《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7章

层递删截，通盘格空：文件、目录的删除及磁盘格式化

使用znFAT实现一些定时周期性的数据存储功能时，有人又提出了这样一个问题："SD卡的容量终归是有限的，文件数据写满之后，能不能把前面的数据或文件删掉，再继续写入数据呢？"这一问题揭示了znFAT在功能上的欠缺。我们还需要实现数据、文件和目录的删除功能，其中目录的删除较有难度。此时，振南要问一个问题："如何清空磁盘上的所有数据？"有人会说："znFAT不是有通配功能吗？挨个删除就行了！"非也。这种情况下，格式化将比删除来得更直接、更便捷。其实格式化不光可以清空磁盘，它还是我们基于FAT32进行各种文件操作的重要前提。它的工作就如同在磁盘上"画格子"，使其符合FAT32协议标准。但是现在对磁盘的格式化，我们都是借助计算机来完成的，znFAT本身并没有格式化功能。因此，格式化功能的实现将标志其在功能上进一步完备，自成体系。好，请看本章正文。

7.1 文件数据的倾倒

7.1.1 何为数据倾倒

如果把一个存有数据的文件看作是一桶水的话，那么对文件数据的删除就如同倾倒桶中之水，如图7.1所示。"倒水"有一个显著的特点，即我们永远都只能倒出水上方到液面的部分，不可能直接倒出中间的部分。我们这里要实现的数据删除功能，也有此意，如图7.2所示。

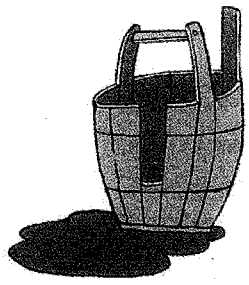
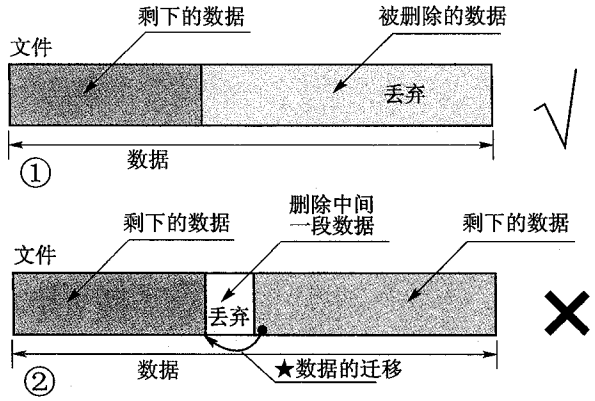


图7.1 删除文件数据如同倾倒桶中之水

我们要按照图7.2中的第一种情况来实现数据删除功能，为什么呢？我们可以看到，第二种情况删除的是文件中间的一段数据，这将引发"数据的迁移"，即把后面的数据全部复制拼接到前面来，工作量较大，效率也比较低下，而且还要耗费更多的内存资源。另一方面的原因是在实际



注：① 它将文件拉腰斩断，留前去后。★★我们实现这一项★★
② 删除文件中间一段，引发后面数据的迁移，效率低下！

图 7.2　znFAT 中的数据删除只实现第一种情况

应用过程中我们基本上都不会只删除文件中间的一段数据，大多都是如第一种情况那样删除文件中间某一位置后面的所有数据，从而解决磁盘写满的问题，誊出空间以便继续向文件写入数据。既然我们要实现的数据删除功能有“倾倒”之意，那就把它对应的函数起名为 znFAT_Dump_Data(dump 意为抛弃、倾倒)。

7.1.2　数据倾倒的实现

前面我们说过，FAT 表及簇链基本贯穿于所有的文件操作中，数据的删除也不例外，其实质就是簇链的销毁。一个完整的系统，要有收有放，有入有出。文件创建及数据写入时对簇链的构造就是“收入”，此处要讲的簇链的销毁就是“放出”。

簇链的销毁在实现上比较简单，就是把簇链上的所有 FAT 簇项都清零即可，代码如下(znFAT.c)：

```
UINT8 Destroy_FAT_Chain(UINT32 cluster)
{
 UINT32 next_cluster = 0;
 do
 {
  next_cluster = Get_Next_Cluster(cluster); //销毁前先将下一簇记录下来
  Modify_FAT(cluster,0); //将簇项清零
  cluster = next_cluster; //将下一簇赋给当前簇
 }while(! IS_END_CLU(cluster)); //如果不是最后一个簇，则继续循环
 return 0;
}
```

上面寥寥几行代码就完成了簇链的销毁，但是就像前面讲预建簇链时候一样，这种频繁调用 Modify_FAT 函数的实现方式效率是很低的，所以改为下面这种实现方

式(znFAT.c):

```
UINT8 Destroy_FAT_Chain(UINT32 cluster)
{
 UINT32 clu_sec = 0,temp1 = 0,temp2 = 0,old_clu = 0,nclu = 1;
 struct FAT_Sec * pFAT_Sec;
 if(cluster<(pInit_Args->Free_Cluster))
                //如果要销毁的簇链开始簇比空簇参考值小,则将空簇赋值为它
{
  pInit_Args->Free_Cluster = cluster;
 }
 old_clu = cluster;
 znFAT_Device_Read_Sector((old_clu/128) + (pInit_Args->FirstFATSector),
                                  znFAT_Buffer); //计算开始簇项所在的 FAT 扇区
 pFAT_Sec = (struct FAT_Sec * )znFAT_Buffer;
            //将内部缓冲区地址强转为 FAT 扇区结构指针,以便对簇项进行操作
 cluster = Bytes2Value(((pFAT_Sec->items)[cluster % 128]).Item,4);
                                                    //计算开始簇的下一簇
 while(! IS_END_CLU(cluster)) //如果当前簇不是簇链的最后一个簇
{
  nclu++ ; //统计簇链包含的总簇数
  clu_sec = cluster/NITEMSINFATSEC; //计算当前簇项所在的 FAT 扇区
  temp2 = old_clu/NITEMSINFATSEC; //计算上一簇项所在的 FAT 扇区
  temp1 = old_clu % NITEMSINFATSEC; //计算上一簇项所在 FAT 扇区内的位置
  ((pFAT_Sec->items)[temp1]).Item[0] = 0; //将上一簇项清零
  ((pFAT_Sec->items)[temp1]).Item[1] = 0;
  ((pFAT_Sec->items)[temp1]).Item[2] = 0;
  ((pFAT_Sec->items)[temp1]).Item[3] = 0;
  if(temp2! = clu_sec) //如果当前簇项与上一簇项所在的 FAT 扇区不是同一扇区
  {
   znFAT_Device_Write_Sector(temp2 + (pInit_Args->FirstFATSector),
                                  znFAT_Buffer); //回写上一簇项所在 FAT 扇区
   znFAT_Device_Write_Sector(temp2 + (pInit_Args->FirstFATSector
                                  + pInit_Args->FATsectors),znFAT_Buffer);
   znFAT_Device_Read_Sector(clu_sec + (pInit_Args->FirstFATSector),
                                  znFAT_Buffer); //读取当前簇项所在 FAT 扇区
  }
  old_clu = cluster;
  cluster = Bytes2Value(((pFAT_Sec->items)[cluster % 128]).Item,4);
 }
 temp2 = old_clu/NITEMSINFATSEC; //计算最后一个簇项所在 FAT 扇区内的位置
 temp1 = old_clu % NITEMSINFATSEC; //计算最后一个簇项所在 FAT 扇区
 ((pFAT_Sec->items)[temp1]).Item[0] = 0; //将最后一个簇项清零
 ((pFAT_Sec->items)[temp1]).Item[1] = 0;
```

```
((pFAT_Sec->items)[temp1]).Item[2] = 0;
((pFAT_Sec->items)[temp1]).Item[3] = 0;
znFAT_Device_Write_Sector(temp2 + (pInit_Args->FirstFATSector),
                          znFAT_Buffer); //回写最后一个簇项所在 FAT 扇区
znFAT_Device_Write_Sector(temp2 + (pInit_Args->FirstFATSector
                                  + pInit_Args->FATsectors),znFAT_Buffer);
pInit_Args->Free_nCluster += nclu; //更新剩余空簇数,空簇回收
return 0;
}
```

这种实现方式的效率要比前一种高得多。另外,在上面的程序中还有一些额外的操作。一是对空簇参考值进行更新。如果要销毁的簇链的开始簇小于当前的空簇参考值,那么就将空簇参考值更新为这个开始簇。因为我们使用的空簇搜索算法是“接力式搜索”,即从当前空簇开始继续向后搜索下一空簇。所以,这样做是为了使空簇尽量靠前,否则被清空的簇链无法再得到重新利用。二是对剩余空簇数的更新。簇链的销毁将释放更多的空簇,剩余空簇数自然随之增加。关于这些操作,请看图 7.3。

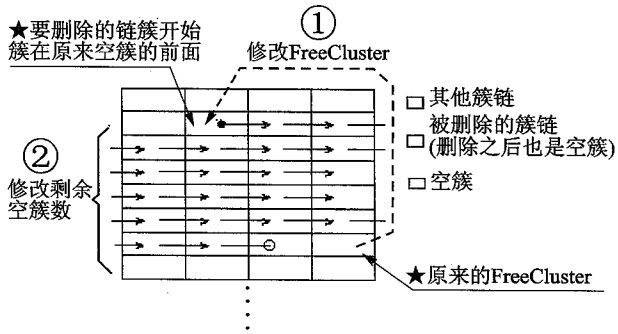


图 7.3 簇链销毁过程中对空簇参考值及剩余空簇数的修改

关于簇链的销毁似乎就这么多内容,但实际上还有一些更为深层的问题。这些问题仍然主要来自于 CCCB。如果使用了 CCCB 机制的话,那么一个文件的簇链就可能不光存在于 FAT 扇区中了。所以,要重新考虑簇链销毁的实现方法。关于这部分内容不再赘述,读者可以参见 znFAT 源代码。

有了簇链的销毁,数据的倾倒(删除)就很简单了,请看如下代码(znFAT.c):

```
UINT8 znFAT_Dump_Data(struct FileInfo * pfi,UINT32 offset)
{
 if(offset>=(pfi->File_Size)) //目标偏移量超出文件范围
 {
  return 1;
 }
 znFAT_Seek(pfi,offset); //定位到目标位置
 Destroy_FAT_Chain(pfi->File_CurClust); //销毁以文件当前簇开始的簇链
```

```
if(offset>0) //如果不是要删除文件所有数据
{
 Modify_FAT(pfi->File_CurClust,0X0FFFFFFF); //簇链封口
}
pfi->File_Size = offset; //更新文件大小
#ifdef RT_UPDATE_FILESIZE
Update_File_Size(pfi); //更新文件大小到物理扇区
#endif
if(0 == pfi->File_Size) //如果文件大小为 0
{
 Update_File_sClust(pfi,0); //更新文件开始簇为 0
}
#ifdef RT_UPDATE_FSINFO
Update_FSINFO(); //更新 FSINFO 扇区
#endif
return 0;
}
```

这里可能会产生这样的疑问："数据删除难道不用将簇里的数据也清零吗？只是销毁簇链就可以了？"当然不用，一条簇链在被销毁之后，其中的各簇即处于闲置状态，簇中的数据具体是什么其实已无关紧要，直到它们被重新利用，被写入新的有效数据。

我们应该听说过，一些公司在处理存有机密文件的磁盘时，都不只是删除那么简单，而是直接进行物理销毁，比如粉碎、消磁、高温等处理。根本原因就是文件被删除后其数据依然存在于簇中，只不过是用于组织这些簇的簇链被销毁了。通过一些很智能的算法是有可能把簇链进行重建的，从而实现数据的恢复，这就是诸如 FinalData、EasyRecovery 等数据恢复软件的基本原理。数据恢复是文件系统技术的另一重要分支，但这不是本书的重点，所以这里不再详述，有兴趣的读者可以参见清华大学出版社出版的《文件系统与数据恢复》一书。

7.2 文件的删除

7.2.1 文件删除的实质

如果仍然把文件看作是一桶水，那么文件删除就是先倾倒见底，然后再把桶砸了。前者就是销毁文件的整条簇链，后者就是对文件目录项进行处理。为了揭示文件删除的实质，我们来做一个实验。向 SD 卡中放入一个名为 test.txt 的文件，并向其写入一些数据，如图 7.4 及图 7.5 所示。接下来将这个文件删除，再看看图 7.5 所示的这些参数和数据有何变化，请看图 7.6。

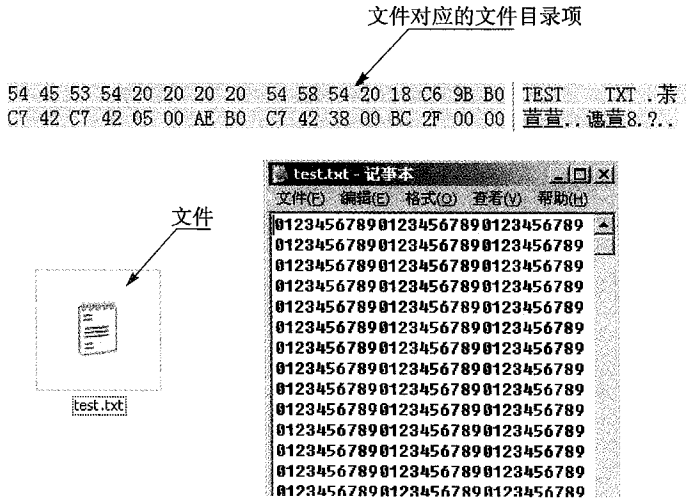


图 7.4 test.txt 文件的内容与文件目录项

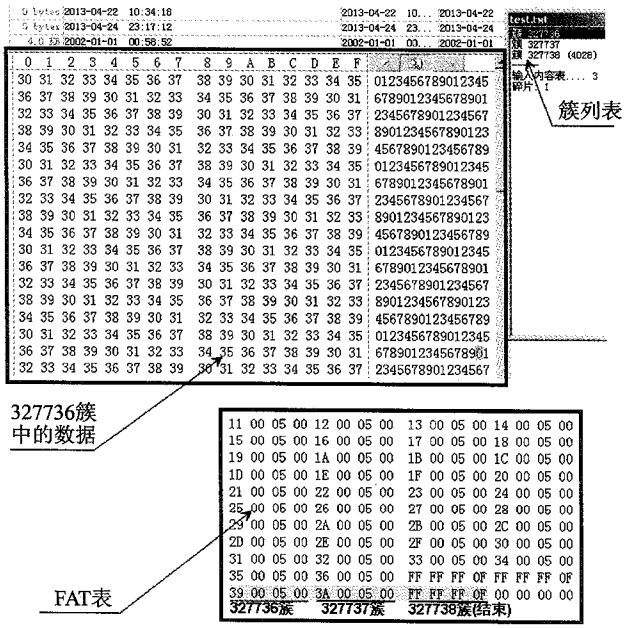


图 7.5 test.txt 文件簇链及簇内数据

可以看到,文件删除之后簇链已经被全部清零,但是簇内的数据却依然如故。这与我们前面所说的是一致的。同时我们发现,文件目录项并没有被清零,而是在原来的基础上有所改动,如图 7.7 所示。很显然,文件删除后文件目录项有两处变化:① 文件名字段的第一个字节被改成了 0XE5;② 文件开始簇的高字被清零。这就进一步为数据的恢复提供了更多的依据。

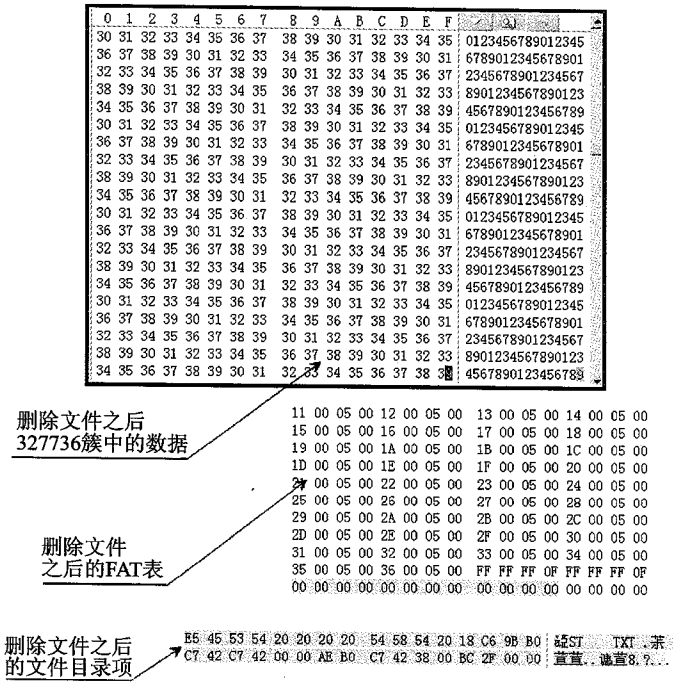


图 7.6　test.txt 文件删除之后的簇链、簇内数据及文件目录项

删除前的文件目录项

```
54 45 53 54 20 20 20 20  54 58 54 20 18 C6 9B B0 | TEST    TXT .苯
C7 42 C7 42 05 00 AE B0  C7 42 38 00 BC 2F 00 00 | 萱萱..谦萱8.?/..
```

删除后的文件目录项

```
E5 45 53 54 20 20 20 20  54 58 54 20 18 C6 9B B0 | 蜕ST    TXT .苯
C7 42 C7 42 00 00 AE B0  C7 42 38 00 BC 2F 00 00 | 萱萱..谦萱8.?/..
```

图 7.7　文件删除前后文件目录项的变化

7.2.2　文件删除的实现

通过上面的实验，我们已经知道了 FAT32 文件系统中文件删除操作的实质，接下来就可以对文件删除函数（znFAT_Delete_File）进行实现了，请看如下代码（znFAT.c）：

```
UINT8 znFAT_Delete_File(INT8 *filepath)
{
 UINT32 fdi_sec = 0; //用于记录文件目录项所在扇区
 UINT8  fdi_pos = 0; //用于记录文件目录项在扇区中的位置
 UINT32 start_clu = 0; //用于记录文件开始簇
 UINT32 cur_clu = 0; //用于记录当前目录簇
 UINT8 err_flag = 1; //用于记录是否删除成功
```

```
INT8 * filename; //用于记录文件名
UINT8 pos = 0; //用于记录文件名在路径中的位置
struct FDIesInSEC * pitems; //指向文件目录项扇区的指针
struct FDI * pitem; //指向文件目录项的指针
pitems = (struct FDIesInSEC * )znFAT_Buffer;
if(!znFAT_Enter_Dir(filepath,&cur_clu,&pos)) //获取文件所在目录的开始簇
{
 filename = filepath + pos; //获取文件名,以便后面进行文件匹配
}
else
{
 return 1; //如果进入目录失败,则直接返回错误
}
do
{
 //在当前簇的所有扇区中对文件进行搜索与匹配(通配)
 {
  //如果匹配成功,err_flag = 0,获取其文件目录项所在扇区
  //及它在扇区中的位置,还有文件开始簇
  {
   if(0! = start_clu) Destroy_FAT_Chain(start_clu);
                    //如果文件开始簇不为 0,即文件数据不为空,则销毁整条簇链
   znFAT_Device_Read_Sector(fdi_sec,znFAT_Buffer); //读取文件目录项所在扇区
   pitem = (pitems - >FDIes) + fdi_pos; //指向文件目录项
   pitem - >Name[0] = 0XE5; //给文件目录项打上"已删除"的标记
                            //即将文件名字段的第一个字节改为 0XE5
   pitem - >HighClust[0] = pitem - >HighClust[1] = 0; //将文件开始簇的高字清零
   znFAT_Device_Write_Sector(fdi_sec,znFAT_Buffer); //回写扇区
  }
 }
 //获取下一目录簇
}while(不是当前目录簇最后一个簇);
return err_flag;
}
```

程序中首先对文件进行了搜索和匹配,这与前面讲过的打开文件函数(znFAT_Open_File)的实现大体相同。然后是对文件整条簇链的销毁,最后对文件目录项进行修改。而且,还加入了“文件名通配”,这使得我们可以一次性删除同一目录下的很多文件,比如 znFAT_Delete_File("/dir1/dir2/ * . txt")。上面的代码对一些重复性的内容进行了精简,使得篇幅不至于过于冗长拖沓。

7.3 目录的删除

7.3.1 目录删除的难处

其实上面所讲的内容都比较好理解，接下面要讲的目录删除就不是那么简单了。到底难处何在？下面振南就让读者来“见识一下”。

目录与文件在存储形式上虽然是相似的，但是目录删除与文件删除在实现上却有着极大的不同。目录有着它所独有的特点：树状结构。删除一个目录并不像销毁簇链、修改文件目录项那么简单。目录下可能还有子目录和文件，而子目录下还可能再有子目录和文件……初遇这一问题，振南也有些犯难。振南第一个念头就是觉得这是一个递归的问题，可以从图 7.8 更加深刻地体会到目录删除的难处。

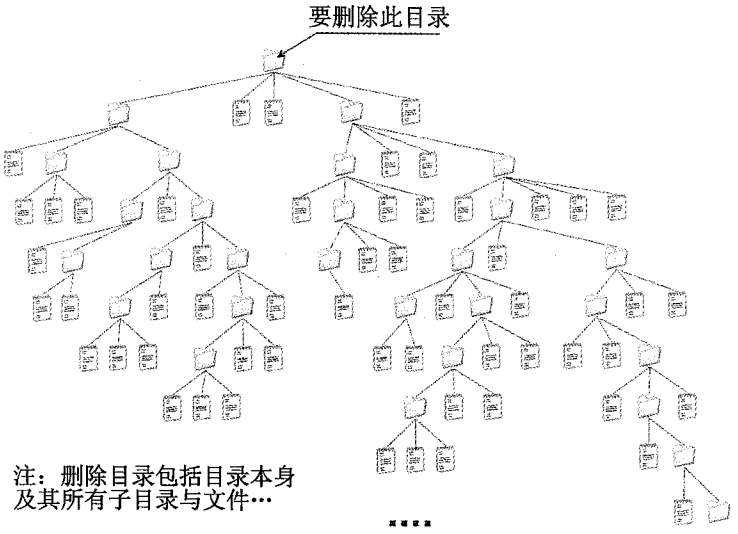


图 7.8 目录的树状结构

图 7.8 所示的就是目录的树状结构。从顶层目录出发，下面可能会有更为复杂的各种目录分支。目录就如同一扇门，它本身“不起眼”，但推开它里面却是“别有洞天”。毁门容易，“捣洞”却难了。要把顶层目录删除，就要将其下面的各级目录及所有文件全部删除，这是一项较有难度的工作。你可能会说：“可以使用递归算法来解决。”不错，这确实是一个递归结构。但是有经验的人都知道，在嵌入式系统中递归是要求代码可重入的(Reentrant)(因为递归调用是一个自身调用自身的过程)。调用次数起决于递归结构的层数与规模(目录的深度)，而且一定要有一个结束条件，递归将以它为终点进行回溯。有一句话是这样说的：“一个没有结束条件的‘递归’，可以