《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3章

数据写入，细微可见：数据写入的实现

第2章实现了文件与目录的创建，更重要的是引出了空簇搜索与簇链构造。这是FAT32的核心，也是本章以及后续各章相关内容的重要基础。还记得在前面使用“偷梁换柱”的方法所做的数据采集与存储实验？这里将使用真正的数据写入函数(znFAT_WriteData)对实验进行重现和拓展。这个函数没有任何功能限制，可以对任意文件进行任意数据的写入。其实在具体的实现上，数据写入和上册中讲过的数据读取是有相似之处的，可以对比阅读。另外，本章还引出一个新的实验——简易数码相机，揭示出一些新的问题，指明后面继续研究和努力的方向。

3.1 初步实现

3.1.1 回顾数据读取

既然数据写入与数据读取在实现上是相似的，先来回忆一下当初是如何对数据读取进行实现的。当时，首先实现了从文件开头读取数据的功能。然后试图从文件的中间开始读取数据，从而引入了数据位置参数的概念。接下来，我们就开始“纠缠”于为了处理当前簇内数据而产生的各种纷杂的情况。这部分是难点也是重点，振南花了较大篇幅来讲解。最后，所有超出当前簇的情况都统一为“从整簇读取数据”，与前面的“从文件开头读取数据”实现了“归一化”。这个过程请看图3.1。

纷杂的簇内数据过程主要是为了处理文件当前簇中的扇区级与字节级的数据。数据读取的起始位置以及数据长度不同，那么具体的实现过程也会不同。尤其是数据读完之后，对数据位置参数的更新。说到这里，读者是否还记得“窘簇”？它是一种很特殊的情况，此时的数据位置参数并不能真实地表达文件数据当前所处的位置。窘簇的出现是因为数据读完之后正好到了文件的结尾，而这个结尾又正好是结束簇的末尾。这就如同一个人站在悬崖边上一样，已无路可走。当时我们处理的方式是“置之不理”，即保留位置参数的值，尤其是File_CurClust(数据当前簇，其实就是文件的结束簇)。为什么要这么做？这一章中就可以看到了。而数据的写入同样会遇到这所有的问题及过程。

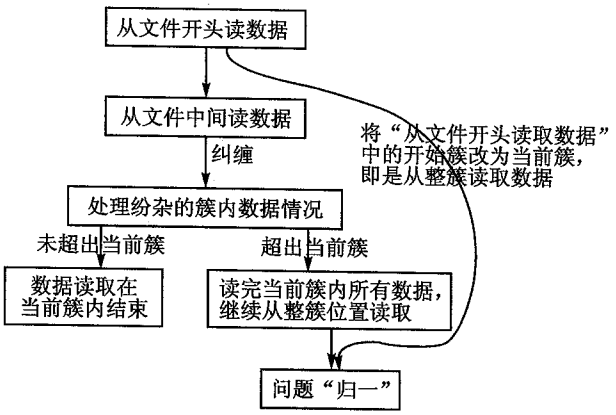

图 3.1　数据读取功能涉及的一些主要阶段和问题

3.1.2　从开头写数据

依然从比较简单的情况入手：从文件的开头写入数据，其实就是向空文件写入数据。前面已经说过，对于一个空文件来说，因为它还没有数据，不占用任何簇，所以它的开始簇为 0，数据位置参数为：

```
File_CurClust = 0; File_CurSec = 0; File_CurPos = 0; File_CurOffset = 0; File_IsEOF = 1;
```

此时，这些参数其实都是一些约定值，也就是在一些不同寻常情况下所取的特殊值（窘簇就是一种特殊情况）。它们虽然不能正确地表达真实的数据位置，只能算是一个标记，但是却可以帮助我们在程序中对这些特殊情况进行处理。

那到底该如何实现从文件的开头写入数据呢？不能直接把 File_CurClust 拿来用，否则会造成数据的损坏或文件系统的崩溃，请看图 3.2。

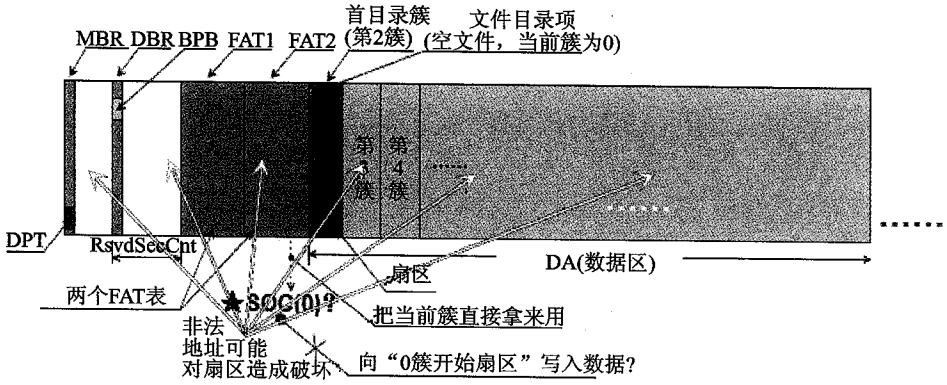

图 3.2　非法的“0 簇”操作将可能造成不良后果

正常情况下簇是从 2 开始的，所以 0 簇是非法的，是不应该进入到我们的计算中

的。如果非要把它等同于普通簇来进行操作，那么计算得到的结果必然是无意义的。比如图 3.2 中使用 SOC 宏计算 0 簇的开始扇区地址，其结果可能会是一个远远超过实际有效物理扇区地址的非法值，有可能造成存储设备的硬件瘫痪，或者造成无法预知的数据坏破，或者因重要核心参数的丢失而造成文件系统的崩溃。

正确的做法应该是：先为空文件分配开始簇，构造簇链，然后再将数据写入到簇中。随着更多的数据被写入，簇链也被同步地构造出来……请看图 3.3。

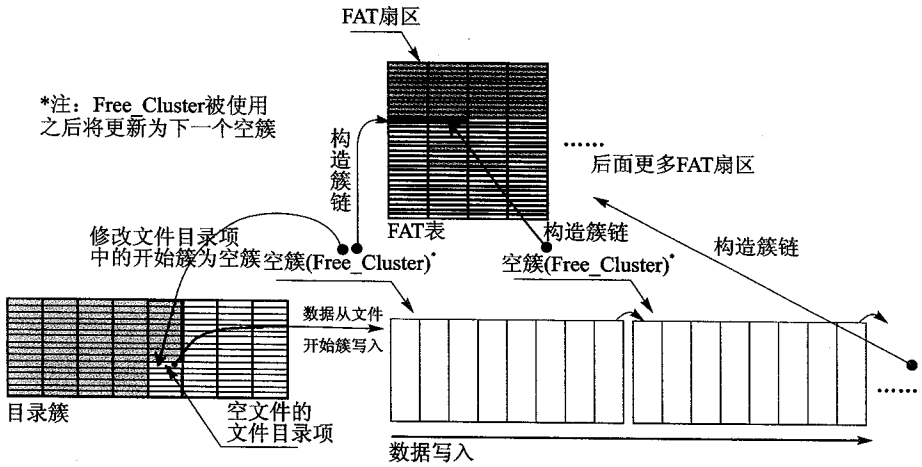

图 3.3　从文件开头写入数据的正确实现过程

这里有一个问题："要把文件的开始簇修改为空簇，那就必须知道文件目录项的位置，那么是不是还要再写一个搜索文件目录项的函数？"不错，涉及对文件目录项的修改时就首先要知道它的位置，但是却不必对它再进行搜索！这是因为打开文件函数(znFAT_Open－_File)或文件创建函数(znFAT_Create_File)已经完成了这项工作。还记得前面讲解这些函数的实现过程中已经记录下了文件目录项所在的扇区及其位置吗(记录到了文件信息体中的 FDI_Sec 与 FDI_Pos 中)？文件开始簇的修改可以通过下面这个函数来完成(znFAT.c)：

```
UINT8 Update_File_sClust(struct FileInfo * pfi,UINT32 clu)
{
 struct FDI * pfdi;
 znFAT_Device_Read_Sector(pfi->FDI_Sec,znFAT_Buffer); //读取文件目录项所在扇区
 pfdi = (((struct FDIesInSEC * )znFAT_Buffer)->FDIes) + (pfi->FDI_Pos);
 (pfdi->HighClust)[0] = clu>>16; //修改文件目录项中的开始簇字段
 (pfdi->HighClust)[1] = clu>>24;
 (pfdi->LowClust )[0] = clu;
 (pfdi->LowClust )[1] = clu>>8;
 znFAT_Device_Write_Sector(pfi->FDI_Sec,znFAT_Buffer); //回写
 pfi->File_StartClust = clu; //更新文件信息
```

```
 return 0;
}
```

接下来就实现从文件开头写数据，会经历 3 个数据阶段：整簇、整扇区以及剩余字节，如图 3.4 所示。具体实现代码如下(znFAT.c)：

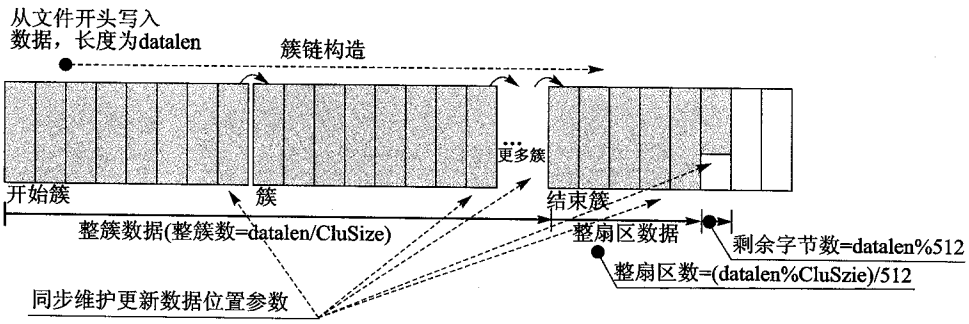

图 3.4　从文件开头写数据将要经历 3 个数据阶段

```
UINT32 WriteData_From_Start(struct FileInfo * pfi,UINT32 len,UINT8 * pbuf)
{
 UINT32 CluSize = ((Init_Args.BytesPerSector) * (Init_Args.SectorsPerClust));
                                                          //计算簇大小
 UINT32 temp = len/CluSize; //计算整簇数
 UINT32 old_clu = 0;
 UINT8 i = 0,j = 0;
 if(0 == len) return 0; //如果要写入的数据长度为 0,则直接返回
 Update_File_sClust(pfi,Init_Args.Free_Cluster);
 //更新文件目录项中的开始簇字段
 //更新文件数据位置参数
 pfi->File_CurClust = Init_Args.Free_Cluster; //当前簇为空簇
 pfi->File_CurPos = 0;
 for(j = 0;j<temp;j++) //写入整簇数据
 {
  old_clu = pfi->File_CurClust;
  pfi->File_CurClust = Init_Args.Free_Cluster; //当前簇为空簇
  pfi->File_CurSec = SOC(pfi->File_CurClust); //当前簇的开始扇区
  Update_Free_Cluster(); //更新空簇,即从当前簇开始遍历查找下一空簇
  for(i = 0;i<(Init_Args.SectorsPerClust);i++) //向簇内扇区写入数据
  {
   znFAT_Device_Write_Sector(pfi->File_CurSec + i,pbuf);
   pbuf + = Init_Args.BytesPerSector;
  }
  Modify_FAT(old_clu,pfi->File_CurClust); //构造簇链
 }
```

```
if(0! = (len % CluSize)) //如果还有数据要写入
{
 old_clu = pfi - >File_CurClust;
 pfi - >File_CurClust = Init_Args.Free_Cluster; //当前簇为空簇
 pfi - >File_CurSec = SOC(pfi - >File_CurClust); //当前簇的开始扇区
 Update_Free_Cluster();
 temp = (len % CluSize)/(Init_Args.BytesPerSector); //剩余数据的整扇区数
 for(i = 0;i<temp;i ++ ) //写入最后一个簇的整扇区数据
 {
  znFAT_Device_Write_Sector(pfi - >File_CurSec,pbuf);
  pfi - >File_CurSec ++ ;
  pbuf + = Init_Args.BytesPerSector;
 }
 temp = len % (Init_Args.BytesPerSector); //最后的剩余字节数据
 if(0! = temp) //最后还有剩余数据要写入
 {
  Memory_Copy(pbuf,znFAT_Buffer,temp);
                          //把最后不足整扇区的字节数据放入内部缓冲区中
  znFAT_Device_Write_Sector(pfi - >File_CurSec,znFAT_Buffer);
                                    //将内部缓冲区中的数据写入扇区中
  pfi - >File_CurPos = temp;
 }
 Modify_FAT(old_clu,pfi - >File_CurClust); //构造簇链
}
Modify_FAT(pfi - >File_CurClust,0X0FFFFFFF); //把 FAT 簇链"关上"
Update_FSINFO(); //同步更新 FSINFO 扇区,它记录了剩余空簇数
pfi - >File_Size + = len;
pfi - >File_CurOffset + = len;
Update_File_Size(pfi); //更新文件大小
return len;
}
```

3.1.3 从整簇写数据

如果当前文件的大小正好是簇大小的整数倍,那该如何向其写入数据呢?如图 3.5所示。首先使用 Seek 函数将数据定位到文件末尾。但是因为文件大小为簇的整数倍,所以此时必定会出现窘簇。我们要从这个窘簇开始向文件写入数据,这就需要从原来的结束簇继续构造簇链。如果我们当初在遇到"窘簇"时,没有保留位置参数的值,那么此时将陷入怎样的困境?对,我们将无法知道后面的簇链应该从哪开始。文件的整条簇链将在这里出现断链。

其实上面的从文件开头写数据函数(WriteData_From_Start)稍加修改就是从整

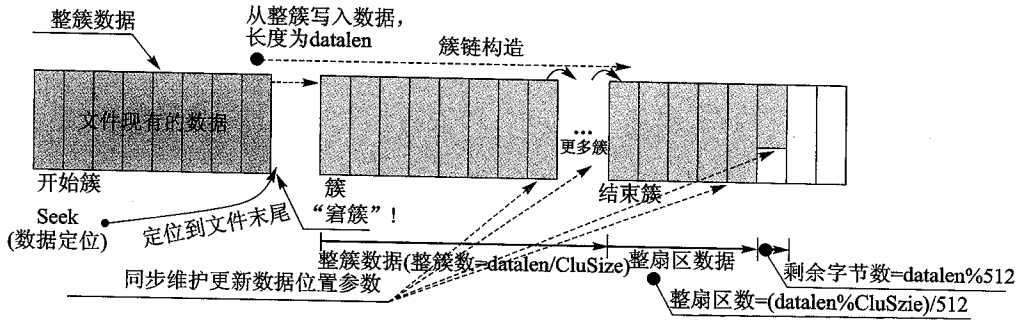

图 3.5 从整簇写数据的实现过程

簇写数据函数(WriteData_From_nClustert)，代码如下(znFAT.c)：

```
UINT32 WriteData_From_nCluster(struct FileInfo * pfi,UINT32 len,UINT8 * pbuf)
{
 //变量定义
 if(0 == len) return 0; //如果要写入的数据长度为 0,则直接返回
 znFAT_Seek(pfi,pfi->File_Size); //文件数据定位到文件末尾
 if(0 == pfi->File_CurClust) //如果是空文件
 {
  Update_File_sClust(pfi,Init_Args.Free_Cluster);
  //更新文件目录项中的开始簇字段
  //更新文件数据位置参数
  pfi->File_CurClust = Init_Args.Free_Cluster; //当前簇为空簇
  pfi->File_CurSec = SOC(pfi->File_CurClust); //当前簇的首扇区
  pfi->File_CurPos = 0;
 }
 //分整簇、整扇区与剩余字节 3 个数据阶段完成数据写入,同 WriteData_From_Start
 return len;
}
```

3.2 数据写入的实现

1. 依旧繁杂的簇内过程

振南在前面所讲的从文件开始写数据、从整簇写数据其实都是数据写入过程中的一些特例，但最终的数据写入函数应该是更为强大、更为普适，无论现有的文件末尾落于何处，它都应该能够正确地完成数据的写入。如同数据读取一样，数据写入也存在簇内的数据过程，也要针对各种情况分别进行处理。

第一类情况：要写入的数据长度不大于当前扇区剩余数据量，如图 3.6 所示。这种情况下数据的写入非常简单，只需要做一个“数据拼接”即可，代码如下(znFAT.c)：

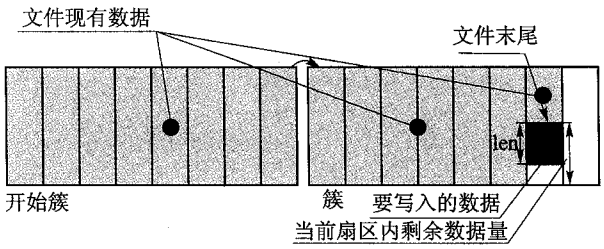

图 3.6　要写入的数据长度小于当前扇区剩余数据量

```
znFAT_Device_Read_Sector(pfi->File_CurSec,znFAT_Buffer);
                                        //读取当前扇区数据,以便作扇区内数据拼接
Memory_Copy(pbuf,znFAT_Buffer+pfi->File_CurPos,len); //扇区数据拼接
znFAT_Device_Write_Sector(pfi->File_CurSec,znFAT_Buffer); //回写扇区数据
//更新数据位置参数
pfi->File_CurPos+=len;
pfi->File_CurOffset+=len;
pfi->File_Size+=len; //更新文件大小
Update_File_Size(pfi);
```

当然,情况不只于此。请继续看图 3.7。这种情况下,数据位置的更新方法稍有不同,请看如下代码(znFAT.c):

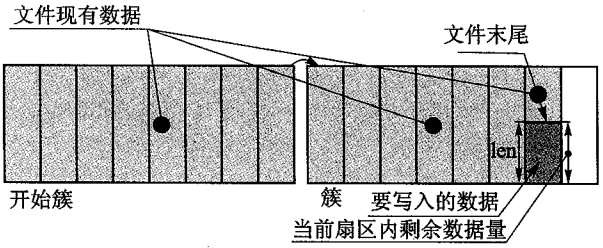

图 3.7　要写入的数据长度等于当前扇区剩余数据量

```
//数据拼接
//更新数据位置参数
pfi->File_CurSec++;
pfi->File_CurPos=0;
pfi->File_CurOffset+=len;
//更新文件大小
```

再想想,如果数据正好写到了簇的末尾该如何处理?请看图 3.8。此时将产生窘簇,具体的处理方法就不赘述了。

第二类情况:要写入的数据长度大于当前扇区的剩余数据量,但并不超出当前簇,如图 3.9 所示。处理方式如下(znFAT.c):

```
len_temp=len; //len_temp 为临时变量
```

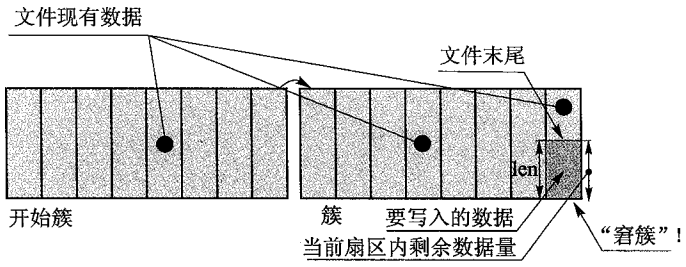

图 3.8　数据正好写到了簇的末尾

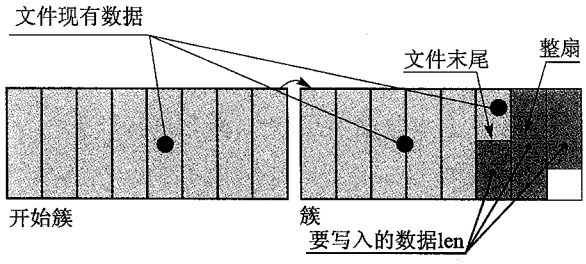

图 3.9　第二类情况中的普通实例

```
temp = ((Init_Args.BytesPerSector) - (pfi - >File_CurPos));
                                                    //计算当前扇区剩余数据量
znFAT_Device_Read_Sector(pfi - >File_CurSec,znFAT_Buffer);
                                                    //读取当前扇区数据,以便作扇区内数据拼接
Memory_Copy(pbuf,znFAT_Buffer + pfi - >File_CurPos,temp); //数据拼接
znFAT_Device_Write_Sector(pfi - >File_CurSec,znFAT_Buffer); //回写扇区数据
len_temp - = temp; //计算要写入的剩余数据量
pbuf + = temp;
pfi - >File_CurSec ++ ;
pfi - >File_CurPos = 0;
temp1 = len_temp/512; //计算剩余数据整扇区数
for(i = 0;i<temp1;i ++ ) //写入整扇区数据
{
 znFAT_Device_Write_Sector(pfi - >File_CurSec + i,pbuf);
 pbuf + = 512;
}
pfi - >File_CurSec + = temp1;
pfi - >File_CurPos = len_temp % 512;
Memory_Copy(pbuf,znFAT_Buffer,pfi - >File_CurPos);
                                                    //将最后的剩余字节装入到内部缓冲中
znFAT_Device_Write_Sector(pfi - >File_CurSec,znFAT_Buffer);//写最后一个扇区
pfi - >File_CurOffset + = len;
pfi - >File_Size + = len; //更新文件大小
```

```
Update_File_Size(pfi);
```

当然，如果数据又正好写到簇末尾，那么窘簇就又出现了。

上面所讲的就是数据写入过程中针对于簇内数据的一些处理方法，其实它比数据读取要简单。

2. 问题的归一

如果要写入的数据超出了当前簇，如图 3.10 所示，可以看到，首先完成当前簇内的数据写入。此时，会在当前簇的末尾暂时产生窘簇。后面的工作就是从整簇写数据了。所以，最终的数据写入函数是这样的(znFAT.c)：

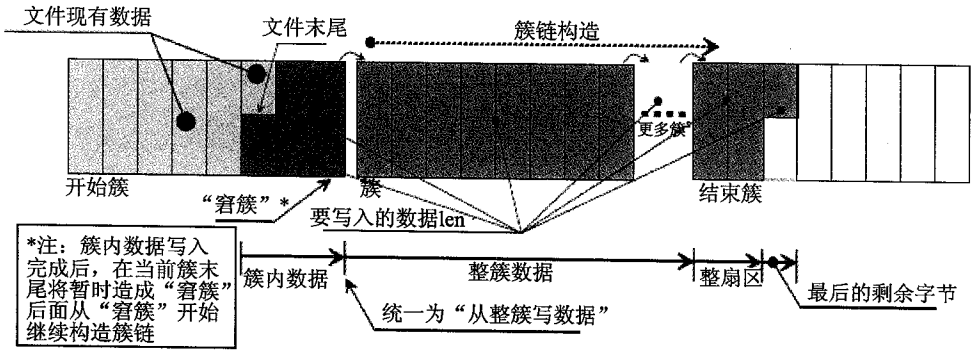

图 3.10　超出当前簇的数据写入过程

```
UINT32 znFAT_WriteData(struct FileInfo * pfi,UINT32 len,UINT8 * pbuf)
{
 UINT32 temp = 0,temp1 = 0,len_temp = len,i = 0;
 UINT32 Cluster_Size =
          ((Init_Args.BytesPerSector) * (Init_Args.SectorsPerClust));
 if(0 == len) return 0; //如果要写入的数据长度为 0,则直接返回 0
 znFAT_Seek(pfi,pfi->File_Size); //定位到文件末尾
 temp = ((Init_Args.BytesPerSector) - (pfi->File_CurPos)); //计算当前扇区剩余数据量
 if((pfi->File_CurOffset % Cluster_Size)! = 0)
 {
  if(len<= temp) //★要写入的数据长度不大于当前扇区剩余数据量
  {
   //处理第一类情况
  }
  else //★要写入的数据长度大于当前扇区剩余数据量
  {
   //将数据写入当前扇区,与现有扇区数据进行拼接
   if(!(IS_END_SEC_OF_CLU(pfi->File_CurSec,pfi->File_CurClust)))
                                        //★如果当前扇区不是当前簇的最后一个扇区
   {
```

```
pfi->File_CurSec++;
pfi->File_CurPos=0;
temp=(LAST_SEC_OF_CLU(pfi->File_CurClust)-(pfi->File_CurSec)+1)*512;
                                                    //★计算当前簇内的剩余数据量
if(len_temp<=temp) //★如果要继续写入的数据不超出当前簇
{
 temp1=len_temp/512; //计算要继续写入的数据的整扇区数
 for(i=0;i<temp1;i++)
 {
  znFAT_Device_Write_Sector(pfi->File_CurSec+i,pbuf);
  pbuf+=512;
 }
 if(len_temp==temp) //如果正好写满当前簇
 {
  //此处产生"窘簇"
  return len; //数据写入完成
 }
 else //如果没有写满当前簇
 {
  pfi->File_CurSec+=temp1;
  pfi->File_CurPos=len_temp%512;
  //将剩余字节写入最后扇区
  return len;
 }
}
else //要写入的数据超出当前簇
{
 temp1=LAST_SEC_OF_CLU(pfi->File_CurClust); //计算当前簇最后扇区
 for(i=pfi->File_CurSec;i<=temp1;i++) //将数据写入当前簇所有剩余扇区
 {
  znFAT_Device_Write_Sector(i,pbuf);
  pbuf+=512;
 }
 len_temp-=temp;
 //此处产生"窘簇"
}
}
else //当前扇区是当前簇最后一个扇区
{
 //此处产生"窘簇"
}
}
```

```
    }
    WriteData_From_nCluster(pfi,len_temp,pbuf); //★从整簇开始写数据
    pfi->File_Size += len;
    Update_File_Size(pfi); //更新文件大小
    return len;
}
```

3.3 数据写入的典型应用

至此，我们已经完成了较为成型的数据写入函数(znFAT_WriteData)，下面介绍两个数据写入的典型应用：数据采集与存储、图像采集与存储(简易数码相机)，方便读者深入理解和实践。

3.3.1 实例 1：数据采集与存储

其实这个实验我们前面已经作过，不过是用“偷梁换柱”的方法去实现的，现在真正的数据写入函数已经“出炉”，那我们就来重新实现；同时，在功能上加以拓展：每分钟创建一个新文件，文件名为当前时间。每秒钟采集一次数据，并存储到新创建的文件中。具体过程如图 3.11 所示。实现过程其实很简单，代码(_main.c)如下：

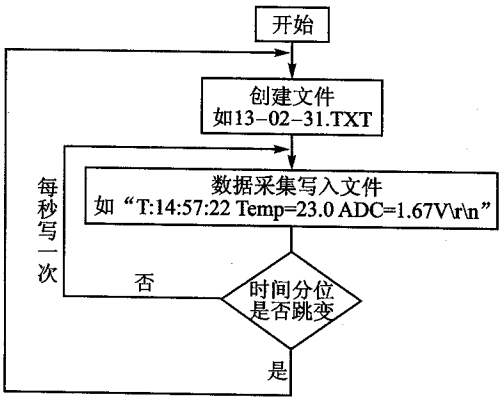

图 3.11 数据采集与存储实验流程

```
void main(void)
{
 char buf[40];
 char fn[20];
 unsigned char old_min = 0,old_sec = 0;
 znFAT_Device_Init(); //存储设备初始化
 znFAT_Init();
 while(1)
```

```
{
 P8563_Read_Time(); //从 PCF8563 中读取时间信息
 if(time.minute! =old_min) //判断分钟是否跳变
 {
  old_min = time.minute;
  Generate_FileName(&time,fn); //用当前时间中的时分秒生成文件名
  //设置文件时间信息
  dt.date.year = time.year; dt.date.month = time.month; dt.date.day = time.day;
  dt.time.hour = time.hour;dt.time.min = time.minute;dt.time.sec = time.second;
  znFAT_Create_File(&fi,fn,&dt); //创建文件
 }
 if(time.second! = old_sec) //判断秒是否跳变
 {
  old_sec = time.second;
  Format_Dat(&time,DS18B20_ReadTemperature(),TLC549_GetValue(),buf);
                                          //将时间、温度以及电压值转为一定格式
  znFAT_WriteData(&fi,32,buf); //向文件中写数据
 }
}
while(1);
}
```

实验效果如图 3.12 所示。

这个实验看起来简单，但实际上经常有人向振南提出疑问。总结起来，主要有两个问题：

① 如何由时间来生成文件名？

② 为什么我写入到文件中的数据是乱码？

下面，振南分别进行回答。

1. 由时间生成文件名

使用文件系统来存储数据时，很多人都希望将不同时段的数据存储为不同的文件，这样会更加便于管理和查阅，比如把当天的数据存到文件 20130503.txt 中，而第二天的数据则存到 20130504.txt 文件中，依此类推，如图 3.13 所示。

其实由时间生成文件名就是一个由数字转为字符串的过程，我们可以编写一个小函数来实现，比如上面程序中的 Generate_FileName(合成文件名)。其实最简单的方法是直接调用字符串格式化函数 sprintf，比如 sprintf(filename,"%d%d%d.txt",year,month,day)。

实际上，振南要说的核心问题还不在于上面的这些。很多人曾经问过："为什么不能创建文件名类似于'201305041303.txt'的文件？"这样的文件名包含了更多的时间信息，可以将数据管理得更为精细。但是当使用 znFAT_Create_File 函数去创建

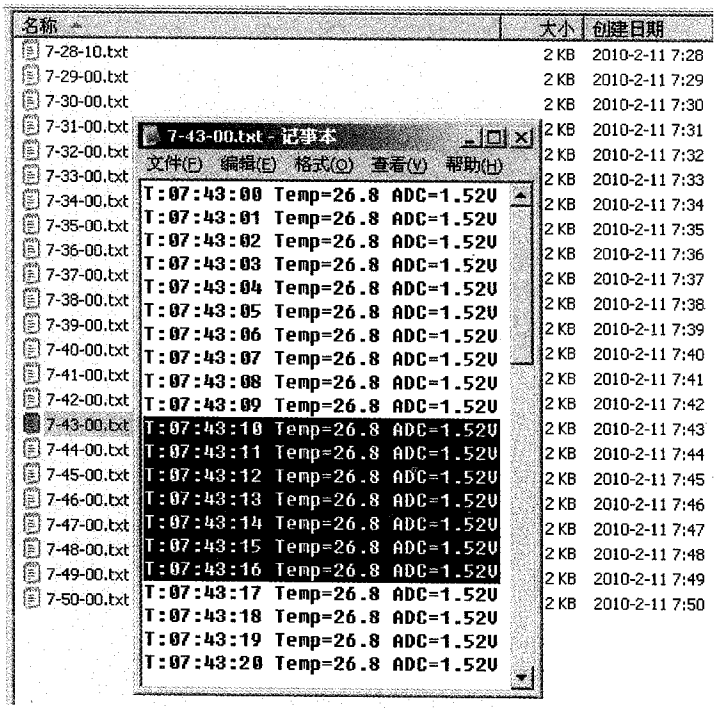

图 3.12　数据采集与存储实验效果

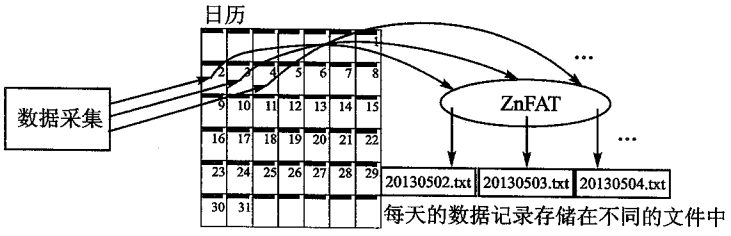

图 3.13　将每天的数据记录到不同的文件中

它的时候，我们确实会发现问题，根本原因就在于文件名是“长名”。前面各章中实现的所有函数基本上都是针对于 8·3 格式短文件的，如果强行输入一个长文件名，那必然会导致出错。归根结底还是因为短名与长名文件在创建原理与实现过程上的较大差异。关于长文件名，振南会在后面的章节进行专门的讲解，到时候我们会让 znFAT 支持长名，诸如上面的这种长名文件的创建将不再是问题。

2. 写入的数据是乱码

很多种原因都会导致写入到文件中的数据显示为乱码，比如内存溢出、函数的错误使用、原始数据本身有误等。振南在这里要说的其实是一种很“令人无语”的错误，但确实有很多人在犯，请看图 3.14。

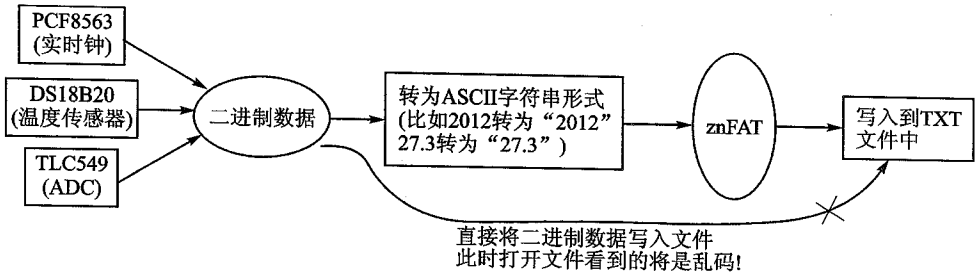

图 3.14　二进制数据不经转换直接写入文件而造成乱码

稍有 C 语言基础的人都应该知道数值意义上的$(2012)_{10}$与由 ASCII 字符组成的字符串“2012”是有本质区别的，如果想以文本方式直接看到数值，那就必须加以转换。上面程序中的函数 Format_Dat 就是在做这样的事情。按原始二进制的方式写入到文件中的数据，我们非要以文本方式来查看，那必然会显示乱码。

3.3.2　实例 2：简易数码相机

自从振南在网上发布了简易数码相机实验之后，咨询的人“络绎不绝”，于是一度成了众多文件系统应用实验中最受人关注的实验之一（其他实验还有诸如 SD 卡视频播放器、录音笔、无线文件传输等，在后面的实验章节可以看到）。所以，振南觉得有必要在这里对它进行一个介绍，一方面为读者提供参考，另一方面也借此揭示当前数据写入函数的一个严重缺陷。

ZN－X 开发板上预留了 OV7670 摄像头模块接口，配合 SD 卡、TFT 液晶、znFAT 以及 BMP 位图文件，便可实现简单的数码相机功能，如图 3.15 所示。实际的硬件效果如图 3.16 所示。实现代码如下（_main.c）（当然，如同其他模块一样，OV7670 模块的驱动也已经做成了现成的程序模块）：

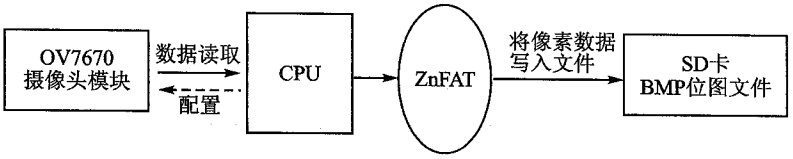

图 3.15　简易数码相机实验的实现示意图

```
#define PICTURE_W 128    //定义图像宽度
#define PICTURE_H 128    //定义图像高度
#define FIFO_BASE_ADDR ((volatile unsigned char xdata *)0x8000)
//注:ZN-X开发板 OV7670 模块上的 FIFO 存储器直接挂到了 51 芯片的外部总线
//上,因此可将其作为外扩 RAM 直接进行读取。硬件电路决定了它的地址空间为
//0X8000～0XFFFF,即 32 KB(请参见附录中的开发板原理图),这样就可以直接
//将它当作应用数据缓冲区来使用
struct FileInfo fileinfo; //文件信息集合
```

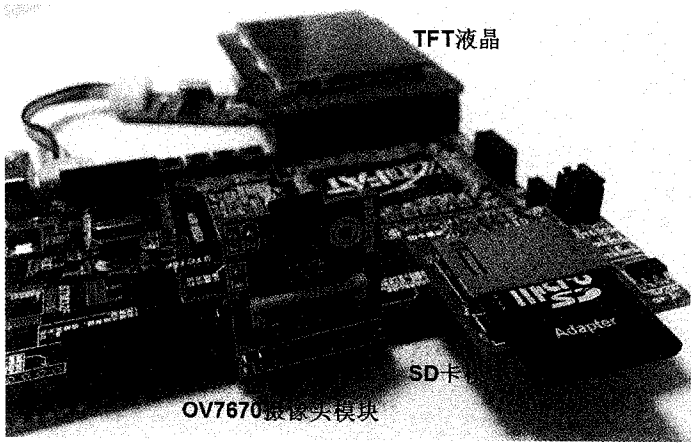

图 3.16　简易数码相机实验硬件效果图

```
    struct DateTime dt; //日期时间结构体变量
    unsigned char cur_status = 0;
    char filename[20] = {'/',0};
    unsigned char code bmp_header[54] =  //BMP 文件数据头(尺寸为 128X128)
    {
     0x42, 0x4D, 0x38, 0x58, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36, 0x00, 0x00, 0x00,
0x28,0x00,
     0x00,0x00,PICTURE_W,PICTURE_W>>8,0x00,0x00,PICTURE_H,PICTURE_H>>8,
     0x00, 0x00, 0x01, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x12,0x0B,
     0x00,0x00,0x12,0x0B,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
    };
    void int0(void) interrupt 0 //OV7670 的场同步信号将触发此中断程序
    {
     EX0 = 0; //关闭中断
     WEN = 0; //将 OV7670 模块上 FIFO 芯片写使能关闭,暂停接收图像数据
     cur_status = 1; //将状态变量进行标记,告诉主程序对图像数据进行读取
    }
    int main()
    {
     //相关变量定义
     UART_Init();
     UART_Send_Str("串口初始化完成\r\n");
     znFAT_Device_Init(); //存储设备初始化
     UART_Send_Str("存储设备初始化完成\r\n");
     znFAT_Init(); //文件系统初始化
     UART_Send_Str("文件系统初始化完成\r\n");
```

```
Sensor_init(); //OV7670 芯片初始化
UART_Send_Str("摄像头芯片初始化完成\r\n");
//设置 OV7670 芯片的图像采集窗口
OV7670_config_window(184 + 2 * PICTURE_W,
                                    10 + 2 * PICTURE_H,PICTURE_W,PICTURE_H);
dt.date.year = 2012; dt.date.month = 12; dt.date.day = 21; //设置文件创建时间
dt.time.hour = 15; dt.time.min = 14; dt.time.sec = 35;
IT0 = 1; //下降沿触发,VSYNC 信号产生时说明 FIFO 中已经有完整的图像
EA = 1; //打开总中断
UART_Send_Str("外部中断已开启,按键即可获取图像数据\r\n");
while(1)
{
 if(!KEY) //检测 KEY 是否按下
 {
  delay(100); //按键去抖
  while(!KEY);
  EX0 = 1; //如果 KEY 按下则打开外部中断,可由 OV7670 的 VSYNC 信号触发
 }
 if(cur_status == 1) //FIFO 中已有完整图像
 {
  cur_status = 0;
  UART_Send_Str("图像已获取\r\n");
  u32tostr(n++ ,filename + 1); //由按键次数生成文件名,如 1.bmp、2.bmp 等
  len = strlen(filename);
  strcpy(filename + len,".bmp");
  znFAT_Create_File(&fileinfo,filename,&dt); //创建文件
  UART_Send_Str("BMP 文件已创建\r\n");
  znFAT_WriteData(&fileinfo,54,bmp_header); //先向文件写入 BMP 数据头
  UART_Send_Str("图像文件数据头已写入\r\n");
  UART_Send_Str("开始将图像写入文件\r\n");
  FIFO_Reset_Read_Addr(); //FIFO 存储器地址归 0,为读取图像数据作好准备
  FIFO_OE = 0; //使能 FIFO 数据端口
  znFAT_WriteData(&fileinfo,128 * 128 * 2,FIFO_BASE_ADDR);
                                                    //将 FIFO 数据直接写入文件
  FIFO_OE = 1; //关闭 FIFO 数据端口使能
  UART_Send_Str("图像写入完成\r\n");
  WEN = 1; //开启 FIFO 的写使能,重新开始接收图像数据
 }
}
return 0;
}
```

实验现象

经实际测试，每存储一个 BMP 文件大约需要 5 s，估计读者会觉得有点太慢了，是因为 51 的运行速度较慢吗？其实不然，ZN－X 开发板配备的是 STC 增强型高速 51，不光主频比普通 51 快 12 倍，而且指令集也得到了较大改进。更重要的是，振南的 SD 卡底层驱动使用的是硬件 SPI，在速度上应该不逊色于 STM32 或其他 CPU，况且读取 FIFO 中的图像数据是通过硬件总线来实现的。其实就算把这个实验放到 STM32 上去实现速度也快不了多少。硬件因素并不是最主要的矛盾，那导致数据写入效率不高的原因到底是什么呢？这正是下一章我们要讨论的问题。

感谢对振南及 znFAT 的关注与支持，希望振南在嵌入式 FAT32 文件系统方面的研究对您有所帮助！

更多内容请关注 振南电子网站

www.znmcu.cn