文件系统介绍

skd2278@gmail.com

TOPIC

- 基础篇

为什么需要文件系统

什么是文件系统

一个简单文件系统的实现

- 高级篇

FAT16/32基本原理

FDB数据库原理

基础篇

EEPROM

EEPROM可以随意读写任何地址的数据，可以以页为单位读写，也可以以一个字节为单位读写。

所以可以把EEPROM看作一块慢速的RAM。在这里不用考虑接口是否与RAM一样，接口都可以用其它方式进行转换。

AT24c256

512个页，每页64字节。

64字节×512页=32K字节

EEPROM读写实例

```
const Address g_address[] =
{
        [ADDR_FORMAT_BYTE]              =       (SYSTEM_INFO_AREA+0),
        [ADDR_WORK_MODE]                =       (SYSTEM_INFO_AREA+1),
        [ADDR_TASK_NUMBER]              =       (SYSTEM_INFO_AREA+2),
        [ADDR_WIRE_NUMBER]              =       (SYSTEM_INFO_AREA+3),
        [ADDR_PHONE_MONITOR_CENTER]     =       (SYSTEM_INFO_AREA+5),
        [ADDR_SYS_STATUS]               =       (SYSTEM_INFO_AREA+21),
        [ADDR_GPS_INFO_NUMBER]          =       (SYSTEM_INFO_AREA+23),
        [ADDR_PHONE_SMS_CENTER]         =       (SYSTEM_INFO_AREA+25),
        [ADDR_WORK_SHEET_ID]            =       (SYSTEM_INFO_AREA+41),
        [ADDR_TOWER_NUMBER]             =       (SYSTEM_INFO_AREA+47),
        [ADDR_TOWER_CUR_INDEX]          =       (SYSTEM_INFO_AREA+49),
        [ADDR_DOWNLOAD_FLAG]            =       (SYSTEM_INFO_AREA+51),

        [ADDR_CONFIG]                   =       (CONFIG_AREA+0),
        [ADDR_GPS_POWER_MODE]           =       (CONFIG_AREA+4),
        [ADDR_MSG_FEEDBACK_MODE]        =       (CONFIG_AREA+5),
        [ADDR_SLEEP_TIME]               =       (CONFIG_AREA+6),
};
WriteDataToEEPROM(ADDR_PHONE_SMS_CENTER, pTel, sizeof(PhoneNumber));
ReadDataFromEEPROM(ADDR_PHONE_SMS_CENTER, pTel, sizeof(PhoneNumber));
```

红色部分可以理解为文件名。

EEPROM读写实例

- 优点

操作简单，对操作的地址了然于心。

- 缺点

维护庞大的地址表，任何一个改变可能会牵一发而动全身。

- 最想改进的地方

不操作地址

EEPROM内存管理

- 隐藏人工管理地址操作

用类似内存管理函数malloc/free管理这块ROM。

- 基本思想

把这块EEPROM看作一块同样大小的内存，malloc/free操作的时候完全不用管它在哪，只有操作实际地址读写时映射到实际的存储器上即可。

EEPROM内存管理实例

- 借用UCOS内存管理思想

创建内存分区

```
OS_MEM  *OSMemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)
```

获取一块内存

```
void  *OSMemGet (OS_MEM *pmem, INT8U *err)
```

释放一块内存

```
INT8U  OSMemPut (OS_MEM  *pmem, void *pblk)
```

主要数据结构

```
typedef struct {                    /* MEMORY CONTROL BLOCK                                  */
   void   *OSMemAddr;               /* Pointer to beginning of memory partition               */
   void   *OSMemFreeList;           /* Pointer to list of free memory blocks                 */
   INT32U  OSMemBlkSize;            /* Size (in bytes) of each block of memory               */
   INT32U  OSMemNBlks;              /* Total number of blocks in this partition              */
   INT32U  OSMemNFree;              /* Number of memory blocks remaining in this partition   */
} OS_MEM;
```

EEPROM内存管理

注意：移植的时候哪些数据在RAM中，哪些数据在EEPROM中。

OSMemAddr

OSMemBlkSize

OSMemNBlke

OSMemNFree

在内存中，但必须在EEPROM中留出一个位置在开关机的时候加载/保存这些数据

OSMemFreeList在EEPROM中，RAM中存有读取备份。

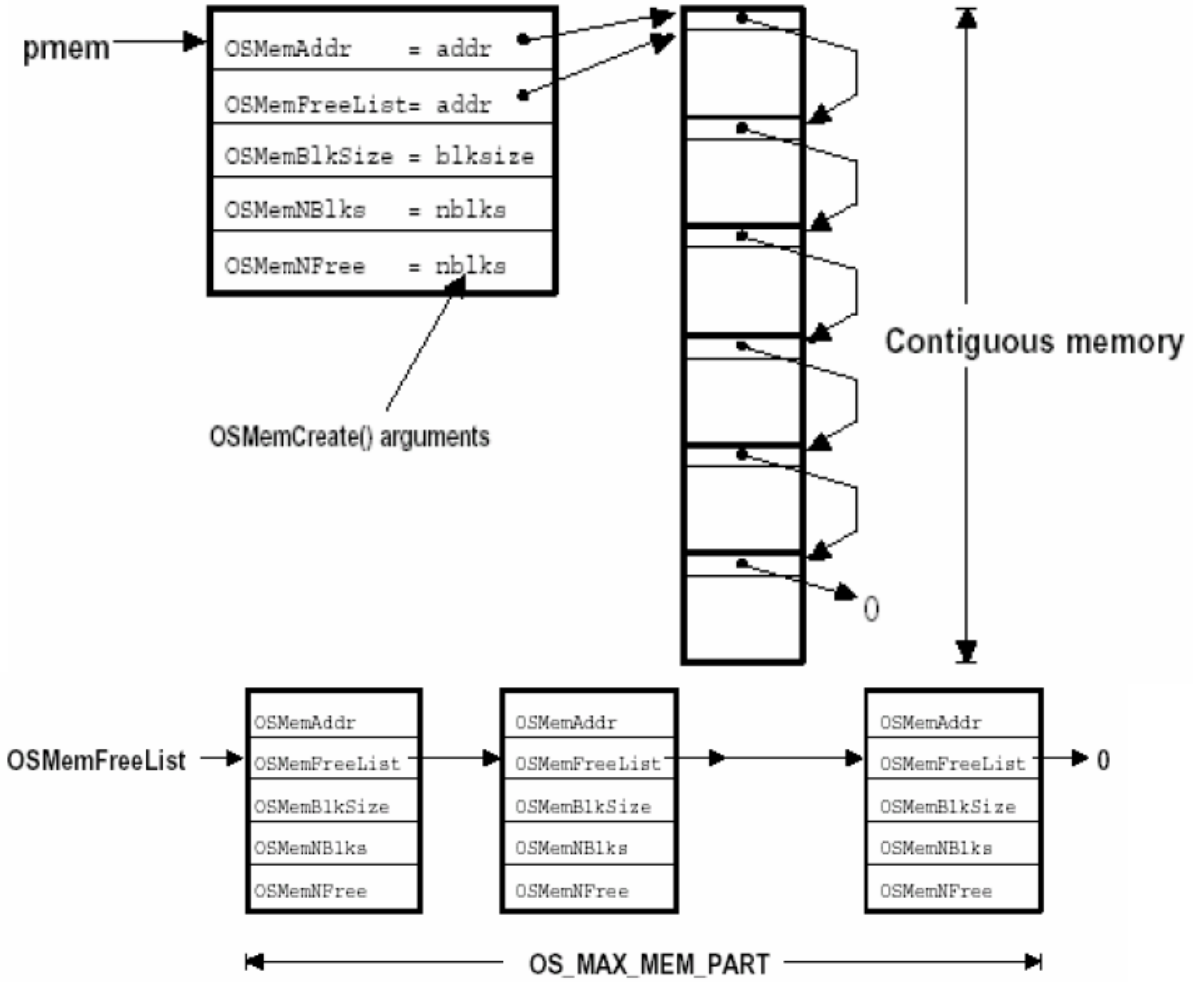


Figure 7-3, List of free memory control blocks.

图 F7.3 空闲内存控制块链表——Figure 7.3

EEPROM内存管理

1.优点

不用管理复杂的地址

2.缺点

数据之间没有体现有组织的关系

3.最想改进的地方

体现数据之间的关系

为什么需要文件系统

- 管理地址
- 管理数据关系

什么是文件系统

- 有组织的管理数据，隐藏复杂地址操作的系统。
- 官方定义：

文件系统是一种存储和组织计算机文件和资料的方法，它使得对其访问和查找变得容易。

写一个文件系统需要明确的

Disk较慢，所以使用较快的RAM，将欲存取使用的文件复制到RAM中，再进行存取，然后写回存储装置。

如果不考虑速度瓶颈，文件系统可以不需要RAM中的数据结构

- RAM-Data Structure

管理在存储装置上所存储的数据及其属性

- ROM-Data Structure

管理从存储装置所读出来的文件及数据或者是逻辑地址对应到实体地址的记录表格

写一个文件系统需要明确的

- 存储器的物理特性

1.大小

2.read/write/erase方式

3.组织方式（扇区，磁道，块，页）

问题：下面的存储器哪些参数是硬件里固有的？怎样得到这个参数？

SDRAM，EEPROM，NOR FLASH，NAND FLASH，

3.5英寸软盘，硬盘。

一个简单的文件系统实现（THIN）

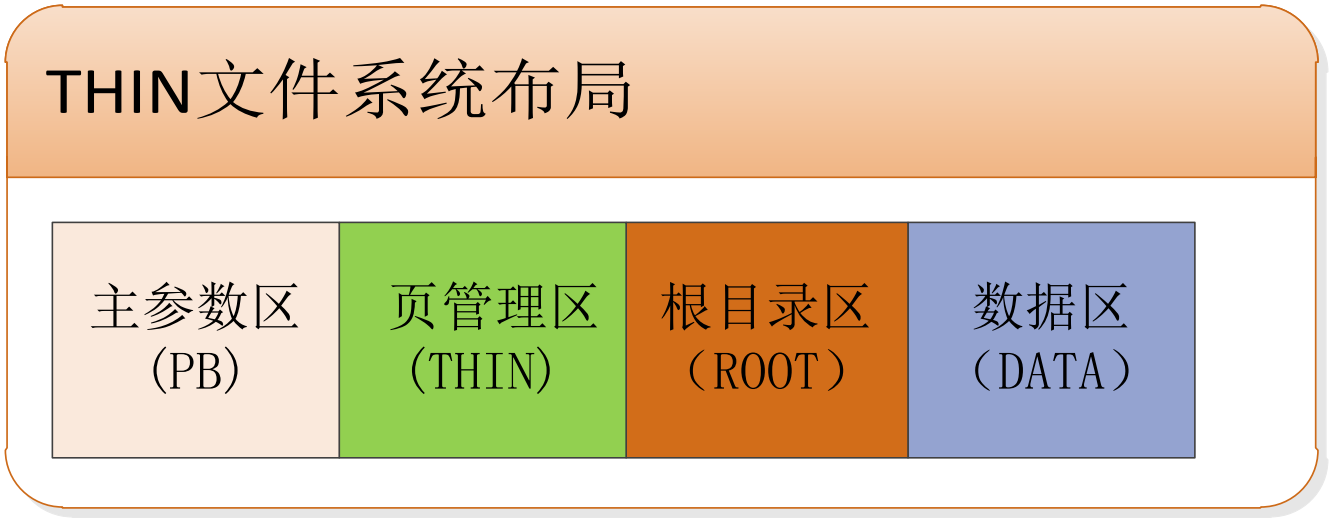
表1. 各型号EEPROM的参数

EEPROM 型号	容量 (bytes)	页大小[1] (bytes)	总页面数	地址位 (bits)	A0-A2使用情况
AT24C08	1k	16	64	8	A2 used for device addressing and A0 A1 used for memory page addressing.[2]
AT24C16	2k	16	128	8	No bit used for device address and A0 A1 A2 used for memory page addressing.
AT24C32	4k	32	128	16	A0 A1 A2 used for device address .
AT24C64	8k	32	256	16	A0 A1 A2 used for device address.
AT24C128	16k	64	256	16	A0 A1 used for device address, A2 is NC.
AT24C256	32k	64	512	16	A0 A1 used for device address, A2 is NC.
AT24C512	64k	128	512	16	A0 A1 used for device address, A2 is NC.
AT24C1024	128k	256	512	16	A1 used for device address ,A0 A2 is NC.

一个简单的文件系统实现（THIN）

- 一个只支持AT24C256的文件系统，将EEPROM分成四个部分：

1.主参数区(PB)　　总得需要知道磁盘参数，这个FS的基本属性
2.页管理区(THIN)　　链接不连续的页
3.根目录区(ROOT)　找文件总得有个入口
4.数据区(DATA)　　就是为了管理这部分的



一个简单的文件系统实现（THIN）

- 主参数区（PB），放在第0页，也可以放在其他任何位置，前提是都遵守这个规则。这个系统规定到0页。

字节位移	字段长度	对应取值	名称和定义
0x00	2	64	页大小
0x02	2	16	根目录项数
0x04	1	?	THIN表占的页数
0x05	2	512	总页数
0x38	2	0xAA55	有效结束标识

一个简单的文件系统实现（THIN）

- 页管理区（THIN）

页管理区每两个字节对应一个页，页号从0开始排序。

规定0表示这个页未占用，0xFFFF为文件数据尾，0xFFF8表示不指向任何页，其它值表示当前页链接到该值的页，是一个链式存储结构。

THIN表（页管理区）

页0	页1	页2	. . .	页N-1	页N
数据页0	数据页1	数据页2	. . .	数据页N-1	数据页N

一个简单的文件系统实现（THIN）

- 根目录区（ROOT）

Dir指目录项，存储一个文件（夹）的属性字段，这里定义为16个字节。

根目录区结构

THIN目录项16字节的表示定义		
字节偏移	字节数	定义
0x00～0x04	5	文件名
0x05	1	属性
0x06～0x07	2	文件最近修改时间
0x08～0x09	2	文件最近修改日期
0x0A～0x0B	2	文件首页号
0x0C～0x0F	4	文件的长度

一个简单的文件系统实现（THIN）

- 数据区（DATA）

存储文件实际的文件数据和目录数据。

文件名命名规则，同WINDOWS操作系统

重要的数据结构

文件系统信息，对应主参数区，内部win用于缓存，一次读取EEPROM中的一页

```
typedef struct {
        BYTE    wflag;          /* win[] dirty flag (1:must be written back) */
        WORD    n_rootdir;      /* Number of root directory entries */
        WORD    pagesize;       /* Bytes per page (64) */
        WORD    last_page;      /* Last allocated page */
        WORD    free_page;      /* Number of free pages */
        WORD    fsize;          /* Pages per THIN */
        WORD    thinbase;       /* THIN start page */
        WORD    dirbase;        /* Root directory start page */
        WORD    database;       /* Data start page */
        WORD    winpage;        /* Current page appearing in the win[] */
        BYTE    win[64];        /* EEPROM access window for Directory */
} THINFS;
```

目录项信息，包含文件的属性

```
/* Directory object structure (DIR) */

typedef struct {
        THINFS* fs;     /* Pointer to the owner file system object */
        WORD    index;  /* Current read/write index number */
        WORD    spage;  /* Table start page (0:Root dir) */
        BYTE*   dir;    /* Pointer to the current entry in the win[] */
        BYTE*   fn;     /* Pointer to the file name */
} DIR;
```

可以把它类似为C语言中的FILE指针

```
/* File object structure (FIL) */

typedef struct {
        FATFS*  fs;             /* Pointer to the owner file system object */
        WORD    fptr;           /* File read/write pointer (0 on file open) */
        WORD    fsize;          /* File size */
        WORD    spage;          /* File start page (0 when fsize==0) */
        WORD    page;           /* Current page */
} FIL;
```

用文件系统操作存储器之前要做的

- 格式化（f_mkfs）

Step1:填充主参数区（PB）

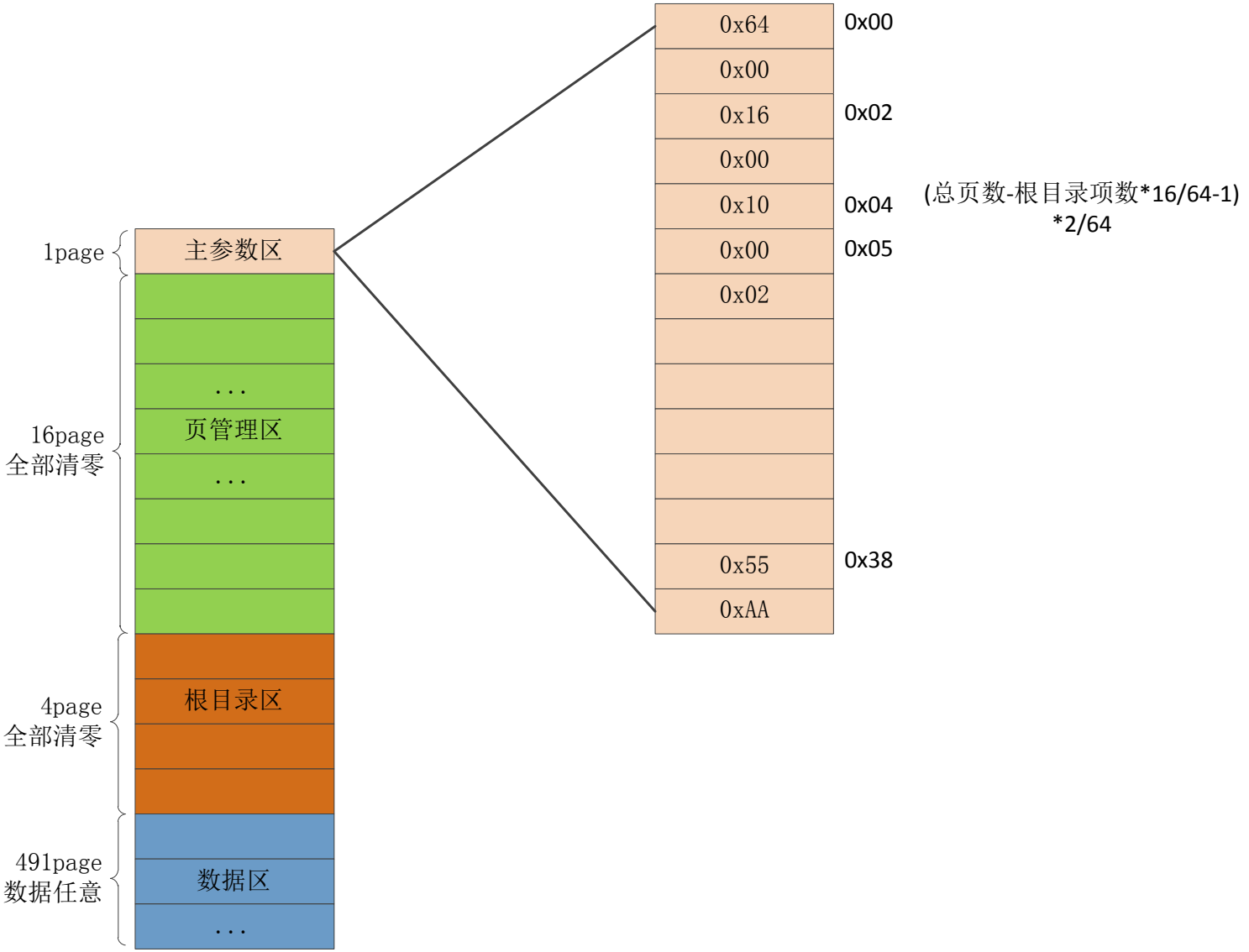
Step2:清除页管理区（THIN）

Step3:清除根目录区（ROOT）

- 挂载（f_mount）

1.定义一个THINFS变量，使系统定义的Fs指针指向它。获取的这个指针在check_mount中进行初始化。

格式化后的EEPROM



THINFS计算

```
typedef struct {
        BYTE    wflag;          =0
        WORD    n_rootdir;      =16
        WORD    pagesize;       =64
        WORD    last_page;      =0
        WORD    free_page;      =512-21=491
        WORD    fsize;          =
（总页数-主参数区页数-根目录项数×目录项大小/页大小）×THIN表项大小/页大小=(512-1-16*16/64)*2/64=16
        WORD    thinbase;       =1
        WORD    dirbase;        =17
        WORD    database;       =17+16*16/64=21
        WORD    winpage;        =0
        BYTE    win[64]; /* EEPROM access window for Directory */
} THINFS;
```

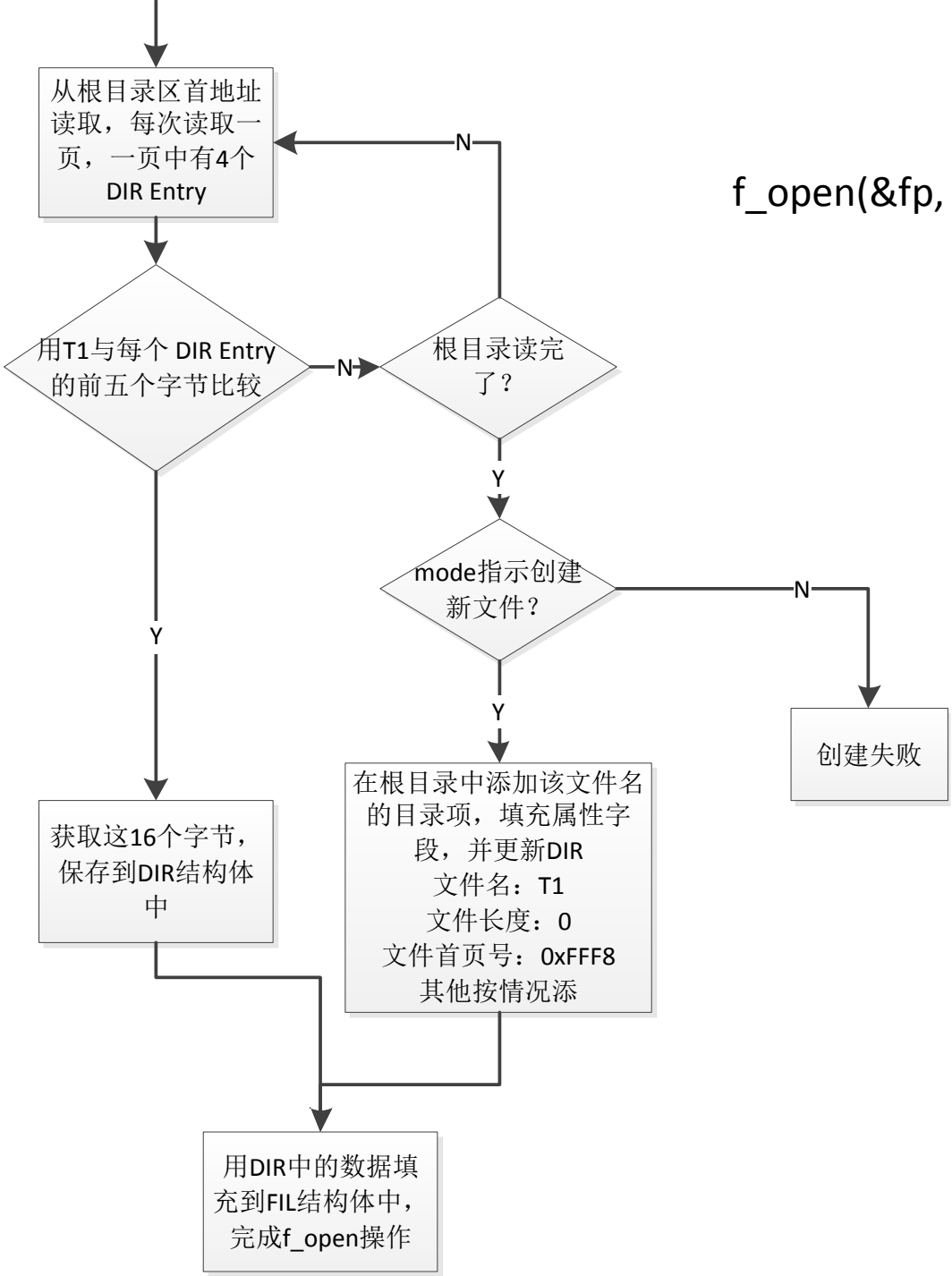
文件操作举例

1.f_open

```
FRESULT f_open (
    FIL *fp,              /* Pointer to the blank file object */
    const TCHAR *path,    /* Pointer to the file name */
    BYTE mode             /* Access mode and file open mode flags */
)
```

例：假设现在有一个0:/T1的文件，看怎么找。

只有一个驱动器，可忽略盘符，直接找T1。



f_open(&fp, “T1”, CREATE)

根目录区

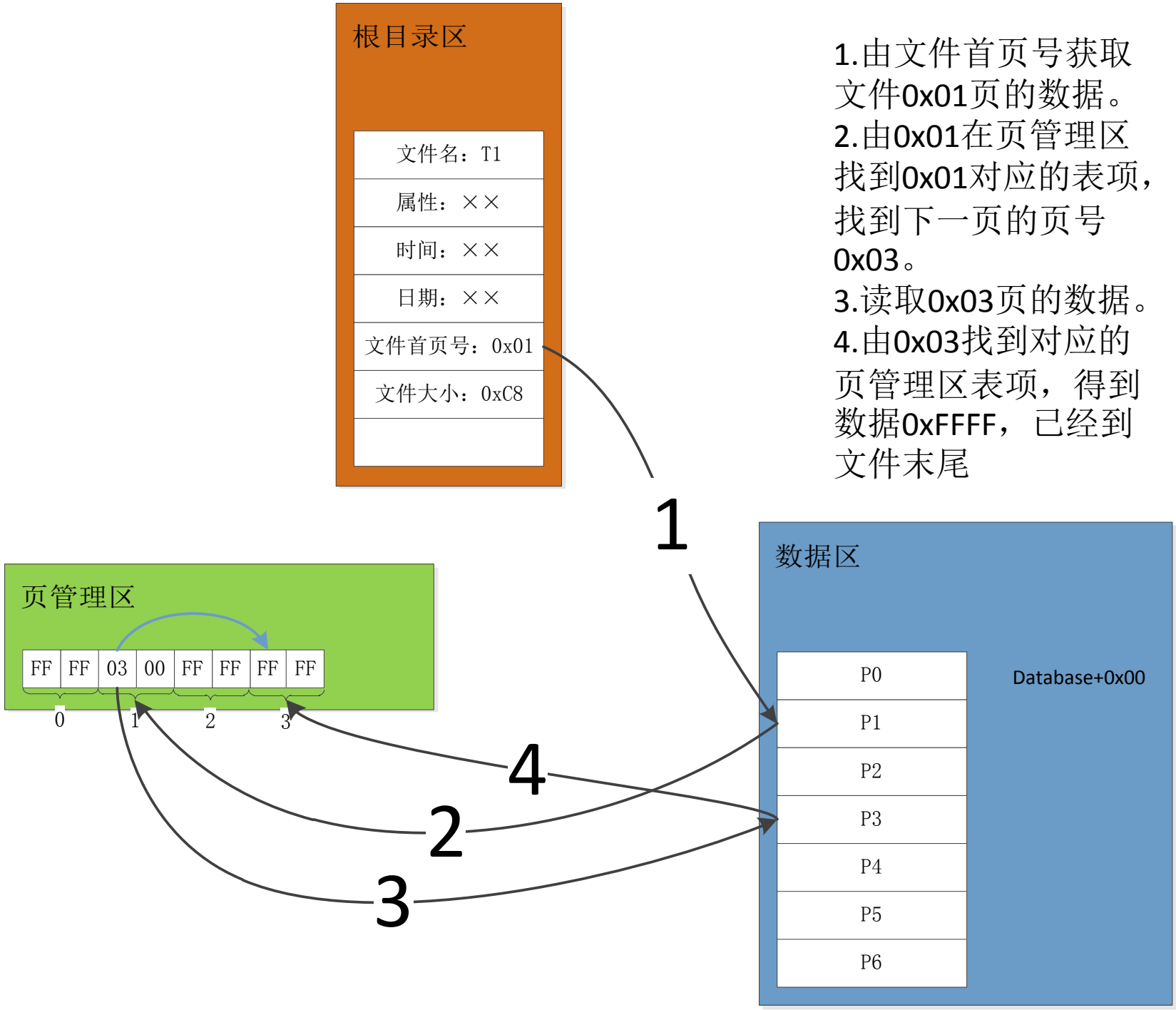
注意：这个打开操作只与根目录区打交道，其他的区域没有涉及

文件操作举例

- f_read

```
FRESULT f_read (
    FIL *fp,          /* Pointer to the file object */
    void *buff,       /* Pointer to data buffer */
    UINT btr,         /* Number of bytes to read */
    UINT *br          /* Pointer to number of bytes read */
)
```

考虑刚才打开的文件是已有的文件，里面存储着120个字节全部是12345678901234567890…



1.由文件首页号获取文件0x01页的数据。
2.由0x01在页管理区找到0x01对应的表项，找到下一页的页号0x03。
3.读取0x03页的数据。
4.由0x03找到对应的页管理区表项，得到数据0xFFFF，已经到文件末尾

文件操作举例

- f_unlink

```
FRESULT f_unlink (
    const TCHAR *path       /* Pointer to the file or directory path */
)
```

考虑删除T1，删除操作并没有FIL指针，所以得根据path，像f_open那样获取DIR ENTRY.

Step1:获取DIR entry

Step2:把DIR entry第一个字节改成0xE5

Step3:根据DIR entry中的文件首页号，把页存储区的该页表项清零，同时把其链接的所有页表项清零

文件操作举例

f_open

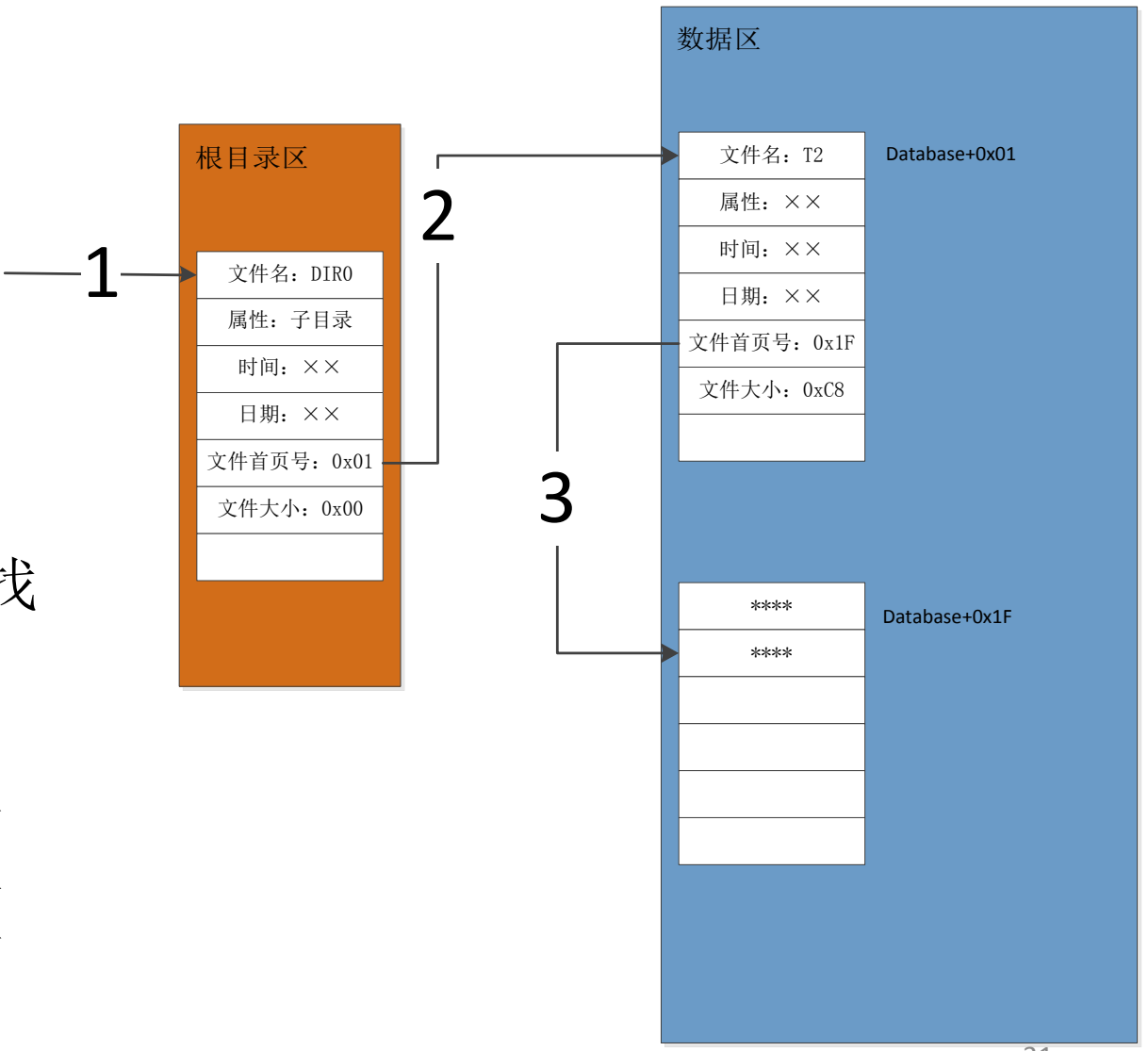
打开已存在的非空文件0：/dir0/t2

Step1:找到dir0

Step2:找到t2。

Step3:通过数据首页号找到数据起始位置

注：根目录下的任何子目录如果是文件夹，其下的文件目录项都存储在数据区。



THIN文件系统小结

1.文件系统包含四个区：主参数区、页管理区、根目录区、数据区。

2.找一个文件，都是从根目录开始查找（类似树的遍历），文件名是唯一的主键，无论如何根目录的首地址是必须知道的。

3.读取文件，一次读取一个页，这样有利于减少I/O操作的时间损耗。

4.删除操作并不扇区数据区，仅仅修改目录项和清除页管理区。

高级篇

FAT文件系统原理

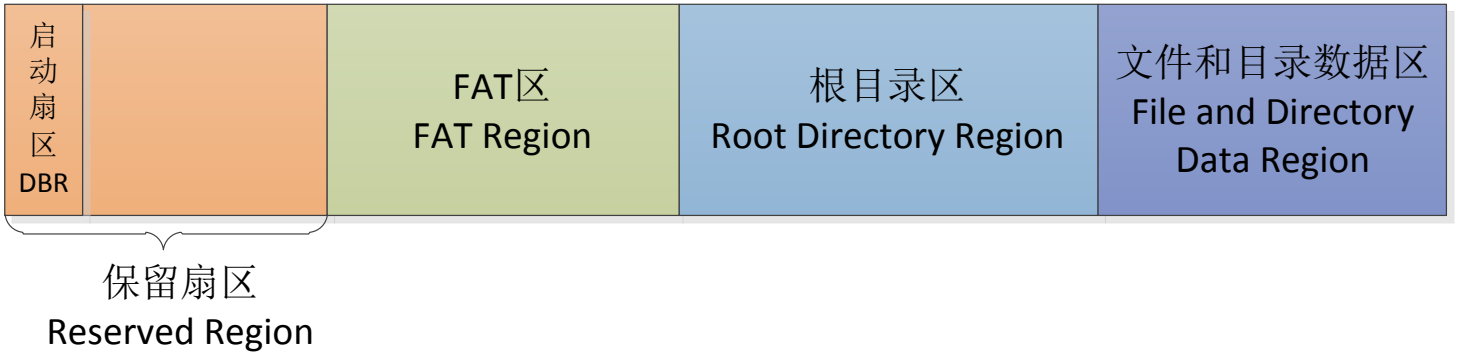
简介

FAT文件系统一共有FAT12/FAT16/FAT32三种。
它们的区别是FAT表中有多少位用来寻址簇。
位数不同，制约了每种系统所能支持的存储器大小。
FAT16簇的大小一般来说不能大于32K。

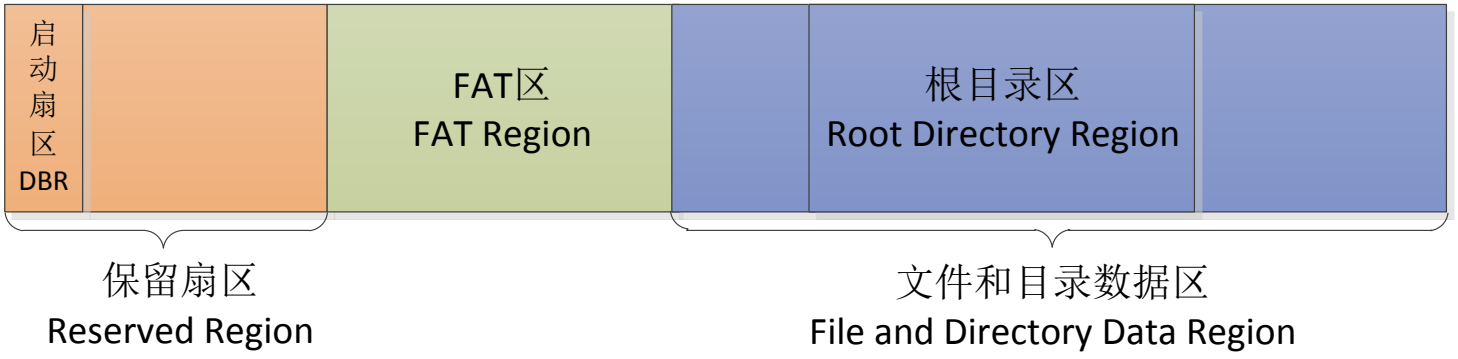
FAT12/16/32最大支持的磁盘大小?

FAT文件系统布局

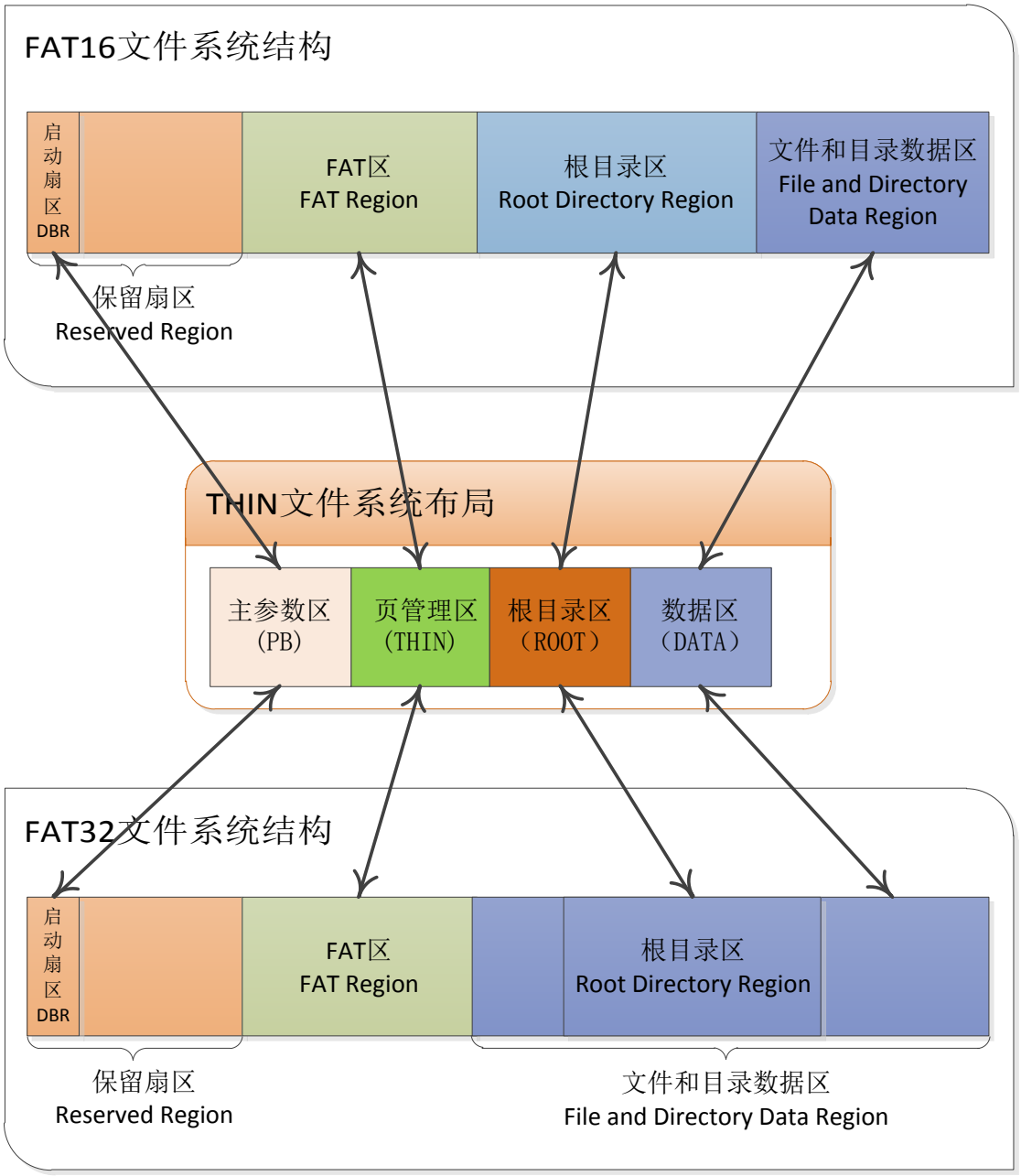
FAT16文件系统结构



FAT32文件系统结构



THIN与FAT的对比



启动扇区DBR

DBR区(DOS BOOT RECORD)即操作系统引导记录区的意思，通常占用分区的第0 扇区共512 个字节(特殊情况也要占用其它保留扇区，我们先说第0 扇)。在这512 个字节中，其实又是由跳转指令，厂商标志和操作系统版本号，

BPB(BIOS Parameter Block)，扩展BPB，os引导程序，结束标志几部分组成。

表 3　FAT32 分区上 DBR 中各部分的位置划分

字节位移	字段长度	字段名	对应图 8 颜色
0x00	3 个字节	跳转指令	
0x03	8 个字节	厂商标志和 os 版本号	
0x0B	53 个字节	BPB	
0x40	26 个字节	扩展 BPB	
0x5A	420 个字节	引导程序代码	
0x01FE	2 个字节	有效结束标志	

Boot Sector and BPB

The first important data structure on a FAT volume is called the BPB (BIOS Parameter Block), which is located in the first sector of the volume in the Reserved Region. This sector is sometimes called the “boot sector” or the “reserved sector” or the “0th sector,” but the important fact is simply that it is the first sector of the volume.

FAT16分区的BPB字段

offset	Length	name	discription
0x0B	2	BPB_BytsPerSec	扇区字节数(Bytes Per Sector) 硬件扇区的大小。本字段合法的十进制值有512、1024、2048和4096。对大多数磁盘来说，本字段的值为512
0x0D	1	BPB_SecPerClus	每簇扇区数(Sectors Per Cluster) 一个簇中的扇区数。由于FAT16文件系统只能跟踪有限个簇(最多为65536个)。因此，通过增加每簇的扇区数可以支持最大分区数。分区的缺省的簇的大小取决于该 分区的大小。本字段合法的十进制值有 1、2、4、8、16、32、64和128。导致簇大于32KB(每扇区字节数*每簇扇区数)的值会引起磁盘错误和软件错误
0x0e	2	BPB_RsvdSecCnt	保留扇区数(Reserved Sector) 第一个FAT开始之前的扇区数，包括引导扇区。本字段的十进制值一般为1
0x10	1	BPB_NumFATs	FAT数(Number of FAT)该分区上FAT的副本数。本字段的值一般为2
0x11	2	BPB_RootEntCnt	根目录项数(Root Entries) 能够保存在该分区的根目录文件夹中的32个字节长的文件和文件夹名称项的总数。在一个典型的硬盘上，本字段的值为512。其中一个项常常被用作卷标号(Volume Label)，长名称的文件和文件夹每个文件使用多个项。文件和文件夹项的最大数一般为511，但是如果使用的长文件名，往往都达不到这个数
0x13	2	BPB_TotSec16	小扇区数(Small Sector) 该分区上的扇区数，表示为16位(<65536)。对大于65536个扇区的分区来说，本字段的值为0，而使用大扇区数来取代它
0x15	1	BPB_Media	媒体描述符(Media Descriptor)提供有关媒体被使用的信息。值0xF8表示硬盘，0xF0表示高密度的3.5寸软盘。媒体描述符要用于MS-DOS FAT16磁盘，在Windows 2000中未被使用
0x16	2	BPB_FATSz16	每FAT扇区数(Sectors Per FAT) 该分区上每个FAT所占用的扇区数。计算机利用这个数和FAT数以及隐藏扇区数来决定根目录在哪里开始。计算机还可以根据根目录中的项数(512)决定该分区的用户数据区从哪里开始
0x18	2	BPB_SecPerTrk	每道扇区数(Sectors Per Trark)
0x1A	2	BPB_NumHeads	磁头数(Number of head)
0x1C	4	BPB_HiddSec	隐藏扇区数(Hidden Sector)该分区上引导扇区之前的扇区数。在引导序列计算到根目录和数据区的绝对位移的过程中使用了该值
0x20	4	BPB_TotSec32	大扇区数(Large Sector) 如果小扇区数字段的值为0，本字段就包含该FAT16分区中的总扇区数。如果小扇区数字段的值不为0，那么本字段的值为0

FAT16分区的BPB字段(starting at offset 0x24)			
offset	Length	name	discription
0x24	1	BS_DrvNum	物理驱动器号(Physical Drive Number) 与BIOS物理驱动器号有关。软盘驱动器被标识为0x00，物理硬盘被标识为0x80，而与物理磁盘驱动器无关。一般地，在发出一个INT13h BIOS调用之前设置该值，具体指定所访问的设备。只有当该设备是一个引导设备时，这个值才有意义
0x25	1	BS_Reserved1	保留(Reserved) FAT16分区一般将本字段的值设置为0
0x26	1	BS_BootSig	扩展引导标签(Extended Boot Signature) 本字段必须要有能被Windows 2000所识别的值0x28或0x29
0x27	2	BS_VolID	卷序号(Volume Serial Number) 在格式化磁盘时所产生的一个随机序号，它有助于区分磁盘
0x2B	11	BS_VolLab	卷标(Volume Label) 本字段只能使用一次，它被用来保存卷标号。现在，卷标被作为一个特殊文件保存在根目录中
0x36	8	BS_FilSysType	文件系统类型(File System Type) 根据该磁盘格式，该字段的值可以为FAT、FAT12或FAT16

FAT32分区的BPB字段

offset	Length	name	discription
0x0B	2	BPB_BytsPerSec	扇区字节数(Bytes Per Sector) 硬件扇区的大小。本字段合法的十进制值有512、1024、2048和4096。对大多数磁盘来说，本字段的值为512
0x0D	1	BPB_SecPerClus	每簇扇区数(Sectors Per Cluster),一簇中的扇区数。由于FAT32文件系统只能跟踪有限个簇(最多为4 294 967 296个)，因此，通过增加每簇扇区数，可以使FAT32文件系统支持最大分区数。一个分区缺省的簇大小取决于该分区的大小。本字段的合法十进制值有1、2、4、8、16、32、64和128。Windows 2000的FAT32实现只能创建最大为32GB的分区。但是，Windows 2000能够访问由其他操作系统(Windows 95、OSR2及其以后的版本)所创建的更大的分区
0x0e	2	BPB_RsvdSecCnt	保留扇区数(Reserved Sector) 第一个FAT开始之前的扇区数，包括引导扇区。本字段的十进制值一般为32
0x10	1	BPB_NumFATs	FAT数(Number of FAT) 该分区上FAT的副本数。本字段的值一般为2
0x11	2	BPB_RootEntCnt	根目录项数(Root Entries)只有FAT12/FAT16使用此字段。对FAT32分区而言,本字段必须设置为 0
0x13	2	BPB_TotSec16	小扇区数(Small Sector)(只有FAT12/FAT16使用此字段)对FAT32分区而言，本字段必须设置为0
0x15	1	BPB_Media	媒体描述符(Media Descriptor)提供有关媒体被使用的信息。值0xF8表示硬盘，0xF0表示高密度的3.5寸软盘。媒体描述符要用于MS-DOS FAT16磁盘，在Windows 2000中未被使用
0x16	2	BPB_FATSz16	每FAT扇区数(Sectors Per FAT)只被FAT12/FAT16所使用,对FAT32分区而言，本字段必须设置为0
0x18	2	BPB_SecPerTrk	每道扇区数(Sectors Per Track) 包含使用INT13h的磁盘的“每道扇区数”几何结构值。该分区被多个磁头的柱面分成了多个磁道
0x1A	2	BPB_NumHeads	磁头数(Number of Head) 本字段包含使用INT 13h的磁盘的“磁头数”几何结构值。例如，在一张1.44MB 3.5英寸的软盘上，本字段的值为 2
0x1C	4	BPB_HiddSec	隐藏扇区数(Hidden Sector) 该分区上引导扇区之前的扇区数。在引导序列计算到根目录的数据区的绝对位移的过程中使用了该值。本字段一般只对那些在中断13h上可见的媒体有意义。在没有分区的媒体上它必须总是为0
0x20	4	BPB_TotSec32	总扇区数(Large Sector) 本字段包含FAT32分区中总的扇区数

FAT32分区的BPB字段（starting at offset 0x24）

offset	Length	name	discription
0x24	4	BPB_FATSz32	每FAT扇区数(Sectors Per FAT)(只被FAT32使用)该分区每个FAT所占的扇区数。计算机利用这个数和 FAT数以及隐藏扇区数(本表中所描述的)来决定根目录从哪里开始。该计算机还可以从目录中的项数决定该分区的用户数据区从哪里开始
0x28	2	BPB_ExtFlags	扩展标志(Extended Flag)(只被FAT32使用)该两个字节结构中各位的值为：
			位0-3：活动 FAT数(从0开始计数，而不是1).
			只有在不使用镜像时才有效
			位4-6：保留
			位7：0值意味着在运行时FAT被映射到所有的FAT
			1值表示只有一个FAT是活动的
			位8-15：保留
0x2A	2	BPB_FSVer	文件系统版本(File ystem Version)只供FAT32使用,高字节是主要的修订号，而低字节是次要的修订号。本字段支持将来对该FAT32媒体类型进行扩展。如果本字段非零，以前的Windows版本将不支持这样的分区
0x2C	4	BPB_RootClus	根目录簇号(Root Cluster Number)(只供FAT32使用) 根目录第一簇的簇号。本字段的值一般为2，但不总是如此
0x30	2	BPB_FSInfo	文件系统信息扇区号(File System Information SectorNumber)(只供FAT32使用) FAT32分区的保留区中的文件系统信息(File System Information, FSINFO)结构的扇区号。其值一般为1。在备份引导扇区(Backup Boot Sector)中保留了该FSINFO结构的一个副本，但是这个副本不保持更新
0x34	2	BPB_BkBootSec	备份引导扇区(只供FAT32使用) 为一个非零值，这个非零值表示该分区保存引导扇区的副本的保留区中的扇区号。本字段的值一般为6，建议不要使用其他值
0x36	12	BPB_Reserved	保留(只供FAT32使用)供以后扩充使用的保留空间。本字段的值总为0

FAT32分区的扩展BPB字段(starting at offset 0x24)续

offset	Length	name	discription
0x40	1	BS_DrvNum	物理驱动器号(Physical Drive Number) 与BIOS物理驱动器号有关。软盘驱动器被标识为0x00，物理硬盘被标识为0x80，而与物理磁盘驱动器无关。一般地，在发出一个INT13h BIOS调用之前设置该值，具体指定所访问的设备。只有当该设备是一个引导设备时，这个值才有意义
0x41	1	BS_Reserved1	保留(Reserved) FAT32分区总是将本字段的值设置为0
0x42	1	BS_BootSig	扩展引导标签(Extended Boot Signature) 本字段必须要有能被Windows 2000所识别的值0x28或0x29
0x43	4	BS_VolID	分区序号(Volume Serial Number) 在格式化磁盘时所产生的一个随机序号，它有助于区分磁盘
0x47	11	BS_VolLab	卷标(Volume Label) 本字段只能使用一次，它被用来保存卷标号。现在，卷标被作为一个特殊文件保存在根目录中
0x52	8	BS_FilSysType	系统ID(System ID) FAT32文件系统中一般取为"FAT32"

FAT32 FSInfo Sector Structure and Backup Boot Sector

名称	Offset (byte)	大小 (byte)	描述
FSI_LeadSig	0	4	值为 0x41615252, 这个标记用来表示该扇区为 FSInfo 扇区。
FSI_Reserved1	4	480	保留为以后扩展使用，FAT32 格式化程序应该把此域全部设置为 0，当前版本的 FAT 程序不可以访问该域。
FSI_StrucSig	484	4	值为 0x61417272，更具体地表明该扇区已经被使用
FSI_Free_Count	488	4	保存最新的剩余簇数量，如果为 0xFFFFFFFF 表示剩余簇未知，需要重新计算，初此之外其他的值都可以用，而且不要求十分精确，但必须保证其值<=磁盘所有的簇数。
FSI_Nxt_free	492	4	该域为 FAT 驱动程序提供一条有利的线索，它告诉驱动程序从哪里开始寻找剩余簇。因为 FAT32 的 FAT 表可能非常的庞大，如果已经分配的簇很多的话要从头开始查找剩余簇将耗费大量时间。通常这个值被设定为驱动程序最后分配出去的簇号。如果值为 0xFFFFFFFF，那么驱动程序必须从簇 2 开始查找，除此之外其他的值都可以使用，当然前提是这个值必须合法的。
FSI_Reserved2	496	12	保留为以后扩展使用，FAT32 格式化程序应该把此域全部设置为 0，当前版本的 FAT 程序不可以访问该域。
FSI_TrailSig	508	4	值为 0xAA550000，此结束标记用来表示这是一个 FSInfo 扇区，注意次域的高两位的偏移量为 510 和 511，这和启动扇区在相同偏移处的标记是一样的。

空磁盘BPB值的计算

计算FAT SIZE

看f_mkfs代码

计算值存储在BPB_FATSz16/ BPB_FATSz32字段

根目录扇区数（FAT32 BPB_RootEntCnt为0， RootDirSectors 也为0）

RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec – 1)) / BPB_BytsPerSec;

数据区起始簇（也就是簇2的第一个扇区）

```
If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;
FirstDataSector = BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors;
```

给定一个簇号计算该簇的起始扇区

FirstSectorofCluster = ((N – 2) * BPB_SecPerClus) + FirstDataSector;

确定磁盘FAT类型

数据区总扇区数

```
If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;
If(BPB_TotSec16 != 0)
    TotSec = BPB_TotSec16;
Else
    TotSec = BPB_TotSec32;
DataSec = TotSec – (BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors);
```

总的簇数

```
CountofClusters = DataSec / BPB_SecPerClus;
```

文件系统类型

```
If(CountofClusters < 4085) {
    /* Volume is FAT12 */
} else if(CountofClusters < 65525) {
    /* Volume is FAT16 */
} else {
    /* Volume is FAT32 */
}
```

这是检测FAT类型的唯一办法。与BS_FilSysType字段中的字符串无任何关系。世上不存在簇数大于4084的FAT32卷，也不存在簇数小于4085或是大于65524的FAT16卷，同样没有哪个FAT32卷的簇数小于65525.

以上限制的依据是什么？

给定一个簇号计算它在FAT表中的位置

```
If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;
If(FATType == FAT16)
    FATOffset = N * 2;
Else if (FATType == FAT32)
    FATOffset = N * 4;
ThisFATSecNum = BPB_ResvdSecCnt + (FATOffset / BPB_BytsPerSec);
ThisFATEntOffset = REM(FATOffset / BPB_BytsPerSec);
```

这个计算经常用在读文件时读完一个簇然后到FAT表找下一个链接的簇号

FAT16/32目录项内容对比

FAT16目录项32个字节的表示定义

字节偏移	字节数	定义	
0x0~0x7	8	文件名	
0x8~0xA	3	扩展名	
0xB	1	属性字节	00000000(读写)
			00000001(只读)
			00000010(隐藏)
			00000100(系统)
			00001000(卷标)
			00010000(子目录)
			00100000(归档)
0xC~0x15	10	系统保留	
0x16~0x17	2	文件的最近修改时间	
0x18~0x19	2	文件的最近修改日期	
0x1A~0x1B	2	表示文件的首簇号	
0x1C~0x1F	4	表示文件的长度	

FAT32短文件目录项32个字节的表示定义

字节偏移	字节数	定义	
0x0~0x7	8	文件名	
0x8~0xA	3	扩展名	
0xB	1	属性字节	00000000(读写)
			00000001(只读)
			00000010(隐藏)
			00000100(系统)
			00001000(卷标)
			00010000(子目录)
			00100000(归档)
0xC	1	系统保留	
0xD	1	创建时间的10毫秒位	
0xE~0xF	2	文件创建时间	
0x10~0x11	2	文件创建日期	
0x12~0x13	2	文件最后访问日期	
0x14~0x15	2	文件起始簇号的高16位	
0x16~0x17	2	文件的最近修改时间	
0x18~0x19	2	文件的最近修改日期	
0x1A~0x1B	2	文件起始簇号的低16位	
0x1C~0x1F	4	表示文件的长度	

FAT表

FAT表项的大小与FAT类型有关，FAT12的表项为12-bit，FAT16为16-bit，而FAT32则为32-bit。对于大文件需要分配多个簇。同一个文件的数据并不一定完整地存放在磁盘中哦你哦个一个连续的区域内，而往往会分成若干段，像链子一样存放。为实现文件的链式存储，文件系统必须准确的记录哪些簇已经被文件占用，还必须为每个已经占用的簇指明存储后继内容的下一个簇号。对文件的最后一个簇，要指明本簇无后继簇。这些都是FAT表来保存的，FAT表的对应表项中记录着它所代表簇的有关信息：是否为空，是否块簇，是否已经是某个文件的尾簇等。

注意：已分配的簇号从2开始(0,1已经被系统占用)

表项	示例代码	描述
0	FFF8	磁盘标识字，必须为 FFF8
1	FFFF	第一簇已经被占用
2	0003	0000h : 可用簇 0002h - FFFEFh: 已用簇，表项中存放文件下个簇的簇号 FFFF0h - FFF6h : 保留簇 FFF7h : 坏簇 FFF8h - FFFFh : 文件的最后一簇
3	0004	
……	……	
N	FFFF	
N+1	0000	
……	……	

FAT12/16/32各FAT表项的取值

FAT 表中各系统记录项的取值含义			
FAT12记录项的取值	FAT16 记录项的取值	FAT32 记录项的取值	对应簇的表现情况
000	0000	00000000	未分配的簇
002~FFF	0002~FFEF	00000002~FFFFFFEF	已分配的簇
FF0~FF6	FFF0~FFF6	FFFFFFF0~FFFFFFF6	系统保留
FF7	FFF7	FFFFFFF7	坏簇
FF8~FFF	FFF8~FFFF	FFFFFFF8~FFFFFFFF	文件结束簇

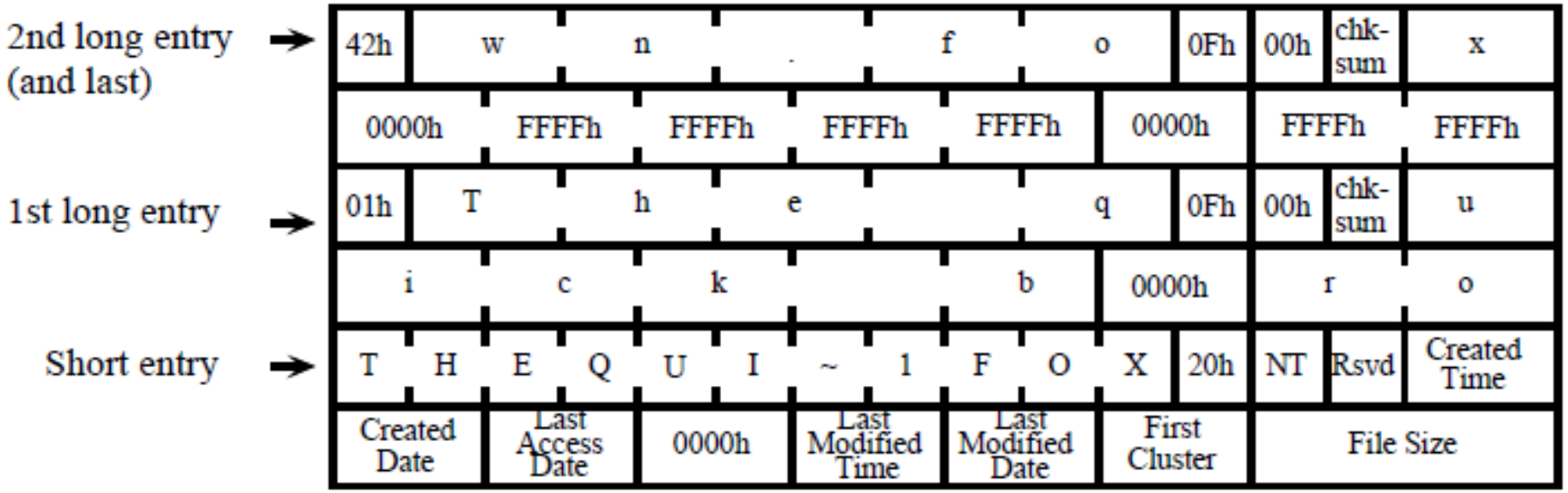
FAT 长目录项结构（FAT Long Directory Entry Structure）

名称	Offset (byte)	大小 (byte)	描述
LDIR_Ord	0	1	该长目录项在本组（long dir set）中的序号，如果标记为 0x40(LAST_LONG_ENTRY)则表明是该组的最后一个长目录项项。长目录项必须以 LDIR_Ord 开始。
LDIR_Name1	1	10	长文件名子项的第 1-5 个字符。
LDIR_Attr	11	1	属性必须为 ATTR_LONG_NAME
LDIR_Type	12	1	如果为 0 表明是长文件名的子项。 NOTE: 其他值保留供以后扩展使用，如果非零表明是其他目录类型
LDIR_Chksum	13	1	短文件名的校验和
LDIR_Name2	14	12	长文件名子项的第 6-11 个字符。
LDIR_FstClusLO	26	2	必须为 0，这是 FAT 的“第一个簇”，对于长目录项，此域没有任何意义，只是为了兼容早期的程序。
LDIR_Name3	28	4	长文件名子项的第 12-13 个字符。

长目录项的序号（Sequence Of Long Directory Entries）

项	序号
第 N 个（最后一个）长目录项	LAST_LONG_ENTRY(0x40) | N
…其他的长目录项	…
第一个长目录项	1
长目录项之前的短目录项	（无）

创建一个 “The quick brown.fox”的文件



长文件名创建条件

- 1.文件名中含有空格、’.’
- 2. “+,;=[]”替换为’_’
- 3.大小写混合
- 4.非ASCII字符

以上的文件名即使符合8.3格式都会创建长文件名

FAT文件名

DIR_NAME[0]

此处特别注释目录项的第一个字节(DIR_NAME[0]).

- 如果 DIR_Name[0] == 0xE5, 则此目录为空(目录项不包含文件和目录)
- 如果 DIR_Name[0] == 0x00, 则此目录为空(同 0xE5),并且此后的不再分配有目录项(此后所有的DIR_Name[0]均为 0).

不同于 0xE5,如果 DIR_Name[0]的值为 0，那么 FAT 程序将不会再去检测其后续的磁盘空间，因为这些空间都是空闲的。

•如果 DIR_Name[0] == 0x05,则文件名在该位的实际值为 0xE5,0xE5 是日文中合法的字符,当需要用 0xE5 来作为 DIR_Name[0]时使用 0x05 来代替，避免程序误认为该目录项为空。

DIR_Name 域实际由两部分组成：8 个字符的主文件名和 3 个字符的扩展名。两部分如果字符数不够的话由空格(0x20)填充（trailing space padded）.

DIR_Name[0]不允许为 0x20,主文件名和扩展名之间的间隔 ‘.’并不真实的存在于 DIR_Name 中，小写字母不允许出现在 DIR_Name 中(这些字符因不同的国家和地区而已).

- 以下字符不允许出现在 DIR_Name 中的任何位置:

  0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D,还有 0x7C。

文件名举例

所有8.3格式的短文件名，字母都是大写的。

以下是一些例子显示用户输入的文件名如何以 DIR_Name 对应:

“foo.bar”	->	“FOO BAR”
“FOO.BAR”	->	“FOO BAR”
“Foo.Bar”	->	“FOO BAR”
“foo”	->	“FOO ”
“foo.”	->	“FOO ”
“PICKLE.A”	->	“PICKLE A ”
“prettybg.big”	->	“PRETTYBGBIG”
“ .big”	->	非法, DIR_Name[0] 不能为 0x20。

WINHEX演示

- 格式化文件系统
- 创建文件

空文件/一簇文件/多簇文件

- 重命名文件
- 寻址文件

根目录文件/子目录文件

- 删除文件
- 恢复文件

FATFS重要的数据结构

FATFS

The FATFS structure holds dynamic work area of individual logical drives. It is given by application program and registerd/unregisterd to the FatFs module with f_mount function. Following members are in standard configuration. There is no member that can be changed from the application program.

```
typedef struct {
    BYTE    fs_type;        /* FAT sub-type (0:Not mounted) */
    BYTE    drv;            /* Physical drive number */
    BYTE    csize;          /* Sectors per cluster (1,2,4...128) */
    BYTE    n_fats;         /* Number of FAT copies (1,2) */
    BYTE    wflag;          /* win[] dirty flag */
    BYTE    fsi_flag;       /* fsinfo dirty flag */
    WORD    id;             /* File system mount ID */
    WORD    n_rootdir;      /* Number of root directory entries (FAT12/16) */
#if _MAX_SS != 512
    WORD    ssize;          /* Sector size (512,1024,2048,4096) */
#endif
#if _FS_REENTRANT
    _SYNC_t sobj;           /* Identifier of sync object */
#endif
#if !_FS_READONLY
    DWORD   last_clust;     /* Last allocated cluster */
    DWORD   free_clust;     /* Number of free clusters */
    DWORD   fsi_sector;     /* fsinfo sector (FAT32) */
#endif
#if _FS_RPATH
    DWORD   cdir;           /* Current directory cluster (0:root) */
#endif
    DWORD   n_fatent;       /* Number of FAT entries (= number of clusters + 2) */
    DWORD   fsize;          /* Sectors per FAT */
    DWORD   fatbase;        /* FAT area start sector */
    DWORD   dirbase;        /* Root directory area start sector (FAT32: cluster#) */
    DWORD   database;       /* Data area start sector */
    DWORD   winsect;        /* Current sector appearing in the win[] */
    BYTE    win[_MAX_SS]; /* Disk access window for Directory, FAT (and Data on tiny cfg) */
} FATFS;
```

FATFS重要的数据结构

可以指向一个目录项，或者文件夹，其实文件夹没有实际的数据，所以就是一个目录项而已。

```
/* Directory object structure (DIR) */

typedef struct {
        FATFS*  fs;                     /* Pointer to the owner file system object */
        WORD    id;                     /* Owner file system mount ID */
        WORD    index;                  /* Current read/write index number */
        DWORD   sclust;                 /* Table start cluster (0:Root dir) */
        DWORD   clust;                  /* Current cluster */
        DWORD   sect;                   /* Current sector */
        BYTE*   dir;                    /* Pointer to the current SFN entry in the win[] */
        BYTE    fn[12];                 /* The SFN (in/out) {file[8],ext[3],status[1]} */
#if _USE_LFN
        WCHAR*  lfn;                    /* Pointer to the LFN working buffer */
        WORD    lfn_idx;                /* Last matched LFN index number (0xFFFF:No LFN) */
#endif
} DIR;
```

The FIL structure (file object) holds state of an open file. It is created by f_open function and discarded by f_close function. There is no member that can be changed by the application program except for cltbl. Note that a sector buffer is defined in this structure under non-tiny configuration so that the FIL structures should not be defined as auto variable.

```
/* File object structure (FIL) */

typedef struct {
        FATFS*  fs;                             /* Pointer to the owner file system object */
        WORD    id;                             /* Owner file system mount ID */
        BYTE    flag;                   /* File status flags */
        BYTE    pad1;
        DWORD   fptr;                   /* File read/write pointer (0 on file open) */
        DWORD   fsize;                  /* File size */
        DWORD   org_clust;              /* File start cluster (0 when fsize==0) */
        DWORD   curr_clust;             /* Current cluster */
        DWORD   dsect;                  /* Current data sector */
#if !_FS_READONLY
        DWORD   dir_sect;               /* Sector containing the directory entry */
        BYTE*   dir_ptr;                /* Ponter to the directory entry in the window */
#endif
#if _USE_FASTSEEK
        DWORD*  cltbl;                  /* Pointer to the cluster link map table (null on file open) */
#endif
#if _FS_SHARE
        UINT    lockid;                 /* File lock ID (index of file semaphore table) */
#endif
#if !_FS_TINY
        BYTE    buf[_MAX_SS];   /* File data read/write buffer */
#endif
} FIL;
```

The FILINFO structure holds a file information returned by f_stat and f_readdir function.

```
/* File status structure (FILINFO) */

typedef struct {
        DWORD   fsize;                  /* File size */
        WORD    fdate;                  /* Last modified date */
        WORD    ftime;                  /* Last modified time */
        BYTE    fattrib;                /* Attribute */
        BYTE    fname[13];              /* Short file name (8.3 format) */
#if _USE_LFN
        TCHAR*  lfname;                 /* Pointer to the LFN buffer */
        UINT    lfsize;                 /* Size of LFN buffer in TCHAR */
#endif
#if _USE_NAV_API
        DWORD   sclust;             /* start cluster of object */
#endif
} FILINFO;
```

如何快速看懂FATFS代码

1.需要阅读的文档

FAT32 File System Specification

FAT文件系统原理

2.“牢记”之前PPT列出的那些表格

3.代码中很多是计算，用winhex获取数据计算一下看自己的推断是否正确。

基于FAT的数据库

基于FAT的数据库

为什么需要数据库？

数据库最重要的用途是为了加快搜索，这就是为什么SQL语言中如此多的select语句。

用这个数据库，加快SD卡中文件的定位。
数据库跟文件读写没任何关系。

基于FAT的数据库

如果没有数据库，是如何定位文件的？

例：通过普通文件操作定位 0:/dir0/dir1/dir2/T1.txt

Step1:从根目录从头到尾搜索文件名dir0

Step2:获取到dir0的入口，到该目录区继续从头到尾搜索dir1

Step3:获取到dir2的入口，到其指向的目录区继续从头到尾搜索T1.txt

特点：

每一次都是从根开始，从头到尾遍历。而且都是I/O操作。

全部都是文件名的比较，而且是长文件名的比较

基于FAT的数据库

数据库加快文件搜索使用的一些策略和做的工作。

1.SD卡初始化时即把所有的文件夹起始簇号和其中指定类型的文件数量存储好

2.加入了CACHE，最近使用的文件重要参数都保存在CACHE中

3.使用短文件名搜索文件

4.在开源的FAT系统数据结构FILINFO中增加了sclust变量，可以保存文件的起始簇号

基于FAT的数据库

为什么数据库能加快文件定位？

1.内存中保存着每一个文件夹的首地址。
2.基于序号。
3.基于缓存。

例：通过数据库定位0:/dir0/dir1/dir2/T1.txt

fdb_database中已经有dir2的起始地址，只需在数据库中定位它，如果缓存中有，直接获取，如果没有再到dir2的目录区搜索T1.txt。

减少了dir0,dir1,dir2三个文件夹的定位操作，而且都是基于RAM的操作。

- 之后的因为属于内部开发的内容，所以隐藏。Sorry