《振南 **znFAT--**嵌入式 **FAT32** 文件系统设计与实现》一书

【上下册】已正式出版发行

全国各渠道全面发售

（在当当、京东、亚马逊、淘宝等网络平台上搜索

关键字"**znFAT**"即可购买，各地实体书店也有售）

此书是市面上 唯一 一套详细全面而深入讲解嵌入式存储技术、**FAT32** 文件系统、**SD** 卡驱动与应用方面的专著。全套书一共 **25** 章，近 **70** 万字。从基础、提高、实践、剖析、创新、应用等很多方面进行阐述，力求通俗，振南用十年磨一剑的精神编著此书，希望对广大工程师与爱好者产生参考与积极意义。

此书在各大电子技术论坛均有**长期的**「抢楼送书活动」，如 21IC、elecfans 等等。

振南的【ZN-X 开发板】是市面上唯一全模块化、多元化的开发板，可支持 51、AVR、STM32(M0/M3/M4)

详情请关注 **www. znmcu. cn** （振南个人主页!!）

第9章

数据读取，纷繁交错：挑战数据读取赛程中的繁杂逻辑

上一章实现了带有簇链的文件数据读取，可以说，加入了 FAT 簇链机制，才算是抓住了 FAT32 的精髓。同时，我们也针对数据读取提出了诸多的问题。本章就专门围绕文件数据读取功能的深化、细化以及具体实现上的方法、技巧展开探讨，将初步实现 znFAT 中的第二个用户级 API 函数——文件数据读取(znFAT_ReadData)。这个函数对簇项簇链等内部操作细节进行封装，为使用者提供了一个简单而强大的功能函数接口。它可以实现对文件任意位置任意长度的数据进行读取，解除了按整扇区或整簇进行数据读取的限制。最后将使用“分步式”的方式来完成 SD 卡 MP3 数码相框实验。这章还将引出新的实验——汉字电子书实验。下面就来看本章的精彩内容吧。

9.1 让数据读取更精细：数据的分层与剥离

1. 关于无效数据

在带簇链的 SD 卡 MP3 数码相框实验中，我们使用的数据读取方式是按整簇进行读取的。也就是说，每次获得一个新的簇，就把这个簇中的所有数据一下子全部写入到 TFT 液晶和 VS1003 中去显示、播放，如图 9.1 所示。

其实，这里面是有问题的。实验中使用的 TEST.MP3 的文件大小为 1 637 901 字节，SD 卡上的簇大小为 8 个扇区，即 4 096 字节。那么这个文件一共占用了 1 637 901/4 096≈399.9 个簇，从后面的小数就可以知道，它最后是有一些不足整簇的数据余量的。也就是说，最后一个簇只占用了一部分，后面的存储空间被浪费掉了(讲簇的时候我们说过，一个簇被占用之后，没有用到的存储空间也不能再被其它文件使用了，而是处于闲置的状态)。所以，实际上这个文件占用了 400 个簇。在 Windows 的文件属性对话框中可以看到，有“大小”与“占用空间”这两项，如图 9.2 所示。

从图中就可以看到，TEST.MP3 文件的“占用空间”为 1 638 400 字节，整整是 400 个簇。最后一个簇中那些并不属于此文件的数据(即最后被浪费掉的存储空间中的数据)，我们就称之为无效数据，如图 9.3 所示。

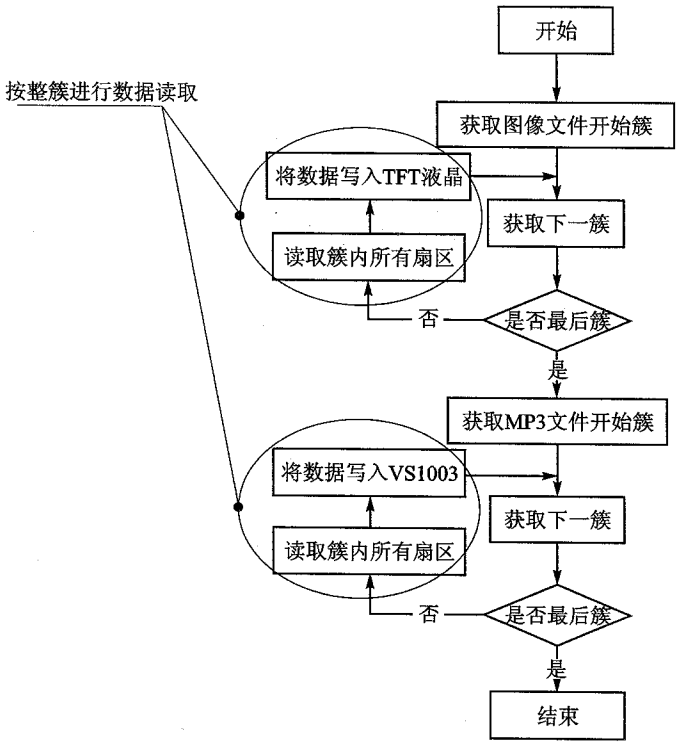

图 9.1　带簇链的 SD 卡 MP3 数码相框实验中按整簇读取数据

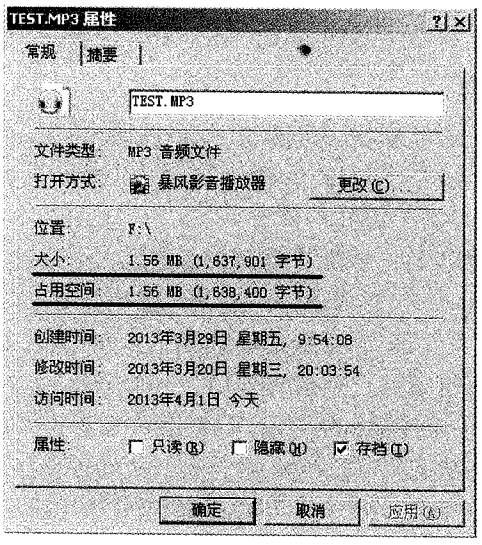

图 9.2　TEST.MP3 文件属性对话框的文件大小与占用空间

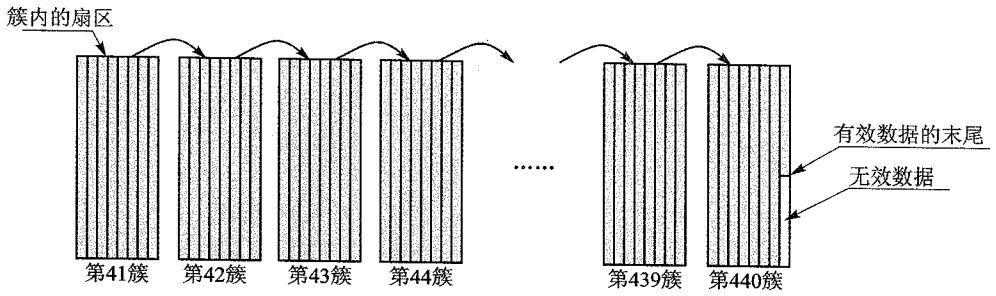

图 9.3 TEST.MP3 文件末簇中的无效数据

回顾前一章的按整簇读取数据的代码，可以看到，虽然 FAT 簇项的属性值可以告诉我们当前簇是不是最后一个簇（看它是否等于 0X0FFFFFFF）、数据读取操作是否应该停止，但是却不能告诉我们最后一个簇中的数据有多少是真正属于文件的、哪些是无效的。

看到这里，可能有些读者会产生这样的疑问："前面实验中确实存在一部分无效数据，但是在图像的显示以及音乐的播放效果中并没有看到异样啊？"确实，看上去似乎影响不大，但如果仔细观察就会发现图片的显示是有问题的。我们使用的 TFT 液晶显示器的分辨率为 320×240，色彩为 16 位色（RGB565），所以图像文件 picture.bin 的数据大小为 320×240×2＝153 600 字节，一共占用了 37.5 个簇（实际上是 38 个簇），无效数据长度为 2 048 字节。这部分多余的数据在写入到 TFT 液晶之后，会反过来将图像前面的若干个像素冲掉（具体是 2 048/2＝1 024 个像素，即 1 024/240≈4.3 行）。更形象的说明，请看图 9.4。

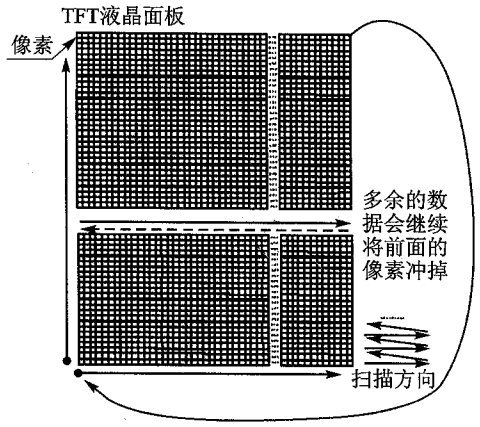

图 9.4 无效数据对图像造成影响的原因

"那 MP3 的播放没有问题，又如何解释呢？它听起来确实没有杂音，VS1003 也没有出现异常！"这只能说明我们使用的 MP3 解码芯片的鲁棒性比较好（鲁棒性是说一个系统或者产品能够承受错误输入的能力）。说白了，就是我们向 VS1003 写入错误的 MP3 音频数据，它也不会崩溃。或者虽然有异样，但因为数据量比较小，所以根本听不出来。总之，从原则上说，不属于文件的数据是根本不应该被读到的。

2. 数据的分层与剥离

如何才能知道最后一个簇中的数据哪些是有效的呢？其实很简单，用文件大小

来判断即可。我们对按整簇读取 MP3 数据的代码进行改进(_main.c):

```
Cur_Clust = fi2.File_StartClust;                    //获取 TEST.MP3 文件的开始簇
nCluster = fi2.File_Size/Cluster_Size;              //计算整簇数
for(i = 0;i<nCluster;i ++ )                         //读取整簇数据"簇"
{
   temp = SOC(Cur_Clust);                           //获取簇开始扇区
   for(j = 0;j<(Init_Args.SectorsPerClust);j ++ )   //读取整簇数据
   {
    SD_Read_Sector(temp + j,buffer);                //读取 SD 卡的 MP3 数据扇区
    SET_VS_XDCS(0);
    for(k = 0;k<512;k ++ )
    {
   VS_Send_Dat(buffer[k]);                          //将 MP3 数据写入 VS1003
  }
    SET_VS_XDCS(1);
  TFT_Update_ProgressBar();
   }
   Cur_Clust = znFAT_GetNextCluter(Cur_Clust); //获取下一簇
}

nSector = (fi2.File_Size % Cluster_Size)/(Init_Args.BytesPerSector);
                                                    //计算最后簇中的整扇区数
temp = SOC(Cur_Clust);                              //获取最后簇开始扇区
for(j = 0;j<nSector;j ++ )                          //读取最后簇中的整扇区数据"扇区"
{
   SD_Read_Sector(temp + j,buffer);
   SET_VS_XDCS(0);
   for(k = 0;k<512;k ++ )
   {
  VS_Send_Dat(buffer[k]);                           //将 MP3 数据写入 VS1003
 }
   SET_VS_XDCS(1);
 TFT_Update_ProgressBar();
}
nByte = (fi2.File_Size % (Init_Args.BytesPerSector)); //计算最后不足扇区的字节数
SD_Read_Sector(temp + j,buffer);                    //读取最后一个 MP3 扇区
SET_VS_XDCS(0);
for(k = 0;k<nByte;k ++ )                    //将最后若干字节的有效数据取出"字节"
{                                           //将有效的若干个字节从扇区中"剥离"出来
 VS_Send_Dat(buffer[k]);                    //将 MP3 数据写入 VS1003
}
```

```
SET_VS_XDCS(1);
TFT_Update_ProgressBar();
```

上面的程序不再使用 do while 循环＋结束簇检测的方式，而是使用了 3 个 for 循环，分别对应于数据读取的 3 个层次：整簇、末尾整扇区、剩余字节。可以看到，在处理最后一个扇区的剩余字节时，先是将整个扇区读入缓冲区中，然后再将有效数据与无效数据进行剥离。数据剥离的结果使我们得到了更为精细的、字节级的数据。

9.2　数据读取函数的实现

我们希望簇链操作以及数据读取的细节对于使用者来说是透明的。其实，振南的 znFAT 作为一个最终供使用者使用的 FAT32 文件系统方案，本来就应该把这些内部的东西进行屏蔽。很多情况下，使用者其实并不关心什么是簇、什么是扇区，也不关心具体如何去读取数据，大多是迫于工作和项目中有文件数据读取或其他文件操作的功能需求才来研究文件系统的，他们希望有现成的函数或功能模块能够直接调用，简单易行地从文件中读到自己想要的字节。（znFAT 是振南经过长期对 FAT32 的研究而研发的一个文件系统方案，为的就是方便人们轻松地实现文件操作。在项目完工或功能达标时来了解一些 FAT32 的相关技术与思想、内部的具体实现与细节，必定会收益良多。）

本章的主要任务就是实现数据读取功能函数（znFAT_ReadData），使用者只需要告诉这个函数要读取数据的开始位置以及数据长度，它就会完成数据读取操作，并将读到的数据放到指定的缓冲区中，等待进一步的处理和应用，如图 9.5 所示。

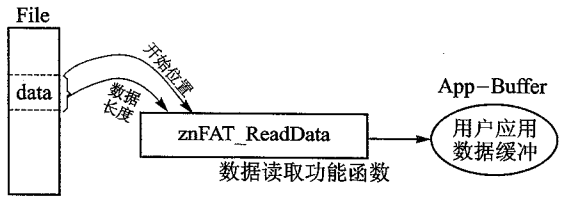

图 9.5　数据读取函数的功能示意

数据读取函数是本书中出现的第一个较为复杂的功能函数；倒不是说它很难，主要是因为我们要把它想得很细致、很全面，实现过程也会比较繁琐。为了能让大家更好地理解实现过程，下面还是以由易到难的方式来进行讲解。

9.2.1　初步实现

其实把上面分层次进行数据读取的过程封装成一个函数，就是数据读取函数的雏形，不同点就在于数据读出之后不是直接写入到 TFT 液晶或 VS1003 中，而是放入一个指定的数据缓冲区中。

在解析 MBR、DBR、文件目录项以及簇项等功能部分的时候，代码里都使用了一个缓冲区(buffer)，用于装载功能扇区数据。但是在读取数据时，buffer 又用于存储从数据区扇区中读出的数据。我们不希望 buffer“身兼数职”，只希望其专门用于 znFAT 的内部功能实现，如参数解析、簇项获取等，而不再去掺和数据缓冲的事情，所以将 buffer 改名为 znFAT_Buffer，并称之为内部缓冲区。那读出来的数据应该放在哪里呢？答：用户应用数据缓冲区，如图 9.5 中的 App_Buffer，它是由使用者自行定义的应用层用户缓冲区，用于装载 znFAT_ReadData 函数读出的数据。这里将内部缓冲区与用户应用数据缓冲区进行了分离，这将使数据读取的效率得以较大的提升。

下面就对数据读取函数进行初步的实现(znFAT.c)：

```
UINT32 znFAT_ReadData(struct FileInfo * pfi,UINT32 len,UINT8 * App_Buffer)
{
 UINT32 Cur_Clust = 0,nCluster = 0,nSector = 0,nByte = 0;
 UINT32 i = 0,j = 0,k = 0,temp = 0,counter = 0;
 UINT32 Cluster_Size = (Init_Args.SectorsPerClust * Init_Args.BytesPerSector);
                                                    //计算簇大小
 Cur_Clust = pfi->File_StartClust;                  //获取文件开始簇
 nCluster = len/Cluster_Size;                       //计算整簇数
 for(i = 0;i<nCluster;i++)                          //读取整簇数据
 {
    temp = SOC(Cur_Clust);                          //获取簇开始扇区
    for(j = 0;j<(Init_Args.SectorsPerClust);j++)    //读取整簇数据
    {
     SD_Read_Sector(temp + j,App_Buffer + counter);  //读取数据放入应用数据缓冲区中
     counter + = (Init_Args.BytesPerSector);
    }
    Cur_Clust = znFAT_GetNextCluter(Cur_Clust);     //获取下一簇
 }
 nSector = (len % Cluster_Size)/(Init_Args.BytesPerSector); //计算最后簇中的整扇区数
 temp = SOC(Cur_Clust);                             //获取最后簇开始扇区
 for(j = 0;j<nSector;j++)                           //读取整扇区数据
 {
    SD_Read_Sector(temp + j,App_Buffer + counter);
    counter + = (Init_Args.BytesPerSector);
 }
 nByte = (len % (Init_Args.BytesPerSector));        //计算最后不足扇区的字节数
 SD_Read_Sector(temp + j,buffer + counter);
 return (counter + nByte);                          //返回读到的数据长度
}
```

函数的 3 个形参分别是：文件信息集合指针、要读取的数据长度以及应用数据

缓冲区指针。从这里也可以看到前一章对文件信息进行封装的重要意义。通过一个 struct FileInfo 结构体指针，可以很方便地将一个文件的所有参数信息引入到功能函数中来，从而实现对某一特定文件的操作。实际上就是给函数传哪个文件的信息集合指针，它就读哪个文件的数据。

9.2.2　SD 卡 MP3 数码相框的分步式实现

有了数据读取函数，就可以提出分步式的思想了。在前面的实验中，文件数据的读取以及向 TFT 液晶、VS1003 的写入等操作都是交织在一起的。而分步式的实现方式是先将文件的数据读入到应用数据缓冲区中，然后再将缓冲区中的数据写入到 TFT 液晶和 VS1003，具体实现过程请看图 9.6。

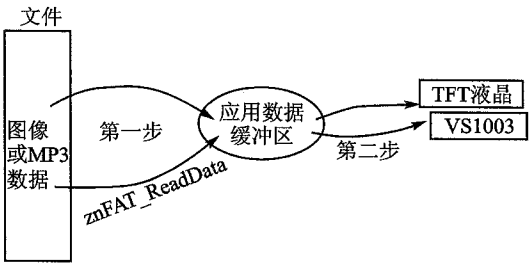

图 9.6　分步式实现方式的示意图

分步式 SD 卡 MP3 数码相框实现代码如下：

```
//对外部变量进行声明
extern struct  znFAT_Init_Arg  Init_Args;
extern UINT8 znFAT_Buffer[512];                    //内部缓冲区
struct FileInfo fi1,fi2;                           //定义两个文件信息集合分别用于装载
                                                   //picture.bin 与 TEST.MP3 的文件信息
UINT8 App_Buffer[2000000];                         //用户应用数据缓冲区 2 MB
void main(void)
{
 UINT32 i = 0,len = 0;
 //相关器件初始化
 znFAT_Init();
 UART_Send_Str("znFAT Init..\r\n");
 SD_Read_Sector(Init_Args.FirstDirSector,znFAT_Buffer); //读取首目录簇开始扇区
 Analyse_FDI(&fi1,&((((struct FDIesInSEC *)znFAT_Buffer)->FDIes)[1]));
                                                   //解析第 1 个 FDI,对应于 PICTURE.BIN
 Analyse_FDI(&fi2,&((((struct FDIesInSEC *)znFAT_Buffer)->FDIes)[2]));
                                                   //解析第 2 个 FDI,对应于 TEST.MP3
 //第一步：读取文件数据到用户应用缓冲区
```