# 一种易于移植和使用的文件系统FatFs Moule

由于微软Windows的广泛应用，在当前的消费类电子产品中，用得最多的还是FAT文件系统，如U盘、MP3、MP4、数码相机等，所以找到一款容易移植和使用、占用硬件资源相对较小而功能又强大的FAT开源文件系统，对于单片机系统设计者来说是很重要的。

## 引 言

随着信息技术的发展，当今社会的信息量越来越大，以往由单片机构成的系统简单地对存储媒介按地址、按字节的读／写已经不满足人们实际应用的需要，于是利用文件系统对存储媒介进行管理成了今后单片机系统的一个发展方向。目前常用的文件系统主要有微软的FATl2、FATl6、FAT32、NTFS，以及Linux系统下的EXT2、EXT3等。由于微软Windows的广泛应用，在当前的消费类电子产品中，用得最多的还是FAT文件系统，如U盘、MP3、MP4、数码相机等，所以找到一款容易移植和使用、占用硬件资源相对较小而功能又强大的FAT开源文件系统，对于单片机系统设计者来说是很重要的。

FatFs Module是一种完全免费开源的FAT文件系统模块，专门为小型的嵌入式系统而设计。它完全用标准C语言编写，所以具有良好的硬件平台独立性，可以移植到8051、PIC、AVR、SH、Z80、H8、ARM等系列单片机上而只需做简单的修改。它支持FATl2、FATl6和FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读／写，并特别对8位单片机和16位单片机做了优化。FatFs Module有个简化版本Tiny—FatFs，它跟完全版FatFs的不同之处主要有两点：

①占用内存更少，只要1 KB RAM；

②1次仅支持1个存储介。

FatFs和Tiny—FatFs的用法一样，仅仅是包含不同的头文件即可,非常方便,本文主要介绍Tiny-FatFs.

## 1 Tiny-FatFs

### 1.1 移植前的准备

FatFs Module 一开始就是为了能在不同的单片机上使用而设计的,所以具有良好的层次结构,如图 1 所示。最顶层是应用层,使用者无需理会 FatFs Module 的内部结构和复杂的 FAT 协议,只需要调用 FatFs Module 提供给用户的一系列应用接口函数,如 f_open,f_read,f_write、f_close 等，就可以像在 PC 上读 / 写文件那样简单。

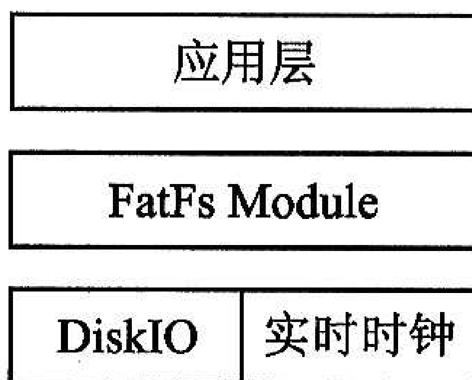


**图 1 FatFs Module 层次结构**

中间层 FatFs Module 实现了 FAT 文件读 / 写协议。FatFs Module 的完全版提供的是 ff．c、ff．h，简化版 Tiny—FatFs 提供的是 tff．c、tff．h。除非有必要，使用者一般不用修改，使用时将需要版本的头文件直接包含进去即可。

需要使用者编写移植代码的是 FatFs Module 提供的底层接口，它包括存储媒介读 / 写接口 DiskIO 和供给文件创建修改时间的实时时钟。

本移植硬件平台使用型号为 ATmegal28 的 AVR 单片机和 SD 卡。ATmegal28 是一种 8 位 RISC 单片机，具有多达 4 KB 的 RAM、128 KB 的内部 Flash 和丰富的外设。软件平台是 WINAVR，具有代码优化能力强和完全免费的优点。

1．2 移植步骤

1．2．1 编写 SPI 和 SD 卡接口代码

本文使用 SD 卡的 SPI 通信模式。SD 卡的 DI 接 MOSI，DO 接 MISO，CS 接 SS。这就需要 ATmegal28 提供 SPI 读／写接口代码，主要包括初始化、读和写。SPI 初始化包括 SPI 相关寄存器的初始化和相关 I／O 口的初始化。将 ATmega 128 的 SPI 配置成主机模式、数据高位先传、时钟速率为二分之一系统时钟等。代码如下：

```
SPCR = (O<<SPIE) | (1<<SPE) | (O<<DORD) | (1<<MSTR) | (O<<CPOL) |
        (O<<CPHA) | (O<<SPR1) |  (O<<SPRO) ；SPSR |= ( 1<< SPI2X )；
```

接着配置 I／O 口的输入／输出。MOSI 脚和 SS 脚配置成输出，MISO 脚配置成输入。然后，就可以进行读／写了。

读 1 个字节的 SPI 接口代码：

```
static BYTE rcvr_spi(void)
    {
          SPDR=OxFF;
           loop_until_bit_is_set(SPSR，SPIF)；
          return SPDR；
    }
```

写 1 个字节的 SPI 接口代码：

```
static void xmit_spi(BYTE dat)
    {
        SPDR=dat；
        loop_until_bit_is_set(SPSR，SPIF)
    }
```

在具备 SPI 读／写接口的基础上编写 SD 卡接口代码，需要编写 3 个基本接口函数：

①向 SD 卡发送 1 条命令：

```
Static BYTE send-cmd(BYTE cmd，DWORD arg)；
```

②向 SD 卡发送 1 个数据包：

Static BOOL xmit—datablock(const BYTE *buff，BYTE token)；

③从 SD 卡接收 1 个数据包：

static BCK]L rcvr-datablock(BYTE*buff，UINT btr)；

1．2．2 编写 DiskIO

编写好存储媒介的接口代码后，就可以编写 DiskIO 了，DiskIO 结构如图 2 所示。

Tiny—FatFs 的移植实际上需要编写 6 个接口函数，分别是：

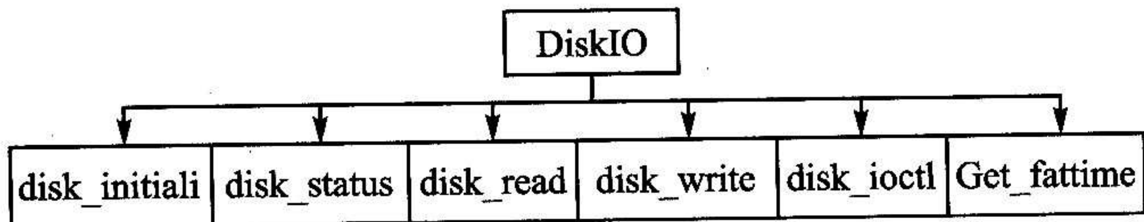


图 2 DiskIO 结构

①DSTATUS disk_initialize(BYTE drv)；

存储媒介初始化函数。由于存储媒介是 SD 卡，所以实际上是对 SD 卡的初始化。drv 是存储媒介号码，由于 Tinv—FatFs 只支持一个存储媒介，所以 drv 应恒为 O。执行无误返回 0，错误返回非 O。

②DSTATUS disk_status(BYTE drV)；

状态检测函数。检测是否支持当前的存储媒介，对 Tinv—FatFs 来说，只要 drv 为 0，就认为支持，然后返回 O。

③DRESULT disk_read(BYTE drv，BYTE*buff，DWORD sector，BYTE．count)；

读扇区函数。在 SD 卡读接口函数的基础上编写，*buff 存储已经读取的数据，sector 是开始读的起始扇区，count 是需要读的扇区数。1 个扇区 512 个字节。执行无误返回 O，错误返回非 0。

④DRESULT disk_write(BYTE drv，const BYTE*buff，DWORD sector，BYTE count)；

写扇区函数。在 SD 卡写接口函数的基础上编写，*buff 存储要写入的数据，sector 是开始写的起始扇区 count 是需要写的扇区数。1 个扇区 512 个字节。执行无误返回 O，错误返回非 0。

⑤DRESULT disk_ioctl(BYTE drv，BYTE ctrl，Voil*buff)；

存储媒介控制函数。ctrl 是控制代码，*buff 存储或接收控制数据。可以在此函数里编写自己需要的功能代码，比如获得存储媒介的大小、检测存储媒介的上电与否存储媒介的扇区数等。如果是简单的应用，也可以不用编写，返回 O 即可。

⑥DWORD get_fattime(Void)；

实时时钟函数。返回一个 32 位无符号整数，时钟信息包含在这 32 位中，如下所示：

bit31：25 年(O．．127)从 1980 年到现在的年数

bit24：21 月(1…12)

bit20：16 日(1．．31)

bitl5．1] 时(O．．23)

bitl0：5 分(O．．59)

bit4：0 秒／2(0．．29)

如果用不到实时时钟，也可以简单地返回一个数。正确编写完 DiskIO，移植工作也就基本完成了，接下来的工作就是对 Tiny—FatFs 进行配置。

## 2 Tiny—FatFs 的配置

Tiny—FatFs 是一款可配置可裁减的文件系统，使用者可以选择自己需要的功能。Tiny—FatFs 总共有 5 个文件，分别是 tff．c、tff．h、diskio．c、diskio．h 和 integer．h。tff_c 和 integer．h 一般不用改动，前面的移植工作主要更改的是 diskio．c，而配置 Tiny—FatFs 则主要修改 tff．h 和 diskio．h。

在 diskio．h 中，使用者可以根据需要使能 disk—write 或 disk_ioetl。以下代码使能 disk_write 和 disk_ioctl：

#define—R'EADONLY 0

#define—USE_IOCTL 1

在 tff．h 中，使用者可以根据需要对整个文件系统进行全面的配置：

① #define_MCU_ENDIAN。有 1 和 2 两个值可设，默认情况下设 1，以获得较好的系统性能。如果单片机是大端模式或者设为 1 时系统运行不正常，则必须设为 2。

② #define_FS_READONLY。设为 1 时将使能只读操作，程序编译时将文件系统中涉及写的操作全部去掉，以节省空间。

③#define_FS_MINIMIZE。有 0、1、2、3 四个选项可设。设 0 表示可以使用全部 Tiny-FatFs 提供的用户函数；设 1 将禁用 f_stat、f_getfree、f_unlink、f_mkdir、f_chmod 和 f_rename；设 2 将在 1 的基础上禁用 f_opendir 和 f_readdir；设 3 将在 1 和 2 的基础上再禁用 f_lseek。使用者可以根据需要进行裁减，以节省空间。

④ ＃define _FAT32。设 1 时将支持 FAT32。

⑤ ＃define _USE_FSINFO。设 1 时提供 FAT32 的磁盘信息支持。

⑥ ＃define_USE_SJIS。设 1 时支持 Shift－JIS 码，一般设 0。

⑦ ＃define_USE_NTFLAG。设 1 时将对文件名大小写敏感。

## 3 TINY-FatFs 的读/写测试

Tiny-FatFs 的功能很强大，提供了丰富而易于使用的用户接口函数，如图 3 所示。

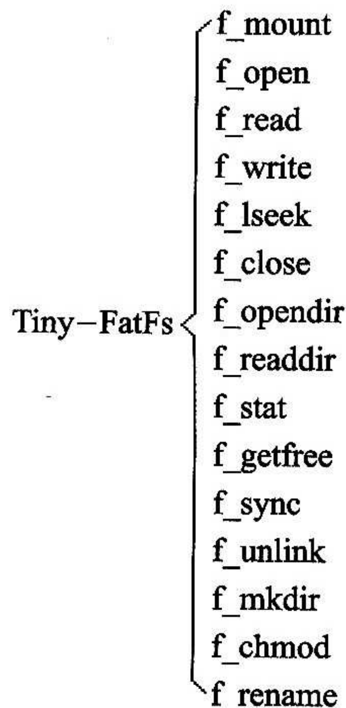


图 3　Tiny－FatFs 用户接口函数

资料来源: http://tech.ddvip.com/2008/10/122475198681677.html　　孟涛整理　2010-9-6

Tiny—FatFs 的功能很全，本文仅测试 f_mount、f_open、f_read、f_write 和 f_close 五个函数来读一个 3.4 MB 的文件和写一个 1MB 的文件，文件名分别为 testl.dat 和 test2.dat。主要代码如下：

```
//初始化：
BYTE buffer[512];
FATFS fs;
FIL fl;
unsigned int r,w,i;
f_mount(0, &fs);
//读 test1.dat：
f_ open ( &fl," test1. dat", FA_
OPEN_EXISTING | FA_READ);
while(1) {
   f_read(&fl,buffer,512,&r);
   if (r  == 0) break;
}
f_close(&fl);
//写 test2.dat：
f_open(&fl,"test2.dat",FA_CREATE_ALWAYS | FA_
WRITE);
for (i=0;i<2048;i++) {
   res = f_write(&fl, buffer,512, &w);
   if (w<512) break;
}
f_close(&fl);
```

经过实际测试，在单片机系统时钟为 11．059 2 MHz 下读一个 3．4 MB 文件耗时约 20 s，平均约 170 KB / s；写一个 1 MB 文件耗时约 6s，平均约 166 KB / s，在资源有限的单片机系统下这个读 / 写速度是相当令人满意的。综上所述，FatFs Module 具有容易移植、功能强大和易于使用的优点，适用于小型嵌入式系统；又是完全的免费和开源，也可以用于教育科研及其商业用途。