摘要

作为ST公司的代表性产品，STM32芯片无论是在工业控制领域还是民用终端领域都占据着举足轻重的地位。本文以一款基于STM32的数字音频播放器为例，详细介绍了嵌入式系统开发的基本过程。在文章中，首先阐述了数字音频的的基本原理，介绍了STM32F103芯片以及SD卡硬件电路，并讨论了如何在MDK开发环境下利用STM32库函数来进行嵌入式系统开发；通过对FATfs文件系统的移植，来实现对带有FAT文件系统的大容量SD卡中数据的读写；文章中间部分详细分析了WAV音频文件格式及其解码算法的实现，同时采用了双缓存的工作模式来解决从SD卡中读取并播放音频数据的连贯性问题。文章最后介绍了如何利用外部按键来实现一个音频播放器的基本功能使之成为一个完成的产品。

**关键词： STM32F103 WAV SD FATfs 双缓存**

ABSTRACT

**Keywords:**

目录

摘要 i

ABSTRACT i

目录 1

第一章 绪论 3

1.1引言 3

1.2音频播放的基本原理 4

1.2.1 声音的概述 4

1.2.2 数字音频的原理 4

1.2.3 音频系统 5

第二章 硬件设计 7

2.1 总体框架 7

2.2 STM32微控制器 8

2.2.1 芯片介绍 8

2.2.2 开发板介绍 8

2.3 SD卡模块 9

2.3.1 SD卡介绍 9

2.3.2 SD卡内部结构及引脚功能 10

2.4 外部按键模块 11

2.4.1 按键控制介绍 11

2.4.2 电路原理图 12

第三章 软件设计 13

3.1 总体框架 13

3.2 STM32固件库 14

3.2.1 什么是STM32库 14

3.2.2 STM32库文件介绍 15

3.3 WAV音频文件 16

3.3.1 WAV简介 16

3.3.2 WAV文件结构 16

3.4 SDIO驱动 20

3.4.1 SDIO简介 20

3.4.2 SDIO的命令格式 21

3.4.3 数据传输格式 22

3.4.4 SDIO驱动移植 23

3.5 FATfs文件系统 24

3.5.1 FATfs简介 24

3.5.2 FATfs目录结构 24

3.5.3 为文件系统添加底层驱动 25

3.5.4 添加简体中文和长文件名支持 27

第四章 系统开发与测试 28

4.1系统概述 28

4.1.1 系统概述 28

4.1.2 开发环境 29

4.2 WAV音频解码算法 30

4.2.1 设计方案 30

4.3.2 流程图 32

4.3 双缓存 34

4.3.1 概述 34

4.3.2 设计方案 35

4.3.2 流程图 36

4.4 播放控制 37

4.4.1 设计需求 37

4.4.2 方案介绍 37

4.4.3 流程图 40

第五章 总结与展望 41

5.1 全文总结 41

5.2 改进及下一步工作 41

5.2.1 系统存在的不足 41

5.2.2 下一步的工作 42

致谢 43

参考文献 44

第一章 绪论

1.1引言

音乐，美妙的音乐，作为人类最宝贵的精神食粮之一，几乎贯穿着人类的整个发展史。从最早中世纪时期的教堂圣咏开始，到以巴赫为代表的巴洛克时期，再到经过启蒙运动的洗礼后诞生的古典主义与浪漫主义时期，音乐一直都是作为艺术的最高表现而被人们所敬仰。

然而，由于技术的落后，作为一种只可现场欣赏的一次性消耗品，音乐，尤其是大型的古典交响乐，一直都被视作一种只有贵族或才能享受到的奢侈品。一直等到二十世纪初，随着黑胶唱片和磁带的出现，音乐终于走入了寻常百姓家，那些无法去音乐会现场的普通人终于通过音乐媒介来欣赏到这一高雅的艺术，而CD碟的出现，则更是将音乐载体的变迁史又推进到下一个阶段。和磁带不同，CD碟是将音乐转化为“0”和“1”的数字格式进行存储，这也为迄今为止最新的一次音乐载体变革买下了伏笔。当更多数字音频压缩格式以及数据存储、传输技术出现的时候，人们迎来了一场延续至今的数字音乐革命。

90年代初，一个多国小组开发出了MP3格式，这是一种全新的有损数字音频压缩格式，能够在保证较好音频质量的同时大大减小存储体积。在当时这样一个存储设备价格极高，而互联网还仅仅处于萌芽状态的时期，MP3的出现极大的满足了普通人对于音乐的追求，能够拥有一部MP3随身听，已经成为当时的年轻人所梦寐以求的事情。音乐，也与人们的生活越来越贴近。

直到今天，随着人们生活水平的提升，人们对于自己的生活品质也有了更高的要求。生产力的解放带动着技术的革新，而技术的革新则推动着世界的进步。随着IT技术和互联网的爆炸性增长，曾经高昂的存储价格和流量费现已不复存在，取而代之的是更大的存储空间，更快的网速，更低廉的价格，以及人们对HIFI音乐的追求。APE格式与FLAC格式虽然有着优质的音质，但经常受到版权问题的困扰，曾经因为太过庞大但却拥有和CD相差无几的高品质音频质量的WAV格式又重新走上了舞台。而本课题，正是在这样的一个前提下提出来的。

1.2音频播放的基本原理

#### 1.2.1 声音的概述

声音在本质上是一种机械振动，它通过空气传播到人耳，刺激神经后使大脑产生一种感觉。在一些专业场合，声音通常被称为声波或音频。声音在物理学上称之为声波，是通过一定介质（如空气、水等）传播的一种连续振动的波，也称为声波。通常把频率范围为20Hz～20kHz的信号称为音频信号。低于20Hz的信号为次声信号；高于20kHz的信号称为超声信号。

#### 1.2.2 数字音频的原理

我们知道，在现实生活中，人耳听到的声音是模拟信号，计数机中能只能存储数字信号，把模拟信号转换成数字信号的原理简单地说就是利用一个固定的频率对模拟信号进行采样，采样后的信号在波形上看就像一串连续的幅值不一的脉冲，把这些脉冲的幅值按一定的精度进行量化，这些量化后的数值被连续地输出、传输、处理或记录到存储介质中，所有这些组成了数字音频的产生过程。

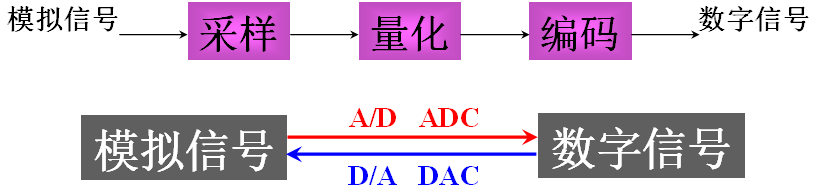


图1-1 转换过程

信号取样定理：**对于上限频率为fm的限带信号f（t），若取样频率为：fs≥2fm，则用频率为fm的理想低通滤波器就可以完全恢复原信号。**

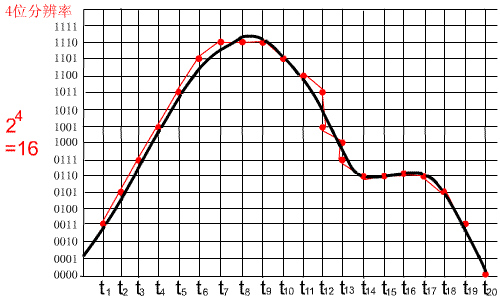


图 1-2 信号量化示意图

声音数字化三要素：

1. 采样频率：每秒钟抽取声波幅度样本的次数，通常来说，采样频率越高，声音质量越好，数据量也越大。常用采样频率：11.025kHz、22.05kHz、44.1kHz；
2. 量化位数：每个采样点用多少二进制位表示数据范围，量化位数越多，音质越好，数据量也越大。常用量化位数：8位＝256、16位＝65,536、24位＝16,777,216;
3. 声道数：使用声音通道的个数，立体声比单声道的表现力更丰富，但数据量翻倍。常用声道数：单声道、立体声。

#### 1.2.3 音频系统

针对不同的数字音频子系统，出现了几种微处理器或DSP与音频器件间用于数字转换的接口，其中最常用的有以下三种接口：

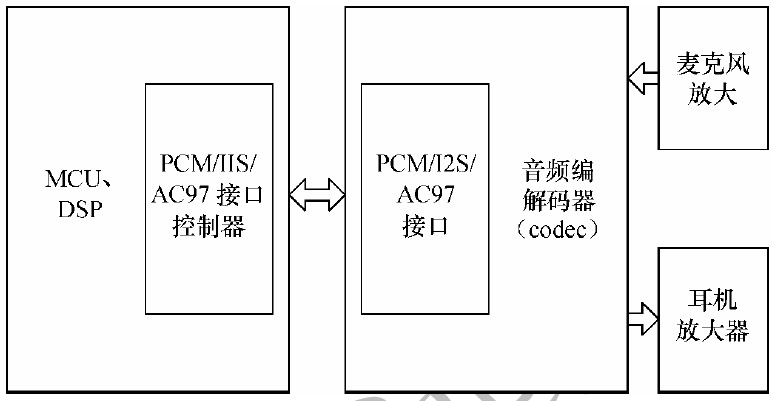


图 1-3 音频系统示意图

1. PCM接口：

PCM（ Pulse Code Modulation，中文称脉冲编码调制）接口是最简单的音频接口，该接口由时钟脉冲（BCLK）、帧同步信号（FS）及接收数据（DR）和发送数据（DX）组成。在FS信号的上升沿，数据传输从MSB（Most Significant Bit）字开始，FS频率等于采样率。FS信号之后开始数据字的传输，单个的数据位按顺序进行传输，1个时钟周期传输1个数据字。

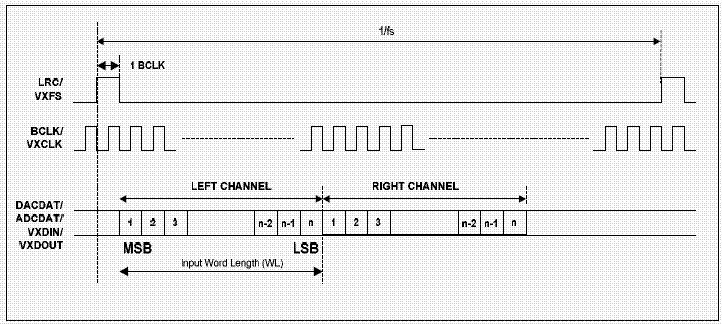


图 1-4 PCM传输示意图

1. I2S接口：

IIS接口（Inter-IC Sound）在20世纪80年代首先被飞利浦用于消费音频，并在一个称为LRCLK（Left/Right CLOCK）的信号机制中经过多路转换，将两路音频信号变成单一的数据队列。当LRCLK为高时，左声道数据被传输；LRCLK为低时，右声道数据被传输。与PCM相比，IIS更适合于立体声系统（因为可以传送多个声道的数据，所以显然更加适合Stereo Single，其实所谓的立体声，也就是多个声道可以发出不一样的声音）。

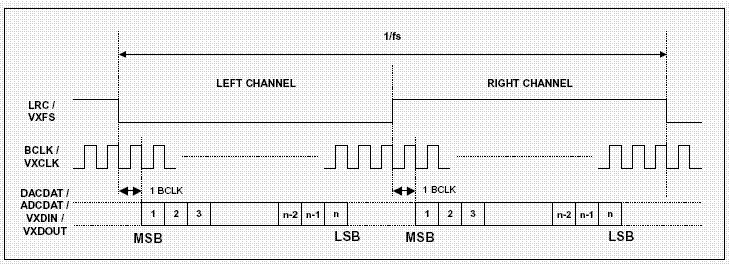


图 1-5 I2S传输示意图

1. AC97接口：

AC'97（Audio Codec 1997）是以Intel为首的五个PC厂商Intel、Creative Labs、NS、Analog Device与Yamaha共同提出的规格标准。与PCM和IIS不同，AC'97不只是一种数据格式，用于音频编码的内部架构规格，它还具有控制功能。 AC97 标准把音频设备中的数字部分和模拟部分分开, 并规定数字信号处理由CPU 来负责,或者采用专门的DSP 芯片; 而模拟部分, 即A/D、D/A 转换与Mix 混音操作, 则由CODEC 芯片(Coder2Decoder: 编码/解码器) 完成。

PCM、IIS和AC97各有其优点和应用范围，例如在CD、MD、MP3随身听多采用IIS接口，移动电话会采用PCM接口，具有音频功能的PDA则多使用和PC一样的AC'97编码格式。经过多方面的考虑，在本项目中，最终选择的是采用PCM接口来作为播放器的数据传输接口。

第二章 硬件设计

2.1 总体框架

本文所设计的作品，是一款基于STM32F103微控制器的WAV数字音频播放器，其主要功能包括从带有FAT文件系统的SD卡中播放WAV音频文件；通过摇杆及按键可以实现对音量的调节，以及基本的音乐播放控制（上一曲、下一曲、暂停／继续播放、停止等）；循环播放文件夹中的所有WAV音频文件。作为ARM嵌入式开发的主流工具，MDK也是目前为数不多的几个完全支持Cortex-M3处理器开发的企业级开发工具，本项目中所使用的MDK 5.1版本更是包含了STM32F103的所有外围设备固件库，可以通过环境配置自由选择所需要的固件库。在硬件方面，本系统分为了六个主要模块，分别是控制模块、文件系统模块、固件下载模块、串口调试模块、播放控制模块和播放模块，系统结构如图2-1所示：

图 2-1系统总体框架图

MCU

SDIO

SD Card

Keys

Main board

Speaker/ earphone

J-Link

JTAG/SW

PC

Usart

在这里我们着重介绍控制模块、文件系统模块和播放控制模块。

2.2 STM32微控制器

#### 2.2.1 芯片介绍

STM32F103ZET6使用高性能的ARM Cortex-M3核心的32位微控制器，LQFP-144封装，512K片内FLASH（相当于硬盘），64K片内RAM（相当于内存），高达72M的时钟频率,丰富的增强I/O端口和联接到两条APB总线的外设，3路共16通道的12位AD输入，2路共2 通道的12位 DA 输出。4个通用定时器，2个高级定时器，2个基本定时器，使得处理速度大大提高，机器的功耗也大幅降低。32位的处理器使得控制更加的稳定、迅速，而窗口型的看门狗，则使得程序的运行更加高效。另外，通过连接J-Link仿真器可以方便实现在线逐步调试功能，大大方便了系统的开发与调试工作。

芯片的管脚如图2-2下所示：



图 2-2 STM32F103ZET6管脚图

#### 2.2.2 开发板介绍

为了降低成本，在本项目中我们使用的是一款搭载了STM32F103ZET6芯片的多功能STM3210E-EVAL开发板，丰富的外设资源为此项目的升级与扩展提供了更多的选择。关于开发板的具体功能在这里就不再赘述，STM32F103芯片的各个管脚与开发板的连接如图2-3所示：

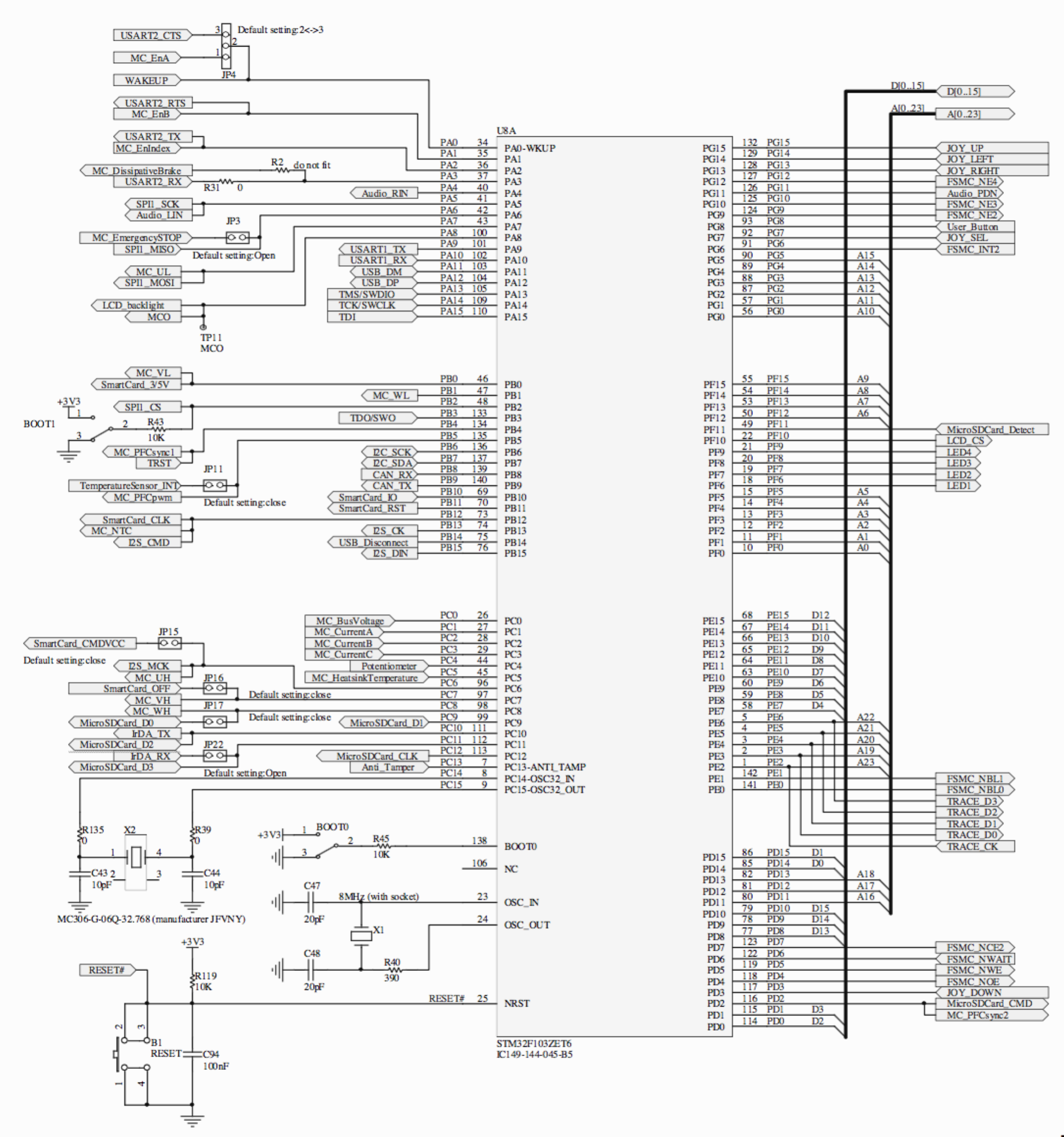


图 2-3 STM32F103ZET6管脚引出电路图

2.3 SD卡模块

#### 2.3.1 SD卡介绍

Secure Digital缩写SD，全名Secure Digital Memory Card，中文翻译为安全数位卡，是一种基于半导体快闪记忆器的新一代记忆设备，它被广泛地于便携式装置上使用。SD卡(Secure Digital Memory Card)是一种基于半导体闪存工艺的存储卡，1999年由日本松下主导概念，参与者东芝和美国SanDisk公司进行实质研发而完成，目前SD卡已成为消费数码设备中应用最广泛的一种存储卡，而作为SD卡的衍生品，miniSD卡和microSD卡虽然出现的比较晚，但由于有着更小的体积，因而再电子市场上有着更为广泛的应用。在此项目中我们使用的是microSD作为音频文件的存储媒介。图2-4为开发板上的microSD管脚电路图。

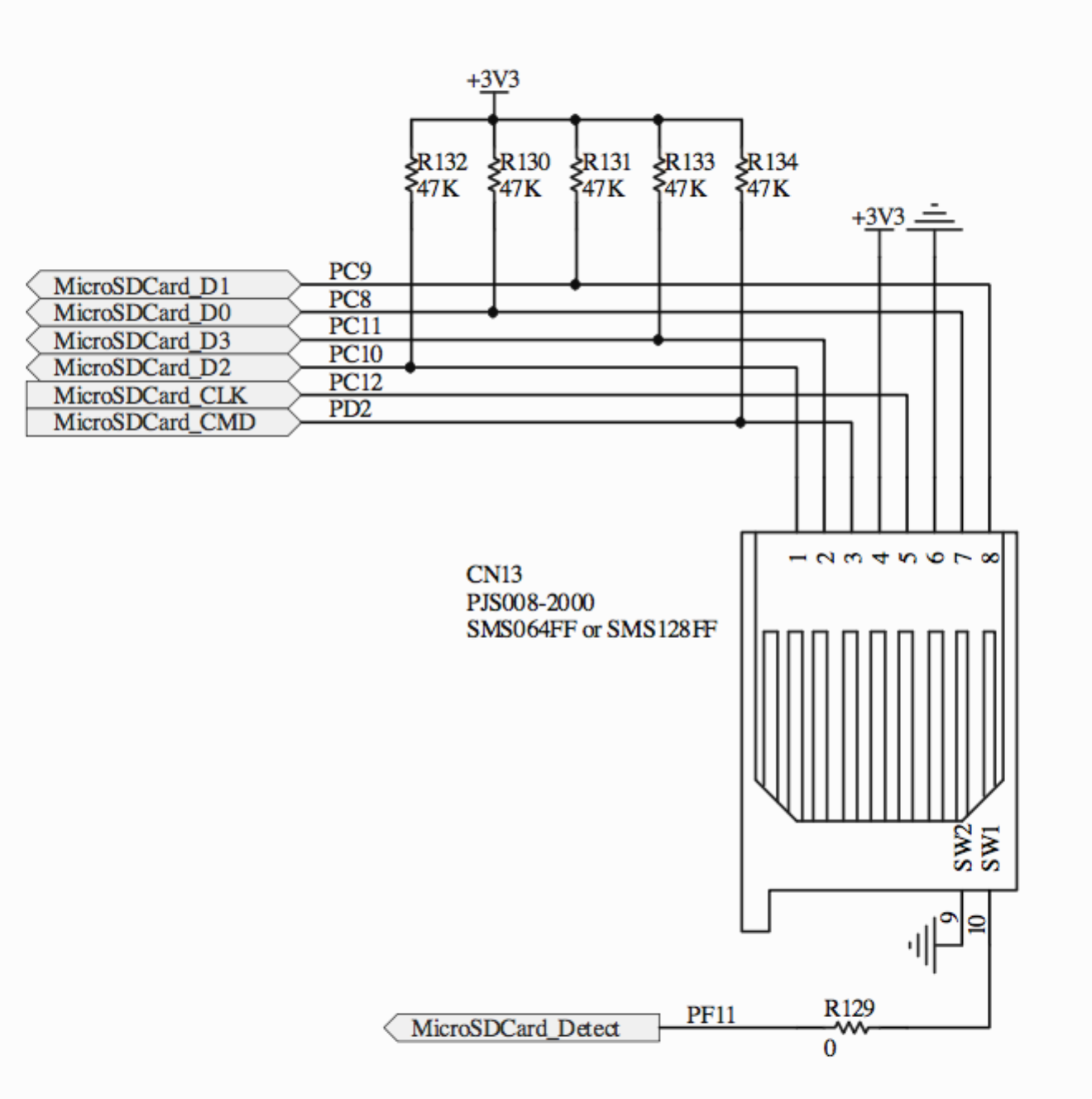


图 2-4 microSD管脚引出电路图

#### 2.3.2 SD卡内部结构及引脚功能

SD卡允许在两种模式下工作，即SD模式和SPI模式，由于本项目使用的是微处理器中的SDIO通信接口来连接SD卡，因此采用的是SD模式。SD卡内部结构及引脚如图2-5所示：

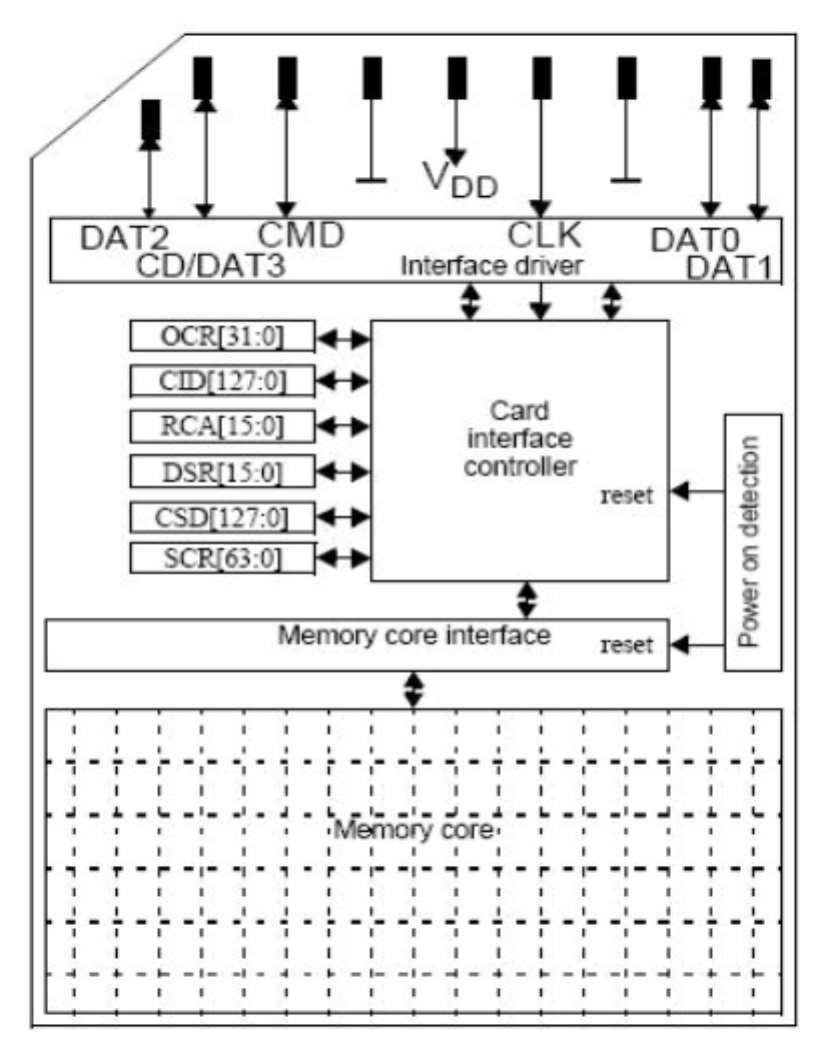


图 2-5 SD卡内部结构及引脚

SD卡的主要引脚和功能：

1. CLK：

CLK是卡的时钟信号线，由主机产生时钟信号，在命令和数据线上，每个时钟周期传输一个命令或数据位，频率可在0～25MHz之间变化。

1. CMD：

CMD为双向命令和回复线，SDIO的所有主机发出的命令及从机对命令的响应都是在这条信号线上传输的。

1. DAT［3:0］：

DAT［3:0］表示四条数据线，主机和从机的数据信号在这四条线上传输。

图2-6为一个SDIO主机与两个SD、SDIO类型卡的连接方式。SDIO接口包含CLK、CMD及4条DAT［3:0］信号线。这六条信号线都是公用总线，即新加入的设备可以并联接入SDIO接口。SDIO主机是通过命令和SD从设备的响应来寻址的，所以不需要片选信号线。关于SDIO的介绍后面在做详细论述。

SD Host

SDIO Card

SDIO Card

DAT［3:0］

DAT［3:0］

CLK

CMD

CMD

CLK

2.4 外部按键模块

图 2-6 SDIO连接图

#### 2.4.1 按键控制介绍

本次项目使用了七个外部按键来实现对音频播放的控制，其中包括一个操纵杆及两个按键。操纵杆包含上、下、左、右、确认五个按键，两个独立按键分别为Wakeup和Key。在这里，我们使用按键触发MCU外部中断来实现对播放系统的控制及音量的调节。

#### 2.4.2 电路原理图

图2-7为操纵杆的电路图，DOWN和UP与MCU的PG3和PG15相连，分别控制着音量的增减；LEFT和RIGHT与MCU的PG14和PG13相连，分别控制着上一曲、下一曲功能；Selection与MCU的PG7相连，在音乐播放时控制播放的暂停与继续播放功能。

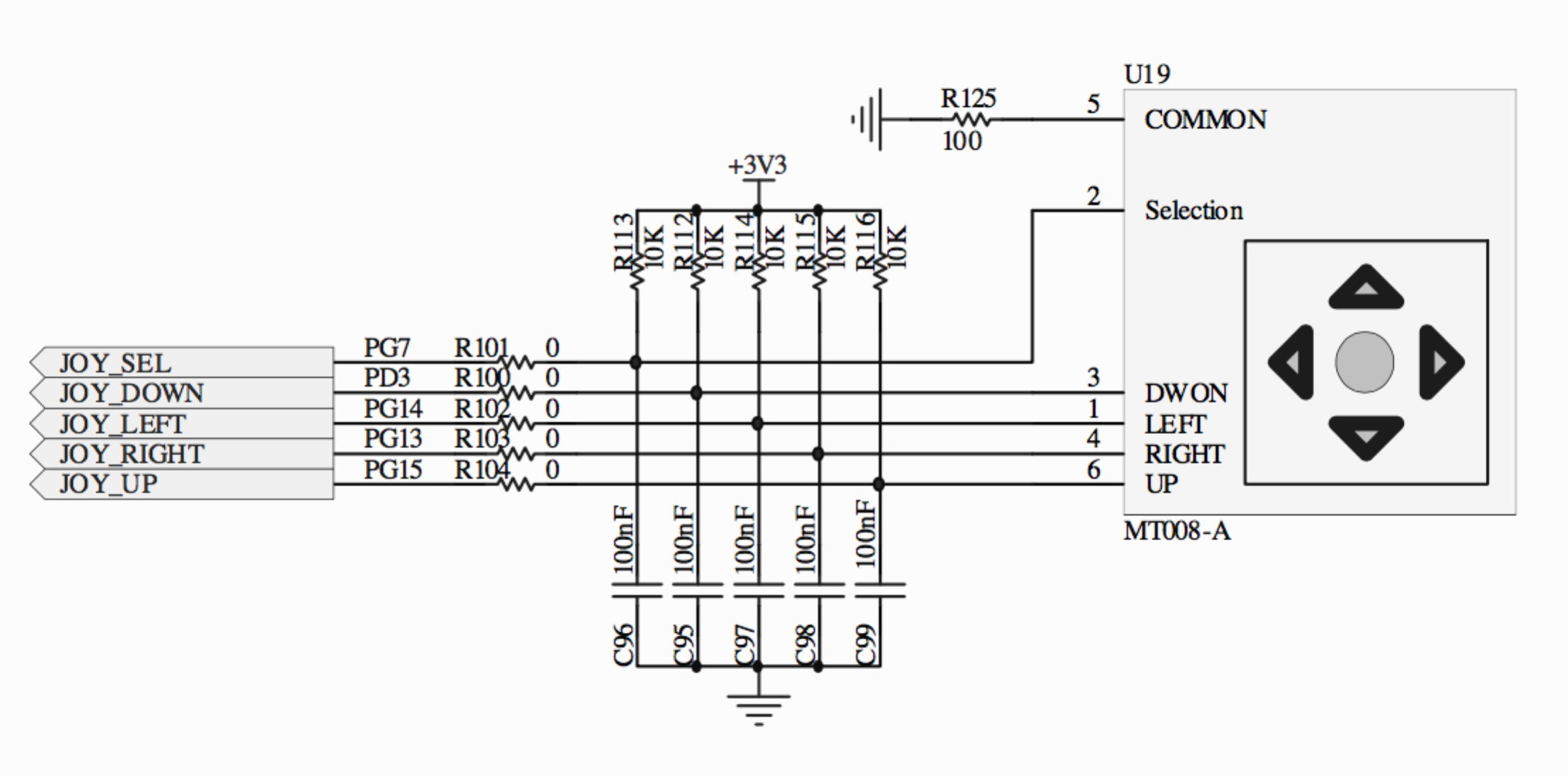


图 2-7 操纵杆管脚引出电路图

图2-8是名为Key按键的电路图，与MCU的PG8相连，负责停止音乐播放。

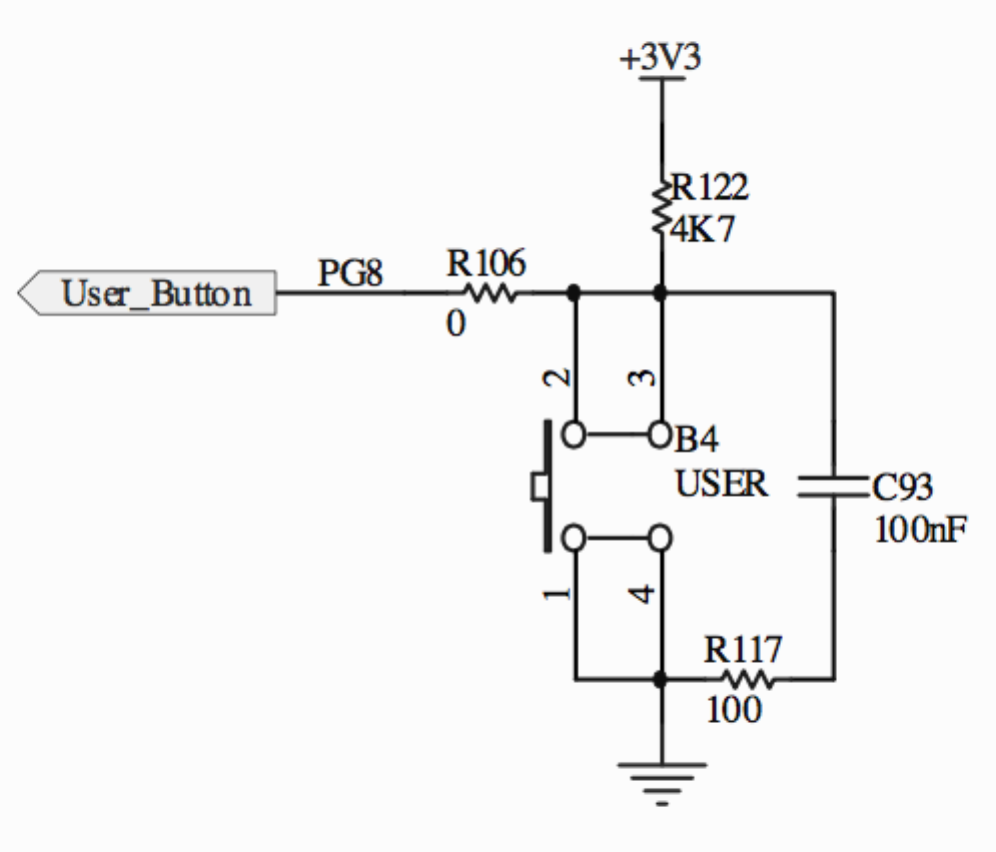


图2-8 Key按键管脚引出电路图

图2-9是名为WAKEUP按键的电路图，与MCU的PA0相连，负责在系统启动后和停止后重新播放音乐。

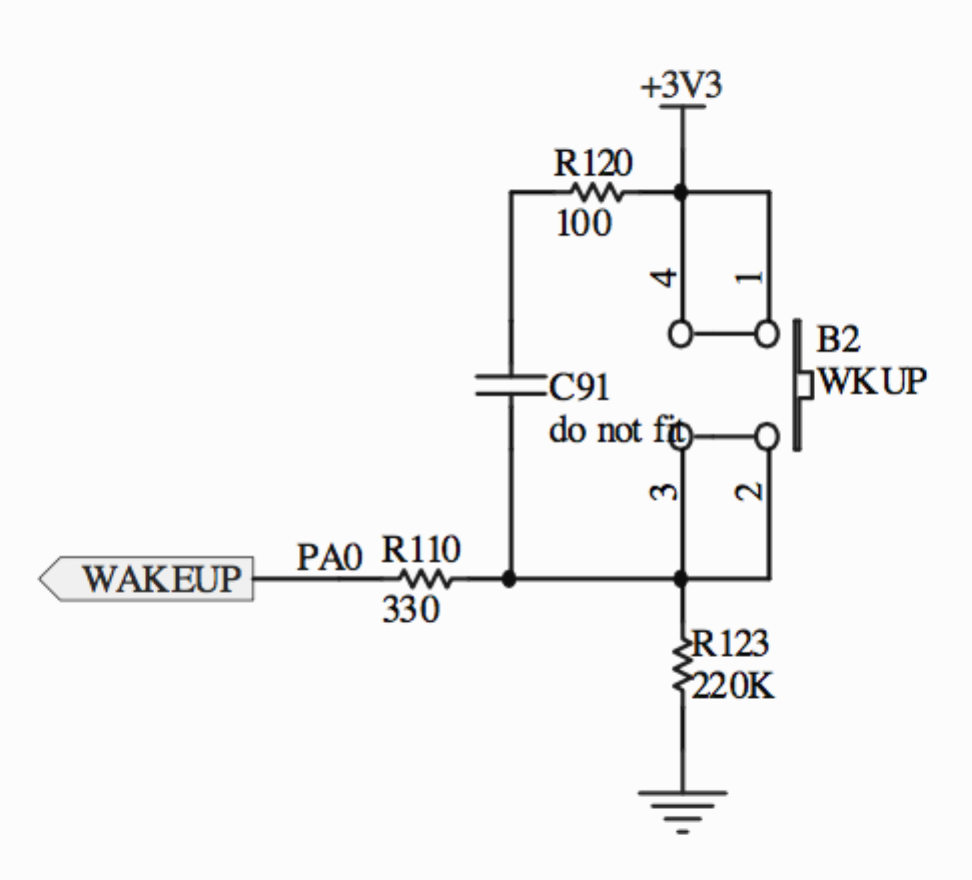


图 2-9 Wakeup按键管脚引出电路图

第三章 软件设计

3.1 总体框架

在嵌入式系统开发中，软件设计是最为重要的一步，对于此项目来说，是否拥有足够的功能与优异的性能，软件设计至关重要。在此项目中，我把软件部分分为了五个主要的模块，如图3-1所示。

图 3-1软件总体框架图

其中，串口通信模块负责输出输入信息，方便调试程序；STM32固件库为程序的开发提供了大量可供调用的API，极大的降低了开发的难度；外部按键模块使用户可以通过按键来控制音乐播放；SDIO和FATfs文件系统为音乐文件的存储与操作提供了良好的基础；而WAV音频文件则详细描述了WAV的原理与结构。这一切都为了第四章系统的设计提供了理论基础。

下面，我们就来详细的介绍一下最主要的几个模块。

3.2 STM32固件库

#### 3.2.1 什么是STM32库

在51单片机的程序开发中，我们通常是直接配置51单片机的寄存器，控制芯片的工作方式，如中断、定时器等。配置的时候，我们常常要查阅寄存器表。这些都是很琐碎的、机械的工作，因为51单片机的软件相对来说比较简单，而且资源很有限，所以可以通过直接配置寄存器的方式来开发。

STM32库是由ST公司针对STM32提供的函数接口，即API（Application Program Interface），开发者可以调用这些函数接口来配置STM32的寄存器，使开发人员得以脱离最底层的寄存器操作，有开发快速、易于阅读、维护成本低等优点。

实际上，库是架设在寄存器与用户驱动层之间的代码，向下处理与寄存器直接相关的配置，向上为用户提供配置寄存器的接口。库开发方式与直接配置寄存器方式的区别件图3-2。

驱动层

库函数层

特殊寄存器层

调用库接口

以函数宏封装配置寄存器的操作

库开发方式

驱动层

特殊寄存器层

直接配置寄存器

直接配置寄存器方式

图 3-2 开发方式对比图

对于STM32，因为外设资源丰富，带来的必然是寄存器的数量和复杂度的增加，这时直接配置寄存器方式的两个缺陷——开发速度慢和可读性差——就凸显出来了，这两个缺陷直接影响了开发效率、程序维护成本的交流成本。而库开发方式则正好弥补了这来那个缺陷。这也是为什么在本项目中采用STM库开发的首要原因。

#### 3.2.2 STM32库文件介绍

图3-3为工程中所使用的库文件目录，其中CMSIS文件夹中包含了内核的库文件，CMSIS是ARM与芯片厂商为了解决不同芯片厂商生产的Cortex微控制器软件的兼容性问题所建立的一个标准——CMSIS标准，所谓CMSIS标准，实际是新建了一个软件抽象层，提供了与芯片生产厂商无关的硬件抽象层，可以为接口外设、实时操作系统提供简单的处理器软件接口，屏蔽了硬件差异。STM32固件库就是按照CMSIS标准建立的。

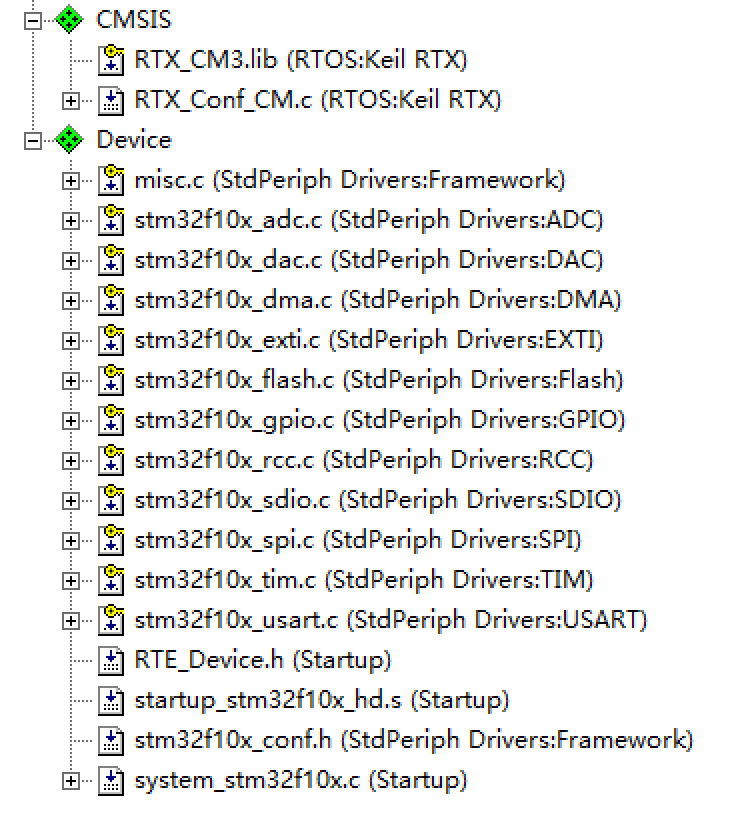


图 3-3 固件库目录

Device文件夹中包含了所需的外设库函数文件，其库文件可以在创建工程文件时的环境配置阶段选择，当然开发过程中也可根据需要自由删减库文件，这也是MDK 5.1所带来的好处之一。

关于STM32库的介绍在这里就不再赘述，如有兴趣请参考《STM32参考手册》和《ARM Cortex-M3权威指南》。

3.3 WAV音频文件

#### 3.3.1 WAV简介

WAV是一种微软公司开发的，用于存放声音文件的重要的文件格式，，它符合RIFF(Resource Interchange File Format)文件规范，尽管现在有MP3，RAM等压缩效率更高的声音文件格式，并且广泛被音乐文件所采用，但是又很多的应用程序仍然采用WAV文件格式。由于WAV文件没有采用压缩技术，所以它的文件很庞大，一般都在几MB以上。但也正是因为没有采用压缩技术，声音的采样数据很容易被读出来，便于用作其他的处理。该格式支持多种量化位数，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位量化位数，因此在声音文件质量和CD相差无几。由于WAV格式的音频文件有着优秀的音质与相对简单的格式，因此在本项目中采用手动实现WAV解码算法来软解码播放。

#### 3.3.2 WAV文件结构

WAV格式符合 RIFF(Resource Interchange File Format)规范。所有的WAV都有一个文件头，这个头文件纪录着这个文件中音频流的编码参数。**数据块的记录方式是little-endian字节顺序，标志符并不是字符串而是单独的符号，**数据本身的格式为PCM或压缩型，属于无损音乐格式的一种。图3-4为WAV音频文件的文件头格式：



图 3-4 WAV文件头文件

图3-5位WAV文件的数据位格式：



图 3-5 WAV声音文件数据位

图3-6是通过Ultra Edit打开的一首WAV音乐的头文件部分：

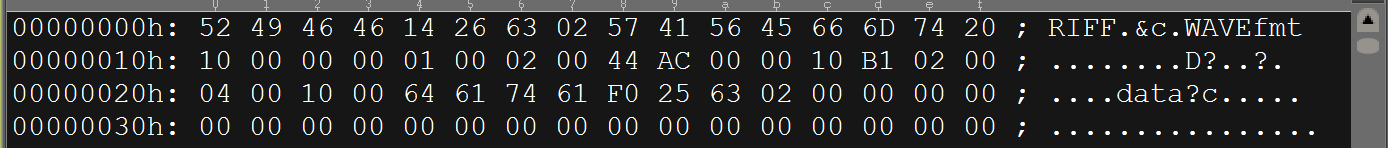


图 3-6 用Ultra Edit解析WAV

其中：

“52 49 46 46”这个是Ascii字符“RIFF”，这部分是固定格式，表明这是一个WAVE文件头。

“14 26 63 02”这个是的WAV文件的数据大小，这个大小包括除了前面4个字节的所有字节，也就是等于文件总字节数减去8。得到40,052,244字节。一定要注意！在计算由多个字节表示的数字时，要按照从右到左的顺序组合之后再计算，以上数据即为02 63 26 14，转换为十进制就是40,052,244。这是由于CPU在读数据的时候是把数据按先后顺序一个字节一个字节的放入栈中，达到指定的位数以后再识别出来。

“57 41 56 45 66 6D 74 20”，也是Ascii字符“WAVEfmt ”，注意最后一个是一位空格。这部分是固定格式，以后是PCMWAVEFORMAT部分。

“10 00 00 00”，这是一个DWORD，对应数字16，这个对应定义中的PCMWAVEFORMAT部分的大小，可以看到后面的这个段内容正好是16个字节。当为16时，最后是没有附加信息的，当为数字18时，最后多了两个字节的附加信息。

“01 00”，这是一个WORD，对应定义为编码格式（WAVE\_FORMAT\_PCM格式用的就是这个）。

“02 00”，这是一个WORD，对应数字2，表示声道数为2，是个立体声WAV，当值为1时为单声道WAV。

“44 AC 00 00”对应数字44,100，代表的是采样频率44,100次／秒，采样率（每秒样本数）表示每个通道的播放速度。

“10 B1 02 00”对应数字176,400，代表的是每秒的数据量，波形音频数据传送数率，其值为通道数×每秒样本数×每个样本的数据位数/8。播放软件利用此值可以估计缓冲区的大小。

“04 00:”对应数字是4，表示块对齐的内容。数据块的调整数（按字节算），其值为通道数×每个样本的数据位置/8。播放软件需要一次处理多个改值大小的字节数据，以便将其值用于缓冲区的调整。

“10 00”，此数值为16，采样大小为16bits，每样本数据位数，表示每个声道中各个样本的数据位数。如果有多个声道，对每个声道而言，样本大小都一样。

“64 61 74 61”，这个是Ascii字符“data”，表示头结束，开始数据区域。

“F0 25 63 02”，对应十进制40,052,208，是数据区的开头以后的数据总数。

知道了WAV头文件的数据格式，就可以通过程序读取音频文件信息。在上述的基础上定义头文件结构体：

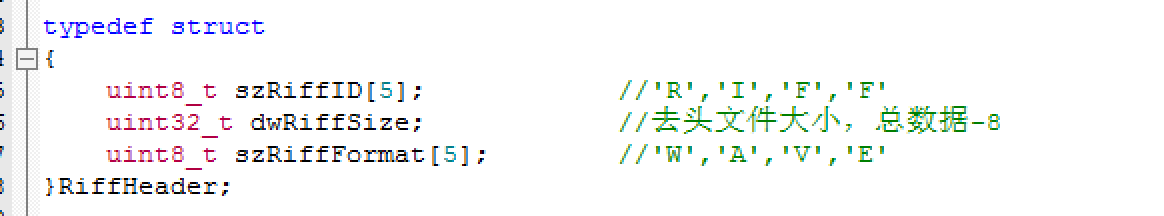


图 3-7 RIFF部分



图 3-8 fmt部分

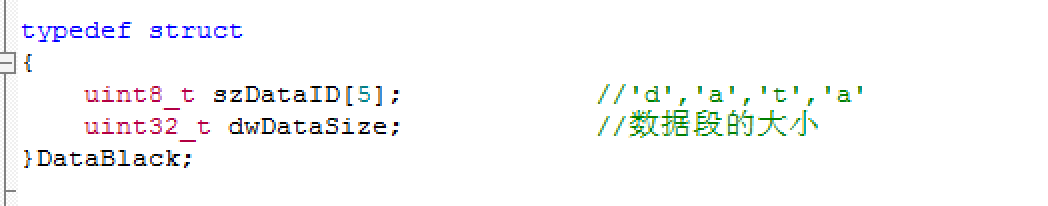


图 3-9 data部分

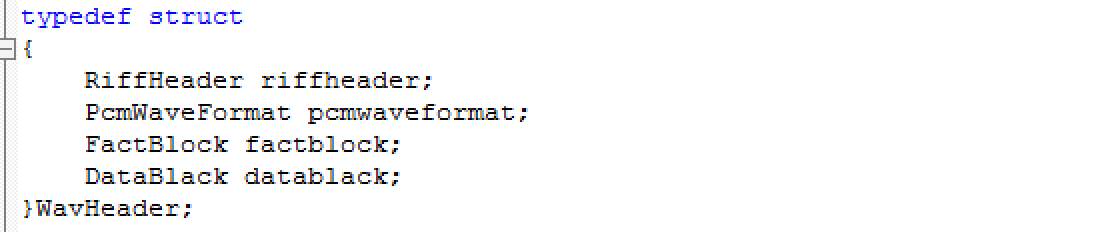


图 3-10 WAV头文件部分

图3-7为经过解码算法解出来的通过串口打印到桌面的头文件信息：



图 3-7 STM32读取WAV头文件信息

可以看到与之前我们分析得出的数据一致。头文件的分析到此为止，下面来分析WAV数据体。

常见的声音文件主要有两种，分别对应单声道（11.025KHz采样率、8Bit的采样值）和双声道（44.1KHz采样率、16Bit的采样值）。对于单声道量化位数为8bits的声音文件，采样数据为8位的短整数；而对于双声道立体声声音文件，每次采样数据位一个16位的整数，高8为和低8位分别代表左右两个声道。

WAVE文件数据块包含以脉冲编码调制（PCM）格式表示样本。WAVE文件是由样本组织而成的。在单声道WAVE文件中，声道0代表左声道，声道1代表右声道。在多声道WAVE文件中，样本是交替出现的。

PCM数据的存放方式如图3-8所示：

样本1

8位单声道

样本2

8位单声道

样本3

8位单声道

样本4

8位单声道

样本1

16位单声道

样本2

16位单声道

样本1

8位左声道

样本1

8位右声道声道

样本2

8位左声道声道

样本2

8位右声道声道

样本1

16位左声道

样本1

16位右声道

图 3-8 PCM数据存放方式

**知道了PCM数据存放方式，以及WAV音频文件的采样频率、量化位数和声道数，我们就可以据此编写WAV音频解码算法了。由于播放部分与文件系统模块相关，因此我们后面再另行讨论。**

3.4 SDIO驱动

在上文中我们介绍了SD卡的基本信息及SDIO各个引脚的主要功能，接下来我们来详细介绍一下STM32的SDIO接口。

#### 3.4.1 SDIO简介

SDIO（Secure Digital Input and Output Card），中文名安全数字输入输出卡，是在SD标准上定义的一种外设接口目前，SDIO主要有两类应用——可移动和不可移动。可移动设备作为Palm和Windows Mobile的扩展设备，用来增加蓝牙、照相机、GPS和802.11b功能。不可移动设备遵循相同的电气标准，但不要求符合物理标准。

SDIO和SD卡规范间的一个重要区别是增加了低速标准。SDIO卡只需要SPI和1位SD传输模式。低速卡的目标应用是以最小的硬件开支支持低速I/ O能力。低速卡支持类似调制解调器、条码扫描仪和GPS接受器等应用。对“组合”卡(存储器+ SDIO)而言，全速和4位操作对卡内存储器和SDIO部分都是强制要求的。

STM32的SDIO接口与标准SDIO有少许区别，区别主要在于数据线的数量。如图3-9，它具有8条数据线——SDIO\_D[7:0]，STM32这样设计是为了支持部分MMC存储卡，这些MMC卡使用的是8条数据线。

图 3-9 STM32的SDIO接口

控制单元

命令通道

数据通道

卡总线

适配器寄存器

FIFO

SDIO\_CK

SDIO\_CMD

SDIO\_D [7:0]

SDIOCLK

HCLK/2

至AHB接口

SDIO适配器

#### 3.4.2 SDIO的命令格式

SDIO的所有命令及命令响应，都是通过SDIO\_CMD引脚来传输的，且命令只能由主机即STM32的SDIO控制器发出。

SDIO协议规定了非常多的命令，把这些命令分类整理也有11种之多，包括基本命令、块读取命令、块写入命令、写保护命令、擦出命令、卡上锁命令、应用指定命令、I/O模式命令、功能选择命令及特殊的应用命令ACMD。其中，在使用ACMD命令前，要首先向卡发送编号为CMD55的应用指定命令。

参照命令格式表，见表3-1。一个命令包含了6个段，分别为起始位、传输位、命令索引、参数、CRC7和结束位，其中除了命令索引段和参数段是需要我们在软件配置的时候设置的，其他段都由硬件完成。命令索引段是指SD协议规定的命令编码，如命令CMD0、CMD1…的编码为0、1…。有的命令会包含参数，如读命令的参数为要读取数据的地址，这些命令参数存放在参数段。

表格 3-1 SDIO命令格式

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 位 | 47 | 46 | ［45:40］ | ［39:8］ | [7:1] | 0 |
| 宽度 | 1 | 1 | 6 | 32 | 7 | 1 |
| 数值 | ‘0’ | ‘1’ | X | X | X | ‘1’ |
| 描述 | 起始位 | 传输位 | 命令索引 | 参数 | CRC7 | 结束位 |

SD卡对主机的各种命令回复称为响应，除了CMD0命令外，SD卡在接收到命令后都会返回一个响应。对于不同的命令，会有不同的响应格式，一共有七种，简称R1～R7.按响应的字节长度又分为长响应型（136bit）和短响应型（48bit）。

见表3-2，这是响应6（R6）的格式。以它为例，R6的主要内容为命令索引段和参数段。

表格 3-2 SDIO命令相应格式（R6）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 位 | 47 | 46 | ［45:40］ | ［39:8］ | | [7:1] | 0 |
| 宽度 | 1 | 1 | 6 | 16 | 16 | 7 | 1 |
| 数值 | ‘0’ | ‘1’ | X | X | X | X | ‘1’ |
| 描述 | 起始位 | 传输位 | 命令索引 | 要重新设置的卡相对地址 | ［15:0］卡状态位：23,22,19,12:0 | CRC7 | 结束位 |

命令段的内容为它响应的命令编码，参数段的内容为命令响应参数。所以当我们需要知道RCA和卡状态时，可以向卡发送CMD3命令，然后等待SD卡对命令的响应。SDIO接口通过CMD信号线接收到响应后，由硬件去除响应的头尾信息，把命令索引段的内容保存到SDIO\_RESPCMD寄存器，把参数段内容存储到SDIO\_RESPx寄存器中，然后通过软件读取这两个寄存器的内容即可获得所需要的信息。

#### 3.4.3 数据传输格式

SD卡的数据写入、读取的最小单位是块，每块的大小为512字节。如图3-10，为多个数据块的写入过程。首先软件通过SDIO接口的CMD信号线发送多块写入的命令，接收到正常的响应后，要写入的数据线从4根DAT信号线传输出去，每块结束后是CRC校验码。接着要检测忙状态，数据传输到SD卡后，SD卡启动内部时序保存数据，这时SD卡会把DAT0信号线拉低，表示处于“忙状态”，忙状态结束后，主机才能发送下一个数据块的数据。

对于SD卡的操作一般是大吞吐量的数据传输，可以采用DMA来提高效率，SDIO采用的是DMA2种的通道4。

command

response

data block

crc

busy

data block

crc

command

response

busy

Multiple block write operation

Block write operation

Data stop operation

CMD

DAT

图 3-10 多个数据块写入时序

#### 3.4.4 SDIO驱动移植

由于SDIO的通信协议相当庞大，要自行编写完整、严谨的SDIO驱动不仅费时费力，而且极易出错。因此在本项目中，我通过移植官方例程中的驱动文件来实现对SD卡的操作。

图3-11为官方提供的SDIO驱动文件

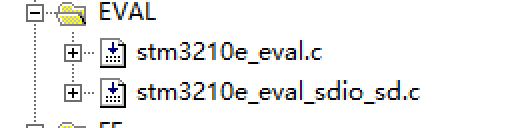


图 3-11 SDIO驱动目录

官方的SDIO驱动文件提供了许多有用的API，通过使用这些API，我们可以很方便的读写SD卡上的数据。下面介绍几个主要的函数：

1. SD\_Init():

在开始阶段调用SD\_Init()来进行SDIO的初始化，同时把返回值附给Status变量，若返回值等于枚举元素SD\_OK，则初始化成功。在SD\_Init()函数中还对纪录了SD卡信息的全局结构体变量SDCardInfo进行了赋值。

1. SD\_WriteMultiBlocks():

调用此函数，可以实现向SD卡中写入多个数据块。同理，SD\_WriteBlock()为写入单个数据块。

1. SD\_ReadMultiBlocks():

调用此函数，可以实现从SD卡中读出多个数据块。同理，SD\_ReadBlock()为读出单个数据块。

使用官方提供的函数，很大的提高了项目的开发速度和可靠性。在接下来的FATfs文件系统移植中，我们就是通过调用这些函数来连接文件系统模块和SDIO驱动，这些我们下面在做详细阐述。

由于本项目中所移植的SDIO驱动来自于STM3210E开发板所提供的Demo工程，而在Demo工程中已经解决了曾在官方原版驱动中的一个会导致程序运行时卡死在循环检测DMA传输结束部分的Bug，因此在这里就不再对此Bug做详细论述。

3.5 FATfs文件系统

#### 3.5.1 FATfs简介

FATfs是一个通用的文件系统模块，用于在小型嵌入式系统中实现FAT文件系统。它完全是由ANSI C语言编写并且完全独立于底层的I/O介质，因此它可以很容易地不加修改地移植到其他处理器当中，例如8051、PIC、AVR、SH、Z80、H8、ARM等。文件系统庞大而复杂，它需要根据应用的文件系统格式而编写，而且一般与驱动层分离开来，以方便移植，所以工程应用中一般是移植现成的文件系统源码。FATfs支持FAT12、FAT16、FAT32等格式，所以利用前面写好的SDIO驱动，再把FATfs文件系统代码移植到工程中，就可以利用文件系统的各种函数，对已格式化的SD卡的文件进行读写了。

#### 3.5.2 FATfs目录结构

在移植FATfs文件系统前，要先到FAT的官网来下载源码，为了稳定起见，此项目并没有使用最新的版本，而是采用了比较常用的R0.09。

移植后的头文件和源代码如图3-12所示：

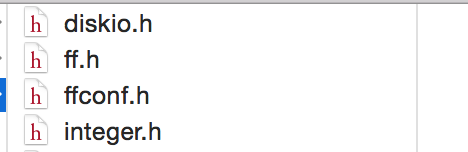




图 3-12FATfs头文件和源代码

其中：

1. integer.h：文件中包含了一些数值类型定义；
2. diskio.c：包含底层存储介质的操作函数。这些函数需要用户自己实现，主要添加底层驱动函数，用来连接FATfs文件系统与SDIO驱动；
3. ff.c：独立于底层介质操作文件的函数，我们就是利用这些函数实现文件的读写；
4. cc936.c：中文字库，时简体中文支持所需要添加的文件，包含了简体中文的GBK和转换函数；
5. ffconf.h：这个头文件包含了对文件系统的各种配置，这也是我们后面配置支持中文和长字节文件名的地方。

#### 3.5.3 为文件系统添加底层驱动

FATfs文件系统与底层介质的驱动分离开来，对底层介质的操作都要交给用户去实现，它仅仅提供了一个接口函数，函数为空，要用户添加代码。因此要把diskio.c中的函数接口与前面的SDIO驱动连接起来。

根据FATfs帮助文档的说明，我们需要提供以下几个函数原型:

1. 存储介质初始化：

DSTATUS disk\_initialize( BYTE drv )

1. 存储介质状态函数：

DSTATUS disk\_status( BYTE drv )

1. 扇区读取函数：

DRESULT disk\_read( BYTE drv, BYTE \*buff, DWORD sector, BYTE count )

1. 扇区写入函数：

DRESULT disk\_write( BYTE drv, const BYTE \*buff, DWORD sector, BYTE count )

1. 其他控制功能：

DRESULT disk\_ioctl( BYTE drv, BYTE ctrl, void \*buff )

这些函数都是操作底层介质的函数，都需要自己实现，FATfs的应用函数就可以调用这些函数来操作我们的SD卡。下面来介绍几个主要的函数实现：

1. 初始化函数接口：

图3-13为文件系统初始化函数接口，文件系统的一些函数会调用这个disk\_initialize()接口来进行底层存储介质的初始化。

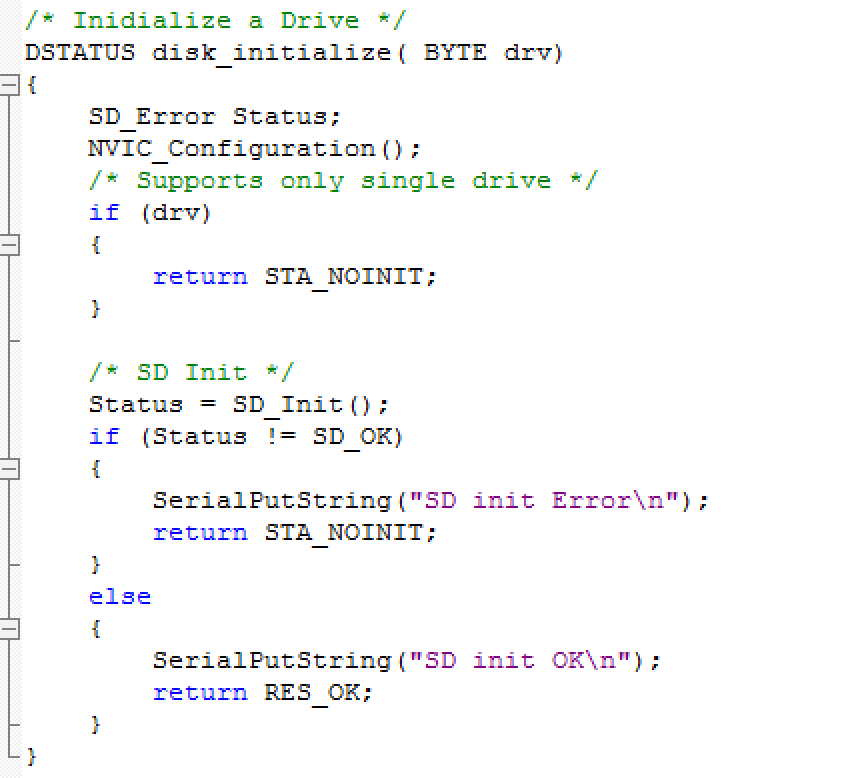


图 3-13 disk\_initialize()接口

这个初始化函数接口调用了SDIO的SD\_Init()函数，返回初始化成功或失败的参数，当文件系统的上层函数调用此函数时，实质调用了SD\_Init()函数对SD卡进行初始化。

1. 扇区读取函数：

接下来时disk\_read()函数，他是文件系统读取SD卡数据会调用的一个函数，所以我们在此加入SDIO驱动的读取函数的接口，如图3-14。

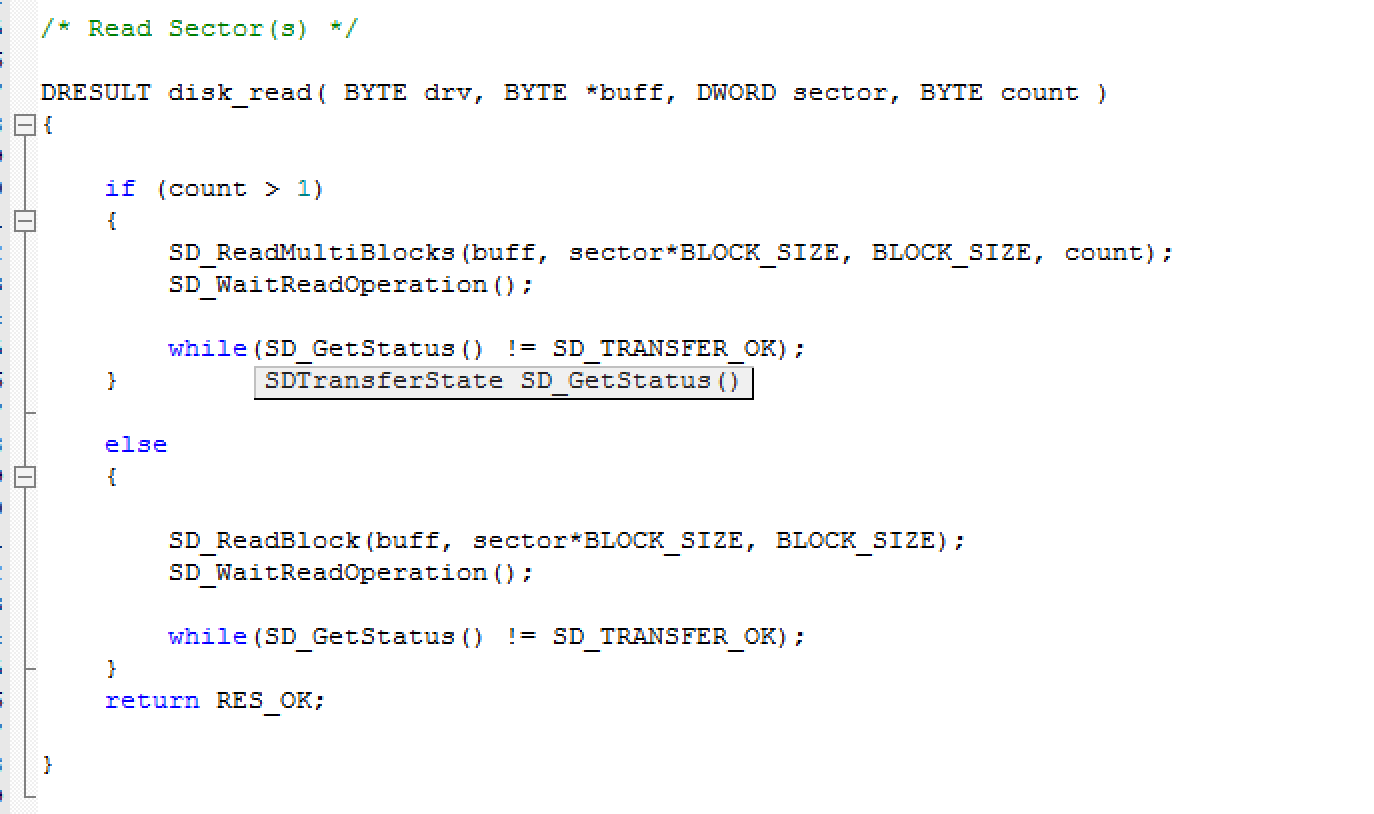


图 3-14 disk\_read()接口

1. 扇区写入函数：

扇区写入函数与扇区读取函数十分类似，也是根据写入的扇区数目是一个还是多个来分别调用不同的SD数据块写入函数。在调用了写入函数之后，也要等待SDIO发送完毕及DMA传输结束。具体代码见图3-15。



图 3-15 disk\_write()接口

#### 3.5.4 添加简体中文和长文件名支持

在使用音乐播放器的时候，我们经常会遇到有些音乐的文件名很长，比如这个：《Janos Starker - Suite No. 1, BWV 1007, In G： Prelude.wav》，而且大多数歌曲都是中文名，比如这个：《久石譲 - 太阳照常升起.wav》，因此，为了让我们的系统支持简体中文和长文件名，就需要对文件进行配置。

在FATfs文件系统里，我们可以在ffconf.h头文件中对文件系统进行剪裁。在这个文件中，修改如下的宏定义，如图3-16。



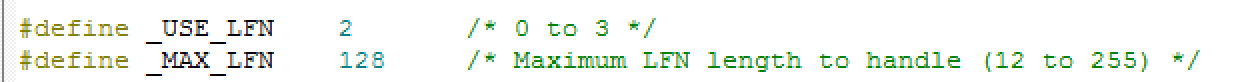


图 3-16 ffconf.h文件中的宏

1. 把\_CODE\_PAGE宏配置成简体中文的code\_page，936。
2. 把\_USE\_LFN宏配置成2，表示开启长文件名支持。
3. 把\_MAX\_LFN宏配置成128，表示可支持的长文件名的最大字节数。

第一个和第三个很好理解，这里要对第二个，也就是\_USE\_LFN宏做一个详细的说明。

图3-17是关于宏\_USE\_LFN各个值的官方定义：

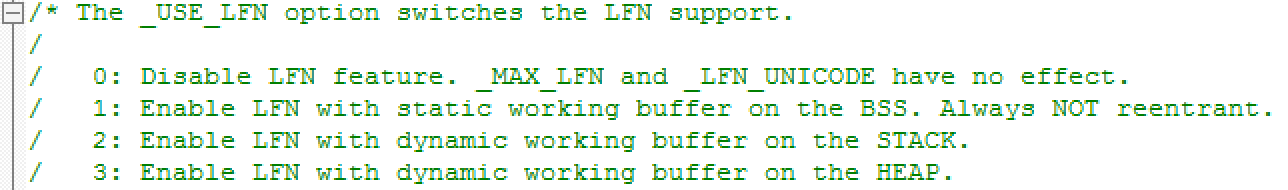


图 3-17宏\_USE\_LFN的官方定义

1表示长文件名的存储在静态存储区。

2表示长文件名的存储在栈区。

3表示长文件名的存储在堆区。

由于栈区存放的内容在子函数调用后会被释放，而且少用全局变量会让代码的移植性更好，因此我们把文件系统长文件名的存储空间放到栈区。

关于软件设计模块的介绍先到此为止，接下来将通过对数字音频播放器系统的分析，由内而外的揭示出它的工作原理。

第四章 系统开发与测试

4.1系统概述

#### 4.1.1 系统概述

数字音频播放器是一个数据流型的软件，在它运行的每时每刻都会有大量的数据被处理。这些数据首先从SD卡中读取出来，然后经过MCU的处理，通过DAC输出到扬声器，最后形成我们所听到的音乐。而如何能够协调有序的实现这一过程，同时还要实现相应的控制功能，则是本章重点讨论的内容。

在音频文件的解码过程中，通常会有两种选择：软解码和硬解码。软解码直接使用CPU来处理数据，而硬解码则通过使用外置解码芯片来处理数据。在查阅了相关资料后，提出了以下两种方案：

1. 在查阅资料的过程中，使用专门的音频解码芯片——也就是硬解码——成为了主流的选择。使用音频解码芯片的好处是可以不必了解复杂的音频解码算法，只需把从文件系统中读取出来的数据直接按顺序输送给芯片即可，后续工作则全部都交给了芯片来完成。然而由于使用了额外的芯片，成本会有相当幅度的增加，并且还需要重新设计电路主板，这对于硬件基础比较薄弱的我而言并不是一个好的选择。
2. 经过与导师的深入交流以及对WAV音频文件的分析后，我发现完全可以通过编写解码算法来实现音频文件的软解码。由于WAV采用PCM编码，因此数据很容易被读取出来，同时播放的音质也非常不错。由于采用的是软解码，主要的难点在于对音频文件的解码算法上，而对于硬件的要求则很少，这正好适合硬件基础比较薄弱的我。

经过对两种方案的考虑，最终选择采用方案二来完成对音频文件的软解码，因此，读取并解码音频文件这一部分将会成为整个项目的设计重点。下面，通过逐一的介绍，我们来一步步地揭开它神秘的面纱。

（由内而外的介绍，按照开发的顺序，先介绍WAV解码算法；然后介绍从带有FAT文件系统的SD卡中读取数据，并于解码算法融合，着重介绍双缓存方式，并配有流程图；最后介绍控制模块的实现、播放列表等）

#### 4.1.2 开发环境

（1）硬件环境：

* + STM3210E-EVAL开发板；
  + J-Link仿真器；
  + 万用表；
  + 扬声器；

（2）软件环境：

* + KeilμVersion5；
  + J-Flash；
  + 串口调试软件：串口调试助手V1.exe；
  + GoldWave；
  + UltraEdit；

4.2 WAV音频解码算法

#### 4.2.1 设计方案

关于WAV音频文件，由于在第三章已经做过介绍，因此在这里就不再赘述。在这一节里我们主要来介绍一下解码算法的实现过程。

我们的解码算法，主要由以下几个函数组成：

1. void WavPlay(WavHeader \*wavheader, char \*Apath)

音频播放的起点，在main函数中调用，执行一首音乐的完整播放，返回值为零。在这个函数中首先执行的是对头文件的初始化，然后执行DAC和中断定时器的初始化，紧接着进入到正式播放的for循环中，通过一边把数据从SD卡中读入缓存，一边通过定时器中断完成DAC音频输出，直到整首音乐播放完毕为止。参数\*wavheader是一个由main函数声明的空的结构体指针，为3.3节中所定义的WAV头文件类型，用来存放即将播放的WAV音频文件的头文件信息；\*Apath是一个字符型指针，存放着即将播放的音频文件的绝对路径，通过这个绝对路径，可以在装有FAT文件系统的SD卡中读取选定文件里的数据。

1. void WavPlayConfig(WavHeader \*wavheader, char \*Apath)

音频播放的初始化函数，在WavPlay函数的起始处调用，主要用来完成对WAV头文件的读取及检测工作，同时完成播放控制和双缓存的初始化工作。在这个函数中，首先调用了f\_open()和f\_read()两个文件操作函数来读取文件的前44个字节的头文件，并存储到header数组中以供头文件解析函数使用；在完成解析函数后， 逐一检测头文件中的RIFF位、WAVE位、fmt位和data位，若符合要求，则头文件解析成功，反之则失败。

1. void WavHeaderInit(WavHeader \*wavheader)

WAV头文件的解析函数，在WavPlayConfig函数中调用。在此函数中，按照WAV头文件的格式，调用WAV\_Info和WAV\_Num两个函数来依次读取存放在header数组中的数据，并逐一填充到\*wavheader结构体中。

1. void WAV\_Info(uint8\_t \*Info, uint32\_t Addr, uint32\_t Num)

从Addr位开始，以一字节为一单位，读取header 数组Num次，并依次放入到\*Info成员中，最后还需要添加一个‘\0’位作为结尾标志。\*Info为需要被填充的头文件结构体成员；Addr为全局变量，表示当前读到头文件成员应该在header数组中的起始地址位，初始化为0，每读完一个头文件的成员，Addr就要增加到下一个成员的起始位，直到读完全部44个字节为止。在这里\*Info指的是‘RIFF’、‘WAVE’、‘fmt ’和‘data’四个成员，因此Num也为4，即读取4次。

1. uint32\_t WAV\_Num(uint32\_t Addr, uint32\_t Byte)

从Addr位开始，以Byte个字节为单位，读取header数组一次，并以整型的类型返回给相应的结构体成员。在这里需要注意的是，由于一次只能读取一个字节，即8bit，因此当Byte大于1时需要重新组合读取的数据，具体操作如图4-1所示：



图 4-1 WAV\_Num函数中数据组合

1. void PrintWavHeader(WavHeader Wavheader)

把已经填充完成的WAV头文件结构体Wavheader通过串口打印到PC上，用于查看当前播放文件的文件属性。

1. void Time\_Init(WavHeader wavheader)

中断定时器初始化函数，用来把wavheader头文件中重要的数据赋值给定时器结构体，初始化并使能中断。在这个函数中主要是把wavheader中的采样频率拿出来，再根据晶振频率来配置定时周期，由于本芯片的晶振频率为72MHz，因此定时器配置如图4-2所示：

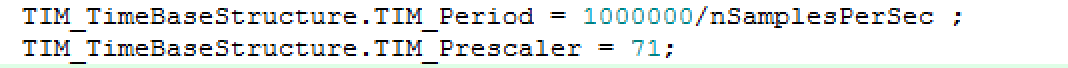


图 4-2 中断定时器定时周期配置

配置完成之后，定时器就会每隔一段时间出发一次定时器中断，而每一次定时器中断就会执行一次中断处理函数。

1. void TIM3\_IRQHandler()

定时器三的中断处理函数，由定时器三触发，实现DAC的音频输出，这个函数也是数字音频播放器的核心函数之一。在完成了之前对头文件数据的读取及初始化以后，就正式进入到了读取播放音频数据的过程。根据第三章所讲到的数据存储格式，我们根据声道数和量化位数的不同来对读入的数据分别进行处理，一次只处理一个音频数据。在第一章介绍声音的基本原理时讲过，声音是由震动产生的，对于扬声器而言，每一次的震动都是由两个不同的电压在切换的时候所产生的，而音频文件中的每个数据都代表着一个电压的相对值，当把这些相对值按照一定的频率来切换时，就可以重现音频文件中所记录的音乐信息。

需要注意的是，由于每一位的缓存都只保存了一个字节——即8bit——的数据，因此在播放16bit的音频文件时需要重新组合数据；同时由于16bit的数据有正负之分，并且内置DAC只有12位，因此还需要做其他处理，具体代码如图4-3所示：

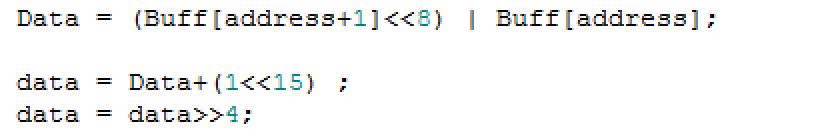


图 4-3 16位数据的处理

下面，将通过流程图的形式来更加直观的展示这一工作流程。

#### 4.3.2 流程图

图4-4为音频解码流程图：

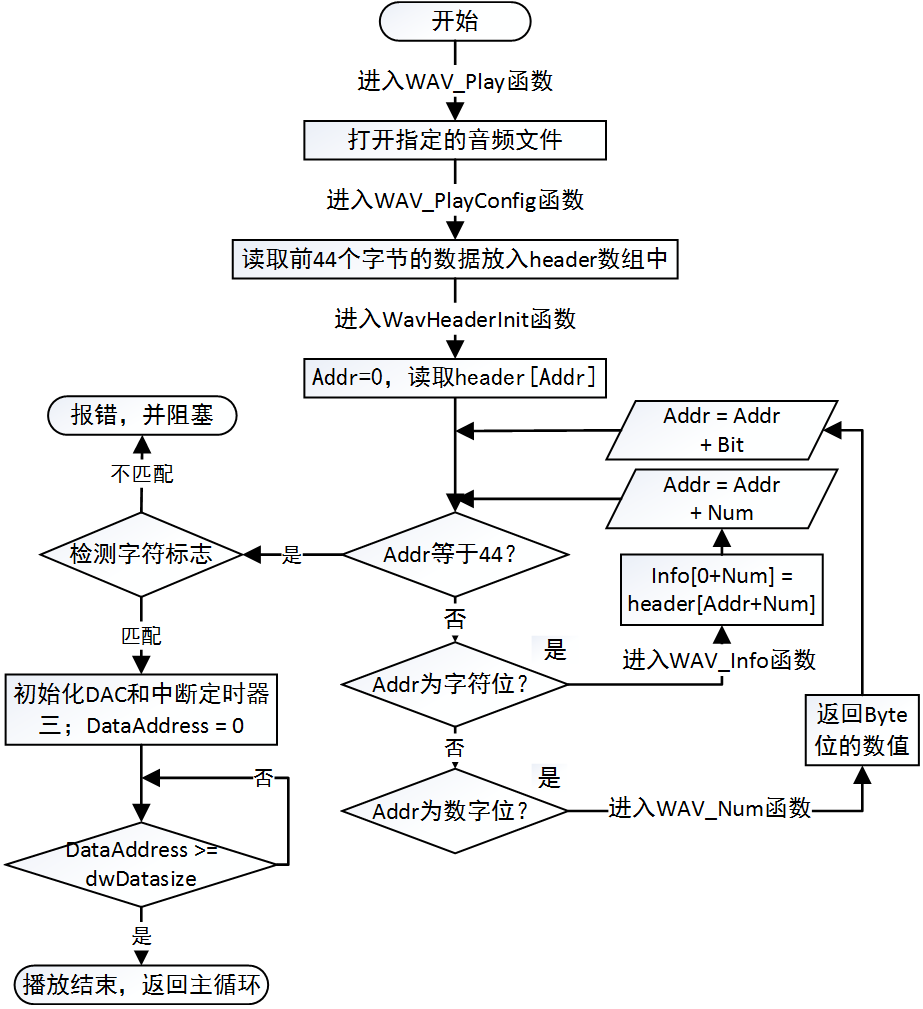


图 4-4 主函数流程图

当完成对DAC和中断定时器三的初始化后，定时器三开始不断触发中断，每触发中断就执行一次中断服务函数，通过中断服务函数来实现DA输出。中断服务函数流程图如图4-5:

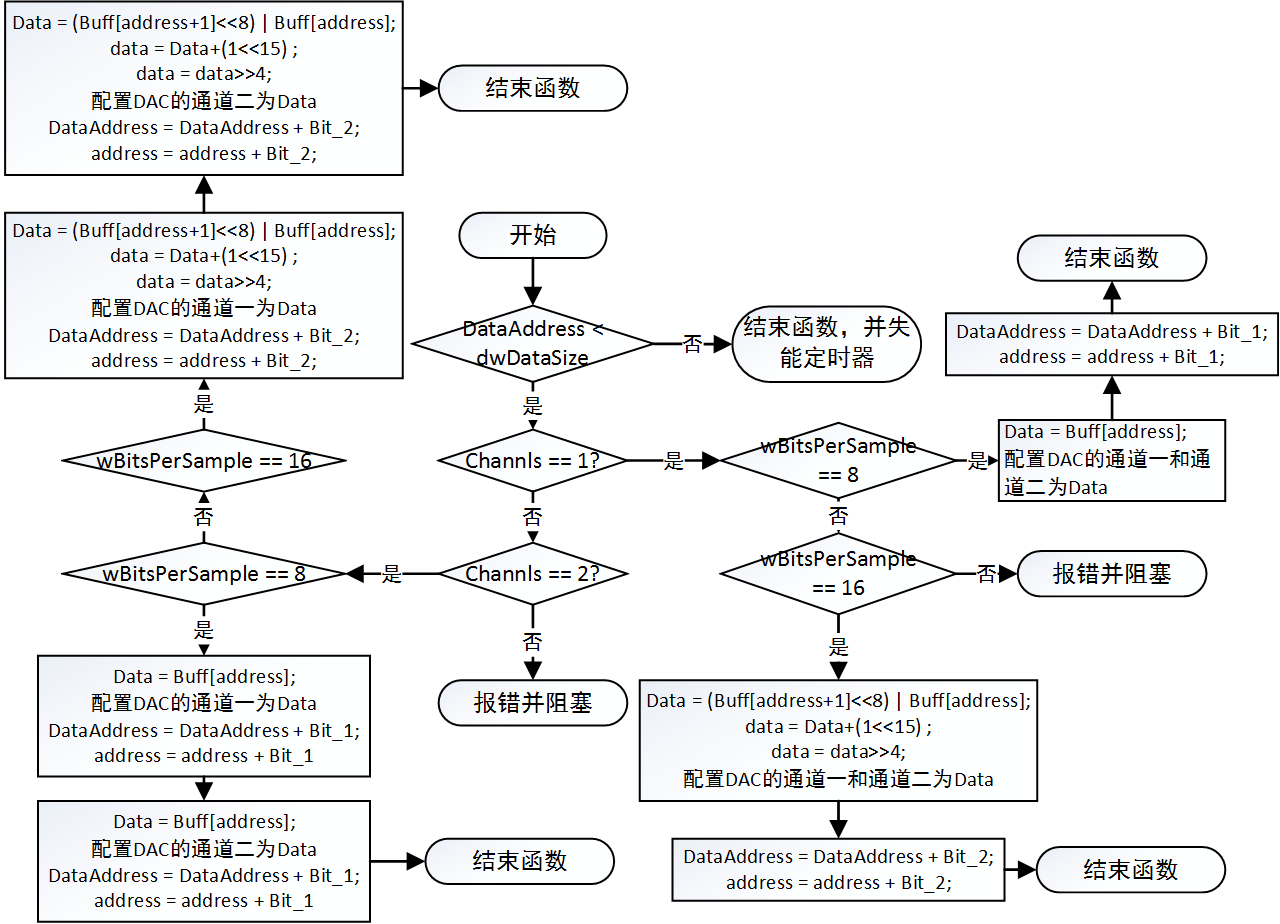


图 4-5 中断服务函数流程图

当DataAddress >= dwDataSize时，失能定时器三，结束定时器中断，同时结束主函数中的for循环，一首歌就这样播放结束了。

4.3 双缓存

#### 4.3.1 概述

在上面的WAV音频解码算法中，当触发定时器中断时，相应的中断服务函数开始以特定的格式从Buff中读取音频数据，经处理后送到DA并输出到扬声器，但由于我们的音频文件数据是存放在具有FAT文件系统的SD卡中，而Buff则是一块在内存中开辟出来的地址空间，因此，在播放之前就需要把SD中的数据先读入到Buff中，然后才能完成后续的播放。

有些人会问了，为什么不直接把SD卡中的数据读入到DAC中呢，那样岂不是效率更高？之所以没有采用这个方法，主要是因为FAT文件系统所致，在使用文件系统的时候，我们通常是通过调用f\_read函数来实现对文件数据的读取，但是，经过测试通过f\_read函数来读取数据的速度远低于直接读取SD卡的速度，而且f\_read中的size的值越大，读取的速度越快，这又是为什么呢？原来在调用f\_read后，首先是一大串需要CPU处理的必要的FAT指令操作，之后才是数据的传输，而由于数据传输是由DMA来完成的，因此整个传输过程的瓶颈就在指令操作那里，当读取的数据流速度低于音频播放的数据流速度时，播放就无法正常进行。因此，就需要调用f\_read读取适量size的数据到Buff中，然后在从Buff中播放，而由于MCU内存的限制，size的值不可能去无限大，因此在本项目中size的值为10\*1024字节。

通过这样的设计，已经可以实现从SD卡中播放音频数据，然而，在调用f\_read从SD卡中读取数据到Buff时，由于读数据需要时间，因此会在整个播放过程中形成一个真空期，Buff中没有正确的数据可被读，这样导致的结果就是在播放的过程中会有明显的卡顿现象。而双缓存的方案，就是为了解决这个问题而出现的。

#### 4.3.2 设计方案

双缓存，顾名思义，就是通过设置两个缓存来实现一边播放一边读取数据。当定时器中断正在播放缓存一中的数据时，调用f\_read往缓存二里写数据，当完成对缓存一中数据播放时，定时器中断就开始播放缓存二中数据，由于缓存二中的数据是紧连着缓存一中的数据，因此音频数据几乎没有受到损失；在播放缓存二的同时，开始调用f\_read往缓存一中写数据。重复此过程，直到音频文件中所有的数据全部被读取为止。

要实现这样的这样的过程，首先需要为每个缓存配置一个状态机，以确定每个缓存所处的状态。图4-6和4-7分别为每个缓存的结构体以及缓存的三个状态：

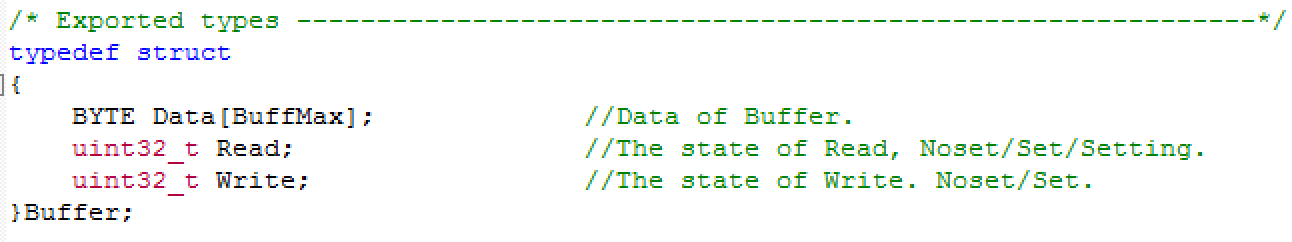


图 4-6 缓存结构体

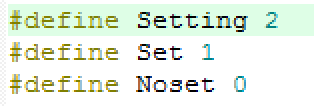


图 4-7 缓存状态

其中Data为缓存空间，char类型，容量BuffMax为之前所定义的的10K字节；状态Read为为此缓存当前读属性的状态，分为Setting、Set和Noset三种，其中Setting表示此缓存正在被播放，Set表示此缓存已被填满正确的数据，Noset表示此缓存中的数据已被播放完成，没有可读数据。图4-8为三种状态之间的转换流程：

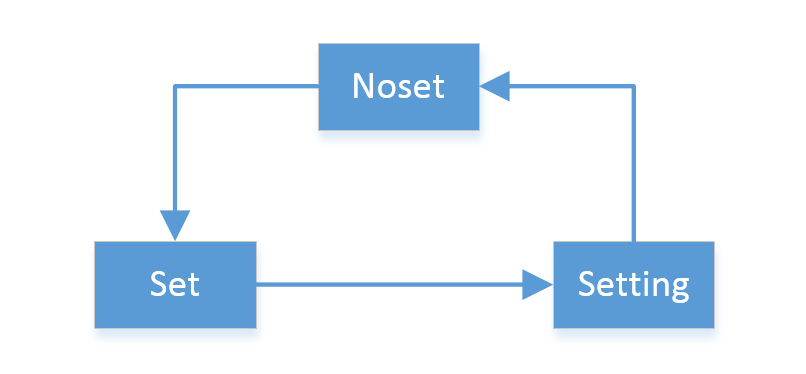


图 4-8

图4-9为声明的双缓存，由于空间过大，因此放在了全局空间中：



图 4-9

Buff为一个char型指针，它指向了当前正在播放的缓存的头地址，这样做的好处是在每次切换需要读取的缓存的时候只需改变Buff指针的地址位即可。

介绍完了这些基本信息后，下面将通过流程图的方式来介绍这一过程。

#### 4.3.2 流程图

图4-9为双缓存的工作流程图：

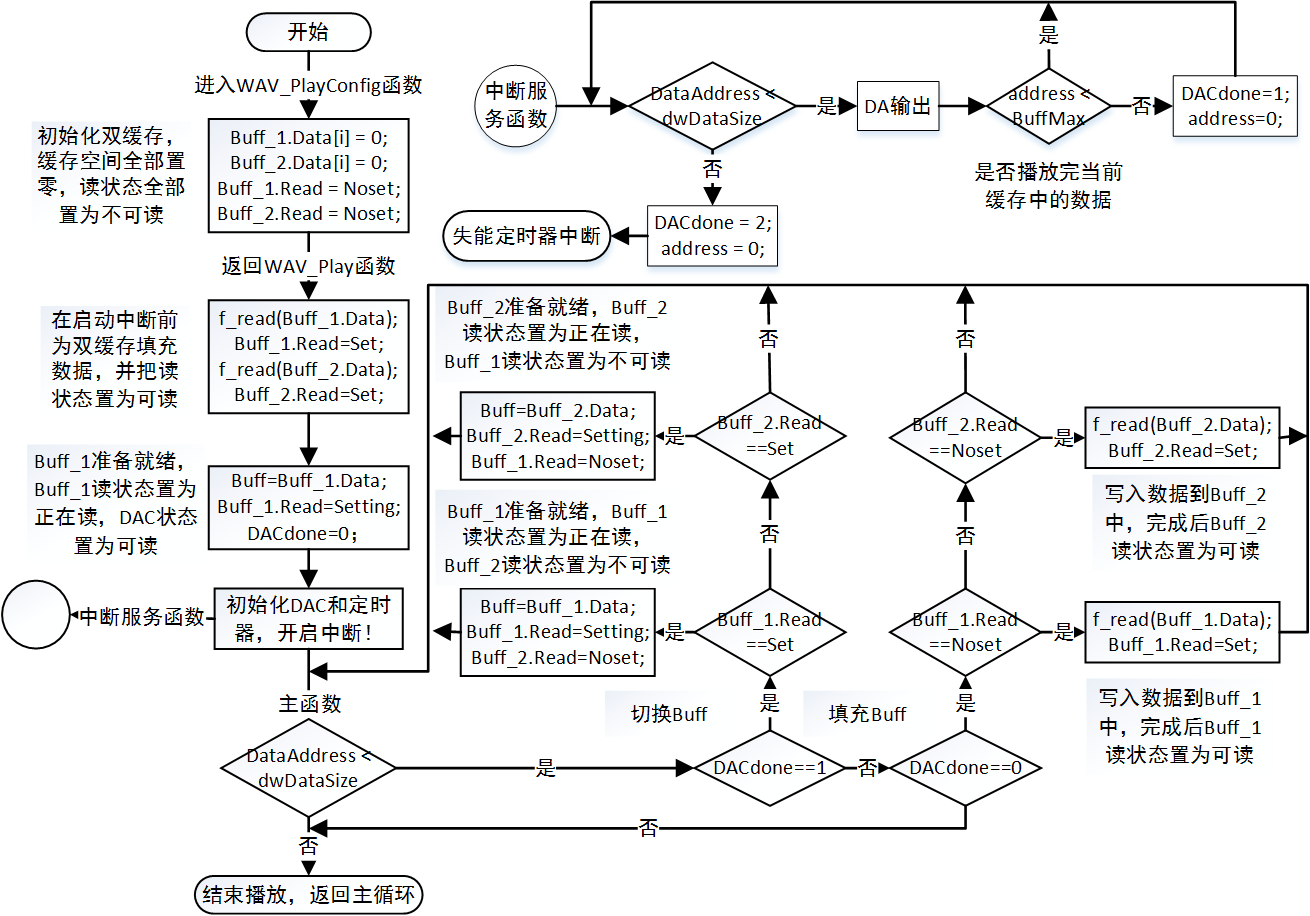


图 4-9 双缓存工作流程

从图中可以看到，双缓存的工作流程与之前的解码流程及播放流程是紧密结合在一起的，这几者相互配合，共同实现了WAV音频文件的读取、解码和播放功能。

4.4 播放控制

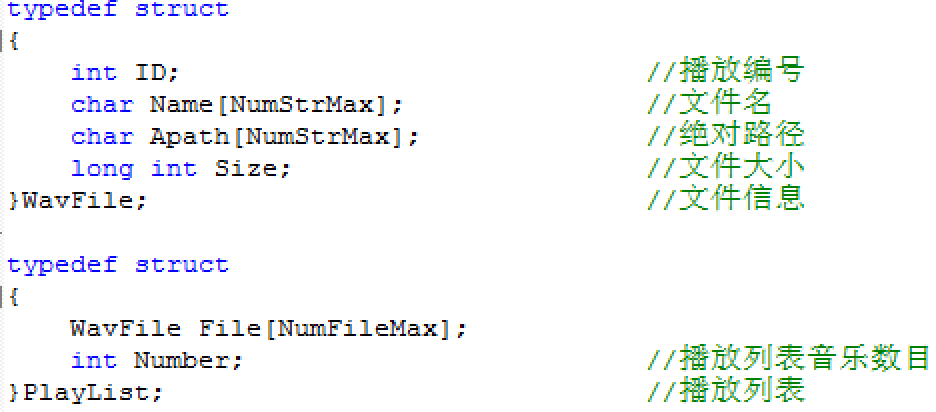
#### 4.4.1 设计需求

在完成了针对WAV文件的音频解码算法后，我们已经可以成功的解码并播放FLASH里的WAV音频文件；在把双缓存模式与带有FAT文件系统的SD卡一同融入到音频解码算法中以后，我们的数字音乐播放器已经具有完整播放一首选定的WAV格式的高品质音乐的能力，这也就意味着本项目的核心部分即驱动层已经基本完成。然而，截止到目前为止，我们对整个系统的控制是通过串口通信的方式来进行的，而要让这个数字音乐播放器可以独立于PC来运行和控制，则需要使用其他的交互方式。因此，我们下一步的工作，就是找到一种新的交互方式来代替串口通信，实现其对整个系统及音乐播放的控制，并优化其内部逻辑，使之可以实现一个播放器应有的基本功能，而最容易想到的，则是使用外部按键来作为我们的交互方式。

#### 4.4.2 方案介绍

STM3210E-EVAL为我们提供了丰富的外部按键，包括一个重启按键Reset；三个自定义按键Wakeup、Tamper、Key；以及一个五向摇杆Joystick，共计九个物理按键，其中Reset键无法定义，Tamper所在的第十三号外部中断通道与Joystick的右键冲突，因此在这里我们使用的是Wakeup、Key、Joystick，共计七个按键来实现操作功能。在上面的硬件设计中我们已经详细介绍了各个按键的工作原理，因此在这里，我们把重点放在如何通过编写程序使用这些按键来实现对系统及音乐播放可控制。

不同于单独播放一首歌曲，一个完整的音频播放器应该能够实现音乐的播放、停止、暂停和继续播放这四个基本功能，同时音量的可控是每个播放器都应该具有的功能，而对所有音乐的上一曲／下一曲操作则更是一个可用的音乐播放器的基本功能。为了实现这些功能，我们首先需要在播放开始前建立一个播放列表，用来存储指定文件夹下每个wav音频文件的编号、文件名、大小和绝对路径，以及音频文件的数量，在这里我们通过建立一个结构体数组来实现这一点，如图4-10所示：



通过调用ShowAllFile()函数来从SD卡中读取指定目录下的文件列表，并放入在主函数中声明的Playlist播放列表中，由于ID编号是从0开始的，因此设置Chose为0来默认选择播放第一首歌。

紧接着便进入主循环中。

为了执行主函数以外的程序，中断是我们最好的选择，因此，这里的七个按键全部采用外部中断来触发相应的中断服务程序。中断服务程序会根据自己的优先级而先后执行，直到执行完成后才把CPU的使用权限交给下一个程序，而当一个中断服务程序占用了太多的CPU资源的时候，整个系统就会因为资源不足而发生卡顿，甚至是死机，俗称跑飞了。因此，为了不妨碍整个系统的运行，中断服务函数所耗时间应该尽可能的短，如图4-10所示：

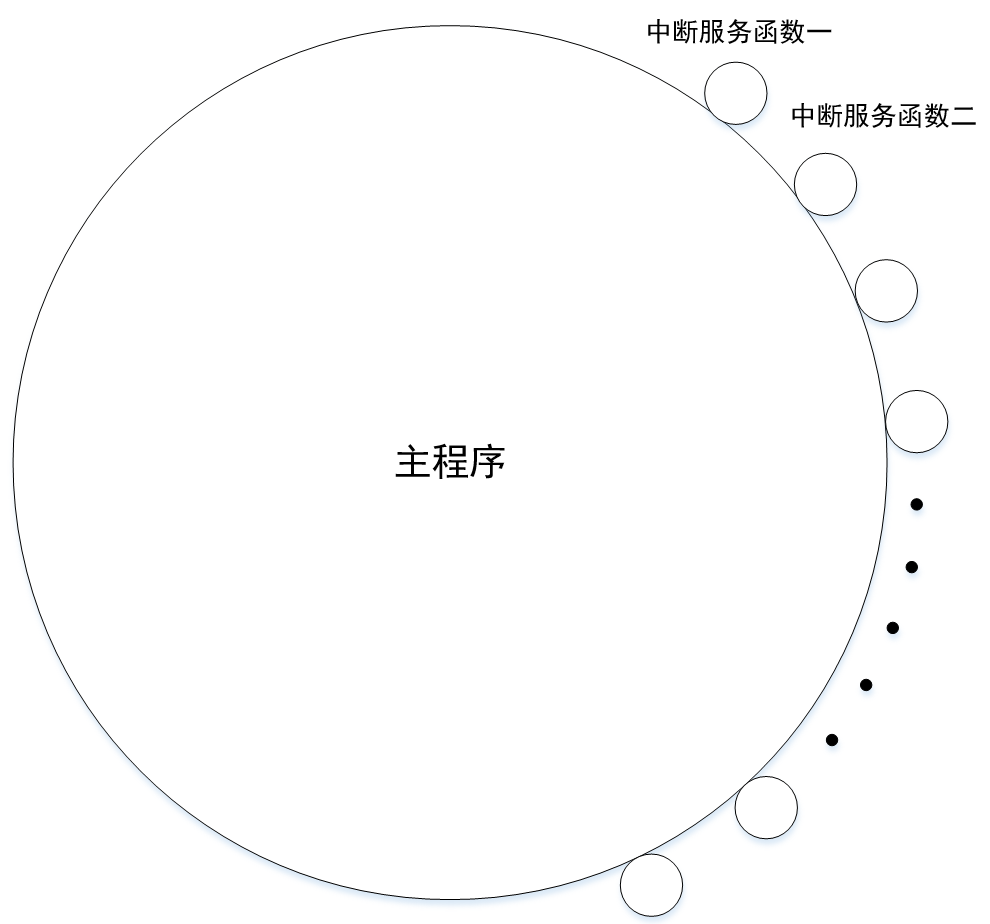


图 4-10 主函数与中断服务函数

由于中断服务函数应该尽可能的短，因此除了那些对时序要求非常高的执行程序外，中断服务函数里的主要过程都应放入主程序中执行，这就需要为每一个中断设置一个缓存，当其中一个中断被触发时，相应的缓存即发生变化，主程序在循环中发现该缓存发生变化后，进入并执行该缓存所对应的中断服务函数，而对时序要求非常高的DA输出部分则全部在中断服务函数中执行。整个播放控制模块便是以这样的模式来运行。

介绍完了播放列表和中断，接下来我们来依次介绍各个功能的实现过程：

1. 开始／停止播放：

在此功能中，我们通过设置PlayIRQBuffer作为开始／停止播放的中断缓存。启动程序后，PlayIRQBuffer的值被初始化为Noset，在初始化结束后进入主循环不断检测PlayIRQBuffer的值，当PlayIRQBuffer的值被由Wakeup按键所触发的中断服务函数修改为Set时，开始播放播放列表中的第一首音乐；当在播放过程中PlayIRQBuffer被由Key按键所触发的中断服务函数修改为Noset，同时DACdone被置为2时，程序跳出播放执行函数，播放停止。

1. 暂停／继续播放：

在此功能中，我们通过设置PlayWait作为暂停／继续播放的中断缓存。启动程序后，PlayWait的值被初始化为Noset，在音乐播放的过程中，当PlayWait的值被由Joystick中的Selection按键所触发的中断服务函数修改为Set时，播放中断定时器失能，播放暂停，PlayWait的值重新置为Noset；当在此触发Selection按键时，PlayWait被修改为Set，播放中断定时器使能，继续播放当前音乐，PlayWait的值又重新置为Noset。

1. 音量加／音量减：

在此功能中，我们通过设置VolumeSub和VolumeAdd作为音量加和音量减的中断缓存，同时设置全局变量volume作为音量的级别，由一到八分为八个等级，初始化为三。启动程序后，VolumeSub和VolumeAdd均被初始化为Noset，当VolumeSub或VolumeAdd的值分别被由Joystick中的UP或DOWN按键所触发的中断服务函数修改为Set时，volume的值便进行相应的增减，然后VolumeSub或VolumeAdd重新被置为Noset，从而达到调节音量的作用。

1. 上一首／下一首：

在此功能中，我们通过设置LastMusic和NextMusic作为上一首和下一首的中断缓存。启动程序后，LastMusic和NextMusic均被初始化为Noset，当LastMusic或NextMusic的值分别被由Joystick中的LEFT或RIGHT按键所触发的中断服务函数修改为Set时，首先停止播放，然后递增或递减全局变量Chose的值，以达到上一曲／下一曲的选择，最后重新进入播放函数，播放新的音乐。当播放到播放列表中的最后一首音乐时，再触发下一曲中断将会跳转到第一曲，同样的，当播放到播放列表中的第一首音乐时，再触发上一曲中断将会跳转到最后一曲。当前歌曲播放完成时，也会调用下一曲的执行函数，自动播放下一曲。

#### 4.4.3 流程图

通过上述的介绍，想必大家已经有了初步的了解，下面将通过流程图的方式，更加通俗易懂的介绍播放流程，如图4-11：

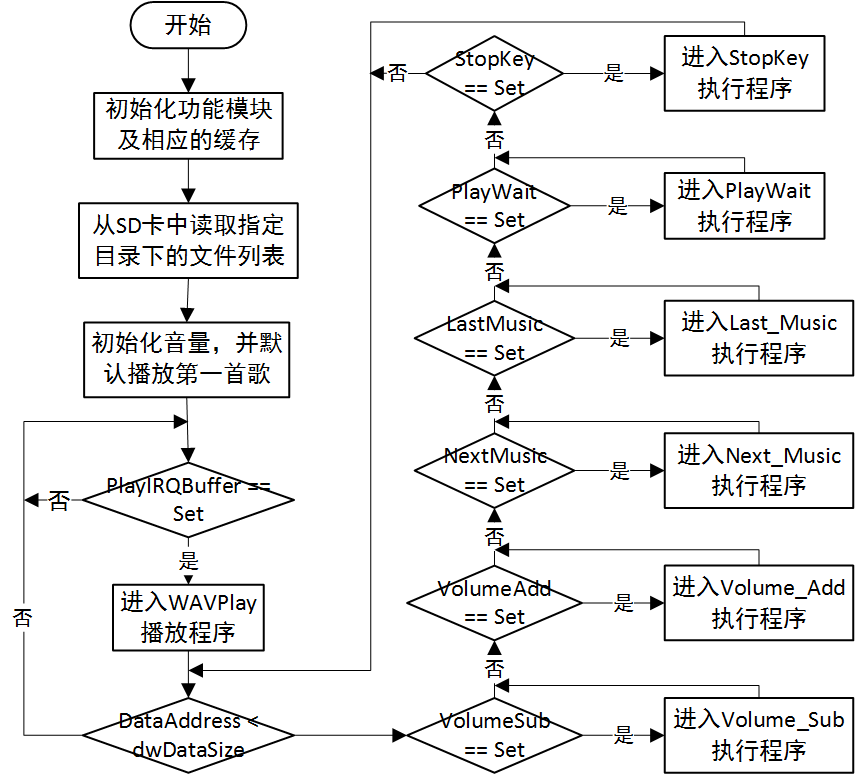


图 4-11 播放流程图

在执行完初始化程序之后，主函数便不断循环等待播放键的触发；进入播放程序后，使能除播放功能以外的所有功能，并在播放循环中等待各个功能键的触发，当某个功能被触发后，主循环进入相应的功能执行函数，之后重新返回主循环中，这样就实现了一个基本的音频播放器。

第五章 总结与展望

5.1 全文总结

自2001年苹果公司发布第一款iPod以后，各式各样的数字音频播放器便层出不穷。飞利浦、索尼、爱国者等国外的大公司相继推出了自己的数字音频播放器，而作为国内最早做数字播放器的厂商，魅族、OPPO等也开始推出具有自己特性的产品。数字音频播放器从出现发展到今天，已经完完全全的改变了人们对于音乐的消费习惯，整个音乐行业发生了翻天覆地的变化，而随着移动互联网的发展，音乐行业迎来了新的变革。

本项目作为公司所给出的一个产品研究性题目，从一开始，便遵循公司导师的要求，严格按照嵌入式系统的开发模式，并以做产品的思路来作为开发导向，以实时操作系统RTOS的设计思想作为设计规范，来进行项目的设计与开发。

在文章中，不仅介绍了声音的基本原理以及把声音数字化的方法，还介绍了用来存储数字音频的WAV文件格式，以及针对WAV文件格式的解码算法；STM32F103作为意法半导体公司所推出的一款基于ARM Cortex-M3的32位微控制器芯片，在此项目中占据了核心地位；而SDIO + FATfs的组合，则为播放器提供了大容量的存储空间以及便捷的文件操作方式。

本项目在设计播放过程的时候，采用了在操作系统中常用的双缓存工作模式，通过对两个缓存分别进行数据的写入与数据的播放，有效的解决无法直接从SD中连续播放音频文件这个问题。

为了更好的介绍本项目的设计与开发，在本文的后半部分，分别以流程图的形式展示了不同模块的工作流程，而从所展示的流程图中可以看出，所有的模块在实际运行中都是相互嵌套，相互协作，共同实现各个功能的。

5.2 改进及下一步工作

#### 5.2.1 系统存在的不足

由于时间的关系，以及对嵌入式系统开发的认识不够，导致此项目还存在着诸多不足之处，具体表现为以下几个方面：

1. 无法正常播放数据流大于0.7Mb/s，即采样频率大于22,050Hz的十六位双声道WAV音频文件。主要原因在于此款MCU，即STM32F103芯片的内存过小，因此所设置的双缓存最大只能为10K，同时由于处理器的时钟频率只有72MHz，在缓存为10K的情况下，从带有FAT文件系统的SD卡中读取数据的速度最快仅为1Mb/s，而数据流小于1Mb/s的且质量最高的只有采样频率为22,050Hz的十六位双声道WAV音频文件，这也就导致目前无法正常播放采样频率为44,100Hz的十六位双声道WAV音频文件；
2. 在播放十六位WAV音频文件时失真率过高。主要原因是由于文件数据是十六位的，而在项目中使用的集成DAC是十二位，因此数据的最后四位在播放过程中丢失，从而导致这个问题。
3. 交互方式不够人性化。由于时间的关系，仅仅采用了七个外部按键来作为唯一交互方式，而用户除了通过听扬声器的方式外并没有其他方式来获取设备信息。

#### 5.2.2 下一步的工作

针对上述已经存在的问题以及公司的要求，在后续的开发过程中将主要集中在以下几个方面：

1. 使用DMA模式来完成DAC的播放功能，降低对CPU资源的使用；
2. 使用开发板上的十六位声卡芯片AK4343来代替内置的DAC，这样可以极大的提高声音播放的质量；
3. 移植libmad库到程序中，使之可以解码并播放MP3格式的音频文件。

致谢

参考文献

1. 苏菲. 数字音频原理与应用. 北京：电子工业出版社，2002.
2. 刘火良，杨森. STM32库开发实战指南. 北京：机械工业出版社，2013.
3. 李云. 嵌入式系统开发之道－－菜鸟成长日志与项目经理的私房菜. 人民邮电出版社. 2011.
4. 周立功. Keil C51使用技巧及实战. 广州：广州致远电子有限公司.
5. 周立功. 基于Stellaris系列ARM的PWM 语音播放器. 广州：广州致远电子有限公司. 2008.
6. STMicroelectronics group of companies. STM32F10xxx参考手册(参照RM0008 Reference Manual Dec. 2009 Rev 10翻译). 意法半导体(中国)投资有限公司. 2010.
7. STMicroelectronics group of companies. 32位基于ARM微控制器STM32F101xx与STM32F103xx固件函数库(参照UM0427 Oct. 2007 Rev 2翻译). 意法半导体(中国)投资有限公司. 2008.
8. FATfs文件系统指南. http://elm-chan.org/fsw/ff/00index\_e.html.
9. STMicroelectronics group of companies. STM32F10xxx Cortex-M3 programming manual Rev 3. STMicroelectronics. April 2010.
10. STMicroelectronics group of companies. How to use the high-density STM32F103xx microcontroller to play audio files with an external I2S audio codec. STMicroelectronics. May 2008.
11. Dan Lavry. Sampling Theory For Digital Audio. Lavry Engineering, Inc. 2004.
12. 马志强，王建刚，孙少林等. 基于STM32的PWM音乐播放器应用设计. 单片机与嵌入式系统应用. 2012年，第12期.