一、通过格式化命令-看磁盘文件系统的建立过程

1、添加 format 命令，单步调试

所有的底层驱动函数都已经准备好。添加格式化命令 format 后，编译下载。

Format 命令的执行主要是调用 f_mkfs（）函数，下面进行单步调试。

以下主要列出函数的主要执行步骤：

res=f_mkfs(0, 1, 4096); //1表示不需要引导扇区。4096是8个扇区。

进入 f_mkfs（）函数，这里只列出主要执行步骤：

if (disk_ioctl(drv, GET_SECTOR_COUNT, &n_part) != RES_OK || n_part < MIN_SECTOR)

return FR_MKFS_ABORTED;这个函数调用后，n_part=0x000F,3400 = 996 352，这是 SD 的总块数。

allocsize /= SS(fs); 等于8/* Number of sectors per cluster */

n_clst = n_part / allocsize; //等于0x1E680 = 124 544 簇。

if (n_clst >= 0xFFF5) fmt = FS_FAT32; 所以文件系统确定为 FAT32类型。

n_fat = ((n_clst * 4) + 8 + SS(fs) - 1) / SS(fs); 等于0x3CE = 974，表示 FAT 要占据974个扇区。

n_rsv = 33 - partition; 保留扇区32个。

n_dir = 0;

b_fat = b_part + n_rsv; /* FATs start sector 32扇区*/

b_dir = b_fat + n_fat * N_FATS; /* Directory start sector 0x3EE =1006，由于 FAT 表个数设为1个，所以目录区=FAT 起始+FAT 占用扇区数*/

b_data = b_dir + n_dir; /* Data start sector */

以上三项确定 FAT 区域、根目录区、数据区的起始扇区。

disk_ioctl(drv, GET_BLOCK_SIZE, &n) != RES_OK，这个函数调用没有正确返回 可擦出扇区的总数。

接下来程序会出错，因此退出，修改 disk_ioctl（）函数后，再次分析。把这个函数返回值直接改为32。并且把 FAT 表的个数定义为2.

N_FATS 改为2后，根目录区、数据区的起始扇区的起始扇区变为0x7BC=1980扇区。继续往下执行。

n = (b_data + n - 1) & ~(n - 1);

n_fat += (n - b_data) / N_FATS;这两句话对 fat 所占扇区数进行了修正，保证擦除时，以32个扇区为一个单位。

n_clst = (n_part - n_rsv - n_fat * N_FATS - n_dir) / allocsize; =0x1E588。

tbl = fs->win; /* Clear buffer */

mem_set(tbl, 0, SS(fs)); 清零文件系统缓冲区。

mem_set(tbl, 0, SS(fs));

ST_DWORD(tbl+BS_jmpBoot, 0x90FEEB); /* Boot code (jmp $, nop) */

ST_WORD(tbl+BPB_BytsPerSec, SS(fs)); /* Sector size */

tbl[BPB_SecPerClus] = (BYTE)allocsize; /* Sectors per cluster */

ST_WORD(tbl+BPB_RsvdSecCnt, n_rsv); /* Reserved sectors */

上面的工作主要是填充 引导扇区缓冲区，也就是常说的 DBR 扇区缓冲，等所有的参数写好，就可以写回磁盘。

ST_WORD(tbl+BS_55AA, 0xAA55); /* Signature */

if (disk_write(drv, tbl, b_part+0, 1) != RES_OK)

return FR_DISK_ERR; //这就是在写有效引导标志 sec[510]=0x55, sec[511]=0xAA。

if (fmt == FS_FAT32)

disk_write(drv, tbl, b_part+6, 1); //FAT32在第六扇区有个备份引导扇区。

for (m = 0; m < N_FATS; m++) {

mem_set(tbl, 0, SS(fs)); /* 1st sector of the FAT */

if (fmt != FS_FAT32) {

n = (fmt == FS_FAT12) ? 0x00FFFF00 : 0xFFFFFF00;

n |= partition;

ST_DWORD(tbl, n); /* Reserve cluster #0-1 (FAT12/16) */

} else {

ST_DWORD(tbl+0, 0xFFFFFFF8); /* Reserve cluster #0-1 (FAT32) */

ST_DWORD(tbl+4, 0xFFFFFFFF);
ST_DWORD(tbl+8, 0x0FFFFFFF); /* Reserve cluster #2 for root dir */ } //簇0和簇1保留，簇2分配给根目录区。
if (disk_write(drv, tbl, b_fat++, 1) != RES_OK)
return FR_DISK_ERR;
mem_set(tbl, 0, SS(fs)); /* Following FAT entries are filled by zero */ //接下来所有的扇区都清0，表示该簇未被占用。
for (n = 1; n < n_fat; n++) {
if (disk_write(drv, tbl, b_fat++, 1) != RES_OK)
return FR_DISK_ERR;
} //第一次是写从0x21扇区开始，总共 0x3CF 个扇区（第一个扇区已经写了）。第二次开始时是备份引导扇区，0x20+0x3D0=0x3F0。同理，第一扇区与0x21扇区相同，后面都清零。
Format 命令要执行较长的时间，主要就是 FAT 表接近2000个扇区要清零。
m = (BYTE)((fmt == FS_FAT32) ? allocsize : n_dir); m=8,每簇8扇区。
do {
if (disk_write(drv, tbl, b_fat++, 1) != RES_OK)
return FR_DISK_ERR;
} while (--m); //以上部分是将根目录区的8个扇区清零，表示目录项未被占用。
if (fmt == FS_FAT32) {
ST_WORD(tbl+BS_55AA, 0xAA55);
ST_DWORD(tbl+FSI_LeadSig, 0x41615252);
ST_DWORD(tbl+FSI_StrucSig, 0x61417272);
ST_DWORD(tbl+FSI_Free_Count, n_clst - 1); //根目录区已经用掉了一簇。
ST_DWORD(tbl+FSI_Nxt_Free, 0xFFFFFFFF);
disk_write(drv, tbl, b_part+1, 1); //分别写入1号和7号扇区。
disk_write(drv, tbl, b_part+7, 1);
} //这是写 FAT32文件系统的 FSI 扇区，包括空闲簇总数和上次分配的簇号。
FAT32文件系统的格式化到此完成。
2、再以 Fdisk 的方式格式化一次，这时候格式系统要写 MBR 扇区，FAT 引导扇区不在 0号扇区了。同时 MBR 要建立分区表，以找到 DBR 引导扇区。其过程与上述差不多，只是多执行了以下代码，详细过程就不叙述了。
if (!partition) {
DWORD n_disk = b_part + n_part;
mem_set(fs->win, 0, SS(fs));
tbl = fs->win+MBR_Table; //分区表从0x1BE 开始。
ST_DWORD(tbl, 0x00010180); /* Partition start in CHS */ Table[0x1BE] = 0x80，表明该分区是活动扇区。00表示开始柱面，01、01表示开始扇区、开始磁头。
if (n_disk < 63UL * 255 * 1024) { /* Partition end in CHS */
n_disk = n_disk / 63 / 255;
tbl[7] = (BYTE)n_disk; //表示结束的柱面。
tbl[6] = (BYTE)((n_disk >> 2) | 63); //结束的扇区。
} else {
ST_WORD(&tbl[6], 0xFFFF); //
}
tbl[5] = 254; //结束的磁头。
if (fmt != FS_FAT32) /* System ID */
tbl[4] = (n_part < 0x10000) ? 0x04 : 0x06;
else
tbl[4] = 0x0c; // 表示该分区类型为 win95 FAT32
ST_DWORD(tbl+8, 63); /*起始扇区 0x3F in LBA */

ST_DWORD(tbl+12, n_part); /*分区的大小，总扇区数减去 MBR 及其占据的一个柱面。 */
ST_WORD(tbl+64, 0xAA55); /* Signature */
if (disk_write(drv, fs->win, 0, 1) != RES_OK)
return FR_DISK_ERR;
partition = 0xF8; //MBR 标志。
} else {
partition = 0xF0;
}
3、观察在有 MBR 区域的情况下，如何检查文件系统
fmt = check_fs(fs, bsect = 0); /* 检查0扇区的时候，没有发现 FAT 文件系统扇区，但是有0x55 0xAA 标志，说明这是有效磁盘，但是返回1. */
if (fmt == 1) { /* 表明可能存在分区 */
/* Check a partition listed in top of the partition table */
tbl = &fs->win[MBR_Table + LD2PT(vol) * 16]; /* Partition table */
if (tbl[4]) { 实际这里应该是0x0c,表示 FAT32系统。
bsect = LD_DWORD(&tbl[8]); /* 这个是文件系统 引导扇区的号码。 */
fmt = check_fs(fs, bsect); /* 再到这个扇区检查是否存在 FAT 文件系统标志。 */ } }
执行过后，仍然能够建立完整的 文件系统信息 结构体，只是里面的 FAT 分配起始扇区、数据区起始扇区地址相对 没有 MBR 的时候改变了，其它都差不多。

二、将 SD 卡格式化成具有两个分区的磁盘。

1、目的

（1）深入理解 MBR、DPT 等概念。

（2）修改 ff.c 中的 f_mkfs 函数，得到一个新函数，f_format（u8 partition,u16 allocsize）,前一个参数是指磁盘等分的个数，接受 1、2、3、4四个参数，默认为1，最大为4。后一个参数是指 每簇占用的字节数。

（3）添加命令 fdisk，调用上述函数。执行完成后，用读卡器在 PC 上读取该 SD 卡，应该显示两个可移动磁盘。

2、f_format（）函数的编写

首先新建一个文件 fext.c，该文件就实现一个函数 f_format.c，首先将 f_mkfs（）函数复制过来，在此基础上修改。

编译后，首先解决警告和错误：包含头文件 ff.h 和 diskio.h。引用了 ff.c 中的静态函数 mem_set()和 mem_clr()，复制过来。定义 Null 为0。将 FATFS * FatFs[_Drives]做外部声明。

同时发现，不同 c 源文件中 #define 同样的宏相互之间是不影响的。说明预处理的时候是一个一个文件处理的，不检查相互之间的关联。但同一个文件中，一个宏不能两次定义。

```
#ifndef NULL
#define NULL 0
#endif //采取这种方式，主要是防止其他 包含的头文件也对 NULL 进行了定义。
```

函数中主要修改的地方就是：

```
n_part = n_part /drv; //进行 drv 等分。每个磁盘的扇区总数就是这么多。
```

在 DPT 对应增加的分区部分，填好分区表16个字节。特别重要的是四个地方：0字节写为00或0x80，第四字节写入0xc0表示 FAT32系统。（第一次调试找不到新磁盘，就是由于这个字节默认为0）。第8-11写入分区引导扇区的线性扇区地址。第12-15写入该磁盘分区的大小。

```
for ( i="0"; i<drv; i++){ //每个分区都要做一次，DBR 的写入，FAT 分配表初始化，根目录初始化。
b_part+= i*n_part; //调整该分区引导扇区和 FAT 表起始地址。
b_fat=b_part + n_rsv;
```

3、其它修改的地方

定义 _DRIVE 为2，定义_MULTI_PARTITION 为1，表示支持多分区。同时初始化磁盘 物理驱动与分区号转换结构体（每个逻辑磁盘对应一个结构体）。

主程序中也要定义两个文件系统结构体，每个对应一个磁盘。然后分别调用 f_mount（）函数。

4、调用执行

```
void UartCmdFdisk(u8 argc,void **argv)
{
FRESULT res;
res=f_format( 2, 2048 ); /2表示格式化成两个磁盘。一簇是4个扇区、2048字节。
```

执行完这个函数后，磁盘被分成两个分区。

实际结果是，命令界面上可以查询到两个分区。但是利用读卡器放到 PC 上，却只看到250M 的第一分区，不知什么原因。

三、添加命令 fchdrive、fmove。

1、目的

（1）fchdrive：在支持相对目录 的情况下改变当前逻辑磁盘。

（2）fmove:两个参数，将文件移动到另一个位置，可以改名，也可以保留原名。

2、fchdrive 命令的实现

（1）添加命令支持

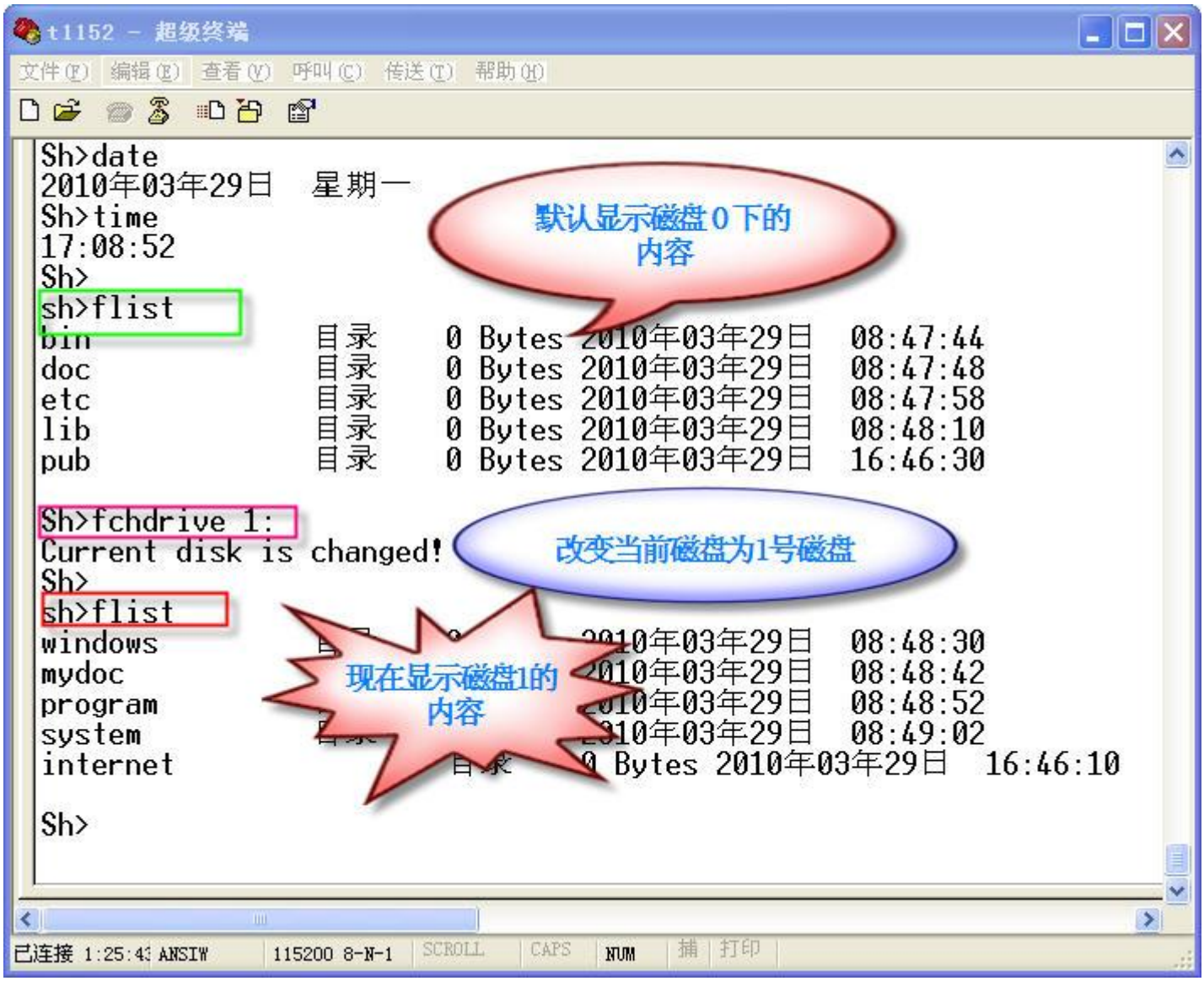
这个不再赘述。

（2）实现代码

```
void UartCmdFChDrive(u8 argc,void **argv){
BYTE Drive;
FRESULT res="FR"_INVALID_DRIVE;
if ( *( (BYTE*)argv[1]+1 )==':' ){ 检查参数，是不是1：类似格式。
Drive =*( (BYTE*)argv[1]) - '0'; 取出磁盘号
res=f_chdrive( Drive ); }
if ( res!=FR_OK) { Uart_PutString( "Invalid path!\r\n");return;}
Uart_PutString( "Current disk is changed!\r\n");
}
```

((BYTE)argv[1])这个意思是先将 argv[1]转换为 BYTE 指针，再取出里面的数据（BYTE 型，一个字节。）

以下是实现的截图：



3、fmove 命令的实现

（1）这个命令带两个参数：第一个是文件的原有路径，第二个是新路径，同时可以更改文件名。

（2）实现的原理

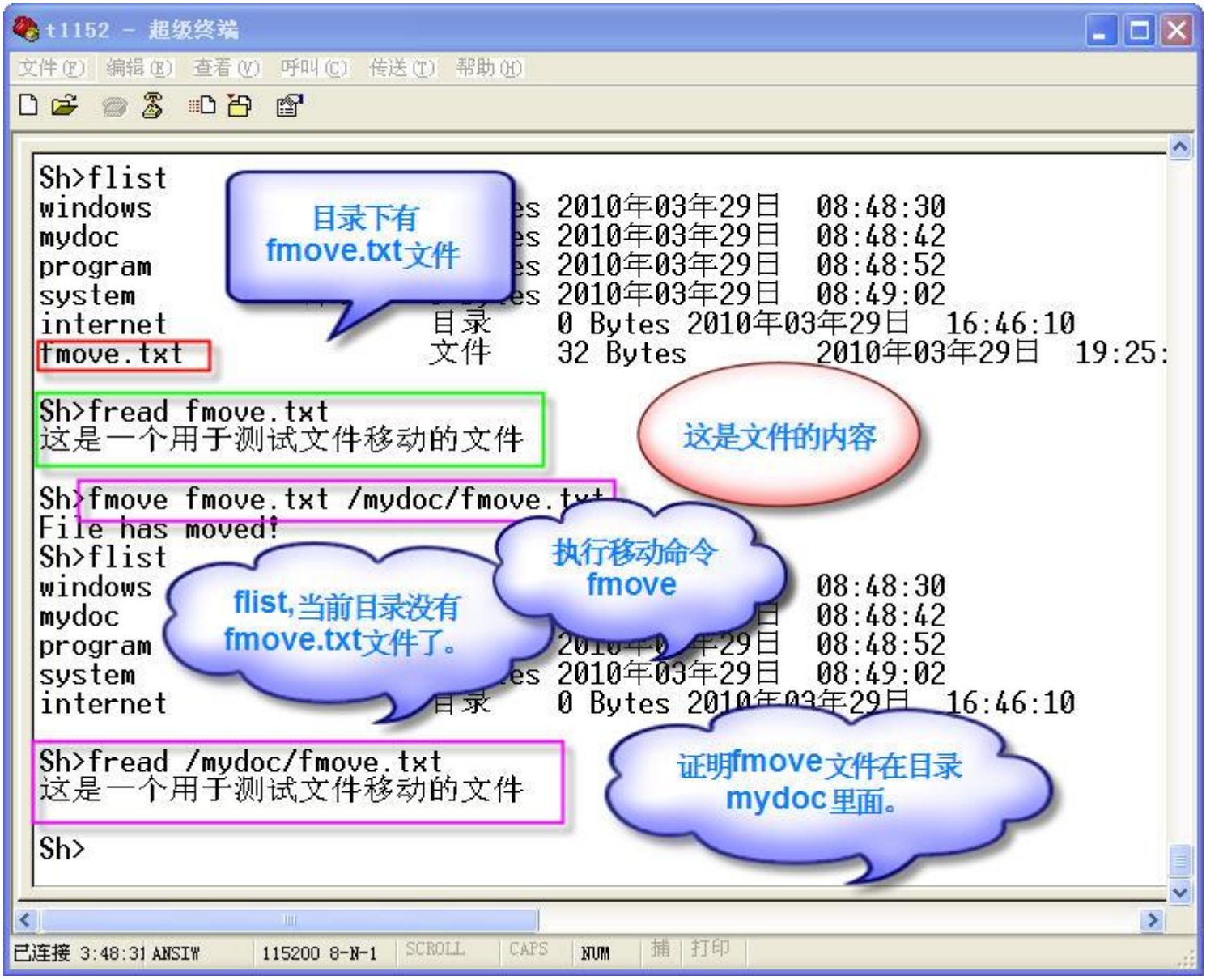
以创建新文件的方式 打开新路径的文件。这样会创建一个目录项，填充一个文件信息结构体。然后打开原有路径的文件，也会得到一个文件信息结构体。将旧结构体的文件大小、起始簇号复制到新的结构体，然后更新新文件的信息。然后删除原有路径的文件。移动操作完成。

（3）代码编写

需要两次打开文件，所以需要两个文件信息结构体。

```
res = f_open ( &FileOld, (const char *)argv[1], FA_WRITE );
res = f_open ( &FileNew, (const char *)argv[2], FA_CREATE_NEW );
FileNew.fsize = FileOld.fsize; //  改变新文件的大小等于原文件
FileNew.org_clust = FileOld.org_clust; //新文件的簇等于原文件的簇。
FileNew.flag |= FA__WRITTEN; //文件属性已改变，需要更新
res=f_close ( &FileNew ); //关闭时进行更新。
res=f_unlink ( (const char *)argv[1] ); //删除原文件
```

编译，下载测试，功能正常。以下是实现的截图。



四、添加命令 fcopy、fpath。

1、目的

（1）fcopy：复制功能，原文件保留，新位置建立文件。

（2）fpath：在支持相对路径的情况下，显示当前磁盘的当前目录。

2、fcopy 命令的实现

（1）复制命令的实现思路

首先要打开两个文件，原文件以可读 FA_READ 的方式打开，新文件以 FA_CREATE_NEW 和 FA_WRITE 的方式打开。

然后建立一个循环，每次从原文件读出一个扇区的数据，写入新文件。然后检查原文件 结束标志，已到结尾，则跳出循环。

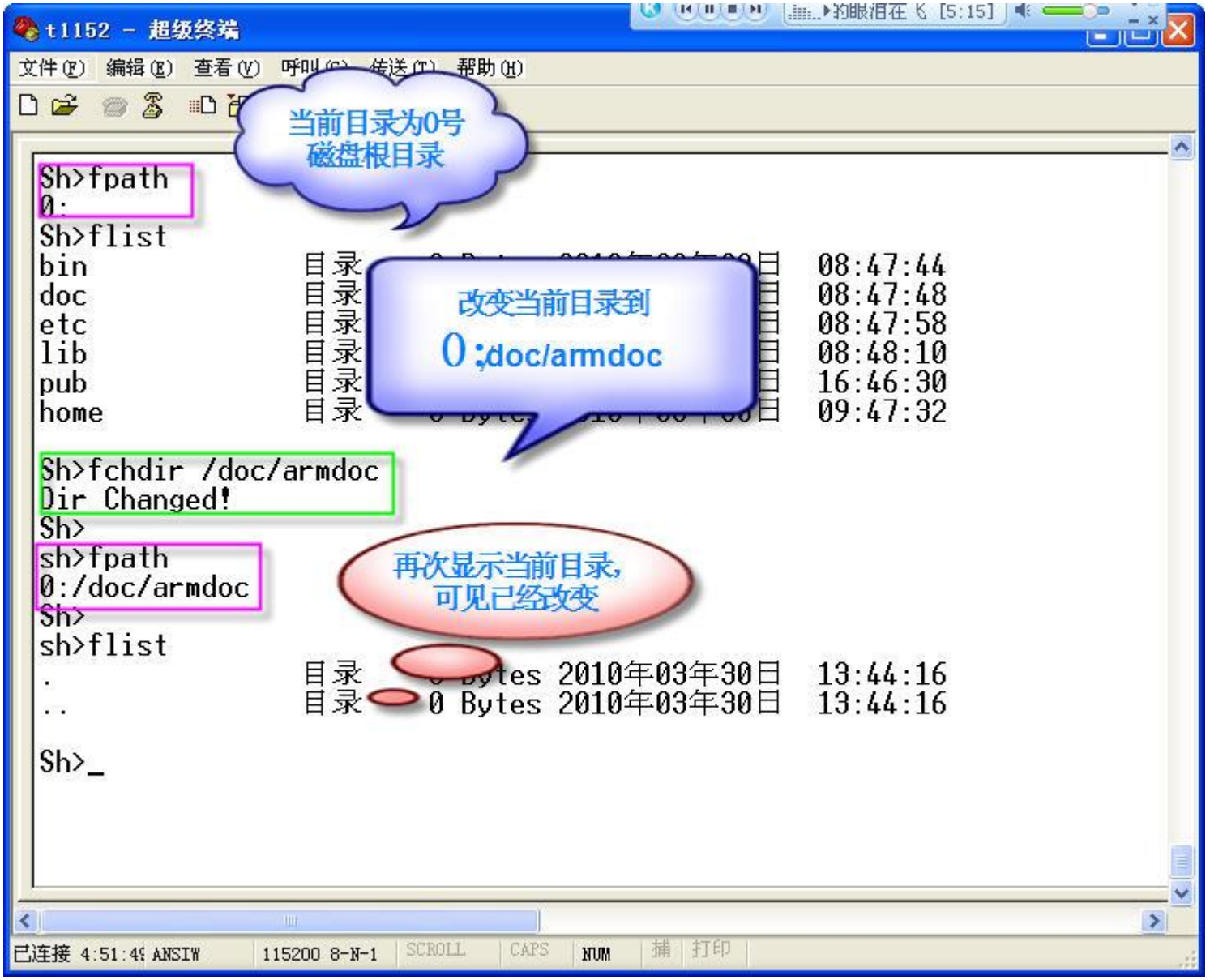
关闭两个文件，更新文件信息。

（2）代码实现

```
res = f_open ( &FileOld, (const char *)argv[1], FA_READ );
res = f_open ( &FileNew, (const char *)argv[2], FA_CREATE_NEW | FA_WRITE );
for ( ; ; ){
res = f_read ( &FileOld, (void*)FileBuf, 512, &ByteRead );
res = f_write ( &FileNew, ( const void *)FileBuf, ByteRead, &ByteWrite );
if (FileOld.fptr== FileOld.fsize) )break;
```

}
f_close (&FileOld);
f_close (&FileNew);
编译，下载，功能正确。
3、fpath 命令的实现
（1）这个命令实现起来稍微有些难度
首先要获取当前磁盘 和 当前磁盘的当前目录所在簇，也就是FatFs[Drive]->cdir，如果为0，表明是在磁盘根目录，显示0：或者1:。如果该簇号等于根目录所在簇号，当前目录也是在跟目录。
如果不是在根目录，那就要逐层往上搜索了。根据 f_opendir (DIR，..目录)可以回溯到上层目录，DIR 结构体得到了上层目录（X+1）的起始簇号，目录项指针指向第0项。用 f_readdir（）逐步读出目录项属性，得到 FILINFO 结构体。如果其簇号 等于 文件系统的当前搜索簇号，则其名称就是所要得到的本层目录名。如果是根目录下则为 0：/X 目录名。
当前搜索簇号设为（X+1）的起始簇号。再次调用 f_opendir(DIR，..目录)，此次得到（X+2）的起始簇号，判断是否根目录。查找当期搜索簇号在（X+2）目录层中对应的目录项，并获取 X+1层的目录名。如果是在根目录，则为0:/X+1目录名/X 目录名。
循环直到到达根目录跳出搜索。
目录名缓冲区的处理方法。定义总长度为100字节。Path[99]=0；初始化时先让指针指向 Path【99】。
得到一个目录，向前移动名称那么长，复制名称。再往前移动1，添加’/’标志。如果是根目录添加’:’，往前移动1，在添加‘驱动号’，然后可以显示整个字符串。
（2）代码实现
```
for ( ; ; ){
if ( CurClust == 0 || CurClust == CurFileSys->dirbase ) {
PathPtr--;
*PathPtr--=':'; /
*PathPtr = CurDrive+'0'; //如果当前簇号对应根目录。
break; }
res = f_opendir ( &DirInf, ".." ); //第一次执行时，DIR 结构体的开始簇号变为X+1层目录的簇号 //第二次时，变为 X+2层目录的簇号。
if ( res!= FR_OK ) break;
do {
res = f_readdir ( &DirInf, &FileAttrib );
TempClust = FileAttrib.sclust;
if ( TempClust == CurClust ) break; //如果该目录项的簇号等于当前搜索簇号，它的名称就是当前需要的目录名
} while( res == FR_OK );
if ( res!= FR_OK ) break;
DirLen= Str_Length ( (const char*)FileAttrib.fname ); //这里要得到字符串的长度。
PathPtr -= DirLen; //文件指针往后退目录名长度
mem_cpy ( (void *)PathPtr, ( const void*)FileAttrib.fname, DirLen);
PathPtr--;
*PathPtr='/'; //添加文件间隔/符号。
CurClust = DirInf.sclust; //当前搜索簇号跟着上移。
}
Uart_PutString( PathPtr); //这是显示的当前目录。
Uart_PutString( "\r\n");
```

编译，下载，显示两层目录是正确的。但是目录一旦到第三层，就进入死循环。
还要接着调试。
（3）调试
经调试发现 res = f_opendir (&DirInf, "..")调用后，并不能每次都自动回溯到上一层目录。
所以改为：f_opendir (&DirInf, (const char*)ddPtr)，ddPtr 初始化成“..”，然后每循环一次，前面加上”../”，第二次变为 ”../..”，第三次变为“../../.. “，这样目录不断回溯。
以下是该命令实现的截图。



；

五、使用文件字符串处理的功能
1、目的
验证 f_printf（）函数的功能，学习变参数函数的编写和使用方法。
掌握格式化字符串的一些定义和用途。
2、有关格式控制串
格式控制串总是以%开始，后跟一些参数：
%【flags】【宽度】【精度控制】type。
前面三个是可选，而 type 是必须的参数。
（1） flag 用于控制输出的 符号+（+-）、对齐方式-（左对齐）等。

（2） 宽度只控制输出的宽度，03，当字符数小于3个时，左边补0.
（3） 精度控制，主要针对浮点数。比如12.3456 如果%.2则只输出2位，变成12.34。
（4） 类型是必须的。这里主要分析这个。

3、源代码分析

```
int f_printf (
FIL* fil, /* Pointer to the file object */
const char* str, /* Pointer to the format string */
... /* Optional arguments... */
){ va_list arp;
UCHAR c, f, r;
ULONG val;
char s[16];
int i, w, res, cc;
va_start(arp, str); //arp 实际是一个通用类指针，指向 str 后的第一个参数。
for (cc = res = 0; cc != EOF; res += cc) {
c = *str++;
if (c == 0) break; /* End of string */
if (c != '%') { /* Non escape cahracter */
cc = f_putc(c, fil); //如果不是%，表示正常字符，写入文件
if (cc != EOF) cc = 1; //cc 是本次写对应的字符数字。当然如果写入不成功，返回 EOF，则跳出循环
continue; //res 是总共写入的数字。
}
w = f = 0; //从此处开始表示遇到了%后的处理
c = *str++; //取出下一个字符。
if (c == '0') { /* Flag: '0' padding */
f = 1; c = *str++; //如果是字符0，表示要控制宽度，flag 第0位标志置位。取出下一字符
}
while (c >= '0' && c <= '9') { /* Precision */
w = w * 10 + (c - '0');
c = *str++; //依次取出后面的字符，如遇到034，则 w=34。字符输出 }
if (c == 'l') { /* Prefix: Size is long int */
f |= 2; c = *str++; //如果遇到字符'l'，表示长整形，标志第1位置位。 }
if (c == 's') { /* Type is string */
cc = f_puts(va_arg(arp, char*), fil); //如果遇到 ‘s’标志，将 arp 转换为 char*指针，并指向下一个参数。
continue; //cc 是字符串的长度，本次输入文件的字符数。
}
if (c == 'c') { /* Type is character */
cc = f_putc(va_arg(arp, int), fil); //字符以 int 类型出现，要占四个字节。
if (cc != EOF) cc = 1;
continue;
} r = 0; //如果字符'0'和'l'后，不是 s 和 c，则下面继续处理
if (c == 'd') r = 10; /* Type is signed decimal */
if (c == 'u') r = 10; /* Type is unsigned decimal *///十进制
if (c == 'X') r = 16; /* Type is unsigned hexdecimal */
if (r == 0) break; /* Unknown type */
if (f & 2) { /* Get the value */
val = (ULONG)va_arg(arp, long);
} else {
val = (c == 'd') ? (ULONG)(long)va_arg(arp, int) : (ULONG)va_arg(arp, unsigned int);
} //如果设置了'l'，标志，则将数据转换为长整形。从参数中取出来。
```

```
/* Put numeral string */ //下面是将数字转换为字符串。
if (c == 'd') {
if (val & 0x80000000) {
val = 0 - val;
f |= 4; //标志第二位表明这是个负数。
} }
i = sizeof(s) - 1; s[i] = 0; //字符串最后一为以 '\0'结束。i 指向尾端，往前扩展。
do { //同时处理十进制和十六进制。
c = (UCHAR)(val % r + '0'); //十进制时，1234/10,余数4+‘0’，变成字符
if (c > '9') c += 7; //十六进制，3A 表示大写 A，但实际的字符 A 是0x41，中间差7.
s[--i] = c; //前一位存储该字符。
val /= r; //val 变成123，也就是把尾端 一位去掉。
} while (i && val); //最多存储15位，要不然溢出。
if (i && (f & 4)) s[--i] = '-'; //如果为负数，还要加上符号标志。
w = sizeof(s) - 1 - w; //要求的宽度计算。8位宽，则填充从第7位开始。7-14，正好8位。
while (i && i > w) s[--i] = (f & 1) ? '0' : ' ';//如果0填充置位填充0，否则填充空格。i 退到0，或者 i 推导 W 都结束填充。
cc = f_puts(&s[i], fil);
} va_end(arp);
return (cc == EOF) ? cc : res;
}
```

把代码看了一遍，基本还是能看懂。

4、移植。

（1）添加命令 fstring，该命令一个参数，是用户字符串。以附加的写入到指定的文件 fstring.txt。

（2）实习方法

以写文件的方式，打开文件，然后指针移动到文件末尾。调用代码 f_printf(&File，”输入字符串%s/r/n”,（const char*）argv[1])来写入文件。

（3）代码实现

```
res=f_open( &FileStr, "0:/doc/fstring.txt",FA_WRITE ); //
res=f_lseek( &FileStr, FileStr.fsize );
f_printf (&FileStr,"用户输入:%s\r\n",(char*)argv[1] );
f_close (&FileStr );
```

功能比较简单，测试通过。

六、支持长文件名（一）。

1、代码页的功能

（1）在原来的程序中，我把 #define _CODE_PAGE 936，代码页定义为936.

这时候可以在磁盘上新建中文名称的文件夹和文件，（通过 fwrite 命令实现，只要数目小于4个就行，也可以用 fread 命令读出来），也可以显示中文文件名。（通过命令 flist 来实现）。

（2）我把 #define _CODE_PAGE 1，代码页定义为1后，调用 flist 能正常显示中文名。但是调用 fread 和 fwrite 命令不能读取和创建中文名文件了。

进入代码分析，显示时关键在函数 get_fileinfo（），它原封不动的将系统标准名 转换为一般字符串，对字符串不做任何处理，送到串口以后自然可以正常显示。

但是读取文件和写入文件时，首先调用 f_open（）函数，进入 follow_path（）函数，然后调用 create_name（），当它遇到大于 0x80 的字符时，产生如下效果：

```
if (c >= 0x80) { /* Extended char */
#ifdef _EXCVT
c = cvt[c - 0x80]; /* Convert extend char (SBCS) */
#else
b |= 3; /* Eliminate NT flag if ext char is exist */
#if !_DF1S /* ASCII only cfg */
```

return FR_INVALID_NAME;
#endif
#endif
}
返回 FR_INVALID_NAME，所以它不识别中文 GB2312（OEM）代码。
如果定义了 define _CODE_PAGE==936，则同时就会定义 DF1S，也同时定义了 IsDBCS1(c)和 IsDBCS2(d)。因此可以将中文 OEM 代码转换为标准文件名，存储于文件系统的目录项里面。
2、如果定义长文件名_USE_LFN，而不定义 UNICODE 码。有什么效果呢？
根据要求，先要添加 ff_convert() and ff_wtoupper()两个函数。
结果发现，光两个转换表就要150K 左右，我这个128k 的空间是远远不够了。只能看代码而不能测试了。不定义 UNICODE 码的含义是用户提供的路径名不是以 UNICODE 形式出现的。
以下，就只能纸上谈兵了。
3、如果0:/doc 目录下有一个文件名 （这是一个 very long 的文件.它里面是空的.txt 文档）。
如果要显示这个目录下所有的文件信息，我会在命令界面上输入：
Flist 0:/doc。
执行这个命令，程序首先调用
F_opendir（DIR dj，const void *path）函数，对输入的路径进行分析。这个函数首先调用函数 chk_mounted（）检查磁盘上是否有文件系统，如果已经取得了信息，则很快返回。如果没有，则初始化磁盘，并读取信息，填充文件系统信息结构体 FATFS。它去掉了路径前面的 0:。
F_opendir 接下来调用 follow_path(dj, path)，这个函数根据路径填充目录信息结构体。这个函数首先设置 dj->sclust，如果前面有/，会去掉这个符号。接下来一直读取目录名直到下一个/。这个工作调用 create_name(dj, &path)来完成。
进入 create_name(dj, &path)，首先做的工作是根据路径填充 dj->lfn 指向的内存单元，由于我的这个目录是短文件名：doc，所以填入缓冲区的是 ‘d0‘，’o0‘，’c0‘，由于目录到此结束，所以 NS_LAST 属性被设置，表明已经到路径末尾。同时 lfn[di] = 0，di 此时指向 c0后面那个内存单元。注意，存储进长目录区是以 unicode 的形式，而 path 用的是 ANSi 码，汉字两个字节，英文字母一个字节。
create_name(dj, &path)继续执行，mem_set(dj->fn, ' ', 11)，先将 dj 结构体的标准短文件名填充空格。
if (si) cf |= NS_LOSS | NS_LFN 通过这句话的判断，长文件名前面没有空格和.，没有超出8.3格式。当然后面还要继续判断。执行 while (di && lfn[di - 1] != '.') di—这句后，di=0，表示没有找到扩展名(doc 没有扩展名)。dj->fn[i++] = (BYTE)w，通过这个方式将 doc 存进了短文件名存储区。dj->fn[NS] = cf，短文件名存储区12个字节，前11个是标准短文件名，最后一个是路径属性字节：是否.目录、大小写标志，超出8.3格式，是否长文件名、是否路径结束标志等等。
所以 create_name(dj, &path)的工作主要是根据路径填充 dj->lfn 和 dj->fn，前者每个字符16Byte，以‘\0’结尾，后者12个字节，以路径的整体属性结尾。
执行路径返回 follow_path(dj, path)，接下来调用 dir_find(dj)，这个函数的作用是在当前目录（对应 dj->sclust）下，搜索 dj->lfn 和 dj->fn 对应的目录项，找到以后让 dj->index 和 dj->dir 指向该目录项的数据。追踪那个进入该函数。
在函数 dir_find(dj)里，首先 dir_seek(dj, 0)，将索引定位于0，目录第一项。ord = sum = 0xFF 长目录项索引和校验和设为-1。c = dir[DIR_Name]取出目录项的第一个字符进行判断，a = dir[DIR_Attr] & AM_MASK 取出属性进行判断。建立一个循环，不断移动目录缓冲（每次下移32个字节），通过 dir_next(dj, FALSE)来实现。当找到”doc “目录项时，if (!(dj->fn[NS] & NS_LOSS) && !mem_cmp(dir, dj->fn, 11)) break 跳出循环，此时的有效信息就是 dj->sclust 加上 dj->index 和 dj->dir，它指向根目录里的“doc”目录项。
执行路径返回 follow_path(dj, path)，由于 last = *(dj->fn+NS) & NS_LAST，标志被置位，很快跳出路径分解的循环，返回到 F_opendir（DIR dj，const void *path）。通过下面这个计算：
dj->sclust = ((DWORD)LD_WORD(dir+DIR_FstClusHI) << 16) | LD_WORD(dir+DIR_FstClusLO)真正得到了分配给 doc 目录的起始簇的地址。接下来就可以读取 0:/doc 目录下所有的目录项，并显示它包含文件、目录的属性信息了。
回到主程序执行路径，接下来调用：
f_readdir（）这个函数功能是根据 DIR 结构体提供的信息，填充文件属性结构体，包括文件名字符串、

文件大小、文件修改日期等。进入该函数进行追踪。这里就快要找到我的长文件名目录项《这是一个 very long 的文件.它里面是空的.txt 文档》了，它在目录项里是怎样的存在形式呢？
它共有29个字（包括空格和.），共占据3个长目录项和一个短目录项：其格式应该如下。

43	t文档		0000H		FFFFH	0FH	00H	Num	FFFFH		长文件名
	FFFFH FFFFH FFFFH FFFFH					0000	FFFFH		FFFFH		
02	的文件.它					0FH	00H	Num	里		
面是空的.						0000	tx				
01	这是一个v					0FH	00H	Num	e		
ry lo						0000	ng				
这	是	一	~	1	T	X	T	20H	NT	XX	创建时间	短文件名
创建日期	访问日期	起始簇号高位	修改时间		修改日期		起始簇号低位		文件的大小		

f_readdir（）这个函数调用 dir_read（）函数，读取各个目录项的信息，然后调用 get_fileinfo（）函数得到文件的各项信息。下面进入 dir_read（）进行跟踪，看它是怎样对待长目录项的。
进入 dir_read（）函数，首先 ord, sum = 0xFF，读取属性 a = dir[DIR_Attr] & AM_MASK，if (a == AM_LFN)，也就是当遇到长目录索引“43”的时候，sum = dir[LDIR_Chksum];c &= 0xBF; ord = c; dj->lfn_idx = dj->index;取得校验和，取得索引（这里是3），使长目录索引指向“43”目录项的索引号。
如果 pick_lfn(dj->lfn, dir)返回 TRUE，则索引自动 ord=ord-1变为2。进入 pick_lfn(dj->lfn, dir)，它的作用是将当前目录项里的长文件名部分写入 dj->lfn 指向的名字缓冲区。先从偏移26开始存入“t 文档”三个 unicode 字符，然后写入0，返回 TRUE。也就是 ord=2。
继续执行，dir_next(dj, FALSE)，dj->index 指向下一项，a = dir[DIR_Attr] & AM_MASK，重新取得属性。此时 c = dir[DIR_Name]=02，满足(c == ord && sum == dir[LDIR_Chksum] && pick_lfn(dj->lfn, dir))这个表达式，所以索引又被设置为 ord=1。
同上，再次读取 ord=1的那一项，填充长文件名缓冲区，至此 unicode 名称已经完全存入 dj->lfn 指向的缓冲区，并且 ord=0，表明长目录项已经结束。
通过 dir_next（）再次循环，接下来 dj->index 指向对应的短目录项，只要 sum_sfn(dir)计算出来的校验和与前面长目录项里取得的校验和相等，dir_read（）函数的任务就算完成了。它获得的信息包括：dj->lfnidx（长目录项开始索引），dj->lfn（完整的 unicode 长文件名），dj->index 和 dj->dir 指向短目录项。执行路径回到 f_readdir（）。
f_readdir（）函数接下来调用 get_fileinfo(dj, fno)获取目录项的详细信息。追踪该函数的执行。这个函数共获取六个信息：大小、属性、修改时间、日期、短文件名、长文件名字符串（通过读取 dj->lfn 指向的缓冲区，并将 unicode 转换为 OEM 代码）。有了这个文件属性结构体的数据，用户就可以在串口终端显示目录下所有目录项的信息了。