了因时差诱发的失真，为用户节省了购买抵抗音频抖动的专业设备的费用。

STM32F4 自带了 2 个全双工 I2S 接口，其特点包括：

●支持全双工/半双工通信

●主持主/从模式设置

●8 位可编程线性预分频器，可实现精确的音频采样频率(8~192Khz)

●支持 16 位/24 位/32 位数据格式

●数据包帧固定为 16 位（仅 16 位数据帧）或 32 位（可容纳 16/24/32 位数据帧）

●可编程时钟极性

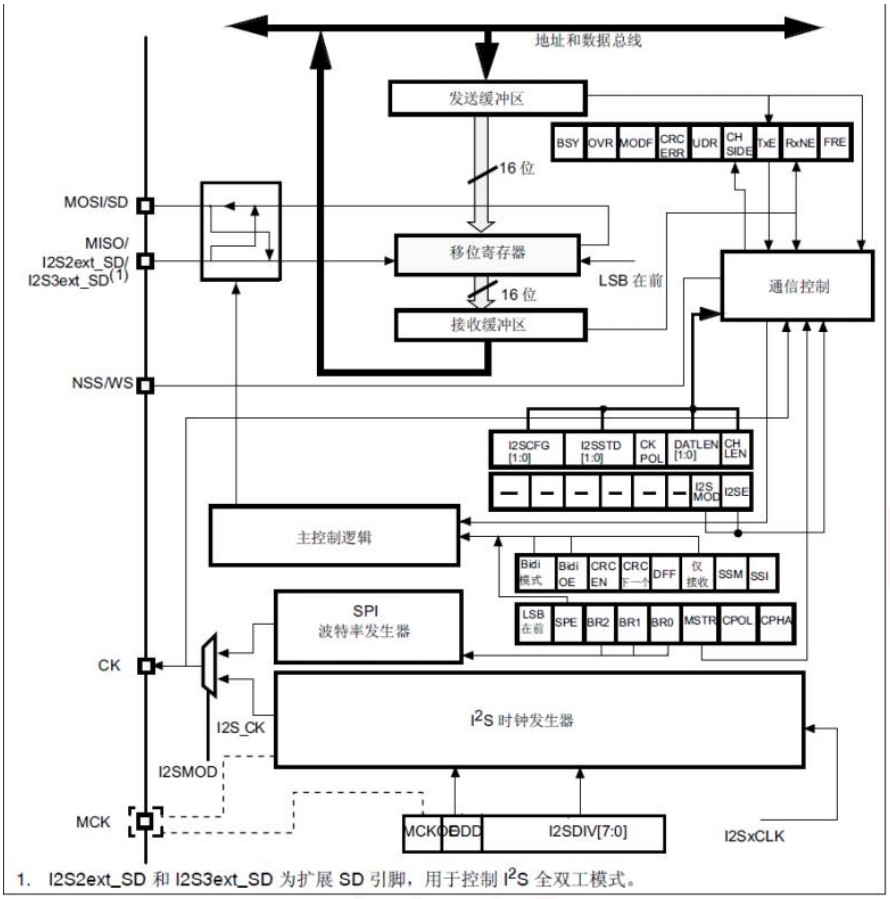
●支持 MSB 对齐（左对齐）、LSB 对齐（右对齐）、飞利浦标准和 PCM 标准等 I2S 协议

●支持 DMA 数据传输（16 位宽）

●数据方向固定位 MSB 在前

●支持主时钟输出（固定为 256*fs，fs 即音频采样率）

STM32F4 的 I2S 框图如下图所示：



STM32F4 的 I2S 是与 SPI 部分共用的，通过设置 SPI_I2SCFGR 寄存器的 I2SMOD 位即可开启 I2S 功能，I2S 接口使用了几乎与 SPI 相同的引脚、标志和中断。I2S 用到的信号有：

1，SD：串行数据（映射到 MOSI 引脚），用于发送或接收两个时分复用的数据通道上的数据（仅半双工模式）。

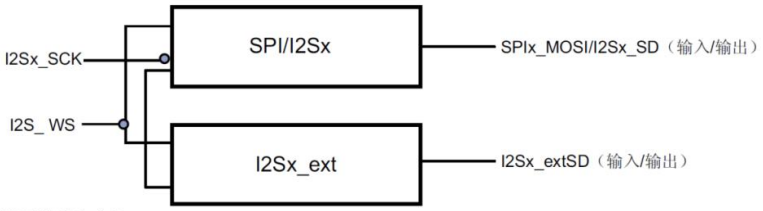
2，WS：字选择（映射到 NSS 引脚），即帧时钟，用于切换左右声道的数据。WS 频率等于音频信号采样率（fs）。

3，CK：串行时钟（映射到 SCK 引脚），即位时钟，是主模式下的串行时钟输出以及从模式下的串行时钟输入。CK 频率=WS 频率（fs）*2*16（16 位宽），如果是 32 位宽，则是：CK 频率=WS 频率（fs）*2*32（32 位宽）

4，I2S2ext_SD 和 I2S3ext_SD：用于控制 I2S 全双工模式的附加引脚（映射到 MISO 引脚）。

5，MCK：即主时钟输出，当 I2S 配置为主模式（并且 SPI_I2SPR 寄存器中的 MCKOE 位置 1）时，使用此时钟，该时钟输出频率 256×fs，fs 即音频信号采样频率（fs）。

为支持 I2S 全双工模式，除了 I2S2 和 I2S3，还可以使用两个额外的 I2S，它们称为扩展 I2S（I2S2_ext、I2S3_ext），如下图所示：



1. 其中 X 可以是 2 或 3。

因此，第一个 I2S 全双工接口基于 I2S2 和 I2S2_ext，第二个基于 I2S3 和 I2S3_ext。注意：I2S2_ext 和 I2S3_ext 仅用于全双工模式。

I2Sx 可以在主模式下工作。因此：

1，只有 I2Sx 可在半双工模式下输出 SCK 和 WS

2，只有 I2Sx 可在全双工模式下向 I2S2_ext 和 I2S3_ext 提供 SCK 和 WS。

扩展 I2S (I2Sx_ext)只能用于全双工模式。I2Sx_ext 始终在从模式下工作。I2Sx 和 I2Sx_ext 均可用于发送和接收。

STM32F4 的 I2S 支持 4 种数据和帧格式组合，分别是：1，将 16 位数据封装在 16 位帧中； 2，将 16 位数据封装在 32 位帧中；3，将 24 位数据封装在 32 位帧中；4，将 32 位数据封装在 32 位帧中。

将 16 位数据封装在 32 位帧中时，前 16 位(MSB)为有效位，16 位 LSB 被强制清零，无需任何软件操作或 DMA 请求（只需一个读/写操作）。如果应用程序首选 DMA，则 24 位和 32 位数据帧需要对 SPI_DR 执行两次 CPU 读取或写入操作，或者需要两次 DMA 操作。24 位的数据帧，硬件会将 8 位非有效位扩展到带有 0 位的 32 位。

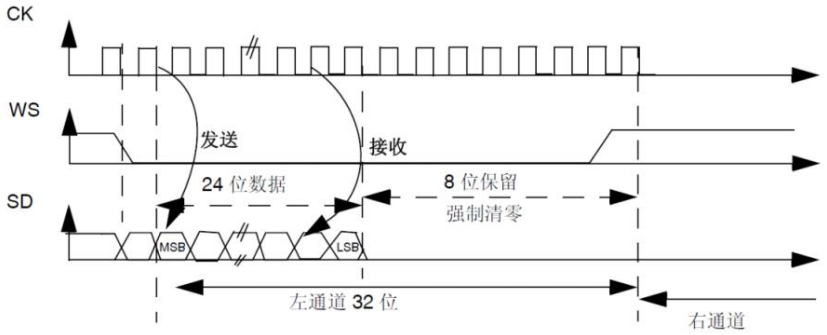
对于所有数据格式和通信标准而言，始终会先发送最高有效位（MSB 优先）。

STM32F4 的 I2S 支持：MSB 对齐（左对齐）标准、LSB 对齐（右对齐）标准、飞利浦标准和 PCM 标准等 4 种音频标准，本章我们用飞利浦标准，仅针对该标准进行介绍，其他的请大家参考《STM32F4xx 中文参考手册》第 27.4 节。

I2S 飞利浦标准，使用 WS 信号来指示当前正在发送的数据所属的通道。该信号从当前通道数据的第一个位(MSB)之前的一个时钟开始有效。发送方在时钟信号(CK)的下降沿改变数据，接收方在上升沿读取数据。WS 信号也在 CK 的下

降沿变化。这和我们前面小节介绍的是一样的。

本章我们使用 16 位/24 位数据格式，16 位时采用扩展帧格式（即将 16 位数据封装在 32 位帧中），以 24 位帧为例，I2S 波形（飞利浦标准）如下图所示：

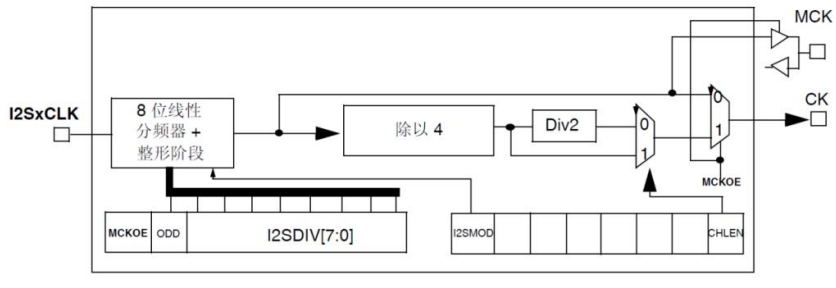


这个图和图 48.1.2.1 是一样的时序，在 24 位模式下数据传输，需要对 SPI_DR 执行两次读取或写入操作。比如我们要发送 0X8EAA33 这个数据，就要分两次写入 SPI_DR，第一次写入：0X8EAA，第二次写入 0X33xx（xx 可以为任意数值），这样就把 0X8EAA33 发送出去了。

顺便说一下 SD 卡读取到的 24 位 WAV 数据流，是低字节在前，高字节在后的，比如，我们读到一个声道的数据（24bit），存储在 buf[3]里面，那么要通过 SPI_DR 发送这个 24 位数据，过程如下：

```
SPI_DR=((u16)buf[2]<<8)+buf[1];
SPI_DR=(u16)buf[0]<<8;
```

这样，第一次发送高 16 为数据，第二次发送低 8 位数据，完成一次 24bit 数据的发送。接下来，我们介绍下 STM32F4 的 I2S 时钟发生器，其架构如下图所示：



1. 其中，X 可以是 2 或 3。

图中 I2SxCLK 可以来自 PLLI2S 输出（通过 R 系数分频）或者来自外部时

钟（I2S_CKIN 引脚），一般我们使用前者作为 I2SxCLK 输入时钟。

一般我们需要根据音频采样率（fs，即 CK 的频率）来计算各个分频器的值，常用的音频采样率有：22.05Khz、44.1Khz、48Khz、96Khz、196Khz 等。

根据是否使能 MCK 输出，fs 频率的计算公式有 2 种情况。不过，本章只考虑 MCK 输出使能时的情况，当 MCK 输出使能时，fs 频率计算公式如下：

```
fs=I2SxCLK/[256*(2*I2SDIV+ODD)]
```

其中：I2SxCLK=(HSE/pllm)*PLLI2SN/PLLI2SR。HSE 我们是 8Mhz，而 pllm 在系统时钟初始化就确定了，是 8，这样结合以上 2 式，可得计算公式如下：

```
fs= (1000*PLLI2SN/PLLI2SR )/[256*(2*I2SDIV+ODD)]
```

fs 单位是：Khz。其中：PLL2SN 取值范围：192~432；PLLI2SR 取值范围：2~7；I2SDIV 取值范围：2~255；ODD 取值范围：0/1。根据以上约束条件，我们便可以根据 fs 来设置各个系数的值了，不过很多时候，并不能取得和 fs 一模一样的频率，只能近似等于 fs，比如 44.1Khz 采样率，我们设置 PLL2SN=271，PLL2SR=2，I2SDIV=6，ODD=0，得到 fs=44.108073Khz，误差为：0.0183%。晶振频率决定了有时无法通过分频得到我们所要的 fs,所以,某些 fs 如果要实现 0 误差，大家必须得选用外部时钟才可以。

如果要通过程序去计算这些系数的值，是比较麻烦的，所以，我们事先计算好常用 fs 对应的系数值，建立一个表，这样，用的时候，只需要查表取值就可以了，大大简化了代码，常用 fs 对应系数表如下：

```
1.  //表格式:采样率/10,PLLI2SN,PLLI2SR,I2SDIV,ODD
2.  const u16 I2S_PSC_TBL[][5]=
3.  {
4.      {800 ,256,5,12,1}, //8Khz 采样率
5.      {1102,429,4,19,0}, //11.025Khz 采样率
6.      {1600,213,2,13,0}, //16Khz 采样率
7.      {2205,429,4, 9,1}, //22.05Khz 采样率
8.      {3200,213,2, 6,1}, //32Khz 采样率
9.      {4410,271,2, 6,0}, //44.1Khz 采样率
10.     {4800,258,3, 3,1}, //48Khz 采样率
11.     {8820,316,2, 3,1}, //88.2Khz 采样率
12.     {9600,344,2, 3,1}, //96Khz 采样率
13.     {17640,361,2,2,0}, //176.4Khz 采样率
14.     {19200,393,2,2,0}, //192Khz 采样率
15. };
```

有了上面的 fs-系数对应表，我们可以很方便的完成 I2S 的时钟配置。

接下来，我们看看本章需要用到的一些相关寄存器。

首先，是 SPI_I2S 配置寄存器：SPI_I2SCFGR，该寄存器各位描述如下图所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				I2SMOD	I2SE	I2SCFG		PCMSYNC	Reserved	I2SSTD		CKPOL	DATLEN		CHLEN
				rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw

I2SMOD 位，设置为 1，选择 I2S 模式，注意，必须在 I2S/SPI 禁止的时候，设置该位。

I2SE 位，设置为 1，使能 I2S 外设，该位必须在 I2SMOD 位设置之后再设置。

I2SCFG[1:0]位， 这两个位用于配置 I2S 模式，设置为 10，选择主模式（发送）。

I2SSTD[1:0]位，这两个位用于选择 I2S 标准，设置为 00，选择飞利浦模式。

CKPOL 位，用于设置空闲时时钟电平，设置为 0，空闲时时钟低电平。

DATLEN[1:0]位，用于设置数据长度，00，表示 16 位数据；01 表示 24 位数据。

CHLEN 位，用于设置通道长度，即帧长度，0，表示 16 位；1，表示 32 位。

第二个是 SPI_I2S 预分配器寄存器：SPI_I2SPR，该寄存器各位描述如下图所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						MCKOE	ODD	I2SDIV							
						rw	rw	rw							

位 15:10 保留，必须保持复位值。

位 9 **MCKOE**：主时钟输出使能 (Master clock output enable)

0：禁止主时钟输出

1：使能主时钟输出

注意：应在 I²S 禁止时配置此位。只有在 I²S 为主模式时，才会使用此位。不适用于 SPI 模式。

位 8 **ODD**：预分频器的奇数因子 (Odd factor for the prescaler)

0：实际分频值为 = I2SDIV *2

1：实际分频值为 = (I2SDIV * 2)+1

注意：应在 I²S 禁止时配置此位。只有在 I²S 为主模式时，才会使用此位。

位 7:0 **I2SDIV**：I2S 线性预分频器 (I2S Linear prescaler)

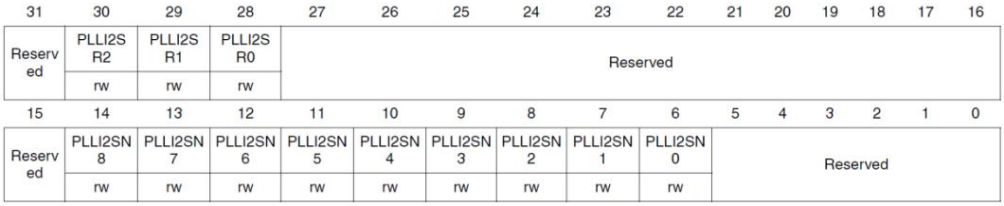
I2SDIV [7:0] = 0 或 I2SDIV [7:0] = 1 为禁用值。

注意：应在 I²S 禁止时配置这些位。只有在 I²S 为主模式时，才会使用此位。

本章我们设置 MCKOE 为 1，开启 MCK 输出，ODD 和 I2SDIV 则根据不同的

fs，查表进行设置。

第三个是 PLLI2S 配置寄存器：RCC_PLLI2SCFGR，该寄存器各位描述如下图所示：



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	PLLI2SR2	PLLI2SR1	PLLI2SR0	Reserved											
	rw	rw	rw												

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	PLLI2SN8	PLLI2SN7	PLLI2SN6	PLLI2SN5	PLLI2SN4	PLLI2SN3	PLLI2SN2	PLLI2SN1	PLLI2SN0	Reserved					
	rw	rw	rw	rw	rw	rw	rw	rw	rw						

该寄存器用于配置 PLLI2SR 和 PLLI2SN 两个系数，PLLI2SR 的取值范围是：2~7，PLLI2SN 的取值范围是：192~432。同样，这两个也是根据 fs 的值来设置的。

此外，还要用到 SPI_CR2 寄存器的 bit1 位，设置 I2S TX DMA 数据传输，SPI_DR 寄存器用于传输数据，本章用 DMA 来传输，所以直接设置 DMA 的外设地址位 SPI_DR 即可。

2.6 WM8978 配置步骤

最后，我们看看要通过 STM32F4 的 I2S，驱动 WM8978 播放音乐的简要步骤。这里需要说明一下，I2S 相关的库函数申明和定义跟 SPI 是同文件的，在 stm32f4xx_spi.c 以及头文件 stm32f4xx_spi.h 中。具体步骤如下：

（1）初始化 WM8978

这个过程就是在前面小节 wm8978 那十几个寄存器的配置，包括软复位、DAC 设置、输出设置和音量设置等。在我们实验工程中是在文件 wm8978.c 中，大家可以打开实验工程参考。

（2）初始化 I2S

此过程主要设置 SPI_I2SCFGR 寄存器，设置 I2S 模式、I2S 标准、时钟空闲电平和数据帧长等，最后开启 I2S TX DMA，使能 I2S 外设。

在库函数中初始化 I2S 调用的函数为：

```
1.    void I2S_Init(SPI_TypeDef* SPIx, I2S_InitTypeDef* I2S_InitStruct);
```

第一个参数比较好理解，我们来着重看下第二个参数，这里我们主要讲解结构体 I2S_InitTypeDef 各个成员变量的含义。结构体 I2S_InitTypeDef 的定义为：

```
typedef struct
{
    uint16_t I2S_Mode;
    uint16_t I2S_Standard;
    uint16_t I2S_DataFormat;
    uint16_t I2S_MCLKOutput;
    uint32_t I2S_AudioFreq;
    uint16_t I2S_CPOL;
}I2S_InitTypeDef;
```

第一个参数用来设置 I2S 的模式，也就是设置 SPI_I2SCFGR 寄存器的 I2SCFG 相关位。可以配置为主模式发送 I2S_Mode_MasterTx，主模式接受 I2S_Mode_MasterRx，从模式发送 I2S_Mode_SlaveTx 以及从模式接受 I2S_Mode_SlaveRx 四种模式。

第二个参数 I2S_Standard 用来设置 I2S 标准，这个前面已经讲解过。可以设置为：飞利浦标准 I2S_Standard_Phillips,MSB 对齐标准 I2S_Standard_MSB，LSB 对齐标准 I2S_Standard_LSB 以及 PCM 标准 I2S_Standard_PCMShort。

第三个参数 I2S_DataFormat 用来设置 I2S 的数据通信格式。这里实际包含设置 SPI_I2SCFGR 寄存器的 HCLEN 位（通道长度）以及 DATLEN 位（传输的数据长度）。当我们设置为 16 位标准格式 I2S_DataFormat_16b 的时候，实际上传输的数据长度为 16 位，通道长度为 16 位。当我们设置为其他值的时候，通道长度都为 32 位。

第四个参数 I2S_MCLKOutput 用来设置是否使能主时钟输出。我们实验会使能主时钟输出。

第五个参数 I2S_AudioFreq 用来设置 I2S 频率。实际根据输入的频率值，会来计算 SPI 预分频寄存器 SPI_I2SPR 的预分频奇数因子以及 I2S 线性预分频器的值。这里支持 10 中频率：

```
#define I2S_AudioFreq_192k ((uint32_t)192000)
#define I2S_AudioFreq_96k ((uint32_t)96000)
#define I2S_AudioFreq_48k ((uint32_t)48000)
#define I2S_AudioFreq_44k ((uint32_t)44100)
#define I2S_AudioFreq_32k ((uint32_t)32000)
#define I2S_AudioFreq_22k ((uint32_t)22050)
#define I2S_AudioFreq_16k ((uint32_t)16000)
#define I2S_AudioFreq_11k ((uint32_t)11025)
```

```
#define I2S_AudioFreq_8k ((uint32_t)8000)
#define I2S_AudioFreq_Default ((uint32_t)2)
```

第六个参数 I2S_CPOL 用来设置空闲状态时钟电平，这个比较好理解。取值为高电平 I2S_CPOL_High 以及低电平 I2S_CPOL_Low。

（3）解析 WAV 文件，获取音频信号采样率和位数并设置 I2S 时钟分频器

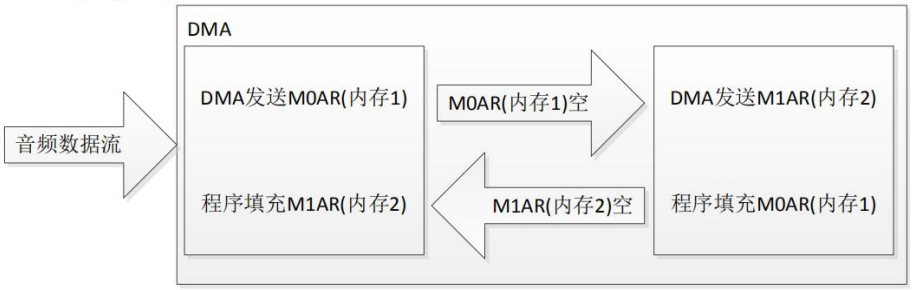
这里，要先解析 WAV 文件，取得音频信号的采样率（fs）和位数（16 位或 32 位），根据这两个参数，来设置 I2S 的时钟分频，这里我们用前面介绍的查表法来设置即可。这是我们单独写了一个设置频率的函数为 I2S2_SampleRate_Set，我们后面程序章节会讲解。

（4）设置 DMA

I2S 播放音频的时候，一般都是通过 DMA 来传输数据的，所以必须配置 DMA，本章我们用 I2S2，其 TX 是使用的 DMA1 数据流 4 的通道 0 来传输的。并且，STM32F4 的 DMA 具有双缓冲机制，这样可以提高效率，大大方便了我们的数据传输，本章将 DMA1 数据流 4 设置为：双缓冲循环模式，外设和存储器都是 16 位宽，并开启 DMA 传输完成中断（方便填充数据）。DMA 具体配置过程请参考我们工程代码，前面 DMA 实验我们已经讲解过 DMA 相关配置过程。

（5）编写 DMA 传输完成中断服务函数

为了方便填充音频数据，我们使用 DMA 传输完成中断，每当一个缓冲数据发送完后，硬件自动切换为下一个缓冲，同时进入中断服务函数，填充数据到发送完的这个缓冲。过程如下图所示：



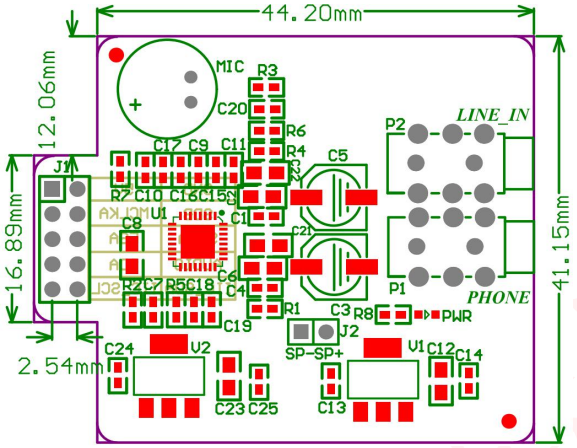
（6）开启 DMA 传输，填充数据

最后，我们就只需要开启 DMA 传输，然后及时填充 WAV 数据到 DMA 的两个缓存区即可。此时，就可以在 WM8978 的耳机和喇叭通道听到所播放音乐了。操作方法为：

```
DMA_Cmd(DMA1_Stream4,ENABLE);//开启 DMA TX 传输,开始播放
```

2.7 结构尺寸

PZ-WM8978 模块的尺寸结构如下图所示：



3 硬件设计

3.1 硬件准备

本实验所需要的硬件资源如下：

①T100&T200 开发板 1 个

②PZ-WM8978 模块 1 个

③SD 卡 1 个（预先在 SD 卡根目录下新建 MUSIC 文件夹，并存放一些 wav 歌曲在 MUSIC 文件夹下。用户可直接拷贝资料“\3--SD 卡根目录文件”下文件夹）

④耳机线 1 条

⑤USB 线一条（用于供电和模块与电脑串口调试助手通信）

3.2 模块与开发板连接

下面我们介绍下 PZ-WM8978 模块与 T100&T200 开发板的连接关系，如下图所示：（PZ-WM8978 模块-->STM32 IO）

```
FSA-->PB12
SCKA-->PB13
SDB-->PC2
```

```
SDA-->PC3
MCLKA-->PC6
IIC_SCL-->PB8
IIC_SDA-->PB9
5V-->5V
GND-->GND
```

4 软件设计

4.1 音乐播放器实验

打开本章实验工程目录可以看到，我们在工程根目录文件夹下新建了一个 AudioDecod 文件夹。在 AudioDecod 文件夹里面新建了 audioplay.c 和 audioplay.h 两个文件和一个 wav 文件夹。在 wav 文件夹里面新建了 wavplay.c 和 wavplay.h 两个文件。同时，我们把相关的源文件引入工程相应分组，同时将 AudioDecod 和 wav 文件夹加入头文件包含路径。然后，我们在 APP 文件夹下新建了 wm8978 和 i2s 两个文件夹，在 wm8978 文件夹里面新建了 wm8978.c 和 wm8978.h 两个文件，在 i2s 文件夹里面新建了 i2s.c 和 i2s.h 两个文件。最后将 wm8978.c 和 i2s.c 添加到工程 APP 组下。同时相应的头文件加入到 PATH 中。

本章代码比较多，我们就不全部贴出来给大家介绍了，这里仅挑一些重点函数给大家介绍下。

4.1.1 i2s.c

首先是 i2s.c 里面，重点函数代码如下：

```
1.  //I2S2 初始化
2.
3.  //参数 I2S_Standard:  @ref SPI_I2S_Standard  I2S 标准,
4.  //I2S_Standard_Phillips,飞利浦标准;
5.  //I2S_Standard_MSB,MSB 对齐标准(右对齐);
6.  //I2S_Standard_LSB,LSB 对齐标准(左对齐);
```

```
7.  //I2S_Standard_PCMShort,I2S_Standard_PCMLong:PCM 标准
8.  // 参 数 I2S_Mode:  @ref  SPI_I2S_Mode  I2S_Mode_SlaveTx: 从 机 发 送 ;I2S_Mode_SlaveRx: 从 机 接 收;I2S_Mode_MasterTx:主机发送;I2S_Mode_MasterRx:主机接收;
9.  // 参 数 I2S_Clock_Polarity  @ref  SPI_I2S_Clock_Polarity:  I2S_CPOL_Low, 时 钟 低 电 平 有 效;I2S_CPOL_High,时钟高电平有效
10. // 参 数 I2S_DataFormat :  @ref  SPI_I2S_Data_Format  : 数 据 长 度 ,I2S_DataFormat_16b,16 位 标 准  ;I2S_DataFormat_16bextended,16  位  扩  展  (frame=32bit);I2S_DataFormat_24b,24 位;I2S_DataFormat_32b,32 位.
11. void I2S2_Init(u16 I2S_Standard,u16 I2S_Mode,u16 I2S_Clock_Polarity,u16 I2S_DataFormat)
12. {
13.     I2S_InitTypeDef I2S_InitStructure;
14.
15.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);//使能 SPI2 时钟
16.
17.     RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI2,ENABLE); //复位 SPI2
18.     RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI2,DISABLE);//结束复位
19.
20.     I2S_InitStructure.I2S_Mode=I2S_Mode;//IIS 模式
21.     I2S_InitStructure.I2S_Standard=I2S_Standard;//IIS 标准
22.     I2S_InitStructure.I2S_DataFormat=I2S_DataFormat;//IIS 数据长度
23.     I2S_InitStructure.I2S_MCLKOutput=I2S_MCLKOutput_Disable;//主时钟输出禁止
24.     I2S_InitStructure.I2S_AudioFreq=I2S_AudioFreq_Default;//IIS 频率设置
25.     I2S_InitStructure.I2S_CPOL=I2S_Clock_Polarity;//空闲状态时钟电平
26.     I2S_Init(SPI2,&I2S_InitStructure);//初始化 IIS
27.
28.
29.     SPI_I2S_DMACmd(SPI2,SPI_I2S_DMAReq_Tx,ENABLE);//SPI2 TX DMA 请求使能.
30.     I2S_Cmd(SPI2,ENABLE);//SPI2 I2S EN 使能.
31. }
32. //采样率计算公式:Fs=I2SxCLK/[256*(2*I2SDIV+ODD)]
33. //I2SxCLK=(HSE/pllm)*PLLI2SN/PLLI2SR
34. //一般 HSE=8Mhz
35. //pllm:在 Sys_Clock_Set 设置的时候确定，一般是 8
36. //PLLI2SN:一般是 192~432
37. //PLLI2SR:2~7
38. //I2SDIV:2~255
39. //ODD:0/1
40. //I2S 分频系数表@pllm=8,HSE=8Mhz,即 vco 输入频率为 1Mhz
41. //表格式:采样率/10,PLLI2SN,PLLI2SR,I2SDIV,ODD
42. const u16 I2S_PSC_TBL[][5]=
43. {
44.     {800 ,256,5,12,1},      //8Khz 采样率
45.     {1102,429,4,19,0},      //11.025Khz 采样率
46.     {1600,213,2,13,0},      //16Khz 采样率
```

```
47.         {2205,429,4, 9,1},      //22.05Khz 采样率
48.         {3200,213,2, 6,1},      //32Khz 采样率
49.         {4410,271,2, 6,0},      //44.1Khz 采样率
50.         {4800,258,3, 3,1},      //48Khz 采样率
51.         {8820,316,2, 3,1},      //88.2Khz 采样率
52.         {9600,344,2, 3,1},      //96Khz 采样率
53.         {17640,361,2,2,0},      //176.4Khz 采样率
54.         {19200,393,2,2,0},      //192Khz 采样率
55.     };
56.     //设置 IIS 的采样率(@MCKEN)
57.     //samplerate:采样率,单位:Hz
58.     //返回值:0,设置成功;1,无法设置.
59.     u8 I2S2_SampleRate_Set(u32 samplerate)
60.     {
61.         u8 i=0;
62.         u32 tempreg=0;
63.         samplerate/=10;//缩小 10 倍
64.
65.         for(i=0;i<(sizeof(I2S_PSC_TBL)/10);i++)//看看改采样率是否可以支持
66.         {
67.             if(samplerate==I2S_PSC_TBL[i][0])break;
68.         }
69.
70.         RCC_PLLI2SCmd(DISABLE);//先关闭 PLLI2S
71.         if(i==(sizeof(I2S_PSC_TBL)/10))return 1;//搜遍了也找不到
72.           RCC_PLLI2SConfig((u32)I2S_PSC_TBL[i][1],(u32)I2S_PSC_TBL[i][2]);// 设 置 I2SxCLK 的 频 率
    (x=2)  设置 PLLI2SN PLLI2SR
73.
74.         RCC->CR|=1<<26;                     //开启 I2S 时钟
75.         while((RCC->CR&1<<27)==0);      //等待 I2S 时钟开启成功.
76.         tempreg=I2S_PSC_TBL[i][3]<<0; //设置 I2SDIV
77.         tempreg|=I2S_PSC_TBL[i][4]<<8;     //设置 ODD 位
78.         tempreg|=1<<9;                      //使能 MCKOE 位,输出 MCK
79.         SPI2->I2SPR=tempreg;          //设置 I2SPR 寄存器
80.         return 0;
81.     }
82.     //I2S2 TX DMA 配置
83.     //设置为双缓冲模式,并开启 DMA 传输完成中断
84.     //buf0:M0AR 地址.
85.     //buf1:M1AR 地址.
86.     //num:每次传输数据量
87.     void I2S2_TX_DMA_Init(u8* buf0,u8 *buf1,u16 num)
88.     {
89.         NVIC_InitTypeDef   NVIC_InitStructure;
```

```
    DMA_InitTypeDef  DMA_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,ENABLE);//DMA1 时钟使能

    DMA_DeInit(DMA1_Stream4);
    while (DMA_GetCmdStatus(DMA1_Stream4) != DISABLE){}//等待 DMA1_Stream1 可配置

    /* 配置 DMA Stream */

    DMA_InitStructure.DMA_Channel = DMA_Channel_0;  //通道 0 SPI2_TX 通道
    DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)&SPI2->DR;//外设地址为:(u32)&SPI2->DR
    DMA_InitStructure.DMA_Memory0BaseAddr = (u32)buf0;//DMA 存储器 0 地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToPeripheral;//存储器到外设模式
    DMA_InitStructure.DMA_BufferSize = num;//数据传输量
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量模式
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//存储器增量模式
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;//外设数据长度:16 位
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;//存储器数据长度： 16 位
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;//高优先级
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable; //不使用 FIFO 模式
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_1QuarterFull;
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;//外设突发单次传输
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;//存储器突发单次传输
    DMA_Init(DMA1_Stream4, &DMA_InitStructure);//初始化 DMA Stream

    DMA_DoubleBufferModeConfig(DMA1_Stream4,(u32)buf1,DMA_Memory_0);//双缓冲模式配置

    DMA_DoubleBufferModeCmd(DMA1_Stream4,ENABLE);//双缓冲模式开启

    DMA_ITConfig(DMA1_Stream4,DMA_IT_TC,ENABLE);//开启传输完成中断

    NVIC_InitStructure.NVIC_IRQChannel = DMA1_Stream4_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x00;//抢占优先级 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x00;//子优先级 0
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//使能外部中断通道
    NVIC_Init(&NVIC_InitStructure);//配置

}
//I2S DMA 回调函数指针
void (*i2s_tx_callback)(void);  //TX 回调函数
```

```
//DMA1_Stream4 中断服务函数
void DMA1_Stream4_IRQHandler(void)
{
    if(DMA_GetITStatus(DMA1_Stream4,DMA_IT_TCIF4)==SET)////DMA1_Stream4,传输完成标志
    {
        DMA_ClearITPendingBit(DMA1_Stream4,DMA_IT_TCIF4);
        i2s_tx_callback();  //执行回调函数,读取数据等操作在这里面处理
    }
}
```

其中，I2S2_Init 完成 I2S2 的初始化，通过 4 个参数设置 I2S2 的详细配置信息。另外一个函数：I2S2_SampleRate_Set，则是用前面介绍的查表法，根据音频采样率来设置 I2S 的时钟部分。函数 I2S2_TX_DMA_Init，用于设置 I2S2 的 DMA 发送，使用双缓冲循环模式，发送数据给 WM8978，并开启了发送完成中断。而 DMA1_Stream4_IRQHandler 函数，则是 DMA1 数据流 4 发送完成中断的服务函数，该函数调用 i2s_tx_callback 函数（函数指针，使用前需指向特定函数）实现 DMA 数据填充。在 i2s.c 里面，还有 2 个函数：I2S_Play_Start 和 I2S_Play_Stop，用于开启和关闭 DMA 传输，这里我们没贴出来了，请大家参考本例程源码。

4.1.2 wm8978.c

再来看 wm8978.c 里面的几个函数，代码如下：

```
//WM8978 初始化
//返回值:0,初始化正常
//    其他,错误代码
u8 WM8978_Init(void)
{
    u8 res;
    GPIO_InitTypeDef  GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC, ENABLE);          //使能外设 GPIOB,GPIOC 时钟

    //PB12/13 复用功能输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
```

```
17.       GPIO_Init(GPIOB, &GPIO_InitStructure);//初始化
18.
19.       //PC2/PC3/PC6 复用功能输出
20.       GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3|GPIO_Pin_6;
21.       GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
22.       GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽
23.       GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
24.       GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
25.       GPIO_Init(GPIOC, &GPIO_InitStructure);//初始化
26.
27.       GPIO_PinAFConfig(GPIOB,GPIO_PinSource12,GPIO_AF_SPI2); //PB12,AF5  I2S_LRCK
28.       GPIO_PinAFConfig(GPIOB,GPIO_PinSource13,GPIO_AF_SPI2);  //PB13,AF5  I2S_SCLK
29.       GPIO_PinAFConfig(GPIOC,GPIO_PinSource3,GPIO_AF_SPI2);   //PC3 ,AF5  I2S_DACDATA
30.       GPIO_PinAFConfig(GPIOC,GPIO_PinSource6,GPIO_AF_SPI2);   //PC6 ,AF5  I2S_MCK
31.       GPIO_PinAFConfig(GPIOC,GPIO_PinSource2,GPIO_AF6_SPI2);  //PC2 ,AF6  I2S_ADCDATA  I2S2ex
    t_SD 是 AF6!!!
32.
33.
34.
35.       IIC_Init();//初始化 IIC 接口
36.       res=WM8978_Write_Reg(0,0);  //软复位 WM8978
37.       if(res)return 1;            //发送指令失败,WM8978 异常
38.       //以下为通用设置
39.         WM8978_Write_Reg(1,0X1B);     //R1,MICEN 设置为 1(MIC 使能),BIASEN 设置为 1(模拟器工
    作),VMIDSEL[1:0]设置为:11(5K)
40.       WM8978_Write_Reg(2,0X1B0);  //R2,ROUT1,LOUT1 输出使能(耳机可以工作),BOOSTENR,BOOSTENL 使
    能
41.       WM8978_Write_Reg(3,0X6C);   //R3,LOUT2,ROUT2 输出使能(喇叭工作),RMIX,LMIX 使能
42.       WM8978_Write_Reg(6,0);      //R6,MCLK 由外部提供
43.       WM8978_Write_Reg(43,1<<4);    //R43,INVROUT2 反向,驱动喇叭
44.       WM8978_Write_Reg(47,1<<8);    //R47 设置,PGABOOSTL,左通道 MIC 获得 20 倍增益
45.       WM8978_Write_Reg(48,1<<8);    //R48 设置,PGABOOSTR,右通道 MIC 获得 20 倍增益
46.       WM8978_Write_Reg(49,1<<1);    //R49,TSDEN,开启过热保护
47.       WM8978_Write_Reg(10,1<<3);    //R10,SOFTMUTE 关闭,128x 采样,最佳 SNR
48.       WM8978_Write_Reg(14,1<<3);    //R14,ADC 128x 采样率
49.       return 0;
50.   }
51.
52.   //WM8978 DAC/ADC 配置
53.   //adcen:adc 使能(1)/关闭(0)
54.   //dacen:dac 使能(1)/关闭(0)
55.   void WM8978_ADDA_Cfg(u8 dacen,u8 adcen)
56.   {
57.       u16 regval;
```

```
    regval=WM8978_Read_Reg(3);  //读取 R3
    if(dacen)regval|=3<<0;          //R3 最低 2 个位设置为 1,开启 DACR&DACL
    else regval&=~(3<<0);       //R3 最低 2 个位清零,关闭 DACR&DACL.
    WM8978_Write_Reg(3,regval); //设置 R3
    regval=WM8978_Read_Reg(2);  //读取 R2
    if(adcen)regval|=3<<0;          //R2 最低 2 个位设置为 1,开启 ADCR&ADCL
    else regval&=~(3<<0);       //R2 最低 2 个位清零,关闭 ADCR&ADCL.
    WM8978_Write_Reg(2,regval); //设置 R2
}

//WM8978 输出配置
//dacen:DAC 输出(放音)开启(1)/关闭(0)
//bpsen:Bypass 输出(录音,包括 MIC,LINE IN,AUX 等)开启(1)/关闭(0)
void WM8978_Output_Cfg(u8 dacen,u8 bpsen)
{
    u16 regval=0;
    if(dacen)regval|=1<<0;      //DAC 输出使能
    if(bpsen)
    {
        regval|=1<<1;       //BYPASS 使能
        regval|=5<<2;       //0dB 增益
    }
    WM8978_Write_Reg(50,regval);//R50 设置
    WM8978_Write_Reg(51,regval);//R51 设置
}

//设置 I2S 工作模式
//fmt:0,LSB(右对齐);1,MSB(左对齐);2,飞利浦标准 I2S;3,PCM/DSP;
//len:0,16 位;1,20 位;2,24 位;3,32 位;
void WM8978_I2S_Cfg(u8 fmt,u8 len)
{
    fmt&=0X03;
    len&=0X03;//限定范围
    WM8978_Write_Reg(4,(fmt<<3)|(len<<5));  //R4,WM8978 工作模式设置
}
```

以上代码 WM8978_Init 用于初始化 WM8978，这里只是通用配置（ADC&DAC），初始化之后，并不能正常播放音乐，还需要通过 WM8978_ADDA_Cfg 函数，使能 DAC，然后通过 WM8978_Output_Cfg 选择 DAC 输出，通过 WM8978_I2S_Cfg 配置 I2S 工作模式，最后设置音量才可以接收 I2S 音频数据，实现音乐播放。这里设置音量、EQ、音效等函数，没有贴出了，请大家参考光盘本例程源码。

4.1.3 wavplay.c

接下来，看看 wavplay.c 里面的几个函数，代码如下：

```
__wavctrl wavctrl;      //WAV 控制结构体
vu8 wavtransferend=0;   //i2s 传输完成标志
vu8 wavwitchbuf=0;      //i2sbufx 指示标志

//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*)mymalloc(SRAMIN,sizeof(FIL));
    buf=mymalloc(SRAMIN,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
            {
                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!
                {
                    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);
    myfree(SRAMIN,ftemp);//释放内存
    myfree(SRAMIN,buf);
    return 0;
}

//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;
    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
            if(bread<size)//不够数据了,补充 0
            {
                for(i=bread;i<size-bread;i++)buf[i]=0;
            }
        }
        return bread;
}
//WAV 播放时,I2S DMA 传输回调函数
void wav_i2s_dma_tx_callback(void)
{
    u16 i;
    if(DMA1_Stream4->CR&(1<<19))
    {
        wavwitchbuf=0;
        if((audiodev.status&0X01)==0)
        {
            for(i=0;i<WAV_I2S_TX_DMA_BUFSIZE;i++)//暂停
            {
                audiodev.i2sbuf1[i]=0;//填充 0
            }
        }
    }else
    {
        wavwitchbuf=1;
        if((audiodev.status&0X01)==0)
        {
            for(i=0;i<WAV_I2S_TX_DMA_BUFSIZE;i++)//暂停
            {
                audiodev.i2sbuf2[i]=0;//填充 0
            }
        }
    }
    wavtransferend=1;
```

```
}
//得到当前播放时间
//fx:文件指针
//wavx:wav 播放控制器
void wav_get_curtime(FIL*fx,__wavctrl *wavx)
{
    long long fpos;
    wavx->totsec=wavx->datasize/(wavx->bitrate/8); //歌曲总长度(单位:秒)
    fpos=fx->fptr-wavx->datastart;                    //得到当前文件播放到的地方
    wavx->cursec=fpos*wavx->totsec/wavx->datasize; //当前播放到第多少秒了?
}
//播放某个 WAV 文件
//fname:wav 文件路径.
//返回值:
//KEY0_PRES:下一曲
//KEY1_PRES:上一曲
//其他:错误
u8 wav_play_song(u8* fname)
{
    u8 key;
    u8 t=0;
    u8 res;
    u32 fillnum;
    audiodev.file=(FIL*)mymalloc(SRAMIN,sizeof(FIL));
    audiodev.i2sbuf1=mymalloc(SRAMIN,WAV_I2S_TX_DMA_BUFSIZE);
    audiodev.i2sbuf2=mymalloc(SRAMIN,WAV_I2S_TX_DMA_BUFSIZE);
    audiodev.tbuf=mymalloc(SRAMIN,WAV_I2S_TX_DMA_BUFSIZE);
    if(audiodev.file&&audiodev.i2sbuf1&&audiodev.i2sbuf2&&audiodev.tbuf)
    {
        res=wav_decode_init(fname,&wavctrl);//得到文件的信息
        if(res==0)//解析文件成功
        {
            if(wavctrl.bps==16)
            {
                WM8978_I2S_Cfg(2,0);    //飞利浦标准,16 位数据长度
                 I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,I2S_DataForm
at_16b);     //飞利浦标准,主机发送,时钟低电平有效,16 位帧长度
            }else if(wavctrl.bps==24)
            {
                WM8978_I2S_Cfg(2,2);    //飞利浦标准,24 位数据长度
                 I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,I2S_DataForm
at_24b);     //飞利浦标准,主机发送,时钟低电平有效,24 位扩展帧长度
            }
            I2S2_SampleRate_Set(wavctrl.samplerate);//设置采样率
```

```
170.              I2S2_TX_DMA_Init(audiodev.i2sbuf1,audiodev.i2sbuf2,WAV_I2S_TX_DMA_BUFSIZE/2); //配置 TX DMA
171.                  i2s_tx_callback=wav_i2s_dma_tx_callback;                    //回调函数指 wav_i2s_dma_callback
172.              audio_stop();
173.              res=f_open(audiodev.file,(TCHAR*)fname,FA_READ);     //打开文件
174.              if(res==0)
175.              {
176.                  f_lseek(audiodev.file, wavctrl.datastart);      //跳过文件头
177.                  fillnum=wav_buffill(audiodev.i2sbuf1,WAV_I2S_TX_DMA_BUFSIZE,wavctrl.bps);
178.                  fillnum=wav_buffill(audiodev.i2sbuf2,WAV_I2S_TX_DMA_BUFSIZE,wavctrl.bps);
179.                  audio_start();
180.                  while(res==0)
181.                  {
182.                      while(wavtransferend==0);//等待 wav 传输完成;
183.                      wavtransferend=0;
184.                      if(fillnum!=WAV_I2S_TX_DMA_BUFSIZE)//播放结束?
185.                      {
186.                          res=KEY0_PRESS;
187.                          break;
188.                      }
189.                       if(wavwitchbuf)fillnum=wav_buffill(audiodev.i2sbuf2,WAV_I2S_TX_DMA_BUFSIZE,wavctrl.bps);//填充 buf2
190.                       else fillnum=wav_buffill(audiodev.i2sbuf1,WAV_I2S_TX_DMA_BUFSIZE,wavctrl.bps);//填充 buf1
191.                      while(1)
192.                      {
193.                          key=KEY_Scan(0);
194.                          if(key==KEY_UP_PRESS)//暂停
195.                          {
196.                              if(audiodev.status&0X01)audiodev.status&=~(1<<0);
197.                              else audiodev.status|=0X01;
198.                          }
199.                          if(key==KEY0_PRESS||key==KEY2_PRESS)//下一曲/上一曲
200.                          {
201.                              res=key;
202.                              break;
203.                          }
204.                             wav_get_curtime(audiodev.file,&wavctrl);//得到总时间和当前播放的时间
205.                          audio_msg_show(10,160,wavctrl.totsec,wavctrl.cursec,wavctrl.bitrate);
206.                          t++;
207.                          if(t==20)
```

```
                        {
                            t=0;
                            LED1=!LED1;
                        }
                        if((audiodev.status&0X01)==0)delay_ms(10);
                        else break;
                    }
                }
                audio_stop();
            }else res=0XFF;
        }else res=0XFF;
    }else res=0XFF;
    myfree(SRAMIN,audiodev.tbuf);   //释放内存
    myfree(SRAMIN,audiodev.i2sbuf1);//释放内存
    myfree(SRAMIN,audiodev.i2sbuf2);//释放内存
    myfree(SRAMIN,audiodev.file);   //释放内存
    return res;
}
```

以上，wav_decode_init 函数，用来对 wav 文件进行解析，得到 wav 的详细信息（音频采样率，位数，数据流起始位置等）；wav_buffill 函数，用 f_read 读取数据，填充数据到 buf 里面，注意 24 位音频的时候，读出的数据需要经过转换后才填充到 buf；wav_i2s_dma_tx_callback 函数，则是 DMA 发送完成的回调函数（i2s_tx_callback 函数指针指向该函数），这里面，我们并没有对数据进行填充处理（暂停时进行了填 0 处理），而是采用 2 个标志量：wavtransferend 和 wavwitchbuf，来告诉 wav_play_song 函数是否传输完成，以及应该填充哪个数据 buf（i2sbuf1 或 i2sbuf2）；

最后，wav_play_song 函数，是播放 WAV 的最终执行函数，该函数解析完 WAV 文件后，设置 WM8978 和 I2S 的参数（采样率，位数等），并开启 DMA，然后不停填充数据，实现 WAV 播放，该函数还进行了按键扫描控制，实现上下取切换和暂停/播放等操作。该函数通过判断 wavtransferend 是否为 1 来处理是否应该填充数据，而到底填充到哪个 buf（i2sbuf1 或 i2sbuf2），则是通过 wavwitchbuf 标志来确定的，当 wavwitchbuf=0 时，说明 DMA 正在使用 i2sbuf2，程序应该填充 i2sbuf1；当 wavwitchbuf=1 时，说明 DMA 正在使用 i2sbuf1，程序应该填充 i2sbuf2；

4.1.4 audioplay.c

接下来，看看 audioplay.c 里面的几个函数，代码如下：

```
1.  //播放音乐
2.  void audio_play(void)
3.  {
4.      u8 res;
5.      DIR wavdir;         //目录
6.      FILINFO *wavfileinfo;//文件信息
7.      u8 *pname;          //带路径的文件名
8.      u16 totwavnum;      //音乐文件总数
9.      u16 curindex;       //当前索引
10.     u8 key;             //键值
11.     u32 temp;
12.     u32 *wavoffsettbl;  //音乐 offset 索引表
13.
14.     WM8978_ADDA_Cfg(1,0);   //开启 DAC
15.     WM8978_Input_Cfg(0,0,0);//关闭输入通道
16.     WM8978_Output_Cfg(1,0); //开启 DAC 输出
17.     while(f_opendir(&wavdir,"0:/MUSIC"))//打开音乐文件夹
18.     {
19.         printf("MUSIC 文件夹错误!\r\n");
20.         delay_ms(500);
21.     }
22.     totwavnum=audio_get_tnum("0:/MUSIC"); //得到总有效文件数
23.     while(totwavnum==NULL)//音乐文件总数为 0
24.     {
25.         printf("没有音乐文件!\r\n");
26.         delay_ms(500);
27.     }
28.     wavfileinfo=(FILINFO*)mymalloc(SRAMIN,sizeof(FILINFO)); //申请内存
29.     pname=mymalloc(SRAMIN,_MAX_LFN*2+1);                    //为带路径的文件名分配内存
30.     wavoffsettbl=mymalloc(SRAMIN,4*totwavnum);              //申请 4*totwavnum 个字节的内存,用
    于存放音乐文件 off block 索引
31.     while(!wavfileinfo||!pname||!wavoffsettbl)//内存分配出错
32.     {
33.         printf("内存分配失败!\r\n");
34.         delay_ms(500);
35.     }
36.     //记录索引
37.     res=f_opendir(&wavdir,"0:/MUSIC"); //打开目录
38.     if(res==FR_OK)
39.     {
```

```
40.             curindex=0;//当前索引为 0
41.             while(1)//全部查询一遍
42.             {
43.                 temp=wavdir.dptr;                                   //记录当前 index
44.                 res=f_readdir(&wavdir,wavfileinfo);                 //读取目录下的一个文件
45.                 if(res!=FR_OK||wavfileinfo->fname[0]==0)break;    //错误了/到末尾了,退出
46.                 res=f_typetell((u8*)wavfileinfo->fname);
47.                 if((res&0XF0)==0X40)//取高四位,看看是不是音乐文件
48.                 {
49.                     wavoffsettbl[curindex]=temp;//记录索引
50.                     curindex++;
51.                 }
52.             }
53.         }
54.         curindex=0;                                             //从 0 开始显示
55.         res=f_opendir(&wavdir,(const TCHAR*)"0:/MUSIC");     //打开目录
56.         while(res==FR_OK)//打开成功
57.         {
58.             dir_sdi(&wavdir,wavoffsettbl[curindex]);                //改变当前目录索引
59.             res=f_readdir(&wavdir,wavfileinfo);                     //读取目录下的一个文件
60.               if(res!=FR_OK||wavfileinfo->fname[0]==0)break;              //错误了/到末尾了,退
    出
61.             strcpy((char*)pname,"0:/MUSIC/");                       //复制路径(目录)
62.             strcat((char*)pname,(const char*)wavfileinfo->fname);    //将文件名接在后面
63.             printf("当前播放音乐名：%s\r\n",wavfileinfo->fname);
64.             audio_index_show(10,140,curindex+1,totwavnum);
65.             audio_vol_show(10+80,140,audio_mvol);
66.             key=audio_play_song(pname);                         //播放这个音频文件
67.             if(key==KEY2_PRESS)     //上一曲
68.             {
69.                 if(curindex)curindex--;
70.                 else curindex=totwavnum-1;
71.             }else if(key==KEY0_PRESS)//下一曲
72.             {
73.                 curindex++;
74.                 if(curindex>=totwavnum)curindex=0;//到末尾的时候,自动从头开始
75.             }else break;    //产生了错误
76.         }
77.         myfree(SRAMIN,wavfileinfo);         //释放内存
78.         myfree(SRAMIN,pname);               //释放内存
79.         myfree(SRAMIN,wavoffsettbl);        //释放内存
80.     }
81.
82.     //播放某个音频文件
```

```
u8 audio_play_song(u8* fname)
{
    u8 res;
    res=f_typetell(fname);
    switch(res)
    {
        case T_WAV:
            res=wav_play_song(fname);
            break;
        default://其他文件,自动跳转到下一曲
            printf("can't play:%s\r\n",fname);
            res=KEY0_PRESS;
            break;
    }
    return res;
}
```

这里，audio_play 函数在 main 函数里面被调用，该函数首先设置 WM8978 相关配置，然后查找 SD 卡里面的 MUSIC 文件夹，并统计该文件夹里面总共有多少音频文件（统计包括：WAV/MP3/APE/FLAC 等），然后，该函数调用 audio_play_song 函数，按顺序播放这些音频文件。

在 audio_play_song 函数里面，通过判断文件类型，调用不同的解码函数，本章，只支持 WAV 文件，通过 wav_play_song 函数实现 WAV 解码。其他格式：MP3/APE/FLAC 等，需要移植相应的解码驱动，大家可以参考网上代码自行移植，这里就不做介绍了。

4.1.5 main.c

最后，我们看看主函数代码：

```
#include "system.h"
#include "SysTick.h"
#include "led.h"
#include "usart.h"
#include "malloc.h"
#include "sdio_sdcard.h"
#include "flash.h"
#include "ff.h"
#include "fatfs_app.h"
#include "key.h"
#include "wm8978.h"
```

```
#include "audioplay.h"


int main()
{

    SysTick_Init(168);
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);  //中断优先级分组 分 2 组
    LED_Init();
    KEY_Init();
    USART1_Init(115200);
    EN25QXX_Init();             //初始化 EN25Q128
    WM8978_Init();              //初始化 WM8978
    WM8978_HPvol_Set(40,40);    //耳机音量设置
    WM8978_SPKvol_Set(50);      //喇叭音量设置

    my_mem_init(SRAMIN);        //初始化内部内存池
    my_mem_init(SRAMCCM);       //初始化 CCM 内存池

    while(SD_Init()!=0)
    {
        printf("SD Card Error!\r\n");
        delay_ms(500);
    }
    FATFS_Init();                           //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1);                  //挂载 SD 卡
    f_mount(fs[1],"1:",1);                //挂载 FLASH.

    printf("音乐播放器实验\r\n");
    printf("KEY0:NEXT   KEY2:PREV\r\n");
    printf("KEY_UP:PAUSE/PLAY\r\n");

    while(1)
    {
        audio_play();
    }
}
```

该函数就相对简单了，在初始化各个外设后，通过 audio_play 函数，开始音频播放。软件部分就介绍到这里，其他未贴出代码，请参考本例程源码。

4.2 录音机实验

4.2.1 I2S 录音简介

本章涉及的知识点基本上在上一章都有介绍。本章要实现 WAV 录音，还是和上一章一样，要了解：WAV 文件格式、WM8978 和 I2S。WAV 文件格式，我们在上一章已经做了详细介绍了，这里就不作介绍了。

PZ-WM8978 模块将板载的一个 MIC 分别接入到了 WM8978 的 2 个差分输入通道（LIP/LIN 和 RIP/RIN）。代码上，我们采用立体声 WAV 录音，不过，左右声道的音源都是一样的，录音出来的 WAV 文件，听起来就是个单声道效果。

WM8978 上一章也做了比较详细的介绍，本章我们主要看一下要进行 MIC 录音，WM8978 的配置步骤：

1，寄存器 R0（00h），该寄存器用于控制 WM8978 的软复位，写任意值到该寄存器地址，即可实现软复位 WM8978。

2，寄存器 R1（01h），该寄存器主要要设置 MICBEN(bit4)和 BIASEN(bit3)两个位为 1，开启麦克风(MIC)偏置，以及使能模拟部分放大器。

3，寄存器 R2（02h），该寄存器要设置 SLEEP(bit6)、INPGAENR(bit3)、INPGAENL(bit2)、ADCENR(bit1)和 ADCENL(bit0)等五个位。SLEEP 设置为 0，进入正常工作模式;INPGAENR 和 INPGAENL 设置为 1,使能 IP PGA 放大器;ADCENL 和 ADCENR 设置为 1，使能左右通道 ADC。

4，寄存器 R4（04h），该寄存器要设置 WL(bit6:5)和 FMT(bit4:3)等 4 个位。WL(bit6:5)用于设置字长（即设置音频数据有效位数），00 表示 16 位音频，10 表示 24 位音频；FMT(bit4:3)用于设置 I2S 音频数据格式（模式），我们一般设置为 10，表示 I2S 格式，即飞利浦模式。

5，寄存器 R6（06h），该寄存器我们直接全部设置为 0 即可，设置 MCLK 和 BCLK 都来自外部，即由 STM32F4 提供。

6，寄存器 R14（0Eh），该寄存器要设置 ADCOSR128(bit3)为 1，ADC 得到最好的 SNR。

7，寄存器 R44（2Ch），该寄存器我们要设置 LIP2INPPGA(bit0)、LIN2INPPGA(bit1)、RIP2INPPGA(bit4)和 RIN2INPPGA(bit5)等 4 个位，将这 4

个位都设置为 1，将左右通道差分输入接入 IN PGA。ADCOSR128(bit3)为 1，ADC 得到最好的 SNR。

8，寄存器 R45（2Dh）和 R46（2Eh），这两个寄存器用于设置 PGA 增益（调节麦克风增益），一个用于设置左通道（R45），另外一个用于设置右通道（R46）。这两个寄存器的最高位（INPPGAUPDATE）用于设置是否更新左右通道的增益，最低 6 位用于设置左右通道的增益，我们可以先设置好两个寄存器的增益，最后设置其中一个寄存器最高位为 1，即可更新增益设置。

9，寄存器 R47（2Fh）和 R48（30h），这两个寄存器也类似，我们只关心其最高位(bit8)，都设置为 1，可以让左右通道的 MIC 各获得 20dB 的增益。

10，寄存器 R49（31h），该寄存器我们要设置 TSDEN(bit1)这个位，设置为 1，开启过热保护。

以上，就是我们用 WM8978 录音时的设置，按照以上所述，对各个寄存器进行相应的配置，即可使用 WM8978 正常录音了。不过我们本章还要用到播放录音的功能，WM8978 的播放配置在前面章节已经介绍过了，请大家参考前面内容。

上一章我们向大家介绍了 STM32F4 的 I2S 放音，通过上一章的了解，我们知道：STM32F4 的全双工需要用到扩展的 I2Sx_ext（x=2/3），和 I2Sx 组成全双工 I2S。在全双工模式下，I2Sx 向 I2Sx_ext 提供 CK 和 WS 时钟信号。

本章我们必须向 WM8978 提供 WS，CK 和 MCK 等时钟，同时又要录音，所以只能使用全双工模式。主 I2Sx 循环发送数据 0X0000，给 WM8978，以产生 CK、WS 和 MCK 等信号，从 I2Sx_ext，则接收来自 WM8978 的 ADC 数据（I2Sxext_SD），并保存到 SD 卡，实现录音。

本章我们还是采用 I2S2 的全双工模式来录音，I2S2 的相关寄存器，我们在上一章已经介绍的差不多了。至于 I2S2ext 的寄存器，则有一套和 I2S2 一样的寄存器，不过仅仅少数几个对我们有用，他们是：I2S2ext_I2SCFGR、I2S2ext_CR2 和 I2S2ext_DR，这三个寄存器对应为的功能和描述，完全同 I2S2。寄存器描述，我们这里就不再介绍了，大家可以看前面章节，也可以看《STM32F4xx 中文参考手册》第 27.5 节。

最后，我们看看要通过 STM32F4 的 I2S，驱动 WM8978 实现 WAV 录音的简要步骤。这一章用到的硬件部分知识点实际上一章节已经讲解，这里我们主要讲

解一下步骤：

（1）初始化 WM8978

这个过程就是前面所讲的 WM8978 MIC 录音配置步骤，让 WM8978 的 ADC 以及其模拟部分工作起来。

（2）初始化 I2S2 和 I2S2ext

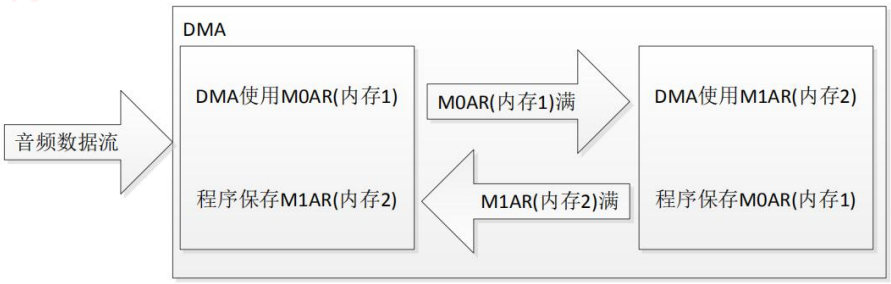
本章要用到 I2S2 的全双工模式，所以，I2S2 和 I2S2ext 都需要配置，其中 I2S2 配置为主机发送模式，I2S2ext 设置为从机接收模式。他们的其他配置（I2S 标准、时钟空闲电平和数据帧长）基本一样，只是一个是发送一个是接收，且都要使能 DMA。同时，还需要设置音频采样率，不过这个只需要设置 I2S2 即可，还是通过上一章介绍的查表法设置。

（3）设置发送和接收 DMA

放音和录音都是采用 DMA 传输数据的，本章放音起始就是个幌子，不过也得设置 DMA（使用 DMA1 数据流 4 的通道 0），配置同上一章一模一样，不过不需要开启 DMA 传输完成中断。对于录音，则使用的是 DMA1 数据流 3 的通道 3 实现的 DMA 数据接收，我们需要配置 DMA1 的数据流 3，本章将 DMA1 数据流 3 设置为：双缓冲循环模式，外设和存储器都是 16 位宽，并开启 DMA 传输完成中断（方便接收数据）。

（4）编写接收通道 DMA 传输完成中断服务函数

为了方便接收音频数据，我们使用 DMA 传输完成中断，每当一个缓冲接数据满了，硬件自动切换为下一个缓冲，同时进入中断服务函数，将已满缓冲的数据写入 SD 卡的 wav 文件。过程如下图所示：



（5）创建 WAV 文件，并保存 wav 头

前面 4 步完成，其实就可以开始读取音频数据了，不过在录音之前，我们需要先在创建一个新的文件，并写入 wav 头，然后才能开始写入我们读取到的的 PCM 音频数据。

（6）开启 DMA 传输，接收数据

然后，我们就只需要开启 DMA 传输，然后及时将 I2S2ext 读到的数据写入到 SD 卡之前新建的 wav 文件里面，就可以实现录音了。

（7）计算整个文件大小，重新保存 wav 头并关闭文件

在结束录音的时候，我们必须知道本次录音的大小（数据大小和整个文件大小），然后更新 wav 头，重新写入文件，最后因为 FATFS，在文件创建之后，必须调用 f_close，文件才会真正体现在文件系统里面，否则是不会写入的！所以最后还需要调用 f_close，以保存文件。

4.2.2 recorder.c

打开本章实验工程可以看到，相比上一章实验，我们主要在 AudioDecod 分组下面新增了 recorder.c 和 recorder.h 两个文件。

因为 recorder.c 代码比较多，我们这里仅介绍其中几个重要的函数，代码如下：

```
u8 *i2srecbuf1;
u8 *i2srecbuf2;

FIL* f_rec=0;          //录音文件
u32 wavsize;           //wav 数据大小(字节数,不包括文件头!!)
u8 rec_sta=0;          //录音状态
                       //[7]:0,没有开启录音;1,已经开启录音;
                       //[6:1]:保留
                       //[0]:0,正在录音;1,暂停录音;

//录音 I2S_DMA 接收中断服务函数.在中断里面写入数据
void rec_i2s_dma_rx_callback(void)
{
    u16 bw;
    u8 res;
    if(rec_sta==0X80)//录音模式
    {
        if(DMA1_Stream3->CR&(1<<19))
        {
            res=f_write(f_rec,i2srecbuf1,I2S_RX_DMA_BUF_SIZE,(UINT*)&bw);//写入文件
            if(res)
            {
                printf("write error:%d\r\n",res);
```

```
            }

        }else
        {
            res=f_write(f_rec,i2srecbuf2,I2S_RX_DMA_BUF_SIZE,(UINT*)&bw);//写入文件
            if(res)
            {
                printf("write error:%d\r\n",res);
            }
        }
        wavsize+=I2S_RX_DMA_BUF_SIZE;
    }
}
const u16 i2splaybuf[2]={0X0000,0X0000};//2 个 16 位数据,用于录音时 I2S Master 发送.循环发送 0.
//进入 PCM 录音模式
void recoder_enter_rec_mode(void)
{
    WM8978_ADDA_Cfg(0,1);       //开启 ADC
    WM8978_Input_Cfg(1,1,0);    //开启输入通道(MIC&LINE IN)
    WM8978_Output_Cfg(0,1);     //开启 BYPASS 输出
    WM8978_MIC_Gain(46);        //MIC 增益设置

    WM8978_I2S_Cfg(2,0);        //飞利浦标准,16 位数据长度
     I2S2_Init(I2S_Standard_Phillips,I2S_Mode_MasterTx,I2S_CPOL_Low,I2S_DataFormat_16b);
  //飞利浦标准,主机发送,时钟低电平有效,16 位帧长度
     I2S2ext_Init(I2S_Standard_Phillips,I2S_Mode_SlaveRx,I2S_CPOL_Low,I2S_DataFormat_16b);
  //飞利浦标准,从机接收,时钟低电平有效,16 位帧长度
    I2S2_SampleRate_Set(16000); //设置采样率
    I2S2_TX_DMA_Init((u8*)&i2splaybuf[0],(u8*)&i2splaybuf[1],1);       //配置 TX DMA
    DMA1_Stream4->CR&=~(1<<4); //关闭传输完成中断(这里不用中断送数据)
    I2S2ext_RX_DMA_Init(i2srecbuf1,i2srecbuf2,I2S_RX_DMA_BUF_SIZE/2);   //配置 RX DMA
    i2s_rx_callback=rec_i2s_dma_rx_callback;//回调函数指 wav_i2s_dma_callback
    I2S_Play_Start();   //开始 I2S 数据发送(主机)
    I2S_Rec_Start();    //开始 I2S 数据接收(从机)
    recoder_remindmsg_show(0);
}
//进入 PCM 放音模式
void recoder_enter_play_mode(void)
{
    WM8978_ADDA_Cfg(1,0);       //开启 DAC
    WM8978_Input_Cfg(0,0,0);    //关闭输入通道(MIC&LINE IN)
    WM8978_Output_Cfg(1,0);     //开启 DAC 输出
    WM8978_MIC_Gain(0);         //MIC 增益设置为 0
    I2S_Play_Stop();            //停止时钟发送
```

```
66.         I2S_Rec_Stop();             //停止录音
67.         recoder_remindmsg_show(1);
68.     }
69.     //初始化 WAV 头.
70.     void recoder_wav_init(__WaveHeader* wavhead) //初始化 WAV 头
71.     {
72.         wavhead->riff.ChunkID=0X46464952;    //"RIFF"
73.         wavhead->riff.ChunkSize=0;           //还未确定,最后需要计算
74.         wavhead->riff.Format=0X45564157;     //"WAVE"
75.         wavhead->fmt.ChunkID=0X20746D66;     //"fmt "
76.         wavhead->fmt.ChunkSize=16;           //大小为 16 个字节
77.         wavhead->fmt.AudioFormat=0X01;       //0X01,表示 PCM;0X01,表示 IMA ADPCM
78.         wavhead->fmt.NumOfChannels=2;        //双声道
79.         wavhead->fmt.SampleRate=16000;       //16Khz 采样率 采样速率
80.         wavhead->fmt.ByteRate=wavhead->fmt.SampleRate*4;//字节速率=采样率*通道数*(ADC 位数/8)
81.         wavhead->fmt.BlockAlign=4;           //块大小=通道数*(ADC 位数/8)
82.         wavhead->fmt.BitsPerSample=16;       //16 位 PCM
83.         wavhead->data.ChunkID=0X61746164;    //"data"
84.         wavhead->data.ChunkSize=0;           //数据大小,还需要计算
85.     }
86.     //显示录音时间和码率
87.     //tsec:秒钟数.
88.     void recoder_msg_show(u32 tsec,u32 kbps)
89.     {
90.         //显示录音时间
91.         printf("录音时间:%d:%d    ",tsec/60,tsec%60);
92.         //显示码率
93.         printf("码率:%dKPBS\r\n",kbps/1000);
94.     }
95.     //提示信息
96.     //mode:0,录音模式;1,放音模式
97.     void recoder_remindmsg_show(u8 mode)
98.     {
99.         if(mode==0) //录音模式
100.         {
101.             printf("KEY0:REC/PAUSE\r\n");
102.             printf("KEY1:STOP&SAVE\r\n");
103.             printf("KEY_UP:PLAY\r\n");
104.         }else       //放音模式
105.         {
106.             printf("KEY0:STOP Play\r\n");
107.             printf("KEY_UP:PLAY/PAUSE\r\n");
108.         }
109.     }
```

```
//通过时间获取文件名
//仅限在 SD 卡保存,不支持 FLASH DISK 保存
//组合成:形如"0:RECORDER/REC20120321210633.wav"的文件名
void recoder_new_pathname(u8 *pname)
{
    u8 res;
    u16 index=0;
    while(index<0XFFFF)
    {
        sprintf((char*)pname,"0:RECORDER/REC%05d.wav",index);
        res=f_open(ftemp,(const TCHAR*)pname,FA_READ);//尝试打开这个文件
        if(res==FR_NO_FILE)break;       //该文件名不存在=正是我们需要的.
        index++;
    }
}
//WAV 录音
void wav_recorder(void)
{
    u8 res;
    u8 key;
    u8 rval=0;
    __WaveHeader *wavhead=0;
    DIR recdir;                     //目录
    u8 *pname=0;
    u8 timecnt=0;                   //计时器
    u32 recsec=0;                   //录音时间
    while(f_opendir(&recdir,"0:/RECORDER"))//打开录音文件夹
    {
        printf("RECORDER 文件夹错误!\r\n");
        delay_ms(500);
        f_mkdir("0:/RECORDER");             //创建该目录
    }
    i2srecbuf1=mymalloc(SRAMIN,I2S_RX_DMA_BUF_SIZE);//I2S 录音内存 1 申请
    i2srecbuf2=mymalloc(SRAMIN,I2S_RX_DMA_BUF_SIZE);//I2S 录音内存 2 申请
    f_rec=(FIL *)mymalloc(SRAMIN,sizeof(FIL));      //开辟 FIL 字节的内存区域
    wavhead=(__WaveHeader*)mymalloc(SRAMIN,sizeof(__WaveHeader));//开辟__WaveHeader字节的内存区域
      pname=mymalloc(SRAMIN,30);                                //申请 30 个字节内存,类似"0:RECORDER/REC00001.wav"
    if(!i2srecbuf1||!i2srecbuf2||!f_rec||!wavhead||!pname)rval=1;   if(rval==0)
    {
        recoder_enter_rec_mode();   //进入录音模式,此时耳机可以听到咪头采集到的音频
        pname[0]=0;                 //pname 没有任何文件名
        while(rval==0)
```

```
        {
            key=KEY_Scan(0);
            switch(key)
            {
                case KEY1_PRESS:    //STOP&SAVE
                    if(rec_sta&0X80)//有录音
                    {
                        rec_sta=0;  //关闭录音
                        wavhead->riff.ChunkSize=wavsize+36;      //整个文件的大小-8;
                        wavhead->data.ChunkSize=wavsize;     //数据大小
                        f_lseek(f_rec,0);                        //偏移到文件头.
                         f_write(f_rec,(const void*)wavhead,sizeof(__WaveHeader),&bw);//写入头数据
                        f_close(f_rec);
                        wavsize=0;
                    }
                    rec_sta=0;
                    recsec=0;
                    LED2=1;                          //关闭 DS1
                    break;
                case KEY0_PRESS:    //REC/PAUSE
                    if(rec_sta&0X01)//原来是暂停,继续录音
                    {
                        rec_sta&=0XFE;//取消暂停
                    }else if(rec_sta&0X80)//已经在录音了,暂停
                    {
                        rec_sta|=0X01;  //暂停
                    }else               //还没开始录音
                    {
                        recsec=0;
                        recoder_new_pathname(pname);             //得到新的名字
                        printf("当前录音文件:%s\r\n",pname+11);
                        recoder_wav_init(wavhead);               //初始化 wav 数据
                         res=f_open(f_rec,(const TCHAR*)pname, FA_CREATE_ALWAYS | FA_WRITE);

                        if(res)         //文件创建失败
                        {
                            rec_sta=0;  //创建文件失败,不能录音
                            rval=0XFE;  //提示是否存在 SD 卡
                        }else
                        {
                            res=f_write(f_rec,(const void*)wavhead,sizeof(__WaveHeader),&bw);//写入头数据
                            recoder_msg_show(0,0);
```

```
                            rec_sta|=0X80;  //开始录音
                        }
                    }
                    if(rec_sta&0X01)LED2=0; //提示正在暂停
                    else LED2=1;
                    break;
                case KEY_UP_PRESS:  //播放最近一段录音
                    if(rec_sta!=0X80)//没有在录音
                    {
                        if(pname[0])//如果触摸按键被按下,且 pname 不为空
                        {
                            printf("当前播放文件:%s\r\n",pname+11);
                            recoder_enter_play_mode();  //进入播放模式
                            audio_play_song(pname);     //播放 pname
                            recoder_enter_rec_mode();   //重新进入录音模式
                        }
                    }
                    break;
            }
            delay_ms(5);
            timecnt++;
            if((timecnt%20)==0)LED1=!LED1;//DS0 闪烁
            if(recsec!=(wavsize/wavhead->fmt.ByteRate))  //录音时间显示
            {
                LED1=!LED1;//DS0 闪烁
                recsec=wavsize/wavhead->fmt.ByteRate;    //录音时间
                 recoder_msg_show(recsec,wavhead->fmt.SampleRate*wavhead->fmt.NumOfChannels*
    wavhead->fmt.BitsPerSample);//显示码率
            }
        }
    }
    myfree(SRAMIN,i2srecbuf1);  //释放内存
    myfree(SRAMIN,i2srecbuf2);  //释放内存
    myfree(SRAMIN,f_rec);       //释放内存
    myfree(SRAMIN,wavhead);     //释放内存
    myfree(SRAMIN,pname);       //释放内存
}
```

这里总共 5 个函数，其中：rec_i2s_dma_rx_callback 函数，用于 I2S2ext 的 DMA 接收完成中断回调函数（通过 i2s_rx_callback 指向该函数实现），在该函数里面，实现音频数据的保存。recoder_enter_rec_mode 函数，用于设置 WM8978 和 I2S 进入录音模式(开始录音时用到)。recoder_enter_play_mode 函数，则用于设置 WM8978 和 I2S 进入播放模式（录音回放时用到）。

recoder_wav_init 函数，该函数初始化 wav 头的绝大部分数据，这里我们设置了该 wav 文件为 16Khz 采样率，16 位线性 PCM 格式，另外由于录音还未真正开始，所以文件大小和数据大小都还是未知的，要等录音结束才能知道。该函数__WaveHeader 结构体就是由上一章介绍的三个 Chunk 组成，结构为：

```
1.  //wav 头
2.  typedef __packed struct
3.  {
4.      ChunkRIFF riff; //riff 块
5.      ChunkFMT fmt; //fmt 块
6.      // ChunkFACT fact; //fact 块 线性 PCM,没有这个结构体
7.      ChunkDATA data; //data 块
8.  }__WaveHeader;
```

最后，wav_recorder 函数，实现了我们在硬件设计时介绍的功能（开始/暂停录音、保存录音文件、播放最近一次录音等）。该函数使用上一章实现的 audio_play_song 函数，来播放最近一次录音。recorder.c 的其他代码和 recorder.h 的代码我们这里就不再贴出了，请大家参考本实验的源码。

4.2.3 i2s.c

然后，我们在 i2s.c 里面也增加了几个函数，如下：

```
1.  //I2S2ext 配置
2.  //参数 I2S_Standard:  @ref SPI_I2S_Standard  I2S 标准,
3.  //I2S_Standard_Phillips,飞利浦标准;
4.  //I2S_Standard_MSB,MSB 对齐标准(右对齐);
5.  //I2S_Standard_LSB,LSB 对齐标准(左对齐);
6.  //I2S_Standard_PCMShort,I2S_Standard_PCMLong:PCM 标准
7.  //参数 I2S_Mode:  @ref SPI_I2S_Mode  I2S_Mode_SlaveTx:从机发送;I2S_Mode_SlaveRx:从机接收;
8.  // 参 数 I2S_Clock_Polarity  @ref SPI_I2S_Clock_Polarity:  I2S_CPOL_Low, 时 钟 低 电 平 有效;I2S_CPOL_High,时钟高电平有效
9.  // 参 数 I2S_DataFormat :  @ref SPI_I2S_Data_Format  : 数 据 长 度 ,I2S_DataFormat_16b,16 位 标准  ;I2S_DataFormat_16bextended,16  位  扩  展  (frame=32bit);I2S_DataFormat_24b,24 位;I2S_DataFormat_32b,32 位.
10.
11. void I2S2ext_Init(u16 I2S_Standard,u16 I2S_Mode,u16 I2S_Clock_Polarity,u16 I2S_DataFormat)

12. {
13.     I2S_InitTypeDef I2S2ext_InitStructure;
14.
15.     I2S2ext_InitStructure.I2S_Mode=I2S_Mode^(1<<8);//IIS 模式
```

```
    I2S2ext_InitStructure.I2S_Standard=I2S_Standard;//IIS 标准
    I2S2ext_InitStructure.I2S_DataFormat=I2S_DataFormat;//IIS 数据长度
    I2S2ext_InitStructure.I2S_MCLKOutput=I2S_MCLKOutput_Disable;//主时钟输出禁止
    I2S2ext_InitStructure.I2S_AudioFreq=I2S_AudioFreq_Default;//IIS 频率设置
    I2S2ext_InitStructure.I2S_CPOL=I2S_Clock_Polarity;//空闲状态时钟电平

    I2S_FullDuplexConfig(I2S2ext,&I2S2ext_InitStructure);//初始化 I2S2ext 配置

    SPI_I2S_DMACmd(I2S2ext,SPI_I2S_DMAReq_Rx,ENABLE);//I2S2ext RX DMA 请求使能.

    I2S_Cmd(I2S2ext,ENABLE);        //I2S2ext I2S EN 使能.

}

//I2S2ext RX DMA 配置
//设置为双缓冲模式,并开启 DMA 传输完成中断
//buf0:M0AR 地址.
//buf1:M1AR 地址.
//num:每次传输数据量
void I2S2ext_RX_DMA_Init(u8* buf0,u8 *buf1,u16 num)
{

    NVIC_InitTypeDef   NVIC_InitStructure;
    DMA_InitTypeDef  DMA_InitStructure;


    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,ENABLE);//DMA1 时钟使能

    DMA_DeInit(DMA1_Stream3);
    while (DMA_GetCmdStatus(DMA1_Stream3) != DISABLE){}//等待 DMA1_Stream3 可配置

     DMA_ClearITPendingBit(DMA1_Stream3,DMA_IT_FEIF3|DMA_IT_DMEIF3|DMA_IT_TEIF3|DMA_IT_HTIF3
|DMA_IT_TCIF3);//清空 DMA1_Stream3 上所有中断标志

    /* 配置 DMA Stream */

    DMA_InitStructure.DMA_Channel = DMA_Channel_3;  //通道 3 I2S2ext_RX 通道
          DMA_InitStructure.DMA_PeripheralBaseAddr   =   (u32)&I2S2ext->DR;//  外  设  地  址
为:(u32)&I2S2ext->DR>DR
    DMA_InitStructure.DMA_Memory0BaseAddr = (u32)buf0;//DMA 存储器 0 地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;//外设到存储器模式
    DMA_InitStructure.DMA_BufferSize = num;//数据传输量
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量模式
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//存储器增量模式
```

```
58.         DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;//外设数据长度:16 位
59.         DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;//存储器数据长度: 16 位
60.       DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
61.       DMA_InitStructure.DMA_Priority = DMA_Priority_Medium;//中等优先级
62.       DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable; //不使用 FIFO 模式
63.       DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_1QuarterFull;
64.       DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;//外设突发单次传输
65.       DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;//存储器突发单次传输
66.       DMA_Init(DMA1_Stream3, &DMA_InitStructure);//初始化 DMA Stream
67.
68.       DMA_DoubleBufferModeConfig(DMA1_Stream3,(u32)buf1,DMA_Memory_0);//双缓冲模式配置
69.
70.       DMA_DoubleBufferModeCmd(DMA1_Stream3,ENABLE);//双缓冲模式开启
71.
72.       DMA_ITConfig(DMA1_Stream3,DMA_IT_TC,ENABLE);//开启传输完成中断
73.
74.
75.       NVIC_InitStructure.NVIC_IRQChannel = DMA1_Stream3_IRQn;
76.       NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority =0x00;//抢占优先级 0
77.       NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x01;//子优先级 1
78.       NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//使能外部中断通道
79.       NVIC_Init(&NVIC_InitStructure);//配置
80.   }
81.
82.   //DMA1_Stream3 中断服务函数
83.   void DMA1_Stream3_IRQHandler(void)
84.   {
85.       if(DMA_GetITStatus(DMA1_Stream3,DMA_IT_TCIF3)==SET) //DMA1_Stream3,传输完成标志
86.       {
87.           DMA_ClearITPendingBit(DMA1_Stream3,DMA_IT_TCIF3);    //清除传输完成中断
88.           i2s_rx_callback();  //执行回调函数,读取数据等操作在这里面处理
89.       }
90.   }
91.
92.   //I2S 开始录音
93.   void I2S_Rec_Start(void)
94.   {
95.       DMA_Cmd(DMA1_Stream3,ENABLE);//开启 DMA TX 传输,开始录音
96.   }
97.   //关闭 I2S 录音
98.   void I2S_Rec_Stop(void)
99.   {
```

```
    DMA_Cmd(DMA1_Stream3,DISABLE);//关闭 DMA,结束录音
}
```

这里也是 5 个函数，I2S2ext _Init 函数完成 I2S2ext 的初始化，通过 4 个参数设置 I2S2ext 的详细配置信息。I2S2ext_RX_DMA_Init 函数，用于设置 I2S2ext 的 DMA 接收，使用双缓冲循环模式，接收来自 WM8978 的数据，并开启了传输完成中断。而 DMA1_Stream3_IRQHandler 函数，则是 DMA1 数据流 3 传输完成中断的服务函数，该函数调用 i2s_rx_callback 函数（函数指针，使用前需指向特定函数）实现 DMA 数据接收保存。最后，I2S_ Rec_Start 和 I2S_ Rec_Stop，用于开启和关闭 DMA 传输。

其他代码，我们就不再介绍了，请大家参考本例程源码。

4.2.4 main.c

最后，我们看看主函数代码：

```
#include "system.h"
#include "SysTick.h"
#include "led.h"
#include "usart.h"
#include "malloc.h"
#include "sdio_sdcard.h"
#include "flash.h"
#include "ff.h"
#include "fatfs_app.h"
#include "key.h"
#include "wm8978.h"
#include "audioplay.h"
#include "recorder.h"


int main()
{

    SysTick_Init(168);
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);  //中断优先级分组 分 2 组
    LED_Init();
    KEY_Init();
    USART1_Init(115200);
    EN25QXX_Init();             //初始化 EN25Q128
    WM8978_Init();              //初始化 WM8978
```

```
26.      WM8978_HPvol_Set(40,40);    //耳机音量设置
27.      WM8978_SPKvol_Set(50);      //喇叭音量设置
28.
29.      my_mem_init(SRAMIN);        //初始化内部内存池
30.      my_mem_init(SRAMCCM);       //初始化 CCM 内存池
31.
32.      while(SD_Init()!=0)
33.      {
34.          printf("SD Card Error!\r\n");
35.          delay_ms(500);
36.      }
37.      FATFS_Init();                           //为 fatfs 相关变量申请内存
38.      f_mount(fs[0],"0:",1);                  //挂载 SD 卡
39.      f_mount(fs[1],"1:",1);                //挂载 FLASH.
40.
41.      printf("录音机实验\r\n");
42.
43.      while(1)
44.      {
45.          wav_recorder();
46.      }
47.  }
```

该函数代码同上一章的 main 函数代码几乎一样，十分简单，我们就不再多说了。

5 实验现象

首先，请先确保硬件都已经连接好：

①PZ-WM8978 模块与 T100&T200 开发板连接（可参考硬件设计小节）

②使用一张 SD 卡插入开发板，且存储好.wav 文件格式的音乐文件（**需注意文件存放位置，可参考硬件设计章节**）

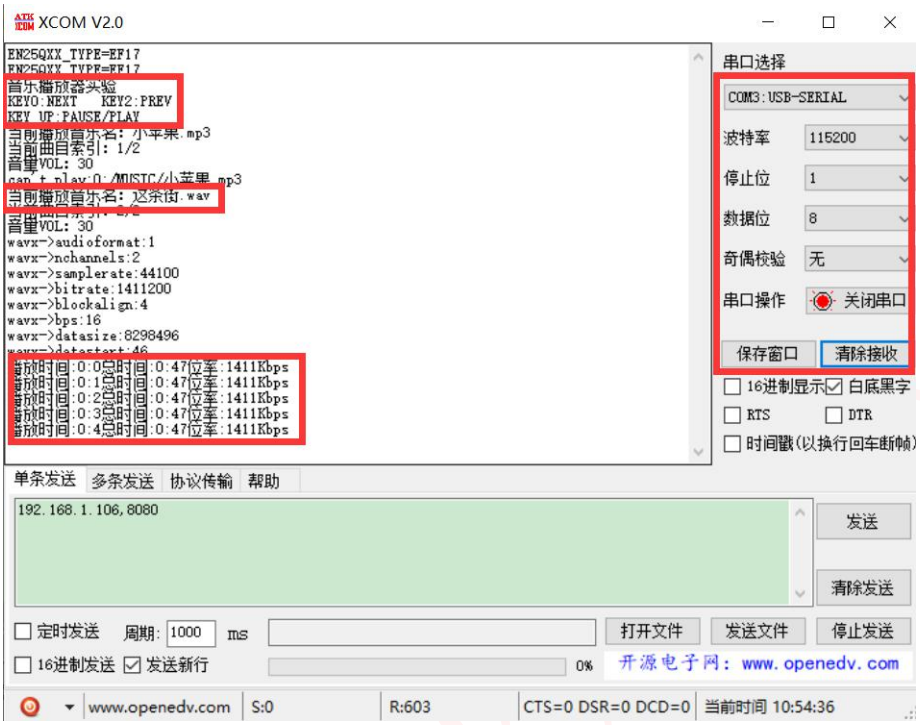
③使用耳机线插入模块 PHONE 接口。

④使用 USB 线给开发板供电，并且通过串口助手查看输出信息。

5.1 音乐播放器实验

将工程程序编译后下载到开发板内，可以看到 DS0 指示灯不断闪烁，表示程

序正常运行。打开串口调试助手可以看到串口助手显示，且自动播放 SD 卡中音乐文件。如下图所示：



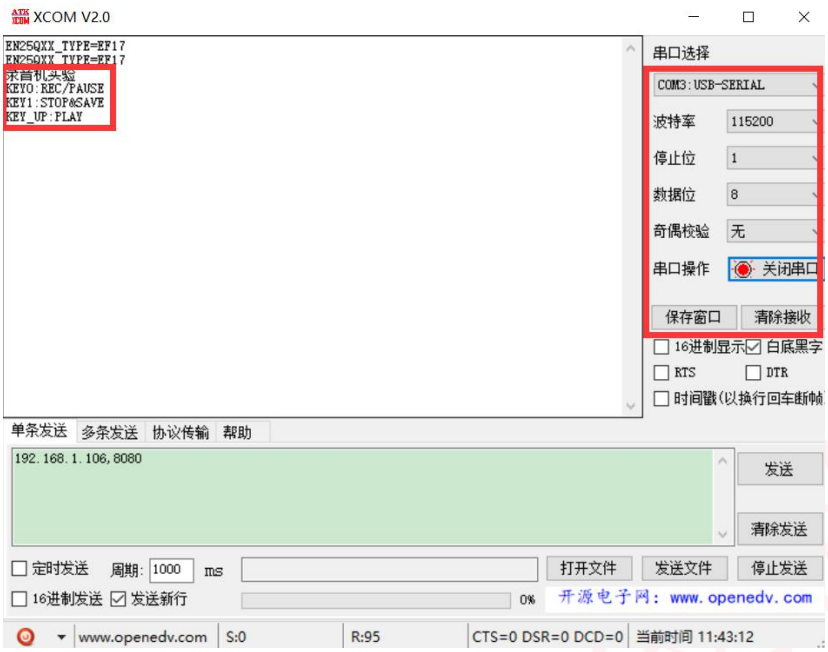
从上图可以看出，当前正在播放第 2 首歌曲，总共 2 首歌曲，歌曲名、播放时间、总时长、码率、音量等信息等也都有显示。此时 DS0 会随着音乐的播放而闪烁。

只要我们在模块的 PHONE 端子插入耳机（或者在 J2 接口插入喇叭），就能听到歌曲的声音了。同时，我们可以通过按 KEY0 和 KEY2 来切换下一曲和上一曲，通过 KEY_UP 控制暂停和继续播放。

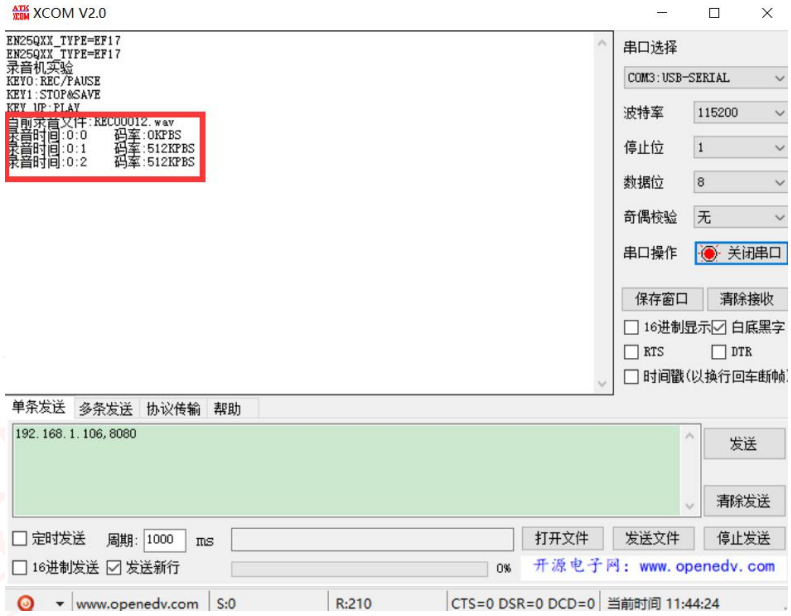
至此，我们就完成了一个简单的音乐播放器了，虽然只支持 WAV 文件，但是大家可以在此基础上，增加其他音频格式解码器，便可实现其他音频格式解码了。

5.2 录音机实验

将工程程序编译后下载到开发板内，可以看到 DS0 指示灯不断闪烁，表示程序正常运行。打开串口调试助手可以看到串口助手显示。如下图所示：



此时，我们按下 KEY0 就开始录音了，此时看到屏幕显示录音文件的名字、码率以及录音时长，如下图所示：



在录音的时候按下 KEY0 则执行暂停/继续录音的切换，通过 DS1 指示录音暂停。通过按下 KEY2，可以停止当前录音，并保存录音文件。在完成一次录音文件保存之后，我们可以通过按 KEY_UP 按键，来实现播放这个录音文件（即播放最近一次的录音文件），实现试听。我们将开发板的录音文件放到电脑上面，可以通过属性查看录音文件的属性，如下图所示：

这和我们预期的效果一样，通过电脑端的播放器可以直接播放我们所录的音频。经实测效果还是不错的。

6 其他

（1）购买地址（普中授权店铺）

http://www.prechin.net/forum.php?mod=viewthread&tid=38746&extra=

（2）资料下载

http://prechin.net/forum.php?mod=viewthread&tid=35264&extra=page%3D1

（3）技术支持

普中官网：www.prechin.cn

普中论坛：www.prechin.net

技术电话：0755-36564227（转技术）