《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6章

摘取参数，精准定位：FAT32中的关键部分——DBR

上一章实现了对MBR的解析，介绍了扇区数据提取和参数计算的方法。这一章将继续这样的工作——解析DBR。其实在本章中解析得到的参数才是FAT32文件系统的核心参数，它们构筑起了FAT32文件系统的整体框架，也是我们今后实现各种文件操作的根本基础。同时，我们将看到znFAT中第一个供用户应用层调用的功能函数雏形——znFAT_Init(文件系统初始化)。下面我们就来看一下本章的具体内容。对了，还记得前面振南说过的“假U盘”吗？研究DBR将使我们能够揭露那些U盘制假者的“卑劣伎俩”。本章最后，振南将介绍如何通过修改DBR把一张4 GB的SD卡变成16 GB。

6.1 定位工具：DOS引导记录DBR

1. DBR简介

DBR是什么？DBR是DOS引导记录(DOS Boot Record)的意思，它是怎样一个东西？怎么会和DOS扯上关系？其实起初FAT(FAT12/16/32)文件系统就是在DOS操作系统上开始使用的，因此与DOS存在着无法割断的关系。虽然现在我们更多的是在Windows或Linux操作系统中使用FAT文件系统，但它内部的一些功能部分和概念仍然会沿用当初DOS时代的名称。

DBR其实就是一个参数的仓库或者说集合，包含了与文件系统相关的很多极为重要的参数信息，比如每扇区的字节数、每簇的扇区数等(“簇”的概念振南在前面已经简单说过，本章将深入讲解)，要提取这些参数就要对DBR进行解析。与MBR一样，我们必须首先了解它的存储结构。

2. DBR(BPB)存储结构

DBR与BPB的存储结构具体定义如图6.1所示。DBR扇区同样遵循功能扇区存储结构的定义，最后以标记码“55 AA”结束。DBR扇区有一个非常明显的标志，那就是它的第一个字节为EB(更严格来说应该是跳转指令，字段为EB 58 90)。DBR扇区中的跳转指令、厂商标志以及后面的引导程序代码我们都不必关心，真正

关心的其实是中间的 79 个字节的数据。这 79 个字节就是 BPB(BOIS 参数块，包含了扩展 BPB)，FAT32 的核心参数就是存储在这里了。从图 6.1 可以看到，BPB 中所包含的参数真是不少，其实我们只需要其中的 6 个参数即可，其它的都是“浮云”。这 6 个参数分别是：BytesPerSec(每扇区字节数)、SecPerClus(每簇扇区数)、RsvdSecCnt(保留扇区数)、NumFATs(FAT 表的个数)、FATSz32(FAT 表所占扇区数)、Total_Sec(分区总扇区数)。

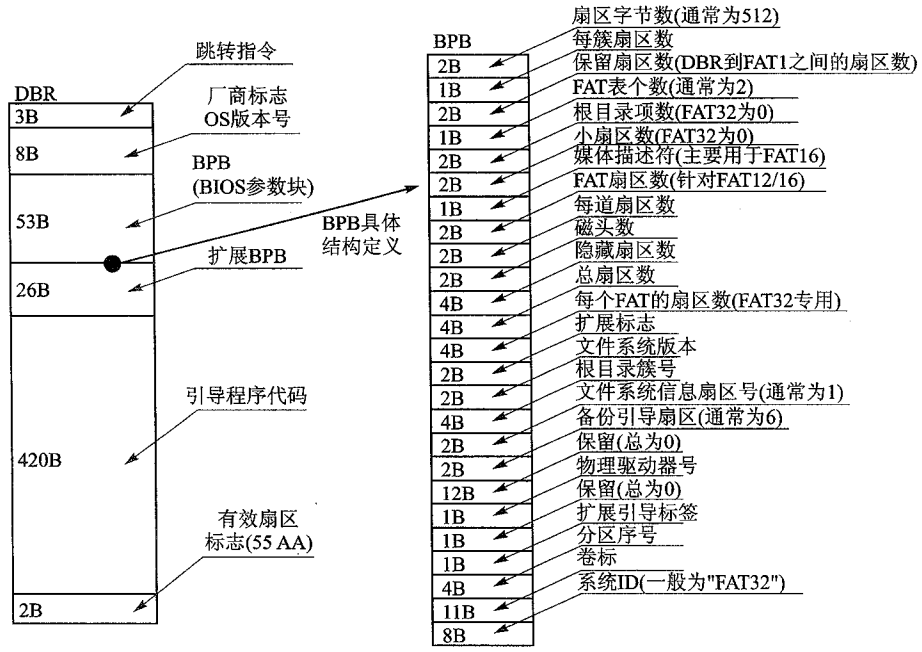

图 6.1　DBR 与 BPB 的存储结构具体定义

这些参数的具体意义可能读者还没有一个形象的认识，还记得第 3 章介绍的 FAT32 整体结构吗？当时我们还画了一张图，标明了主要的几大功能部分及其位置关系。有了上面的这些参数，我们把这些功能部分的位置进行量化。说白了，就是我们可以精准地计算出某个功能部分的具体位置和所占用的扇区数，如图 6.2 所示。

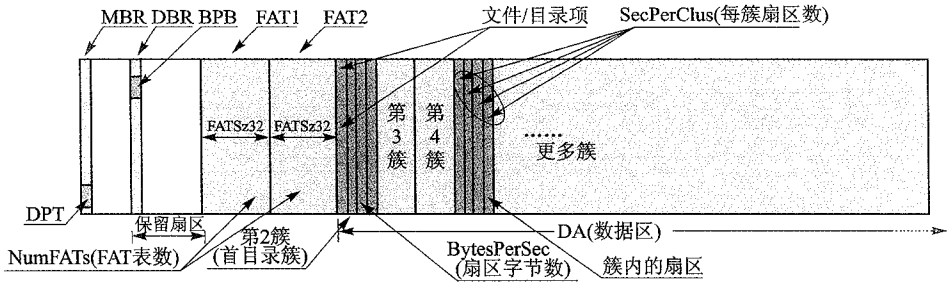

图 6.2　对 FAT32 整体结构的细化与量化

6.2 深入理解：勾勒清晰轮廓

看了上面这张细化后的 FAT32 整体结构图，想必读者心中已经勾勒出了一个更为清晰的轮廓，下面就实际看一下 SD 卡上的 DBR 扇区，并亲手对其中的参数进行解析，最终找出各个功能部分的具体位置，看看它们到底是什么样的？

6.2.1 手工解析 DBR(BPB)

好，现在我们就用 WinHex 打开 SD 卡的 DBR 扇区。说了这么多，我们还不知道 DBR 扇区在哪里？DBR 扇区其实就是分区的开始扇区(逻辑 0 扇区)，这就和第 4 章解析 MBR“挂上钩”了吧。对，分区的开始扇区就是从 MBR 解析得到的。第 4 章解析得到的参数 Part_Start_Sector 的值为 63，那么我们就打开 63 扇区，如图 6.3 所示。

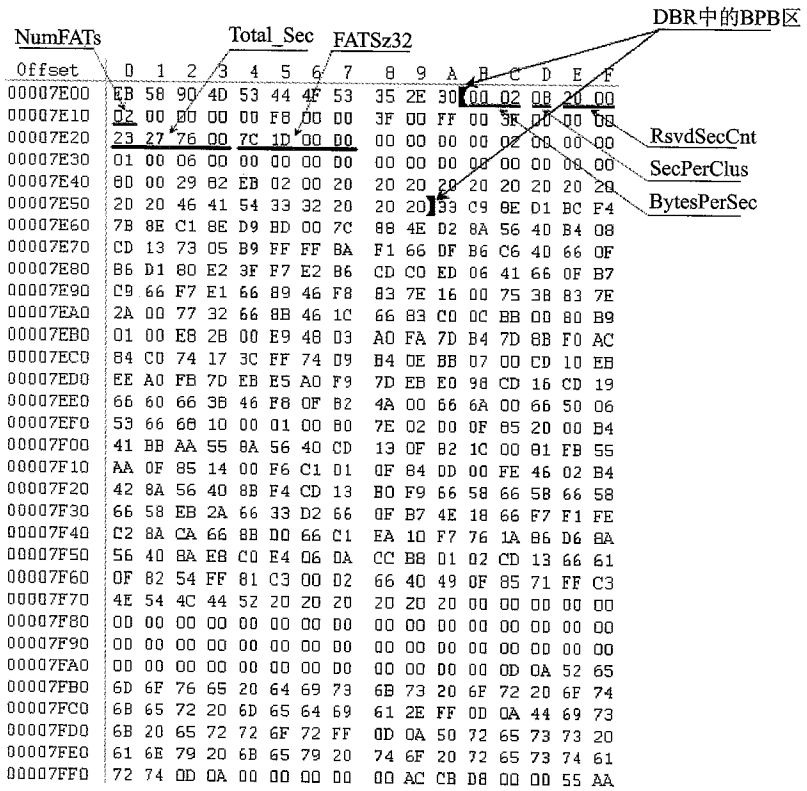

图 6.3 SD 卡上的 DBR 扇区数据及其字段提取

上图中用“【】”标出的部分就是 DBR 中的 BPB，并且我们还标出了上面所说的那 6 个参数所对应的字段位置及长度，计算结果如下：

BytesPerSec＝0X0200（512）　　SecPerClus＝0X08（8）

RsvdSecCnt＝0X0020（32）　　NumFATs＝0X02（2）

FATSz32＝0X00001D7C（7548）　　Total_Sec＝0X00762723(7743267)

解析结果是否正确呢？这一点非常重要！如果这几个核心参数解析的正确性不能得到保证，就不能精准定位各功能部分的位置，又何谈后面的各种文件操作功能呢？

将上面的参数值与 WinHex 软件中的计算结果进行对比，如图 6.4 所示。对比之后可以发现参数的解析是完全正确的。

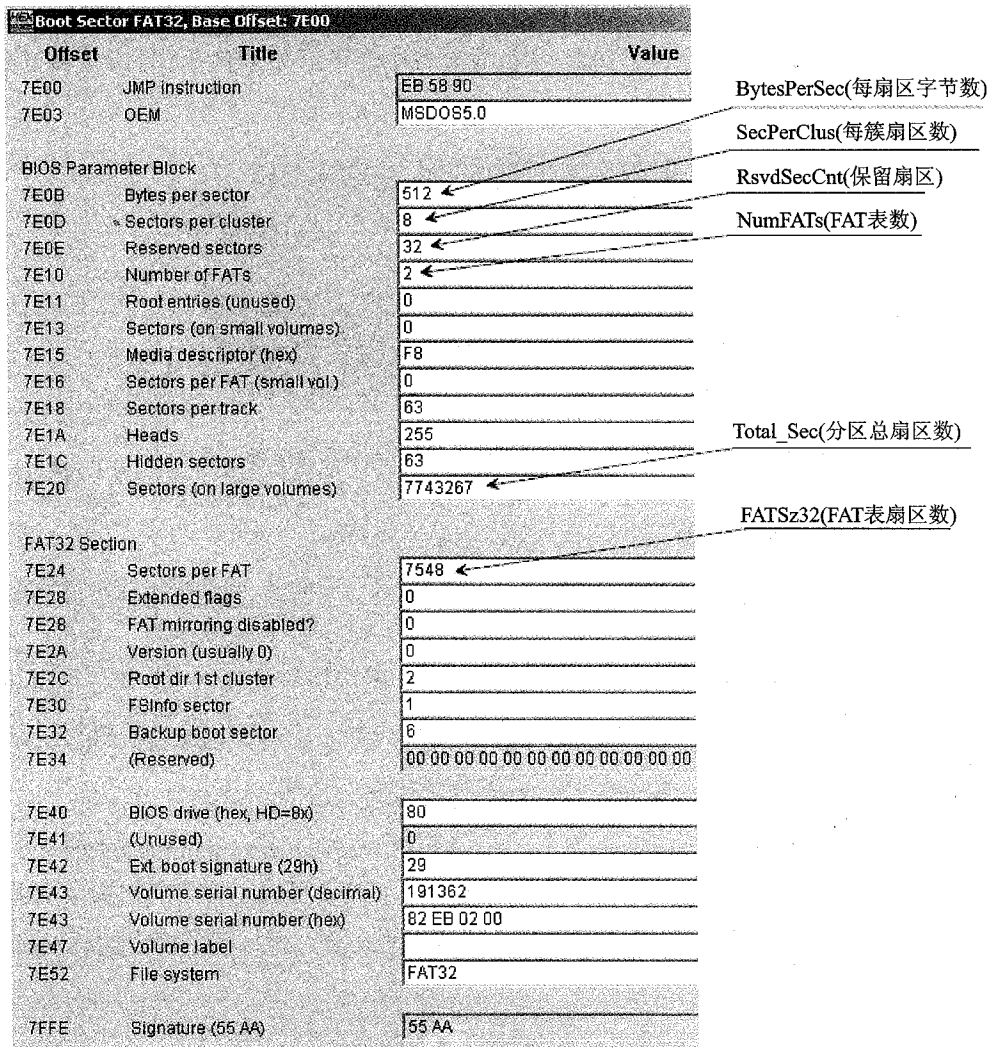

图 6.4　WinHex 软件中计算得到的实际参数值

到这里，可能有读者会问："我现在就想找到 FAT 或首目录簇的开始位置，进到实际扇区里看看它到底长啥样？上面的这 6 个参数好像并没有给出它们的开始扇区地址，那该怎么办呢？"没错！我们可以通过解析的 5 个参数进一步计算出各功能部分的具体位置，计算过程如下：

FirstFATSector(FAT1 的开始扇区)＝Part_Start_Sector(分区开始扇区)＋RsvdSecCnt
＝63＋32＝95

FirstDirSector(首目录簇，即第 2 簇的开始扇区)＝FirstFATSector＋NumFATs×FATSz32
＝95＋2×7 548＝15 191

再将它们与 WinHex 计算结果对比一下，如图 6.5 所示，结果完全一致。不过，还有一些位置参数没有计算，比如 FAT2、后面各个簇的开始扇区等，后面会涉及这些内容。

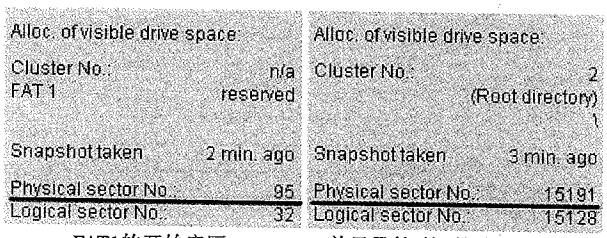

图 6.5　WinHex 软件中计算得到的 FAT1 与首目录簇开始扇区

6.2.2　制作假 U 盘

把 U 盘或 SD 卡插到 PC 上时，Windows 或 Linux 等操作系统就会去读取 DBR 扇区，通过解析其中的参数可计算得到分区容量，最终显示在我们面前。现在已经对 DBR 有了一定的了解，那如果对 DBR 中的参数进行适当的修改，是不是可以骗过 PC 的"眼睛"，显示出"假容量"呢？答案是肯定的。我们来做下面这样一个实验。

将上面所看到的 DBR 扇区数据按图 6.6 进行修改，然后重新将 U 盘或 SD 卡插入 PC 机，看看容量有什么变化，如图 6.7 所示。可以看到磁盘容量已由原来的 3.68 GB 变成了 14.7 GB，一个假 U 盘就这样"诞生"了，而且还可以正常进行各种文件操作，如文件复制、删除等，表面上不会暴露出任何破绽。有一些不法商贩就是用这种方法来给 U 盘"扩容"，以此来蒙骗不懂行的消费者。

分区总扇区数由
7743267改为30973068

簇包含的扇区数由8改为32

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	EB	58	90	4D	53	44	4F	53	35	2E	30	00	02	20	20	00
00000010	02	00	00	00	00	F8	00	00	3F	00	FF	00	3F	00	00	00
00000020	BC	9C	89	1D	7C	1D	00	00	00	00	00	00	02	00	00	00
00000030	01	00	06	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	80	00	29	3D	D0	0C	00	20	20	20	20	20	20	20	20	20
00000050	20	20	46	41	54	33	32	20	20	20	33	C9	8E	D1	BC	F4
00000060	7B	8E	C1	8E	D9	BD	00	7C	88	4E	02	8A	56	40	B4	08
00000070	CD	13	73	05	B9	FF	FF	8A	F1	66	0F	B6	C6	40	66	0F
00000080	B6	D1	80	E2	3F	F7	E2	86	CD	C0	ED	06	41	66	0F	B7
00000090	C9	66	F7	E1	66	89	46	F8	83	7E	16	00	75	38	83	7E
000000A0	2A	00	77	32	66	8B	46	1C	66	83	C0	0C	BB	00	80	B9
000000B0	01	00	E8	2B	00	E9	48	03	A0	FA	7D	B4	7D	8B	F0	AC
000000C0	B4	CD	74	17	3C	FF	74	09	B4	0E	BB	07	00	CD	10	EB
000000D0	EE	A0	FB	7D	EB	E5	A0	F9	7D	EB	E0	98	CD	16	CD	19
000000E0	66	60	66	3B	46	F8	0F	82	4A	00	66	6A	00	66	50	06
000000F0	53	66	68	10	00	01	00	80	7E	02	00	0F	85	20	00	B4
00000100	41	BB	AA	55	8A	56	40	CD	13	0F	82	1C	00	81	FB	55
00000110	AA	0F	85	14	00	F6	C1	01	0F	84	0D	00	FE	46	02	B4
00000120	42	8A	56	40	8B	F4	CD	13	B0	F9	66	58	66	58	66	58
00000130	66	5B	EB	2A	66	33	D2	66	0F	B7	4E	18	66	F7	F1	FE
00000140	C2	8A	CA	66	8B	D0	66	C1	EA	10	F7	76	1A	B6	D6	8A
00000150	56	40	8A	E8	C0	E4	06	0A	CC	B8	01	02	CD	13	66	61
00000160	0F	82	54	FF	81	C3	00	02	66	40	49	0F	85	71	FF	C3
00000170	4E	54	4C	44	52	20	20	20	20	20	20	00	00	00	00	00
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	00	0D	0A	52	65
000001B0	6D	6F	76	65	20	64	69	73	6B	73	20	6F	72	20	6F	74
000001C0	68	65	72	20	6D	65	64	69	61	2E	FF	0D	0A	44	69	73
000001D0	6B	20	65	72	72	6F	72	FF	0D	0A	50	72	65	73	73	20
000001E0	61	6E	79	20	6B	65	79	20	74	6F	20	72	65	73	74	61
000001F0	72	74	0D	0A	00	00	00	00	00	AC	CB	D8	00	00	55	AA

图 6.6　修改之后的 DBR 扇区数据

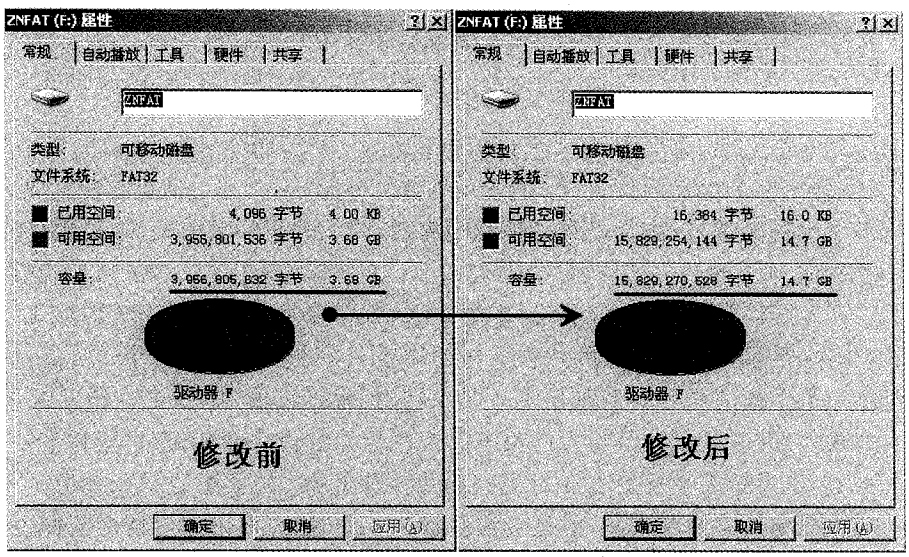

图 6.7　DBR 修改前后磁盘容量被“扩大”了 4 倍

6.2.3 例说“簇”——连锁水桶取水游戏

一直以来，我们都频繁地提到了一个词，那就是“簇”，前文进行过简单的介绍，现在我们已经触及到了 FAT32 较为核心的内容，应该深入理解“簇”的概念了。簇是针对于数据区(DA)来说的，是 FAT32 记录数据的最小存储单位。有人说，扇区不是最小的存储单元吗，现在怎么又说簇？(可见你第 3 章肯定看得不够仔细!)它们其实是针对不同意义或者层面来说的。在物理层面上，扇区确实是存储设备的最小存储单元，是由实际硬件结构决定的。我们编写 SD 卡物理层驱动时是以物理扇区地址来进行操作的，但是我们看到 FAT32 中一个簇包含了若干个扇区(上面我们解析的参数 SecPerClus=8，说明簇中有 8 个扇区，即簇容量为 4 096 字节)，为什么要把最小存储单元重新定义为“簇”？在第三章的基础上，这里再通过下面这个连锁水桶取水游戏来说明，请看图 6.8。

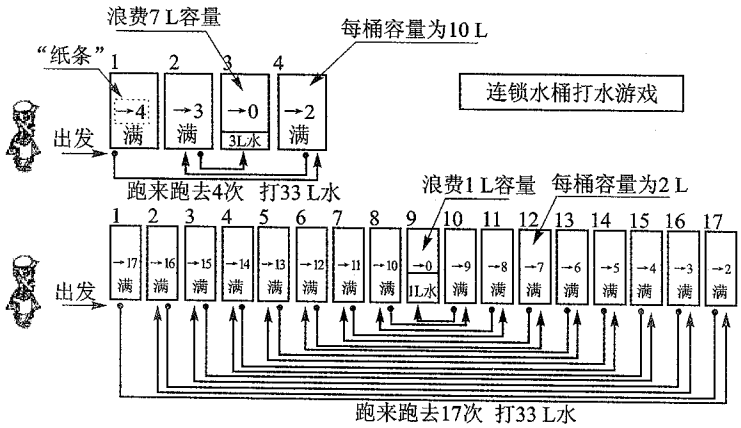

图 6.8 用“连锁水桶取水”实例说明效率问题

在图 6.8 中，上方有 4 个大水桶，其容积均为 10 L，每个水桶上都贴着一个纸条，上面写着下一个水桶的位置(如“→4”，最后一个为“→○”)。只有通过它取水，才能知道下个水桶的编号(游戏规定必须按顺序取水)。如果一共要取 33 L 水(假设前 3 个水桶均满，唯有最后一个水桶不满，仅有 3 L 水，即浪费了 7 L 容量)，那我们就要跑 4 次。下方同样是要取 33 L 水，分装在 17 个小水桶中，每个水桶容积为 2 L，我们就要跑 17 次，足足多了好几倍的路程。但是最后一个水桶仅仅浪费了 1 L 容量。这里就存在一个平衡问题：我们是宁愿浪费一些水桶的容量，还是想少跑几次腿呢？相信大多数人都选择后者，因为人们都不喜欢作劳神费力的事情，都希望作事效率能高一些，尤其是当最终收益相当的时候。翻过头来想想，如果要取的水量是 1 L，按图中上方的情况只需要跑 1 次，但是会浪费桶 9 L 的容量；而下方的情况同样也只需要跑 1 次，却只浪费了 1 L 的容量。此时，似乎上面的情况就丝毫不占优势了。那也没有办法，任何方案都会有其最糟糕的情况，我们只能求得平均意义上的优势，这也

是一个时空平衡的例子，时间（效率、速度）与空间（体积、容量）历来就是一对矛盾。

我们将上面这个例子影射到 FAT32 中的数据读/写。扇区如同小水桶，而簇就像是大水桶：到扇区和簇上去读写数据就相当于在多个水桶中依次取水；水桶上贴的纸条就是 FAT 表中记录的链式关系，告诉我们下一个存储单元在哪里；跑腿的过程就是在 FAT 表中查找和寻址的过程。很明显，在读/写相同量的数据时，簇可以使我们少查几次 FAT 表，而把更多的时间放在读/写数据上，从而极大地提高了数据读/写的效率。

关于簇，我们要深入地去理解，它是 FAT32 进行数据存储的重要概念，后面还会继续从各种不同角度进行介绍。

另外，有人经常会问："为什么簇是从第 2 簇开始的？"确实，我们看到首目录开始于第 2 簇，而没有看到过第 0 簇和第 1 簇。振南只能说这是 FAT32 的定义，0 簇与 1 簇可能另有它用。但是我们知道了首目录簇为第 2 簇，又知道了它的开始扇区（即整个数据区的开始扇区），就可以知道后面任意一个簇的开始扇区。（准确计算簇的开始扇区很重要，否则我们如何到簇里面去读写数据呢？）znFAT 中有这样的一个宏定义，用它就可以计算得到任意簇开始扇区（位于 znFAT.h 中）：

```
#define  SOC(c)  (((c-2)*SecPerClus)+FirstDirSector)
```

其中，SOC 就是 Start Sector of Cluster 的缩写（注意不是"片上系统"的意思，跟那没有关系，此处纯属"雷同"）。这个宏是一个带有参数的宏，参数 c 就是要计算开始扇区的簇。

6.3　znFAT 的初始化函数

6.3.1　DBR 解析的程序实现

直接来看 DBR 解析的相关代码。

znFAT.h 代码如下：

```
//znFAT 中对 DBR 扇区数据结构的定义如下
struct DBR
{
    UINT8 BS_jmpBoot[3];                    //跳转指令
    UINT8 BS_OEMName[8];                    //OEM 名称
     //以下是 BPB 区
    UINT8 BPB_BytesPerSec[2];               //每扇区字节数
    UINT8 BPB_SecPerClus;                   //每簇扇区数
    UINT8 BPB_RsvdSecCnt[2];                //保留扇区数目
    UINT8 BPB_NumFATs;                      //此分区中的 FAT 表数
```

```
    UINT8 BPB_RootEntCnt[2];                //FAT32 固定为 0
    UINT8 BPB_TotSec16[2];                  //FAT32 固定为 0
    UINT8 BPB_Media;                        //存储介质
    UINT8 BPB_FATSz16[2];                   //FAT32 固定为 0
    UINT8 BPB_SecPerTrk[2];                 //磁道扇区数
    UINT8 BPB_NumHeads[2];                  //磁头数
    UINT8 BPB_HiddSec[4];                   //FAT 区前隐扇区数
    UINT8 BPB_TotSec32[4];                  //该分区总扇区数
    UINT8 BPB_FATSz32[4];                   //一个 FAT 表扇区数
    UINT8 BPB_ExtFlags[2];                  //FAT32 特有
    UINT8 BPB_FSVer[2];                     //FAT32 特有
    UINT8 BPB_RootClus[4];                  //根目录簇号
    UINT8 FSInfo[2];                        //保留扇区 FSINFO 扇区数
    UINT8 BPB_BkBootSec[2];                 //通常为 6
    UINT8 BPB_Reserved[12];                 //扩展用
    UINT8 BS_DrvNum;
    UINT8 BS_Reserved1;
    UINT8 BS_BootSig;
    UINT8 BS_VolID[4];
    UINT8 BS_FilSysType[11];
    UINT8 BS_FilSysType1[8];
};
int znFAT_Aanalysis_DBR(void);              //函数声明
```

znFAT.c 代码如下：

```
//以下是 DBR 中的 5 个核心参数
UINT16 BytesPerSec;                         //每扇区字节数
UINT8 SecPerClus;                           //每簇扇区数
UINT16 RsvdSecCnt;                          //保留扇区数
UINT8 NumFATs;                              //FAT 表的个数
UINT32 FATSz32;                             //FAT 表的扇区数
//以下是由 DBR 核心参数计算得到的参数
UINT32 FirstFATSector;                      //FAT1 开始扇区
UINT32 FirstDirSector;                      //首目录簇开始扇区
int znFAT_Aanalyse_DBR(void)
{
 struct DBR * pStru;                        //定义一个结构体指针
 SD_Read_Sector(Part_Start_Sector,buffer);  //将 DBR 扇区数据读到缓冲区
 pStru = (struct DBR * )buffer;
 BytesPerSec = Bytes2Value(pStru->BPB_BytesPerSec,2);
 SecPerClus = pStru->BPB_SecPerClus;
 RsvdSecCnt = Bytes2Value(pStru->BPB_RsvdSecCnt,2);
```

```
 NumFATs = pStru->BPB_NumFATs;
 FATSz32 = Bytes2Value(pStru->BPB_FATSz32,4);
 FirstFATSector = Part_Start_Sector + RsvdSecCnt;
 FirstDirSector = FirstFATSector + NumFATs * FATSz32;
 return 0;
}
```

_main.c 代码如下:

```
#include "tft.h"
#include "sdx.h"
#include "uart.h"
#include "znfat.h"
//对外部变量进行声明
extern UINT8 FileSys_Type;
extern UINT32 Part_Start_Sector;
extern UINT32 Part_Total_nSec;
extern UINT16 BytesPerSec;
extern UINT8 SecPerClus;
extern UINT16 RsvdSecCnt;
extern UINT8 NumFATs;
extern UINT32 FATSz32;
extern UINT32 FirstFATSector;
extern UINT32 FirstDirSector;
void main(void)
{
 UART_Init();
 UART_Send_Str("UART Init..\r\n");
 TFT_Init();
 UART_Send_Str("TFT Init..\r\n");
 SD_Init();
 UART_Send_Str("SD Init..\r\n");
 TFT_Clear(COLOR_WHITE);
 UART_Send_Str("TFT Clear..\r\n");
 znFAT_Analysis_MBR();          //首先解析 MBR,获取 DBR 扇区地址
 UART_Send_Str("MBR Analysis..\r\n");
 znFAT_Analysis_DBR();          //解析 DBR
 //通过 TFT 液晶输出结果
 TFT_Send_Str("DBR Analysis:",8,0,COLOR_BLUE,COLOR_WHITE);
 TFT_Put_Inf("BytePerSec:",BytesPerSec,8,40,COLOR_RED,COLOR_WHITE);
 TFT_Put_Inf("SecPerClus:",SecPerClus,8,72,COLOR_REDCOLOR_WHITE);
 TFT_Put_Inf("RsvdSecCnt:",RsvdSecCnt,8,104,COLOR_RED,COLOR_WHITE);
 TFT_Put_Inf("NumFATs:",NumFATs,8,136,COLOR_RED,COLOR_WHITE);
```

```
    TFT_Put_Inf("FATSz32:",FATSz32,8,168,COLOR_RED,COLOR_WHITE);
    TFT_Put_Inf("FAT1Sec:",FirstFATSector,8,136,COLOR_RED,COLOR_WHITE);
    TFT_Put_Inf("1DirSec:",FirstDirSector,8,168,COLOR_RED,COLOR_WHITE;
    //通过串口输出结果
    UART_Send_Str("DBR Analysis:\r\n");
    UART_Put_Inf("BytePerSec:",BytesPerSec);
    UART_Put_Inf("SecPerClus:",SecPerClus);
    UART_Put_Inf("RsvdSecCnt:",RsvdSecCnt);
    UART_Put_Inf("NumFATs:",NumFATs);
    UART_Put_Inf("FATSz32:",FATSz32);
    UART_Put_Inf("FAT1Sec:",FirstFATSector);
    UART_Put_Inf("1DirSec:",FirstDirSector);
    while(1);
}
```

6.3.2 初始化参数集合

我们看到，无论是上一章解析 MBR，还是本章中解析 DBR，程序中的参数变量都是被定义为单个的全局变量。前面说过，这种方式显得很零乱，那为何不把这些变量都纳入到一个结构体中进行统一管理呢？好，这里就把这个包含了核心参数的结构体称为初始化参数集合。将各个参数装入其中的过程，就是文件系统的初始化(znFAT_Init)。这个结构体变量被定义为全局的，命名为 znFAT_Init_Args，下面是它的具体定义：

znFAT.h 代码如下：

```
struct  znFAT_Init_Arg
{
    UINT32   DBR_Sector_No;          //DBR(BPB)所在扇区
    UINT32   BytesPerSector;         //每个扇区的字节数
    UINT32   FATsectors;             //FAT 表所占扇区数
    UINT32   SectorsPerClust;        //每簇的扇区数
    UINT32   FirstFATSector;         //第一个 FAT 表所在扇区
    UINT32   FirstDirSector;         //第一个目录所在扇区
    UINT32   Total_SizeKB;           //磁盘的总容量
};
```

结构体的第一个参数是从 MBR 中解析得到的(如果 MBR 存在的话)，即 Part_Start_Sector。后面的 6 个参数是从 DBR 解析计算得到的。所以，初始化的过程其实包含了我们这两章的工作。

接下来我们就对 znFAT_Init 函数进行实现，代码如下：

znFAT.h 代码如下：

```
#define MBR_SECTOR (0)                    //MBR扇区
#define DBR_MARK {0XEB,0X58,0X90}         //DBR扇区的标记(跳转指令)
```

znFAT.c代码如下:

```
struct  znFAT_Init_Arg  Init_Args;    //初始化参数集合,用于装载核心参数
int znFAT_Init(void)
{
 struct DBR * pdbr;
 UINT8 dm[3] = DBR_MARK;
 SD_Read_Sector(MBR_SECTOR,buffer);          //读取物理0扇区
 if(! (buffer[0] == dm[0] && buffer[1] == dm[1] && buffer[2] == dm[2]))
                                             //如果物理0扇区不是DBR扇区,则是MBR
 {
  //由MBR计算DBR扇区地址
  Init_Args.DBR_Sector_No = Bytes2Value(((((struct MBR_Sector * )buffer) -> Part) ->
                                                                  StartLBA),4);
 }
 else                                        //如果是DBR扇区
 {
  Init_Args.DBR_Sector_No = 0;
 }
 SD_Read_Sector(Init_Args.DBR_Sector_No,buffer);  //读取DBR扇区
 pdbr = (struct DBR * )buffer;
 Init_Args.BytesPerSector    = Bytes2Value((pdbr -> BPB_BytesPerSec),2);
                                             //装入每扇区字节数到BytesPerSector中
 Init_Args.FATsectors        = Bytes2Value((pdbr -> BPB_FATSz32),4);
                                             //装入FAT表占用的扇区数到FATsectors中
 Init_Args.SectorsPerClust    = pdbr -> BPB_SecPerClus;
                                             //装入每簇扇区数到SectorsPerClust中
 Init_Args.FirstFATSector    = Bytes2Value((pdbr -> BPB_RsvdSecCnt),2)
                                                  + (Init_Args.BPB_Sector_No);
                                             //装入FAT1开始扇区到FirstFATSector中
 Init_Args.FirstDirSector    = (Init_Args.FirstFATSector)
                                  + (pdbr -> BPB_NumFATs) * (Init_Args.FATsectors);
                                             //装入首目录开始扇区到FirstDirSector中
 Init_Args.Total_SizeKB    = Bytes2Value((pdbr -> BPB_TotSec32),4)/2;
                                             //磁盘的总容量,单位是KB
 return 0;
}
```

上面的代码中加入了对DBR扇区的判断。如果前面的3个字节不是"EB 58 90",那么物理0扇区就是MBR,此时DBR扇区(即分区的开始扇区)需要从其中解

析得到；否则，物理 0 扇区就直接是 DBR，Init_Args. BPB_Sector_No 的值取 0。

6.3.3 硬件平台验证实验

下面通过一个测试程序对上面实现的文件系统初始化函数进行验证，看看装入到初始化参数集合中的的参数是否正确。

_main. c 代码如下：

```
//包含相关头文件
extern struct  znFAT_Init_Arg  Init_Args;  //对外部变量进行声明
void main(void)
{
 UART_Init();
 UART_Send_Str("UART Init..\r\n");
 TFT_Init();
 UART_Send_Str("TFT Init..\r\n");
 SD_Init();
 UART_Send_Str("SD Init..\r\n");
 TFT_Clear(COLOR_WHITE);
 UART_Send_Str("TFT Clear..\r\n");
 znFAT_Init();
 UART_Send_Str("znFAT Init..\r\n");
 //通过 TFT 液晶输出结果
 TFT_Draw_Background(); //画背景
 TFT_Put_Inf("DBR_Sec_No:",Init_Args.DBR_Sector_No,8,40,
                                          COLOR_BLUE,COLOR_WHITE);
 TFT_Put_Inf("BytePerSec:",Init_Args.BytesPerSector,8,72,
                                          COLOR_BLUE,COLOR_WHITE);
 TFT_Put_Inf("FAT_Secs:",Init_Args.FATsectors,8,104,
                                          COLOR_BLUE,COLOR_WHITE);
 TFT_Put_Inf("SecPerClu :",Init_Args.SectorsPerClust,8,136,
                                          COLOR_BLUE,COLOR_WHITE);
 TFT_Put_Inf("1stFATSec :",Init_Args.FirstFATSector,8,168,
                                          COLOR_BLUE,COLOR_WHITE);
 TFT_Put_Inf("1DirSec :",Init_Args.FirstDirSector,8,200,
                                          COLOR_BLUE,COLOR_WHITE);
 TFT_Put_Inf("Tt_KB :",Init_Args.Total_SizeKB,8,232,
                                          COLOR_BLUE,COLOR_WHITE);
 //通过串口输出结果
 UART_Send_Str("Init Args:\r\n");
 UART_Put_Inf("DBR_Sec_No:",Init_Args.DBR_Sector_No);
 UART_Put_Inf("BytePerSec:",Init_Args.BytesPerSector);
```

```
    UART_Put_Inf("FATsectors:",Init_Args.FATsectors);
    UART_Put_Inf("SecPerClu :",Init_Args.SectorsPerClust);
    UART_Put_Inf("1stFATSec :",Init_Args.FirstFATSector);
    UART_Put_Inf("1stDirSec :",Init_Args.FirstDirSector);
    UART_Put_Inf("Tt_SizeKB :",Init_Args.Total_SizeKB);
    while(1);
}
```

程序运行结果如图 6.9 与 6.10 所示。可见 DBR 参数解析结果完全正确。znFAT_Init 函数为后面诸多的文件操作函数的实现奠定了重要的基础。

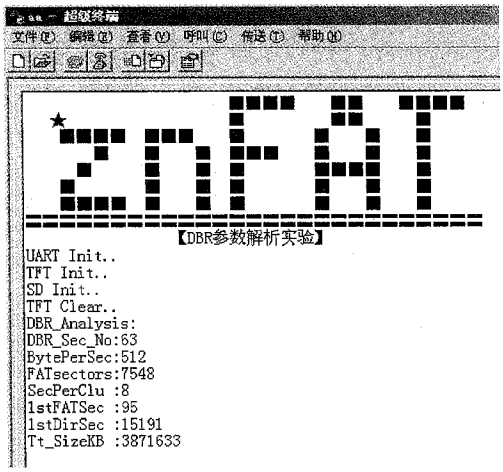

图 6.9　znFAT_Init 函数测试程序串口输出信息

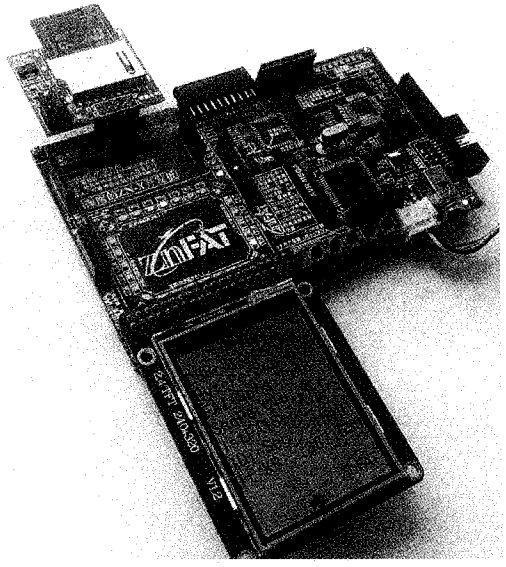

图 6.10　znFAT_Init 测试程序 TFT 液晶输出信息

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