《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5章

轻踏上路，初涉分析：开启 FAT32 文件系统之门

经过上一章的介绍，我们大体了解了 FAT32 文件系统的整体结构以及各功能部分之间的关联，但是认识仍然十分模糊。从本章开始我们就深入到各个部分进行详细讲解。这里将介绍 FAT32 第一个功能部分——MBR（主引导记录）。可以说，它是 FAT32 文件系统的入口，甚至是磁盘上各个分区文件系统的入口；如果对它的定位不准或解析不正确，那么以后的各种文件系统相关的操作均无从谈起。它到底为何如此重要，请看下文。

5.1 FAT32 文件系统的入口——主引导记录 MBR

1. MBR 简介

MBR 是文件系统的入口，为什么会这么说呢？我们知道一个磁盘上是可以有多个分区的，就是平时说的 C 盘、D 盘等。每一个分区可以是不同的文件系统，比如 FAT16、FAT32 或 NTFS 等，如图 5.1 所示。

卷	布局	类型	文件系统	状态	容量	空闲空间
(C:)	磁盘分区	基本	NTFS	状态良好 (系统)	9.71 GB	422 MB
(D:)	磁盘分区	基本	FAT32	状态良好	5.18 GB	264 MB
Mobile Card (F:)	磁盘分区	基本	CDFS	状态良好	27 MB	0 MB
ZNMCU	磁盘分区	基本	FAT32	状态良好 (活动)	955 MB	955 MB
ZNMCU (H:)	磁盘分区	基本	FAT32	状态良好	3.65 GB	3.65 GB

图 5.1 磁盘上各分区的不同文件系统

从图 5.1 中可以看到，C 盘与 D 盘是属于同一个物理磁盘的两个分区，使用的文件系统分别为 NTFS 与 FAT32（分区的文件系统不同，是在格式化的时候决定的，表明分区采用某种文件管理方案进行文件及存储空间的管理）。我们要研究 FAT32，自然要选择文件系统为 FAT32 的分区，以后对文件的各种操作也都是在其上进行的。说白了，分区就是对磁盘进行了划分，每一个分区都有自己的开始扇区和结束扇区，这一区间内的存储空间即为分区的“势力范围”。如何知道一个磁盘上有几个分

区？每一个分区使用的是什么文件系统？分区从哪个扇区开始和结束？分区的容量有多大？这一切都要由 MBR 来告诉我们。有人问了："MBR 在哪呢？"答："绝对 0 扇区。"（如果一个磁盘上有 MBR，那它就必定在绝对 0 扇区，为什么这么说，后面会进行解释。）

2. MBR 结构定义

MBR 以其特定的数据格式记录了分区的相关信息，只有了解这种存储格式才能正确提取出所需的参数。那我们就来看一下 MBR 存储结构的具体定义，如图 5.2 所示。

从图 5.2 中可以看到，前 446 个字节似乎是一团乱麻，其实它们是一些引导代码，不必深究。最后的两个字节是"55 AA"，通常被称为"标记码"，用来告诉我们这是一个合法有效的功能扇区，起到一个校验的作用。最关心的其实是中间的 64 个字节，它们就是 DPT（Disk Partition Table，即磁盘分区表），分区的相关信息就记录在其中了。这 64 个字节包含了 4 个表项，每一个表项 16 字节，具体定义如表 5.1 所列。

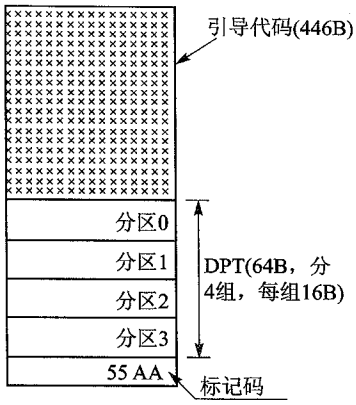

图 5.2 MBR 扇区存储结构的具体定义

表 5.1 DPT 表项的具体定义

字节偏移	字段长度	参数字段功能描述
0	1B	引导指示符：指明此分区是否为活动分区。取值为 0X80，说明可从此分区引导操作系统，通常取值为 0
1	1B	开始磁头
2	2B	前 6 位为开始扇区，后 10 位为开始柱面
4	1B	系统 ID：用于定义分区类型，详见表 5.2
5	1B	结束磁头
6	2B	前 6 位为结束扇区，后 10 位为结束柱面
8	4B	分区的开始扇区地址
12	4B	分区的总扇区数

乍看上去，感觉参数字段的功能比较抽象，其实这些并非都要关心，只选其中 3 个即可，分别是分区开始扇区、总扇区数与文件系统类型 ID。我们要做的就是把这些字段提取出来，并计算得到相应的参数。下面我们就来看看实际的 MBR 是怎样的？又是如何把参数提取计算出来的。

5.2 “手工解析”MBR——基于 WinHex

1. 使用 WinHex 解析

将一张容量为 4 GB 的 SD 卡格式化为 FAT32 格式，使用 WinHex 软件来查看它的 0 扇区(注意此 0 扇区为物理 0 扇区，要使用 WinHex 的物理模式)，如图 5.3 所示。图中的标注之处就是 DPT 中的第一个表项，可以看到它是有数据的，而后面一直到“55 AA”均为 0(后 3 个表项无效)，说明在这张 SD 卡上仅有一个分区(通常情况下，我们很少在 SD 卡这种移动存储介质上分出多个区，所以后面的代码中也只是针对这第一个表项进行解析处理的，而不处理多分区的情况，也就是说后面的 DPT 表项都将忽略)。那我们就结合上面对 DPT 的定义来对这个表项进行尝试性解析，看看会得到怎样的参数，这些参数是否与分区的实际参数相符。这也是我们第一次进行参数块的解析，读者会看到其中的问题和技巧。把这个表项单拿出来，按其定义对字段进行划分，如图 5.4 所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	33	C0	8E	D0	BC	00	7C	FB	50	07	50	1F	FC	50	BE	00
00000010	7C	BF	00	06	B9	00	02	F3	A4	BF	1E	06	57	CB	33	DB
00000020	33	D2	BE	BE	07	B1	04	F6	04	80	74	03	8B	D6	43	83
00000030	C6	10	E2	F3	83	FB	01	74	09	BE	C4	00	B9	17	00	EB
00000040	71	90	52	B4	41	B2	80	BB	AA	55	CD	13	5A	81	FB	55
00000050	AA	75	33	F6	C1	01	74	2E	B8	00	42	BE	AD	07	B1	10
00000060	C6	04	00	46	E2	FA	BE	AD	07	8B	FA	C6	04	10	C6	44
00000070	02	01	C7	44	04	00	7C	8B	5D	08	89	5C	08	8B	5D	0A
00000080	89	5C	0A	EB	0F	90	B8	01	02	BB	00	7C	8B	F2	8B	4C
00000090	02	8A	74	01	B2	80	CD	13	BE	FE	7D	81	3C	55	AA	74
000000A0	09	BE	DB	00	B9	18	00	EB	09	90	33	C0	50	B8	00	7C
000000B0	50	CB	81	C6	00	06	AC	BB	07	00	B4	0E	CD	10	E2	F6
000000C0	B1	0F	E2	FC	49	6E	76	61	6C	69	64	20	70	61	72	74
000000D0	69	74	69	6F	6E	20	74	61	62	6C	65	4D	69	73	73	69
000000E0	6E	67	20	6F	70	65	72	61	74	69	6E	67	20	73	79	73
000000F0	74	65	6D	00	4D	61	73	74	65	72	20	42	6F	6F	74	20
00000100	52	65	63	6F	72	64	20	57	72	6F	74	65	20	62	79	20
00000110	4D	42	52	20	42	79	20	44	69	73	6B	47	65	6E	69	75
00000120	73	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000170	00	00	00	00	00	00	00	00	00	00	00	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	00	00	00
000001B0	00	00	00	00	00	00	00	00	B7	C6	D7	05	72	20	80	01
000001C0	01	00	0B	FE	7F	E1	3F	00	00	00	23	27	76	00	00	00
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA

图 5.3 物理 0 扇区(MBR)中的数据

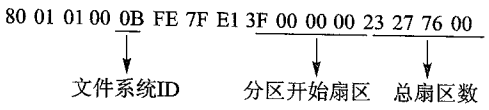

图 5.4 对 DPT 表项字段的划分

其中，“0B”是文件系统类型，这个数字具体代表什么呢，请看表 5.2。从表中就可以知道这个分区上的文件系统是 FAT32 了。

表 5.2 文件系统类型标志码

分区文件系统类型标志码	文件系统	分区文件系统类型标志码	文件系统
00	空，非法值	5C	Priam Edisk
01	FAT32	61	Speed Stor
02	XENIX root	63	GNU HURD or sYS
03	XENIX usr	64	Novell Netware
04	FAT16 <32M	65	Novell Netware
05	Extended	70	Disk Secure Mult
06	FAT16	75	PC/IX
07	HPFS/NTFS	80	Old Minix
08	AIX	81	Minix/Old Linux
09	AIX bootable	82	Linux swap
0A	OS/2 Boot Manage	83	Linux
0B	Win95 FAT32	84	OS/2 hidden C:
0C	Win95 FAT32	85	Linux extended
0E	Win95 FAT16	86	NTFS volume set
0F	Win95 Extended(>8GB)	87	NTFS volume set
10	OPUS	93	Amoeba
11	Hidden FAT12	94	Amoeba BBT
12	Compaq diagnost	A0	IBM Thinkpad hidden
16	HiddenFAT16	A5	BSD/386
14	Hidden FAT16<32GB	A6	Open BSD
17	Hidden HPFS/NTFS	A7	NextSTEP
18	AST Windows swap	B7	BSDI fs
1B	Hidden FAT32	B8	BSDI swap
1C	Hidden FAT32 partition (using LBA-mode INT13 extensions)	BE	Solaris boot partition
		C0	DR-DOS/Novell DOS secured partition
1E	Hidden LBA VFAT partition	C1	DRDOS/sec
24	NEC DOS	C4	DRDOS/sec

续表 5.2

分区文件系统类型标志码	文件系统	分区文件系统类型标志码	文件系统
3C	Partition Magic	C6	DRDOS/sec
40	Venix 80286	C7	Syrinx
41	PPC PreP Boot	DB	CP/M/CTOS
42	SFS	E1	DOS access
4D	QNX4. x	E3	DOS R/O
4E	QNX4. x 2nd part	E4	SpeedStor
4F	QNX4. x 3rd part	EB	BeOS fs
50	Ontrack DM	F1	SpeedStor
51	Ontrack DM6 Aux	F2	DOS 3. 3+ secondary partition
52	CP/M		
53	Ontrack DM6 Aux	F4	SpeedStor
54	Ontrack DM6	FE	LAN step
55	EX-Drive	FF	BBT
56	Golden Bow		

随后是分区的开始扇区与总扇区数，分别对应“3F 00 00 00”与“23 27 76 00”这两个字段。4 个字节？是否联想到了 C 语言里的无符号长整型 unsigned long 呢？对，该字段里存储的就是用于构成长整型变量的那 4 个字节。这样一来，我们就知道了分区开始扇区与总扇区数的值为 0X0000003F＝63 与 0X007627232＝7743267（FAT32 文件系统中存放字段的顺序是低字节在前，高字节在后，这就是小端模式，反之称为大端模式。至于为什么采用这种存储顺序，又为什么称之为“大小端”，我们后面会有解释）。

2. 解析结果的验证

上面对 MBR 中的第一个 DPT 表项进行了手工解析。那解析的结果是否正确呢？是否有一个准确的参考值呢？当然有，WinHex 就可以告诉我们，如图 5.5 所示。可见我们的解析是正确的，与 WinHex 得到的结果完全相符。

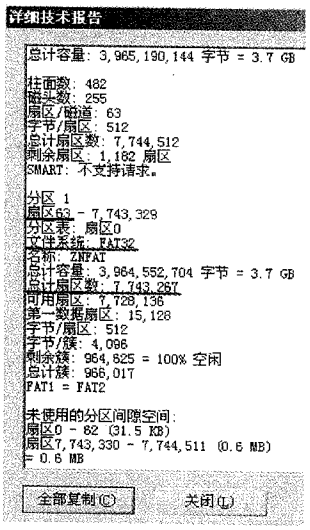

图 5.5 WinHex 中得到的分区信息

5.3 解析MBR的程序实现

5.3.1 笨实现方法

尽管已经得到了我们想要的参数，但最终还是不能用手工方式来人为解析，而是要用程序代码来实现，让CPU去完成解析的工作。振南曾经把MBR解析的程序实现作为一道题目交给一个学生去完成，给出了下面这段代码：

```
#include "tft.h"
#include "sdx.h"
#include "uart.h"
#define MBR_SECTOR 0                    //MBR 扇区
//定义各字段开始位置
#define FILESYS_TYPE        450         //文件系统类型
#define PART_START_SECTOR 454           //分区开始扇区
#define PART_TOTAL_NSEC    458          //分区总扇区数
unsigned char buffer[512];              //扇区数据缓冲区
unsigned char FileSys_Type = 0;
unsigned long Part_Start_Sector = 0;
unsigned long Part_Total_nSec = 0;
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");
 SD_Read_Sector(MBR_SECTOR,buffer);   //将 MBR 扇区数据读到缓冲区
 FileSys_Type = buffer[FILESYS_TYPE]; //提取文件系统类型码
 Part_Start_Sector = (((UINT32)buffer[PART_START_SECTOR + 0]))     |
                     (((UINT32)buffer[PART_START_SECTOR + 1])<<8) |
                     (((UINT32)buffer[PART_START_SECTOR + 2])<<16)|
                     (((UINT32)buffer[PART_START_SECTOR + 3])<<24) ;
                     //计算得到分区开始扇区
 Part_Total_nSec = (((UINT32)buffer[PART_TOTAL_NSEC + 0]))     |
                   (((UINT32)buffer[PART_TOTAL_NSEC + 1])<<8) |
                   (((UINT32)buffer[PART_TOTAL_NSEC + 2])<<16)|
```

```
                (((UINT32)buffer[PART_TOTAL_NSEC + 3])<<24) ;
                //分区总扇区数
    //通过 TFT 液晶输出结果
    TFT_Send_Str("MBR Analysis:",8,0,0x001f,0xffff);
    TFT_Put_Inf("FS Type:",FileSys_Type,8,40,0xf800,0xffff);
    TFT_Put_Inf("Start Sec:",Part_Start_Sector,8,72,0xf800,0xffff);
    TFT_Put_Inf("Total:",Part_Total_nSec,8,104,0xf800,0xffff);
    //通过串口输出结果
    UART_Send_Str("MBR Analysis:\r\n");
    UART_Put_Inf("FS Type:",FileSys_Type);
    UART_Put_Inf("Start Sec:",Part_Start_Sector);
    UART_Put_Inf("Total:",Part_Total_nSec);
    while(1);
}
```

这段代码其实写得比较规范，功能上也确实是正确的。它首先定义了各字段在扇区数据中的开始位置，然后通过移位拼接的方式计算得到相应的参数值。看似没有问题，其实这种实现方法并不是很好：

① 当我们要提取的参数比较多时，程序中会有大量的字段位置宏定义，使程序较为凌乱，可读性差；

② 程序中无法体现出数据的存储结构，看程序时必须结合相关文档的说明，才能有一个比较形象的认识；

③ 用于记录参数的变量是分散开的，如 FileSys_Type 或 Part_Start_Sector，犹如一盘碎石，散落在程序中，不好管理。

5.3.2 改进方法 1：结构化实现方法

结构化的实现方法就是使用结构体的方式来完成数据的解析。我们根据上面对 MBR 扇区的定义可以得到这样一个结构体，代码如下：

```
struct Part_Record
{
    UINT8 Active;                   //引导指示符
    UINT8 StartHead;                //分区的开始磁头
    UINT16 StartCylSect;            //开始柱面与扇区
    UINT8 PartType;                 //分区类型
    UINT8 EndHead;                  //分区的结束头
    UINT16 EndCylSec;               //结束柱面与扇区
    UINT32 StartLBA;                //分区的第一个扇区
    UINT32 TotalSector;             //分区的总扇区数
};
struct MBR_Sector
```

```
{
    UINT8 PartCode[446];            //MBR 的引导程序
    struct PartRecord Parts[4];     //4 个分区记录
    UINT8 BootSectSig0;             //55
    UINT8 BootSectSig1;             //AA
};
```

这两个结构体就对 MBR 的结构定义进行了全面而清晰的描述，相信程序中有了它，就算不看文档也能知道 MBR 的存储结构和字段的意义了。其实结构体就是定义了一个模板或者说框架，使得原始的、无具体意义的扇区数据在逻辑上有了特定的结构，得以功能性地划分。图 5.6 进行了更加形象说明（读者最好要有 C 语言指

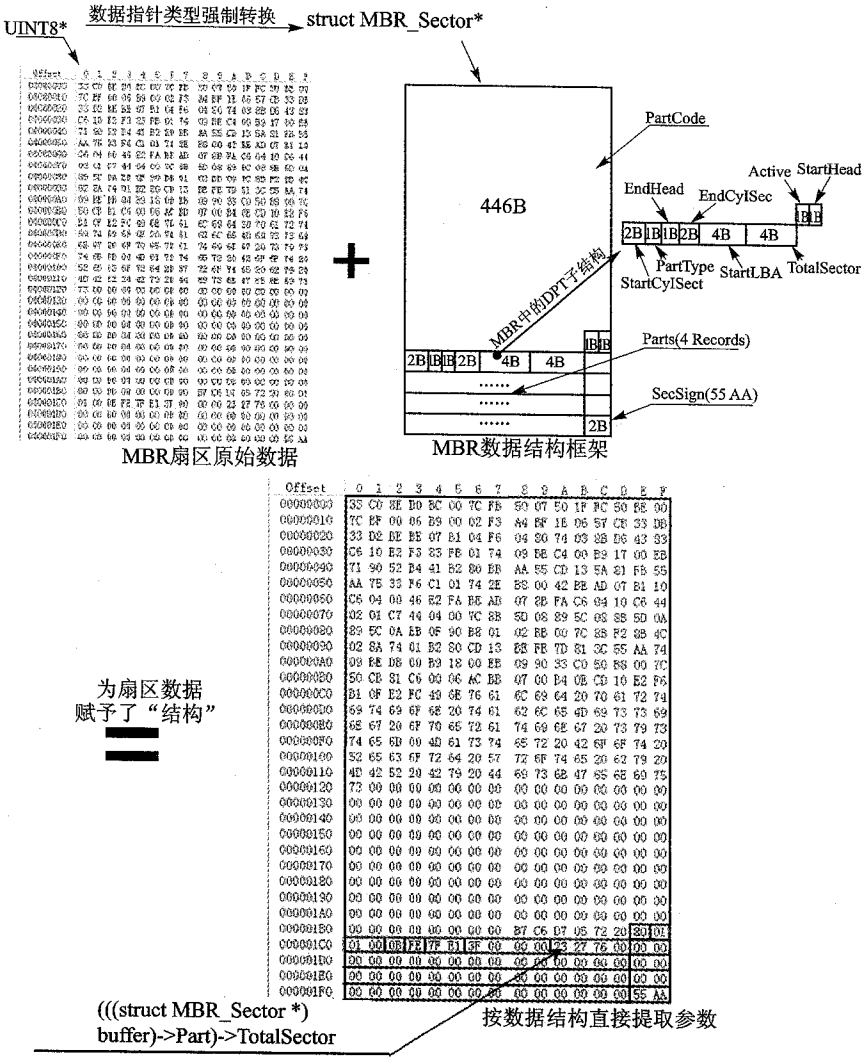

图 5.6　使用结构化的方法直接提取参数

针类型强制转换的思想)。如果从图 5.6 中还是不能参透其中要领,那就来看看具体的代码实现吧:

```
#define MBR_SECTOR 0                    //MBR 扇区
SD_ReadSector(MBR_SECTOR,buffer);       //将 MBR 扇区数据读到缓冲区
struct MBR_Sector * pStru;              //定义一个结构体型指针
pStru = (struct PartSector * )buffer;   //将缓冲区首地址强转为结构体指针,赋给 pStru
FileSys_Type = (pStru -> Parts) -> PartType;
Part_Start_Sector = (pStru -> Part) -> StartLBA; //直接提取参数,字段的 4 个字节会
                                                 //依结构体的定义,自然组成长整型
Part_Total_nSec = (pStru -> Part)[0] -> TotalSector;
```

从 SD 卡扇区中读到的 512 字节的数据是没有任何意义的原始数据(用 UINT8 * 指针来指向它)。也就是说,我们只知道这是一堆字节,但如果没人告诉我们数据的结构定义,那谁也不可能知道哪些字节表示什么参数、表达什么意义。而如果将它强制转换为 struct MBR_Sector * ,则为这堆原始的字节数据赋予了一定的结构,依照结构体的定义,原始数据中对应字段中的若干个字节就会自然结合起来表达某一参数,而且参数同样可以像操作结构体成员变量一样被直接提取出来。可见,结构化的方法在对数据进行解析的过程中比其他方法更为方便、直观而简洁。

上面的代码在对 CPU 访存方式有所了解的读者眼中其实是有严重问题的,什么问题?答:大小端!

5.3.3 关键:大小端问题

在讲解什么是大小端问题之前,先来讲一则小故事,也是大小端(Big Endian & Little Endian)这一名词的来历。

Endian 这个词来源于 Jonathan Swift 在 1726 年写的讽刺小说《Gulliver's Travels》(《格利佛游记》)。该小说在描述 Gulliver 畅游小人国时碰到了如下的场景:在小人国里的小人因为非常小(身高只有 6 英寸),所以总是碰到一些意想不到的问题。有一次因为对水煮蛋该从大的一端(Big-End)剥开还是小的一端(Little-End)剥开的问题而引发了一场战争,并形成了两支截然对立的队伍。支持从 Big-End 剥开的人 Swift 就称作 Big-Endians,而支持从 Little-End 剥开的人就称作 Little-Endians(后缀 ian 表明的就是支持某种观点的人)。Endian 这个词由此而来。这个故事似乎在谈论一个有关顺序或方向的问题,大小端就是这个意思。

我们来想一下,CPU 在向内存中存入一个长整型变量时是如何进行的?它是将长整型变量的 4 个字节依次存放到内存的连续空间中,如图 5.7 所示。图中 CPU 把 4 个字节中的第一个字节,即 0x78,放在了前面(低地址),而 0x12 则放到了最后(高地址)。这种存放的顺序就被称为小端模式。当然,CPU 完全可以把 0x78 放到最后,而把 0x12 放在前面,这就是大端模式了,如图 5.8 所示。

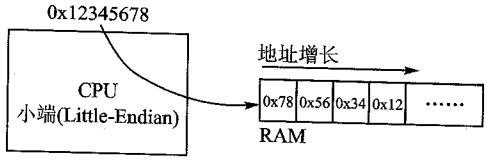

图 5.7　小端 CPU 对长整型变量的存储

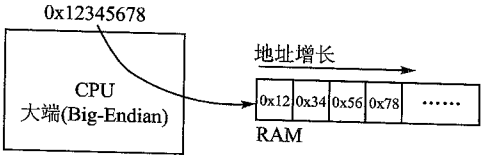

图 5.8　大端 CPU 对长整型变量的存储

那么如何控制 CPU 采用哪种存储方式呢？通常来说，CPU 是大端还是小端，是硬件上决定的。CPU 的研制者把它设计成怎样的访存方式，那就是固定了，无法改变。除非 CPU 在硬件上提供了大小端切换开关（在一些 ARM 芯片上是有这样的开关的，比如三星的 S3C44B0X 等）。我们看到 MBR 的参数字段是采用小端模式来进行存放的（其实 FAT32 其他功能扇区的数据也是如此），这是因为起初 FAT32 文件系统是在 PC 上使用的（比如当初的 DOS），并没有考虑到会在嵌入式领域有所应用。而 PC 使用的 CPU 多以采用小端存储方式的 X86 内核为主，因此 FAT32 的参数字段也是按小端模式来进行存储的，这样扇区数据被读入内存之后，就可以直接转化为参数值了，如图 5.9 所示。

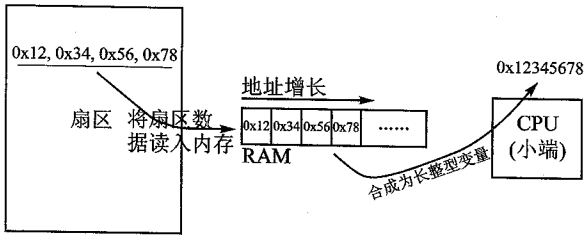

图 5.9　小端 CPU 将扇区数据直接转为参数

可能有人还是不太理解，我们来举一个例子，看下面的代码：

```
UINT8 buf[4] = {0x11,0x22,0x33,0x44};  //有 4 个字节的数组
UINT32 result;                         //定义一个 32 位的长整型变量(由 4 个字节组成)
result = *((UINT32 *)buf);             //将数组首地址转为长整型指针，并取出其值
```

代码中首先定义了一个有 4 个字节的数组，其实就是一个连续的存储空间，就如同 MBR 扇区数据中的字段一样，从低地址到高地址分别存储了 0x11、0x22、0x33、0x44。随后将数组的这 4 个字节组合成一个长整型（这是一个指针类型强制转换的过程），再取其值赋给 result，最终 result 的值为 0x44332211（如果这段代码运行的 CPU 是小端模式的）。可见，这种直接转换为参数的方法比 5.3.1 小节介绍的那种笨方法中移位拼接的方法要简洁而高效得多，这主要得益于字节存储顺序与 CPU 大小端模式的一致。

此时，你一定已经意识到了问题的所在：如果字节存储顺序与 CPU 大小端模式不一致，又会怎样？比如上面如果 CPU 是大端模式（比如 51）会怎么样呢？还拿上

面的代码来说，此时 result 的值为 0x11223344，这就大错而特错了。所以我们前面在讲解 MBR 的结构化解析方法的时候所编写的代码是有严重问题的，它不能保证在任何 CPU 上运行都能得到正确的结果。

其实那段代码除了有大小端问题，还有一个结构体数据对齐的问题，这里就不详细讲了，有兴趣有读者可以研究一下。既然因为大小端问题，使结构化的解析方法不能通用，那有没有更好的方法可以对其进行改进呢？好，我们继续看下面的内容。

5.3.4 改进方法 2：通用化的解析方法

有什么方法可以使程序能够避开大小端问题，而“投之四海而皆准”呢？也就是说，对于参数的解析，我们的程序不论在任何一种 CPU 上运行都能得到正确的结果。可以看到，结构化的方法可以使我们不用专门定义各字段的开始位置，大小端的差异所引发的问题主要产生在字段中字节组合转换为参数值的过程中，那么我们仍然可以使用移位拼接的方式来计算参数值，而不采用依赖于字节顺序的强制转换方法。

我们先把 DPT 表项的结构体定义进行改进：

```
struct PartRecord
{
    UINT8 Active;                   //引导指示符
    UINT8 StartHead;                //分区的开始磁头
    UINT8 StartCylSect[2];          //开始柱面与扇区
    UINT8 PartType;                 //分区类型
    UINT8 EndHead;                  //分区的结束头
    UINT8 EndCylSect[2];            //结束柱面与扇区
    UINT8 StartLBA[4];              //分区的第一个扇区
    UINT8 TotalSector[4];           //分区的大小
};
```

可以看到，这里把原来的 UINT32 和 UINT16 都改为了字节数组，从而不再让 CPU 依照自己的大小端去合成参数，而按照我们指定的一种明确固定的方式去合成。

```
#define MBR_SECTOR 0                    //MBR 扇区
struct MBR_Sector *pStru;               //定义一个结构体型指针
pStru = (struct MBR_Sector *)buffer;
SD_ReadSector(MBR_SECTOR,buffer);  //将 MBR 扇区数据读到缓冲区
FileSys_Type = (pStru->Parts)->PartType;
Part_Start_Sector = Bytes2Value(((pStru->Parts)->StartLBA),4);
                                        //找到字段首地址，将字节移位拼接为参数
```

```
Part_Total_nSec = Bytes2Value(((pStru->Parts)->TotalSector),4);
```

代码中的Bytes2Value是一个小函数，用于将若干个字节按小端方式移位拼接为32位变量，具体实现如下：

```
UINT32 Bytes2Value (UINT8   *dat,UINT8   len)
{
    UINT32 temp = 0;

    if (len>=1)  temp|=((UINT32)(dat[0]))     ;
    if (len>=2) temp|=((UINT32)(dat[1]))<<8 ;
    if (len>=3)  temp|=((UINT32)(dat[2]))<<16;
    if (len>=4) temp|=((UINT32)(dat[3]))<<24;
    return temp;
}
```

振南的znFAT代码中对于参数的解析大量使用了这种方法，以保证较好的可移植性。znFAT作为一个由振南不断改进维护的开源项目，受到了很多人的关注。有些人专门对它进行了研究，并希望通过研读代码来了解其内部实现，但普遍反映比较难懂。振南曾经在百度知道里看到有人在咨询znFAT中参数解析的相关问题，主要还是指针类型强转和大小端的问题，一些网友作了解答，但比较片面，希望本章的内容能够给读者一个满意的答复。

5.4 硬件平台上的验证

前面介绍了解析MBR的方法，但是最终还是要把代码放到硬件平台上去运行，实验使用串口和TFT液晶对解析后的参数值进行输出和显示，这样就能很直观地看到程序运行的结果是否正确了。

5.4.1 编写测试代码

将MBR解析封装为一个函数，代码如下：

```
int znFAT_Aanalyse_MBR(void)                    //对 MBR 进行参数解析
{
 struct MBR_Sector *pStru;                      //定义一个结构体型指针
 SD_Read_Sector(MBR_SECTOR,buffer);             //将 MBR 扇区数据读到缓冲区
 pStru = (struct MBR_Sector *)buffer;
 FileSys_Type = (pStru->Parts)->PartType;
 Part_Start_Sector = Bytes2Value(((pStru->Parts)->StartLBA),4);
 Part_Total_nSec = Bytes2Value(((pStru->Parts)->TotalSector),4);
 return 0;
```

```
}
```

下面要做的就是在主函数 main 中对这个函数进行调用，然后再把 FileSys_Type、Part_Start_Sector 和 Part_Total_nSec 这 3 个变量的值输出即可。此时振南要提一个问题：znFAT_Aanalyse_MBR 这个函数应该放在程序的什么位置？有人不假思索："当然是 main 函数的前面了，放在后面也行，不过要在前面声明一下。"他的意思是这样的，如下所示：

```
#include  相关头文件
#define  宏定义
变量定义
int znFAT_Aanalyse_MBR(void) //MBR 参数解析函数放在 main 函数前面
{
 ......
}
int main(void)
{
//相关初始化
......
 znFAT_Aanalyse_MBR();
 串口输出
 TFT 液晶输出
 return 0;
}
```

这是一种大家都已经很习惯使用的编程方法，或者叫代码组织方法。这种方法把所有代码都放在一个 C 源文件里，有一些模块化思想的编程者可能会把一部分代码，比如函数声明、宏定义等放到.h 文件中。振南称这种编程方法为"流水式编程方法"，代码的编写如同流水一般顺序的进行，一条语句挨着一条语句，一个函数摞着一个函数地都码在了一个.c 文件里，这水流来流去也就这么一条水道。有人说："这种编程方法我觉得很直观，也很方便。"那振南要问："如果一个代码工程中有成百上千，成千上万的函数、变量、语句……，你调试起来还会觉得方便吗？你面对着这么大的一个.c 文件，难道你就不头大吗？"编程如同治水，要学会引流入支。

有的读者看了上面的这段文字，可能就感觉有些迷糊了："函数不放在 main 的前面，那放在哪里好呢？"还记得我们前几章做实验时使用的各种程序模块和模块化的编程方法吗？我们可以把文件系统相关的代码都放到一个程序模块里：znFAT.c 和 znFAT.h。我们来看看下面的程序：

znFAT.h(用于宏、结构体以及函数的定义与声明)代码如下：

```
#ifndef __ZNFAT_H__
#define __ZNFAT_H__                    //此宏用于防止重包含
```

```
#define MBR_SECTOR 0                    //MBR 扇区
//DPT(分区表记录)结构如下
struct PartRecord
{
 ........
};
//MBR 扇区(物理 0 扇区)定义如下
struct MBR_Sector
{
 ........
};
int znFAT_Aanalyse_MBR(void);          //函数声明
#endif
```

znFAT.c(用于变量的定义及函数的具体实现)代码如下：

```
#include "znfat.h"
#include "sdx.h"
UINT8 buffer[512];                     //扇区数据缓冲区
UINT8 FileSys_Type;                    //分区文件系统类型
UINT32 Part_Start_Sector;              //分区开始扇区
UINT32 Part_Total_nSec;                //分区的总扇区数
UINT32 Bytes2Value(UINT8 *dat,UINT8 len)
{
......
}
int znFAT_Aanalyse_MBR(void)
{
 ......
 return 0;
}
```

_main.c(主文件，调用各程序模块以实现顶层功能)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 unsigned char FileSys_Type;
extern unsigned long Part_Start_Sector;
extern unsigned long Part_Total_nSec;
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();
    //通过 TFT 液晶输出结果
    TFT_Draw_BackGround(); //画背景
    TFT_Send_Str("MBR Analysis:",8,40, COLOR_BLUE, COLOR_WHITE);
    TFT_Put_Inf("FS Type:",FileSys_Type,8,72, COLOR_RED, COLOR_WHITE);
    TFT_Put_Inf("Start Sec:",Part_Start_Sector,8,104, COLOR_RED, COLOR_WHITE);
    TFT_Put_Inf("Total:",Part_Total_nSec,8,136, COLOR_RED, COLOR_WHITE);
    //通过串口输出结果
    UART_Send_Str("MBR Analysis:\r\n");
    UART_Put_Inf("FS Type:",FileSys_Type);
    UART_Put_Inf("Start Sec:",Part_Start_Sector);
    UART_Put_Inf("Total:",Part_Total_nSec);
    while(1);
}
```

我们用图 5.10 来阐明各个程序模块之间的关联。

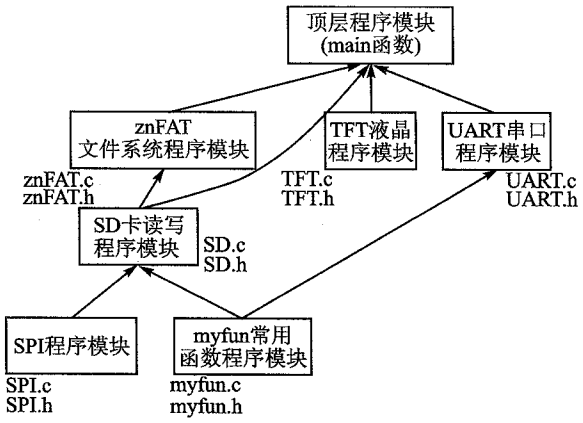

图 5.10　MBR 解析测试程序中的程序模块关联

好,下面来看一下 Keil 开发环境中生成的工程视图(ZN－X 开发板上的 51 与 STM32 均可使用 Keil MDK 来进行开发),它可以体现出工程中所有的程序模块以及模块之间的包含与调用关系,如图 5.11 所示。

其实很多人都非常纳闷，为什么一个工程中可以有这么多的 C 文件，最终却只编译得到一个可执行文件（如 Hex 或 Bin 等）？如果把变量或函数定义在一个 C 文件中，那怎么能让其他 C 文件调用到它们呢？想不通这些问题是因为很多人喜欢把所有代码写在一个 C 文件中，不能接受多 C 文件的模块化、工程化的开发方法。振南在这里进行一个简单的回答，希望大家能够对这种编程思想的理解。

① 虽然一个工程中有多个 C 文件，但它们只是实现各自的那部分功能，最终还是要被顶层文件来调用（如上面代码中的_main. c）。在编译的时候，每一个 C 文件都会单独进行编译（compiling），如图 5. 12 所示。

这个编译的结果并不是一下子就得到最终的可执行文件，而是先得到一些中间文件（比如 OBJ 文件），如图 5. 13 所示。（如果把代码都写在一个 C 文件中，当然就只会得到一个中间文件。）

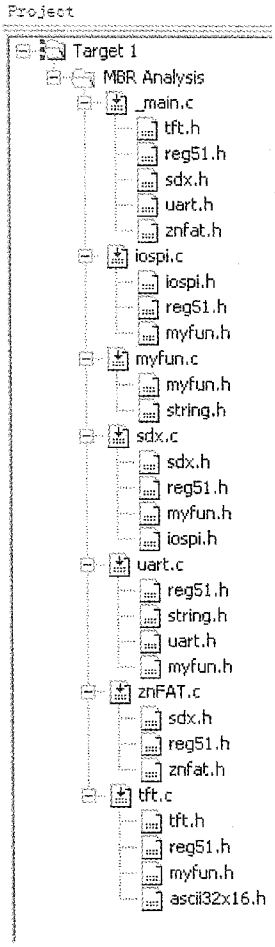

图 5. 11　Keil 的工程视图

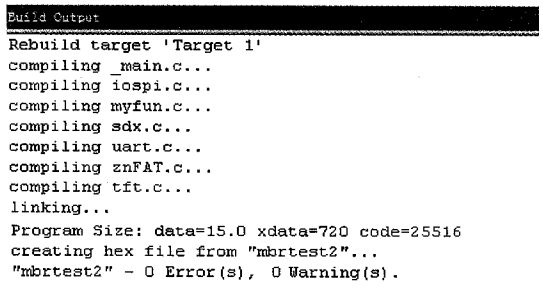

图 5. 12　Keil 编译各个程序模块

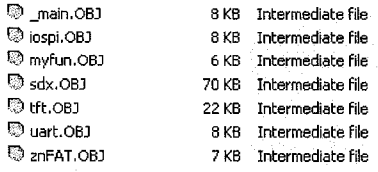

图 5. 13　编译得到的 OBJ 中间文件

这些中间文件再经过一个连接(linking)的过程,将所有中间文件合并为一个可执行文件。这其中,_main.c 是占主导地位的,按照工程的顶层功能需求,对其他程序模块进行适当调用和管理。说白了,其他模块就是“干活”的,主文件才是“领导”,旨在统筹全局。不论有多少程序模块、有多少个 C 文件,最终还是要向主文件靠拢,汇聚成一个可执行文件。

② 程序模块中的.h 文件可以将程序模块中可供外部调用的函数进行声明,这样其他 C 文件只需包含.h 文件即可对函数进行调用。当然,如果在 h 文件中没有对函数进行声明,那么函数是不能被外部调用的(这对于程序模块中的内部函数和具体实现细节的保密是有好处的),如图 5.14 所示。

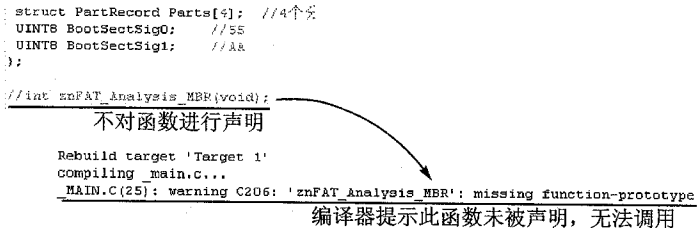

图 5.14 未在.h 文件中对函数声明而导致的编译警告

至于变量的外部引用问题,其实是通过 extern 关键字来实现的。比如在上面的程序中,znFAT.c 文件中定义了 3 个变量,它们是最终要打印输出的变量,但是打印输出操作是在_main.c 文件中来进行的,_main.c 如何去引用 znFAT.c 中的变量呢?我们可以在_main.c 文件中对这 3 个变量使用 extern 声明。这样就告诉编译器,这 3 个变量不是在本文件中定义的,而是要到其它 C 文件中去找。所以,如果把所有代码都写到一个 C 文件中,我们就“一辈子”也用不到 extern 关键字。但是 C 语言中为什么要引入这样一个东西呢?在学 C 语言的时候经常会觉得有一些关键字似乎没什么用,其实引入它必有引入它的道理,看不到它的用途,只能说明编程思想还没有到达使用和驾驭它的地步。

上面讲了这么多,还掺杂了一些编程方法和思想方面的介绍,振南希望在书中讲述文件系统的同时能够为读者揭示一些编程方面的经验,使本书的寓意更加深刻。

5.4.2 验证实验结果

最后来看看程序在 ZN-X 开发板上运行的最终结果。将编译得到的 Hex 文件烧录到单片机中,再重新上电或复位,此时在计算机的超级终端中即可看到如图 5.15 及图 5.16 所示的内容。

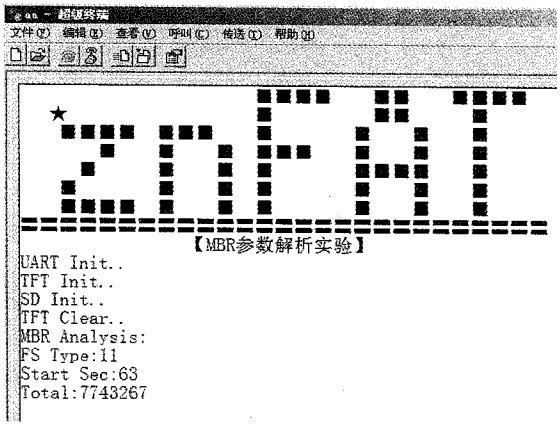

图 5.15　超级终端中所显示的程序运行结果

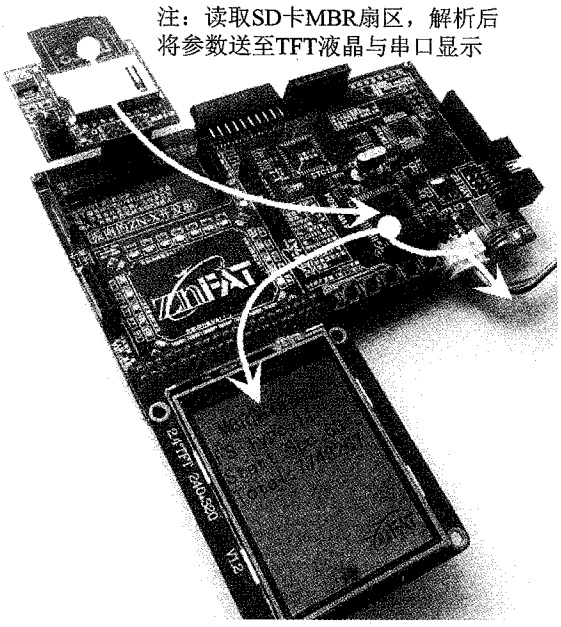

图 5.16　ZN－X 开发板上 TFT 液晶输出的运行结果

可以看到，ZN－X 开发板上运行的结果与实际值是完全相吻合的，说明对 MBR 的解析是正确的。有了从 MBR 中解析出来的参数，我们就可以向新的目标迈进了，那就是 DBR，它是繁杂的，也是非常核心的部分……好戏都在后面，请君翻篇。

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