FAT32 文件系统的存储机制及其在单片机上的实现

FAT32 文件系统您一定不会陌生，最多看到它是在 windows 操作系统里，但在一些嵌入式产品（如手机、MP3、MP4 等）中，也能看到它的身影。从某种意义上来讲，FAT32 文件系统是非常成功的，使我们可以脱离底层储存设备驱动，更为方便高效地组织数据。给单片机系统中的大容量存储器（如 SD 卡、CF 卡、硬盘等）配以 FAT32 文件系统，将是非常有意义的（如创建的数据文件可以在 windows 等操作系统中直接读取等）。

FAT32 本身是比较复杂的，对其进行讲解的最好方法就是实际演练。笔者手里持有一张刚以 FAT32 格式化的 SD 卡，我们就围绕它来讲解 FAT32 的实现机理。

FAT32 分为几个区域，这里将用实例的方法对它们的结构与在文件存储中的功能进行详细的剖析。

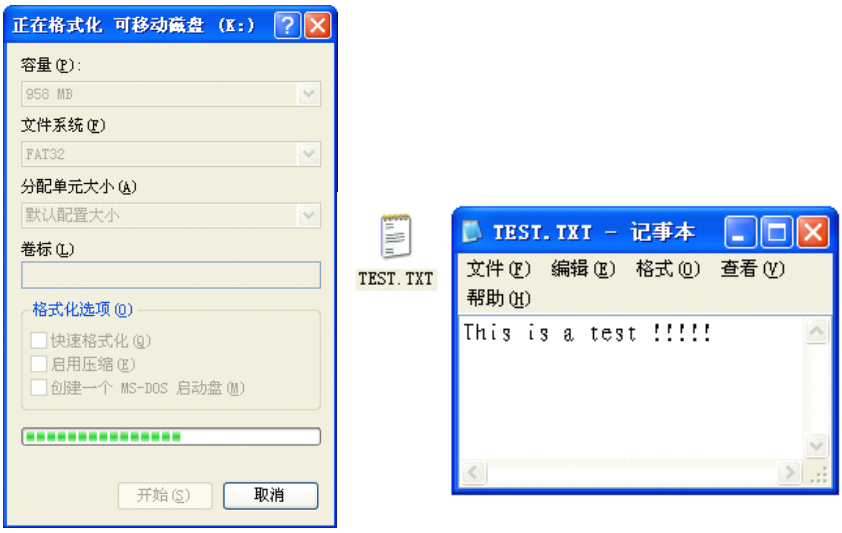
1、 实例说明

此实例首先在一张空的 SD 卡（已被格式化为 FAT32 格式）上创建一个文本文件，并在其中输入 20 个字符。再将它插入到单片机系统中，实现对这个文件的读取，将文件内容输出在调试终端上。

2、 实现过程

1） 格式化与创建文件

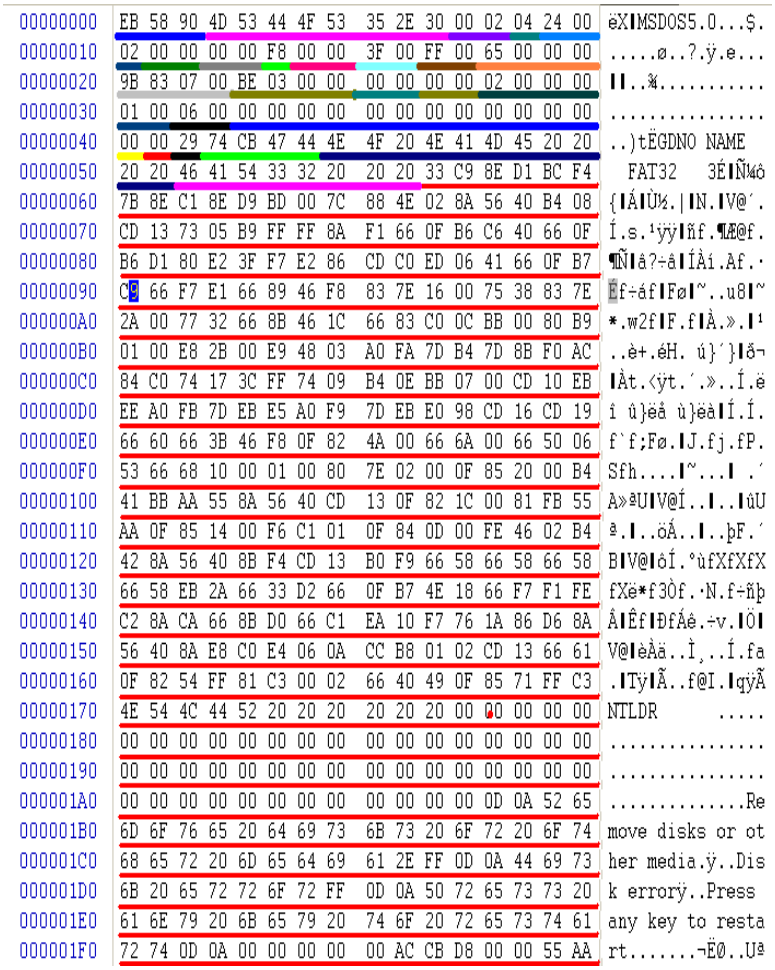
Windows 上的磁盘格式化与文件创建就不用多说了。如下图：



2）DBR（DOS BOOT RECORD 操作系统引导记录区）

DBR 是我们进军 FAT32 的首道防线。其实 DBR 中的 BPB 部分才是这一区域的核心部分（第 12~90 字节为 BPB），只有深入详实的理解了 BPB 的意义，才能够更好的实现和操控 FAT32。关于 DBR 在 FAT32 中的地位就不多说了，

以下面实际的 DBR 内　　图所示：



```
00000000  EB 58 90 4D 53 44 4F 53  35 2E 30 00 02 04 24 00  ëX.MSDOS5.0...$.
00000010  02 00 00 00 00 F8 00 00  3F 00 FF 00 65 00 00 00  .....ø..?.ÿ.e...
00000020  9B 83 07 00 BE 03 00 00  00 00 00 00 02 00 00 00  ..¾.....
00000030  01 00 06 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040  00 00 29 74 CB 47 44 4E  4F 20 4E 41 4D 45 20 20  ..)tËGDNO NAME
00000050  20 20 46 41 54 33 32 20  20 20 33 C9 8E D1 BC F4  FAT32   3É.Ñ¼ô
00000060  7B 8E C1 8E D9 BD 00 7C  88 4E 02 8A 56 40 B4 08  {.ÁÙ½.|.N.VÀ´.
00000070  CD 13 73 05 B9 FF FF 8A  F1 66 0F B6 C6 40 66 0F  Í.s.¹ÿÿ.ñf.¶Æ@f.
00000080  B6 D1 80 E2 3F F7 E2 86  CD C0 ED 06 41 66 0F B7  ¶Ñ.â?÷â.ÍÀí.Af.·
00000090  C9 66 F7 E1 66 89 46 F8  83 7E 16 00 75 38 83 7E  Éf÷áf.Fø.~..u8.~
000000A0  2A 00 77 32 66 8B 46 1C  66 83 C0 0C BB 00 80 B9  *.w2f.F.f.À.».¹
000000B0  01 00 E8 2B 00 E9 48 03  A0 FA 7D B4 7D 8B F0 AC  ..è+.éH. ú}´}.ð¬
000000C0  84 C0 74 17 3C FF 74 09  B4 0E BB 07 00 CD 10 EB  .Àt.<ÿt.´.».Í.ë
000000D0  EE A0 FB 7D EB E5 A0 F9  7D EB E0 98 CD 16 CD 19  î û}ëå ù}ëà.Í.Í.
000000E0  66 60 66 3B 46 F8 0F 82  4A 00 66 6A 00 66 50 06  f`f;Fø.J.fj.fP.
000000F0  53 66 68 10 00 01 00 80  7E 02 00 0F 85 20 00 B4  Sfh....€~... .´
00000100  41 BB AA 55 8A 56 40 CD  13 0F 82 1C 00 81 FB 55  A»ªUVÀÍ...ûU
00000110  AA 0F 85 14 00 F6 C1 01  0F 84 0D 00 FE 46 02 B4  ª..öÁ...þF.´
00000120  42 8A 56 40 8B F4 CD 13  B0 F9 66 58 66 58 66 58  BVÀ.ôÍ.°ùfXfXfX
00000130  66 58 EB 2A 66 33 D2 66  0F B7 4E 18 66 F7 F1 FE  fXë*f3Òf.·N.f÷ñþ
00000140  C2 8A CA 66 8B D0 66 C1  EA 10 F7 76 1A 86 D6 8A  Â.Êf.Ðfáê.÷v.Ö.
00000150  56 40 8A E8 C0 E4 06 0A  CC B8 01 02 CD 13 66 61  V@.èÀä..Ì¸..Í.fa
00000160  0F 82 54 FF 81 C3 00 02  66 40 49 0F 85 71 FF C3  .Tÿ.Ã..f@I..qÿÃ
00000170  4E 54 4C 44 52 20 20 20  20 20 20 00 00 00 00 00  NTLDR       .....
00000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00  00 00 00 00 0D 0A 52 65  ..............Re
000001B0  6D 6F 76 65 20 64 69 73  6B 73 20 6F 72 20 6F 74  move disks or ot
000001C0  68 65 72 20 6D 65 64 69  61 2E FF 0D 0A 44 69 73  her media.ÿ...Dis
000001D0  6B 20 65 72 72 6F 72 FF  0D 0A 50 72 65 73 73 20  k errorÿ..Press
000001E0  61 6E 79 20 6B 65 79 20  74 6F 20 72 65 73 74 61  any key to resta
000001F0  72 74 0D 0A 00 00 00 00  00 AC CB D8 00 00 55 AA  rt.......¬Ë0..Uª
```

上面的数据看起来杂乱不堪，无从下手，其实对我们有用的数据只不过90个字节（如图中彩色线标记的字节）。仅仅是这 90 个字节就可以告诉我们关于磁盘的很多信息，比如每扇区字节数、每簇扇区数、磁道扇区数等等。对于这些信息的读取，只要遵循 DBR 中的字段定义即可。（比如图中紫色字段的两个字节表示这张磁盘的每一个扇区有 512 个字节，具体的计算方法见下文）

字段定义如下表（BPB 后面的 422 个字节对我们的意义不大，表中省略）：

字段名称	长度	含义	偏移量
jmpBoot	3	跳转指令	0
OEMName	8	这是一个字符串，标识了格式化该分区的操作系统的名称和版本号	3
BytesPerSec	2	每扇区字节数	11
SecPerClus	1	每簇扇区数	13
RsvdSecCnt	2	保留扇区数目	14
NumFATs	1	此卷中 FAT 表数	16
RootEntCnt	2	FAT32 为 0	17

TotSec16	2	FAT32 为 0	19
Media	1	存储介质	21
FATSz16	2	FAT32 为 0	22
SecPerTrk	2	磁道扇区数	24
NumHeads	2	磁头数	26
HiddSec	4	FAT 区前隐扇区数	28
TotSec32	4	该卷总扇区数	32
FATSz32	4	FAT 表扇区数	36
ExtFlags	2	FAT32 特有	40
FSVer	2	FAT32 特有	42
RootClus	4	根目录簇号	44
FSInfo	2	文件系统信息	48
BkBootSec	2	通常为 6	50
Reserved	12	扩展用	52
DrvNum	1	—	64
Reserved1	1	—	65
BootSig	1	—	66
VolID	4	—	67
FilSysType	11	—	71
FilSysType1	8	—	82

DBR 的实现代码：

```
struct FAT32_DBR
{
 unsigned char BS_jmpBoot[3];     //跳转指令              offset: 0
 unsigned char BS_OEMName[8];     //                      offset: 3
 unsigned char BPB_BytesPerSec[2];//每扇区字节数          offset:11
 unsigned char BPB_SecPerClus[1]; //每簇扇区数            offset:13
 unsigned char BPB_RsvdSecCnt[2]; //保留扇区数目          offset:14
 unsigned char BPB_NumFATs[1];    //此卷中 FAT 表数       offset:16
 unsigned char BPB_RootEntCnt[2]; //FAT32 为 0            offset:17
 unsigned char BPB_TotSec16[2];   //FAT32 为 0            offset:19
 unsigned char BPB_Media[1];      //存储介质              offset:21
 unsigned char BPB_FATSz16[2];    //FAT32 为 0            offset:22
 unsigned char BPB_SecPerTrk[2];  //磁道扇区数            offset:24
 unsigned char BPB_NumHeads[2];   //磁头数                offset:26
 unsigned char BPB_HiddSec[4];    //FAT 区前隐扇区数      offset:28
```

```
unsigned char BPB_TotSec32[4];    //该卷总扇区数              offset:32

unsigned char BPB_FATSz32[4];     //一个 FAT 表扇区数     offset:36
unsigned char BPB_ExtFlags[2];    //FAT32 特有              offset:40
unsigned char BPB_FSVer[2];       //FAT32 特有              offset:42
unsigned char BPB_RootClus[4];    //根目录簇号              offset:44
unsigned char FSInfo[2[;          //保留扇区 FSINFO 扇区数 offset:48
unsigned char BPB_BkBootSec[2];   //通常为 6                offset:50
unsigned char BPB_Reserved[12];   //扩展用                  offset:52
unsigned char BS_DrvNum[1];       //                        offset:64
unsigned char BS_Reserved1[1[;    //                        offset:65
unsigned char BS_BootSig[1];      //                        offset:66
unsigned char BS_VolID[4];        //                        offset:67
unsigned char BS_FilSysType[11]; //                         offset:71
unsigned char BS_FilSysType1[8]; //"FAT32    "              offset:82
};
```

在程序中我们采用以上的结构体指针对扇区数据指针进行转化，就可以直接读取数据中的某一字段，如要读取 **BPB_BytesPerSec**，可以这样来作：

```
((struct FAT32_DBR *)pSector)-> BPB_BytesPerSec
```

用如上语句就可以得到这一字段的首地址。

心细的读者可能会发现读回来的字节拼在一起，与实际的数据并不吻合。例如 BPB_BytesPerSec 读出来的内容是“00 02”，在程序中我们把 00 作为 int 型变量的高字节，把 02 作为其低字节，那么这个变量的值为 2，而实际的 SD 卡里的扇区大小为 512 个字节，这 512 与 2 之间相去甚远。是什么造成这种现象的呢？

这就是大端模式与小端模式在作怪。上面我们合成 int 型变量的方法（00 为高字节，02 为低字节）为小端模式。而如果我们改用大端模式来进行合成的话，结果就会不同：将 02 作高字节，而把 00 作低字节，变量值就成了 0x0200（十进制的 512），这样就和实际数据吻合了。可见 FAT32 中字节的排布是采用小端模式的。在我们程序中需要将它转为大端模式的表达方式。在笔者的程序有这样一个函数 lb2bb，专门垃圾将小端模式转为大端模式，程序如下：

```
unsigned long lb2bb(unsigned char *dat,unsigned char len) //小端转为大端
{
 unsigned long temp=0;
 unsigned long fact=1;
 unsigned char i=0;
 for(i=0;i<len;i++)
 {
  temp+=dat[i]*fact;
  fact*=256;
 }
 return temp;
}
```

这样就可以从 BPB 中读出关于磁盘的各种参数信息，为我们后面的工作作准备。而这个从 BPB 中读取参数装入到参数表中以备后用的过程就是 FAT32 的初始化了。具体的实现如下：

先定义用来装入从 BPB 中读取的参数的结构：

```
struct FAT32_Init_Arg
{
 unsigned char BPB_Sector_No;      //BPB 所在扇区号
 unsigned long Total_Size;         //磁盘的总容量
 unsigned long FirstDirClust;      //根目录的开始簇
 unsigned long FirstDataSector;    //文件数据开始扇区号
 unsigned int  BytesPerSector;     //每个扇区的字节数
 unsigned int  FATsectors;         //FAT 表所占扇区数
 unsigned int  SectorsPerClust;    //每簇的扇区数
 unsigned long FirstFATSector;     //第一个 FAT 表所在扇区
 unsigned long FirstDirSector;     //第一个目录所在扇区
 unsigned long RootDirSectors;     //根目录所占扇区数
 unsigned long RootDirCount;       //根目录下的目录与文件数
};
```

当然也可以用零散的变量来存储参数，但用结构体更方便管理，也会使程序更为整洁。FAT32 的初始化将向结构中装入参数，实现如下：

```
void FAT32_Init(struct FAT32_Init_Arg *arg)
{
struct FAT32_BPB *bpb=(struct FAT32_BPB *)(FAT32_Buffer);
//将数据缓冲区指针转为 struct FAT32_BPB 型指针
arg->BPB_Sector_No =FAT32_FindBPB();
//FAT32_FindBPB()可以返回 BPB 所在的扇区号
arg->Total_Size =FAT32_Get_Total_Size();
//FAT32_Get_Total_Size()可以返回磁盘的总容量，单位是兆
arg->FATsectors =lb2bb((bpb->BPB_FATSz32) ,4);
//装入 FAT 表占用的扇区数到 FATsectors 中
arg->FirstDirClust =lb2bb((bpb->BPB_RootClus) ,4);
//装入根目录簇号到 FirstDirClust 中
arg->BytesPerSector =lb2bb((bpb->BPB_BytesPerSec),2);
//装入每扇区字节数到 BytesPerSector 中
arg->SectorsPerClust =lb2bb((bpb->BPB_SecPerClus) ,1);
//装入每簇扇区数到 SectorsPerClust 中
arg->FirstFATSector=lb2bb((bpb->BPB_RsvdSecCnt) ,2)+arg-
>BPB_Sector_No;
//装入第一个 FAT 表扇区号到 FirstFATSector 中
arg->RootDirCount =lb2bb((bpb->BPB_RootEntCnt) ,2);
//装入根目录项数到 RootDirCount 中
arg->RootDirSectors =(arg->RootDirCount)*32>>9;
//装入根目录占用的扇区数到 RootDirSectors 中
arg->FirstDirSector=(arg->FirstFATSector)+(bpb-
>BPB_NumFATs[0])*(arg->FATsectors);
//装入第一个目录扇区到 FirstDirSector 中
arg->FirstDataSector =(arg->FirstDirSector)+(arg->RootDirSectors);
//装入第一个数据扇区到 FirstDataSector 中
```

}

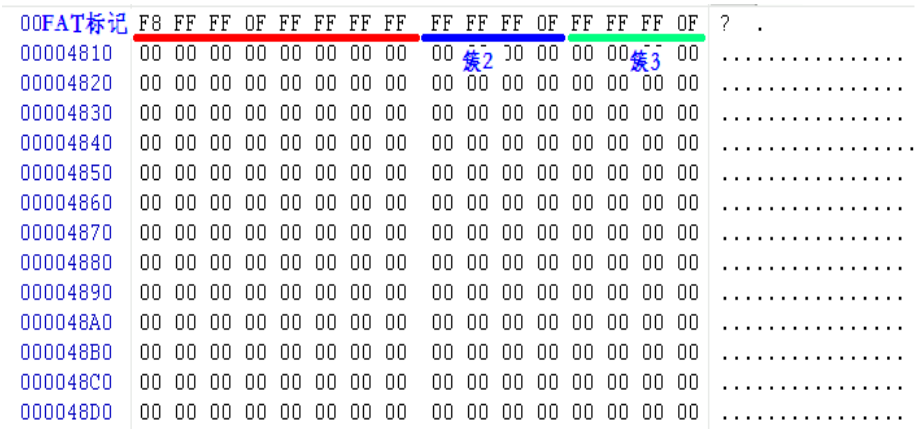
3）FAT（文件分配表）

FAT 表是 FAT32 文件系统中用于磁盘数据（文件）索引和定位引进的一种链式结构。可以说 FAT 表是 FAT32 文件系统最有特色的一部分，它的链式存储机制也是 FAT32 的精华所在，也正因为有了它才使得数据的存储可以不连续，使磁盘的功能发挥得更为出色。

FAT 表到底在什么地方？它到底是什么样子的呢？

从第一步从 BPB 中提取参数中的 **FirstFATSector** 就可以知道 FAT 表所在的扇区号。其实每一个 FAT 表都有另一个与它一模一样的 FAT 存在，并且这两个 FAT 表是同步的，也就是说对一个 FAT 表的操作，同样地，也应该在另一个 FAT 表进行相同的操作，时刻保证它们内容的一致。这样是为了安全起见，当一个 FAT 因为一些原因而遭到破坏的时候，可以从另一个 FAT 表进行恢复。

FAT 表的内容如下图所示：



上图就是一个实际的 FAT 表。前 8 个字节“F8 FF FF 0F FF FF FF FF”为 FAT32 的 FAT 表头标记，用以表示此处是 FAT 表的开始。后面的数据每四个字节为一个簇项（从第 2 簇开始），用以标记此簇的下一个簇号。

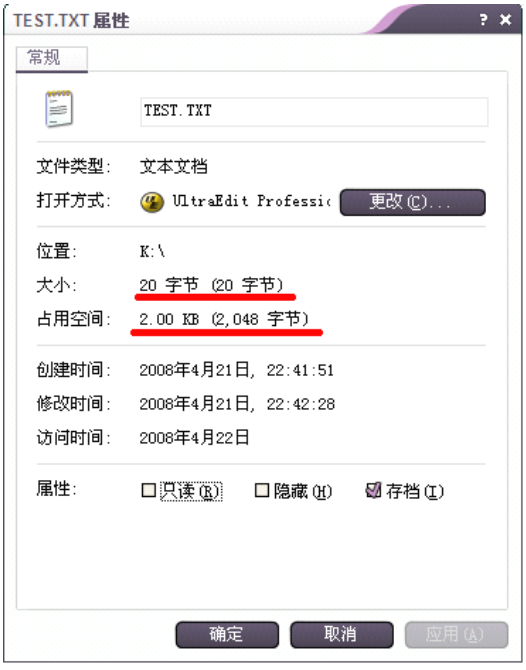
拿我们创建的那个叫 TEST.TXT（大小为 20 个字节）的文件来说，如果这个文件的开始簇为第 2 簇的话，那么就到 FAT 表里来查找，看文件是否有下一个簇（如果文件大小大于一个簇的容量，必须会有数据存储到下一个簇，但下一个簇与上一个簇不一定是连续的），可以看到“簇 2”的内容为“FF FF FF 0F”，这样的标记就说明这个文件到第 2 簇就已经结束了，没有后继的簇，即此文件的大小是小于一个簇的容量的。

上面讲了很多，都是围绕簇这样一个词来讲的，簇又是什么？为什么要将它引入到 FAT32 里来呢？

磁盘上最小可寻址存储单元称为扇区，通常每个扇区为 512 个字节。由于多数文件比扇区大得多，因此如果对一个文件分配最小的存储空间，将使存储器能存储更多数据，这个最小存储空间即称为簇。根据存储设备(磁盘、闪卡和硬盘)的容量，簇的大小可以不同以使存储空间得到最有效的应用。在早期的 360KB 磁盘上，簇大小为 2 个扇区(1,024 字节)；第一批的 10MB 硬盘的簇大小增加到 8 个扇区(4,096 字节)；现在的小型闪存设备上的典型簇大小是 8KB 或 16KB。2GB 以上的硬盘驱动器有 32KB 的簇。如果对于容量大的存储定义了比较小的簇的话，就会使 FAT 表的体积很大，从而造成数据的冗余和效率的下降。

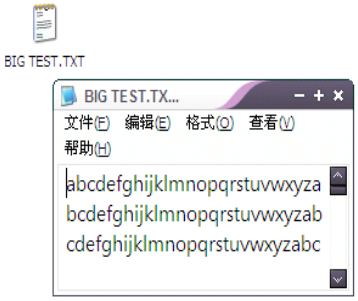
需要指出的是，簇作为 FAT32 进行数据存储的最小单位，内部扇区是不能进一步细分的，即使一个文件的数据写到一个簇中后，簇中还有容量的剩余（里部扇区没有写满），哪怕这个簇只写了一个字节，其它文件的数据也是不能接在后面继续数据的，而只能另外找没有被占用的簇。

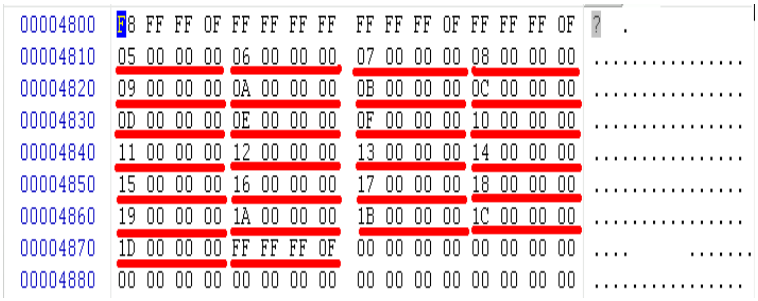
我们按照初始化参数表中的 **SectorsPerClust** 可以知道一个簇中的扇区数，笔者的 SD 卡实测簇大小为 4 个扇区，按照上面的说法，TEST.TXT 这样一个只有 20 个字节的文件，也会占用一个簇的容量，让我们在 Windows 里看看它的实际占用空间的情况。如下图：



从上图可以看到文件大小为 20 个字节，但占用空间却是 2048 个字节（一个簇的容量，4 个扇区）。

TEST.TXT 容量只有 20 个字节，所以只占用了一个簇，可能 FAT 表中还看不出链式结构，现在我们再创建一个文件，使它占用 26 个簇，如下：





可以看到图中红色标记的就是文件所占用的 26 个簇。从第 4 簇开始，簇项 4 的内容为“05 00 00 00”（小端模式），说明下一个簇为第 5 簇，而簇项 5 的内容为“06 00 00 00”,说明下一个簇为第 6 簇……依此类推，直到内容为“FF FF FF 0F”，说明无后继簇，文件数据到此结束。FAT 表中的链式存储结构已经非常明显。把我们从 FAT 表中分析的结果与 Windows 的统计结束进行对比，说明我们的解理是正确的，如下图：

从上面可以看到，当数据结束于某一簇时，FAT32 就用“FF FF FF 0F”来对其进行标记。其实还有其实的标记以表达其它的簇属性，如“00 00 00 00 ”表示未分配的簇，“FF FF FF F7”表示坏簇等。

给出一个簇号，计算出它的后继簇号，是实现 FAT32 的重点，实现如下：

```
unsigned long FAT32_GetNextCluster(unsigned long LastCluster)
{
 unsigned long temp;
 struct FAT32_FAT *pFAT;
 struct FAT32_FAT_Item *pFAT_Item;
 temp=((LastCluster/128)+Init_Arg.FirstFATSector);
//计算给定簇号对应的簇项的扇区号
 FAT32_ReadSector(temp,FAT32_Buffer);
 pFAT=(struct FAT32_FAT *)FAT32_Buffer;
 pFAT_Item=&((pFAT->Items)[LastCluster%128]);
//在算出的扇区中提取簇项
 return lb2bb(pFAT_Item,4);      //返回下一簇号
}
```

那么 FAT 表有多大呢？FAT 表中每四个字节表示一个簇，所以 FAT 表的大小由实际的簇数来决定。从这里也可以看出，如果簇过大，就会则 FAT 表比较小，但会造成空间的浪费，而如果簇过小，可以减小空间的浪费，但会使 FAT 表变得臃肿。FAT 表的大小也可以从 BPB 参数 **FATsectors** 读出。从上面的 BPB 图可以得知笔者的 SD 卡的 FAT 表大小为 958 个扇区（“BE 03 00 00”的大端表示）。如果这 958 个扇区每四个字节都表示一个簇项，则它可以表示（958*512/4）-2= 122622 个簇（减去 2 是因为有 8 个字节的 FAT 表头标识。

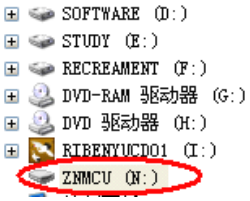
看看我们计算的是否正确呢，下面是 Winhex 计算出来的簇数：

与 Winhex 计算的结果是吻合的，我们对 FAT 表与簇的理解是正确的。

看完上面对 FAT 表的讲解中，你可能会问：一个文件数据的首簇号怎样来确定呢？只有知道了一个文件数据的首簇号才能继续查找下一簇数据的位置，直到数据结束。下面将要讲到的“根目录区”就可以由一个文件的文件名来查到它的首簇。

4）根目录区

在 FAT32 中其实已经把文件的概念进行扩展，目录同样也是文件，从根目录的地位与其它目录是相同的，因此根目录也被看作是文件。既然是文件就会有文件名，根目录的名称就是磁盘的卷标。如笔者的 SD 卡在格式会时设置卷标为 znmcu，则根目录的名称就为 ZNMCU，如下图：

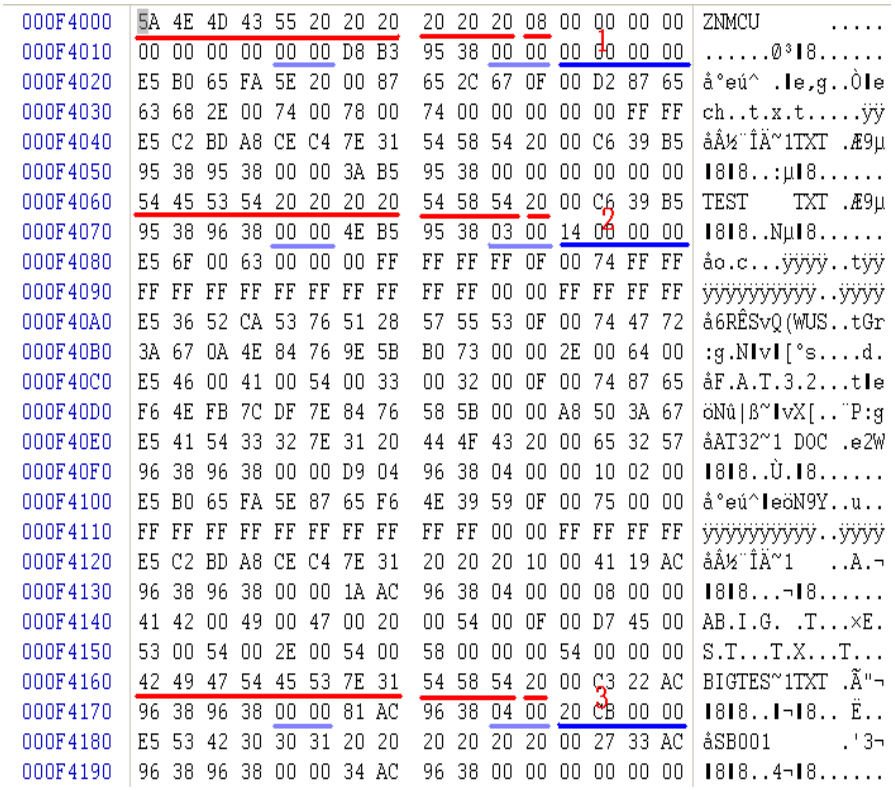


每一个文件都对应一个描述它属性的结构，定义如下：

FAT32 文件目录项 32 个字节的定义			
字节偏移量	字数量	定义	
0~7	8	文件名	
8~10	3	扩展名	
11	1	属性字节	0x00 （读写）
			0x01 （只读）
			0x02 （隐藏）
			0x04 （系统）
			0x08 （卷标）
			0x10 （子目录）
			0x20 （归档）

12	1	系统保留
13	1	创建时间的 10 毫秒位
14~15	2	文件创建时间
16~17	2	文件创建日期
18~19	2	文件最后访问日期
20~21	2	文件起始簇号的高 16 位
22~23	2	文件的最近修改时间
24~25	2	文件的最近修改日期
26~27	2	文件起始簇号的低 16 位
28~31	4	表示文件的长度

根目录区所在扇区可从 BPB 参数 **FirstDirSector** 获取，从 BPB 图得 **FirstDirSector=FirstFATSector+BPB_NumFATs*FATsectors=2053**。根目录区的初始大小为一个簇，实际的内容如下：



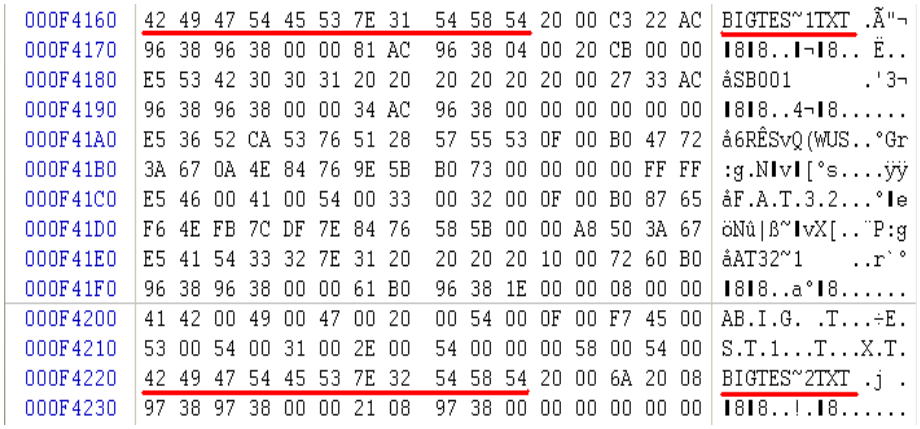
图中的记录 1 描述根目录，前八个字节为文件名“ZNMCU　　”（长度小于 8 的部分用空格符补齐），下面的三个字节为扩展名“　　”（ 长度小于 3 的部分用空格符补齐），08 表示此文件为卷标，开始簇高字节为 00　00，低字节为 00　00，开始簇为 0，文件长度为 0。

记录 2 描述 TEST.TXT 文件，文件名为“TEST　　”，扩展名为“TXT”，20 表示此文件为归档，开始簇为 3（“00　00　00　03”），长度为 20。

记录 3 描述 BIGTEST.TXT 文件，文件名为“BIGTES~1”，扩展名为“TXT”，

开始簇为 4，长度为 5200 字节（00　00　CB　20）。

可以看到 FAT32 中的文件名都以大写字母表示，长度不足的部分用空格符补齐，所以我们要读取的文件 TEST.TXT 就变成了“TEST　　.TXT”，这将有助于文件名的匹配，我们不用去处理不等长文件名所带来的麻烦。另外，还会发现长度过长的部分会被~1 所替换，如果替换后有文件与之重名，则~后面的数字将增加为 2。



文件目录项结构的实现如下：

```
struct direntry
{
  unsigned char deName[8];          // 文件名
  unsigned char deExtension[3];     // 扩展名
  unsigned char deAttributes;       // 文件属性
  unsigned char deLowerCase;        // 系统保留
  unsigned char deCHundredth;       // 创建时间的 10 毫秒位
  unsigned char deCTime[2];         // 文件创建时间
  unsigned char deCDate[2];         // 文件创建日期
  unsigned char deADate[2];         // 文件最后访问日期
  unsigned char deHighClust[2];     // 文件起始簇号的高 16 位
  unsigned char deMTime[2];         // 文件的最近修改时间
  unsigned char deMDate[2];         // 文件的最近修改日期
  unsigned char deLowCluster[2];//   文件起始簇号的低 16 位
  unsigned char deFileSize[4];      // 表示文件的长度
```

```
}
```

我们最终要实现的是对 TEST.TXT 文件的读取，须要作到给定文件名后，可以得到相应文件的首簇。主要的思想就是对根目录区中（本实例只针对根目录中的文件进行读取，至于多级子目录的实现，只须要进行多次首簇定位）的记录进行扫描，对记录中的文件名进行匹配。具体的实现如下：

```
struct FileInfoStruct * FAT32_OpenFile(char *filepath)
{
 unsigned char depth=0;
 unsigned char i,index=1;
 unsigned long iFileSec,iCurFileSec,iFile;
 struct direntry *pFile;
 iCurFileSec=Init_Arg.FirstDirSector;
 for(iFileSec=iCurFileSec;
      iFileSec<iCurFileSec+(Init_Arg.SectorsPerClust);
      iFileSec++)
 {
  FAT32_ReadSector(iFileSec,FAT32_Buffer);
  for(iFile=0;
       iFile<Init_Arg.BytesPerSector;
       iFile+=sizeof(struct direntry))   //对记录逐个扫描
  {
   pFile=((struct direntry *)(FAT32_Buffer+iFile));
   if(FAT32_CompareName(filepath+index,pFile->deName))
//对文件名进行匹配
   {
    FileInfo.FileSize=lb2bb(pFile->deFileSize,4);
    strcpy(FileInfo.FileName,filepath+index);
    FileInfo.FileStartCluster=lb2bb(pFile->deLowCluster,2)
                              +lb2bb(pFile->deHighClust,2)*65536;
    FileInfo.FileCurCluster=FileInfo.FileStartCluster;
```

```
            FileInfo.FileNextCluster=
                                FAT32_GetNextCluster(FileInfo.FileCurCluster);
            FileInfo.FileOffset=0;
            return &FileInfo;
        }
    }
}
}
```

这个函数在找到目标文件后，会将此文件的一些参数信息装入到文件结构中，为以后的文件读取作好准备。文件结构如下：

```
struct FileInfoStruct
{
unsigned char   FileName[12];           //文件名
unsigned long   FileStartCluster;       //文件首簇号
unsigned long   FileCurCluster;         //文件当前簇号
unsigned long   FileNextCluster;        //下一簇号
unsigned long   FileSize;               //文件大小
unsigned char   FileAttr;               //文件属性
unsigned short FileCreateTime;          //文件建立时间
unsigned short FileCreateDate;          //文件建立日期
unsigned short FileMTime;               //文件修改时间
unsigned short FileMDate;               //文件修改日期
unsigned long   FileSector;             //文件当前扇区
unsigned int    FileOffset;             //文件偏移量
};
```

通过对根目录区的扫描，可以得到 TEST.TXT 首簇为 3，下面就可能以它为起点，来读取文件内容了。

5）文件读取

通过上面的讲解，我们已经得到了 TEST.TXT 的首簇。现在到作的就是到相应的簇及其后继簇去读取数据了。一直都在说簇，比如第2 簇、第 3 簇等等。

那这些簇在磁盘的什么位置呢？从FAT表中可以看到，簇号是从2开始的，而第2簇的位置就在第二个FAT表（一共有两个FAT表，它们即时同步）的后面，即根目录所在的簇就为第2簇。

下面就为本篇教程的最后部分，读TEST.TXT文件的内容。主要思想是这样的：在已各文件首簇的前提下，从首簇开始，对于文件满一簇的数据，就把整簇数据读出（其实还是按扇区来读，只是一次性读出所有扇区），对于文件结尾不足一簇的部分，计算它占用了簇内几个扇区，把占用整个扇区部分直接按扇区读出，而最后很有可能是零散的若干个字节，不足一个扇区，即占用了最后一个此文件最后一个扇区的一部分，对于这部分我们也要将整个扇区读出，截选中有效的数据部分。文件信息结构中的FileOffset参数将时刻记录文件读到的位置，它与文件大小的差就是还未读取的数据数量。

具体的实现如下：

```
void FAT32_ReadFile(struct FileInfoStruct *pstru,unsigned long len)
{
 unsigned long Sub=pstru->FileSize-pstru->FileOffset;
 unsigned long iSectorInCluster=0;
 unsigned long i=0;
 while(pstru->FileNextCluster!=0x0fffffff)
 //如果FAT中的簇项为0x0fffffff，说明无后继簇
 {
   for(iSectorInCluster=0;
       iSectorInCluster<Init_Arg.SectorsPerClust;
       iSectorInCluster++)
 //读出整簇数据
   {
    FAT32_ReadSector((((pstru->FileCurCluster)-2)
                      *(Init_Arg.SectorsPerClust))
                      +Init_Arg.FirstDataSector
                      +(iSectorInCluster),FAT32_Buffer);
    pstru->FileOffset+=Init_Arg.BytesPerSector;
```

```
    Sub=pstru->FileSize-pstru->FileOffset;
    for(i=0;i<Init_Arg.BytesPerSector;i++)
    {
     send(FAT32_Buffer[i]);     //将数据发送到终端上显示
    }
   }
   pstru->FileCurCluster=pstru->FileNextCluster;
    pstru->FileNextCluster=FAT32_GetNextCluster(
                                                pstru->FileCurCluster);
//这里是 FAT 簇链的传递
  }
  iSectorInCluster=0;
  while(Sub>=Init_Arg.BytesPerSector)
//处理不足一簇，而足扇区的数据
  {
   FAT32_ReadSector((((pstru->FileCurCluster)-2)
                     *(Init_Arg.SectorsPerClust))
                     +Init_Arg.FirstDataSector
                     +(iSectorInCluster++),FAT32_Buffer);
   pstru->FileOffset+=Init_Arg.BytesPerSector;
   Sub=pstru->FileSize-pstru->FileOffset;
   for(i=0;i<Init_Arg.BytesPerSector;i++)
   {
    send(FAT32_Buffer[i]);
   }
  }
  FAT32_ReadSector((((pstru->FileCurCluster)-2)
                     *(Init_Arg.SectorsPerClust))
                     +Init_Arg.FirstDataSector
                     +(iSectorInCluster),FAT32_Buffer);
```

```
//读取最后一个扇区
 for(i=0;i<Sub;i++)      //Sub 为最后剩余的字节数
 {
  send(FAT32_Buffer[i]);
 }
}
```

6）最终的实现

在主函数中对磁盘驱动及以上函数进行正确合理的调用，就可以达到我们要实现的效果了。主函数如下：

```
#include <reg51.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <uart.h>
#include <fat32.h>

void main()
{
 delay(10000);
 UART_Init();  //串口初始化，用以向调试终端发送数据
 send_s("yahoo!!!"); //发送一个测试字符串
 MMC_Init(); //SD 卡初始化
 delay(10000);
 MMC_get_volume_info();        //获得 SD 卡相关信息，输出到终端
 FAT32_Init(&Init_Arg);   //FAT32 文件系统初始化，装入参数
 Printf("BPB_Sector_No"   ,Init_Arg.BPB_Sector_No);
 Printf("Total_Size"      ,Init_Arg.Total_Size     );
 Printf("FirstDirClust"   ,Init_Arg.FirstDirClust);
 Printf("FirstDataSector",Init_Arg.FirstDataSector);
 Printf("BytesPerSector" ,Init_Arg.BytesPerSector);
```

```
Printf("FATsectors"        ,Init_Arg.FATsectors);
Printf("SectorsPerClust",Init_Arg.SectorsPerClust);
Printf("FirstFATSector" ,Init_Arg.FirstFATSector);
Printf("FirstDirSector" ,Init_Arg.FirstDirSector);
//以上几个语句用以输出参数值到终端
Printf("FAT32_OpenFile"
                    ,(FAT32_OpenFile("\\TEST.TXT"))->FileSize);
//打开根目录下的 TEST.TXT 文件，并输出文件大小
FAT32_ReadFile(&FileInfo);   //读取文件数据，输出到终端
while(1);
}
```

最终实现的效果如下图所示：

至此对于 FAT32 文件系统根目录下的文件读取就已经实现了，至于多级子目录结构可以像查找文件的首簇一样查找某一级目录名的首簇，然后再到此簇下去找下一级目录的首簇，直到最终的文件。