Application Note

Linux 下的文件操作应用

基于 EasyARM-iMX 系列开发套件

AN01010101 V1.00 Date: 2014/08/08 产品应用笔记

类别	内容
关键词	嵌入式 Linux、EasyARM-iMX257/283/287、数据存储
摘　要	本文主要介绍在嵌入式 Linux 下的文件操作方法

修订历史

版本	日期	原因
V0.90	2014/06/20	创建文档
V1.00	2014/08/08	调整文档结构及语言组织

目 录

1. 适用范围

本文主要介绍在嵌入式 Linux 下的文件操作方法，其操作方法适用于在 EasyARM-iMX257/283/287 等开发套件平台基础上，将数据存储至 NAND Flash、U 盘或 TF 卡对应目录。

本文所介绍的技术属于通用技术，其原理应用也适用于其他嵌入式 Linux 操作系统。

2. 技术实现

本节介绍 Linux 下对文件的读、写、创建、删除、移动等操作。

2.1 文件存储与文件系统

进入 Linux 文件系统后，通过“ls”命令可以查看所有目录：

```
root@EasyARM-iMX283 ~# cd /
root@EasyARM-iMX283 /# ls
Settings  dev   home  media  opt   root  sys   usr
bin       etc   lib   mnt    proc  sbin  tmp   var
```

根据命令返回的结果可以发现，根目录“/”下含有很多个子目录，再执行 mount 命令则可以查看文件系统类型。命令执行后可以发现，某些目录被挂载为某种文件系统的格式，并指定了大小和读写属性。mount 命令及返回结果如下：

```
root@EasyARM-iMX283 /# mount
rootfs on / type rootfs (rw)
ubi0:rootfs on / type ubifs (rw,relatime)
proc on /proc type proc (rw,relatime)
sys on /sys type sysfs (rw,relatime)
tmpfs on /dev type tmpfs (rw,relatime,mode=755)
devpts on /dev/pts type devpts (rw,relatime,mode=600)
shm on /dev/shm type tmpfs (rw,relatime)
rwfs on /mnt/rwfs type tmpfs (rw,relatime,size=512k)
rwfs on /var type tmpfs (rw,relatime,size=512k)
usbfs on /proc/bus/usb type usbfs (rw,relatime)
tmpfs on /tmp type tmpfs (rw,relatime,size=16384k)
/dev/mmcblk0p1 on /media/sd-mmcblk0p1 type vfat (rw,relatime,fmask=0022,dmask=0022,codepage=cp437,
iocharset=ascii,shortname=mixed,errors=remount-ro)
/dev/sda4 on /media/usb-sda4 type vfat (rw,relatime,fmask=0022,dmask=0022,codepage=cp437,
iocharset=ascii,shortname=mixed,errors=remount-ro)
```

其中，type 即为文件系统类型，其含义分别如表 2.1 所示：

表 2.1 文件系统类型含义

类型	属性描述
rootfs	可读写、存储的文件系统，对于 EasyARM-iMX257/283/287 等开发套件，其板载 NAND 的 rootfs 分区即为此格式，由于 rootfs 分区使用的是 ubifs 文件系统，因此两种格式为同一个意思，此格式的目录可用于存放掉电保存的数据
ubifs	
tmpfs	临时文件系统，所有信息都存在于内存中，速度快，但掉电即丢失
proc	此四类文件系统属于系统信息、系统设备所使用的目录，它们属于系统的某种功能，Linux 为了简化设计，将“功能”变成了“文件”，使用文件的形式来与程序交互，当多个功能放在一个目录下时，便形成了这些“文件夹”
sysfs	
devpts	
usbfs	
vfat	可读写、存储的文件系统，若接入系统的 U 盘或 TF 卡的文件系统为 FAT 或 FAT32 的文件系统将被识别为 vfat 类型

因此，如果拷贝一个大于 512k 的文件到/var 目录下，是放不下的，并且掉电会丢失。在前面的示例命令中，接入系统的 U 盘和 TF 卡，被分别挂载在“/media/usb-sda4”目录和“/media/sd-mmcblk0p1”目录下，如果需要将数据保存到 U 盘或 TF 卡，则可以存放到其对应的目录下即可。

mount 命令中未提到的其他目录，都是属于“/”目录下的子目录，属于同一个文件系统，如“/root/”、“/home/”等目录，若需要将数据保存至 NAND，则可以将数据存在这些路径下。

需要保存的数据在 Linux 系统中通常都是以文件的形式存在，于是对数据的操作就转换为对文件的操作。在 Linux 系统里主要有两种文件操作方式，即：系统调用和 ANSI C 文件操作，下文将分别介绍这两种文件操作方式。

2.2 Linux系统调用

系统调用常用于 I/O 文件操作，系统调用常用的函数有 open、close、read、write、lseek、ulink 等，这些函数的作用介绍如表 2.2 所示。

表 2.2 Linux 文件操作系统调用函数

函数	作用
open	打开或创建文件
close	关闭文件
read	从指定的文件描述符中读出的数据放到缓冲区中，并返回实际读出的字节数
write	把指定缓冲区的数据写入指定的文件描述符中，并返回实际写入的字节数
lseek	在指定的文件描述符中将文件指针定位到相应的位置
ulink	删除文件

2.2.1 open函数

需要的头文件：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

函数原型：

```
int open(const char* pathname, int flags);
int open(const char* pathname, int flags, int perms);
```

参数说明：

- pathname：被打开的文件名(包括路径名)；
- flags：文件打开方式标志，其取值如下：
  - O_RDONLY：表示“以只读方式打开”；
  - O_WRONLY：表示“以只写方式打开”；
  - O_RDWR：表示“以读写方式打开”；
  - O_CREAT：表示“若文件不存在则新建文档并打开”；
  - O_TRUNC：表示“若文件存在，将其长度缩短为 0，属性不变”；
  - O_APPEND：表示“打开文件后文件指针指向末尾”；
  - O_EXCL：和 O_CREAT 一起使用，用于在执行“O_CREAT”前判断文件是否存在，若存在将导致返回失败（避免原文件被覆写）。

返回值：

- 成功返回文件描述符 fd；
- 失败返回-1。

说明：文件描述符 fd 是一个非负整数的索引值，指向内核中每个进程打开文件的记录表——即底层操作，在进行调用 close、read、write 以及 lseek 函数前均应先调用 open 函数，以获得操作该文件的文件描述符 fd。

2.2.2 close函数

需要的头文件：

```
#include <unistd.h>
```

函数原型：

```
int close(int fd);
```

参数说明：

- fd：文件描述符。

返回值：

- 0：成功；
- -1：出错。

说明：在文件操作过程中，每一个通过 open 打开的文件，在文件操作完后都需要调用 close 进行关闭，否则这个文件资源将长期被占用，影响其他程序对文件的读写操作。

2.2.3 read函数

需要的头文件：

```
#include <unistd.h>
```

函数原型：

```
ssize_t    read(int fd, void *buf, size_t count);
```

参数说明：

- fd：文件描述符；
- buf：存储内容的内存空间（指定存储读出数据的缓冲区）；
- count：读取的字节数。

返回值：

- >0：成功读取的字节数；
- <0：出错；
- 0：表示遇到文件末尾 EOF。

应用示例：

read 函数的应用示例如程序清单 2.1 所示，对应已编译出来的程序文件为附件中的“e1”。示例程序以只读方式打开系统配置文件“/etc/inittab”，从中读取 100 个字符并打印。

程序清单 2.1　read 函数应用示例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int fd;
    char buff[101];
    int iCount;
    buff[100] = '\0';
    fd = open("/etc/inittab", O_RDONLY);
    iCount = read(fd, buff, 100);
    printf("I'v read %d bytes from /etc/inittab, as bellow:\n", iCount);
    printf("%s\n", buff);
    close(fd);
}
```

2.2.4 write函数

需要的头文件：

```
#include <unistd.h>
```

函数原型：

```
ssize_t    write(int fd, void *buf, size_t count);
```

参数说明：

- fd：文件描述符；
- buf：需要写入内容的内存空间（缓冲区的指针）；
- count：写入的字节数。

返回值：

- >0：成功写入的字节数；
- <0：出错；
- 0：表示遇到文件末尾 EOF。

应用示例：

write 函数应用示例如程序清单 2.2 所示，对应已编译出来的程序文件为附件中的“e2”。示例程序将打开或创建当前目录下的“./test”文件，然后写入字符串“Hello,world!”。

程序清单 2.2　write 函数应用示例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int fd;
```

```
    char *str = "Hello,world!";
    int iCount;
    fd = open("./test", O_WRONLY | O_CREAT);
    iCount = write(fd, str, 12);          /* str is 12 bytes length */
    printf("%d bytes written!\n", iCount);
    close(fd);
}
```

注意：若用户需要将 test 文件存放到指定的存储介质，则可以修改示例程序，将 open 函数的路径设置为对应存储介质的绝对路径。

2.2.5 lseek函数

需要的头文件：

```
#include <unistd.h>
#include <sys/types.h>
```

函数原型：

```
ssize_t lseek(int fd, off_t offset, int whence);
```

函数说明：

每一个已打开的文件都有一个读写位置，当打开文件时通常其读写位置是指向文件开头，若是以追加的方式打开文件（如 O_APPEND），则读写位置会指向文件尾。当 read()或 write()时，读写位置会随之增加，lseek()便是用来控制该文件的读写位置。

参数说明：

- fd：文件描述符；
- offset：偏移量（可正可负）；
- whence：取值为 SEEK_SET 时表示“文件开头+offset”为新读写位置，取值为 SEEK_CUR 时表示“当前读写位置+offset”为新位置，取值为 SEEK_END 时表示“文件结尾+offset”为新位置。

返回值：

- >0：定位后文件操作位置相对于文件头的偏移量；
- <0：出错，通常返回-1。

常用方式：

将读写位置移到文件开头：

```
lseek(fd, 0, SEEK_SET)
```

将读写位置移到文件尾：

```
lseek(fd, 0, SEEK_END)
```

将取得目前文件位置：

```
lseek(fd, 0, SEEK_CUR)
```

应用示例：

lseek 函数应用示例如程序清单 2.3 所示，对应已编译出来的程序文件为附件中的“e3”。示例程序通过 lseek 函数计算/etc/inittab 文件的长度。

程序清单 2.3　lseek 函数应用示例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int fd;
    int iCount;
    fd = open("/etc/inittab", O_RDONLY);                    // 打开文件获得文件描述符
    iCount = lseek(fd, 0, SEEK_END);                        // 将读写位置移到文件尾
    printf("%d bytes total!\n", iCount);
    close(fd);
}
```

2.2.6 unlink函数

函数原型：

```
int unlink(const char * path)
```

函数说明：删除文件的一个硬链接。

参数说明：

- path：需要删除的包含文件名的文件路径。

返回值：

- 成功返回 0；
- 失败返回-1。

应用示例：

lseek 函数应用示例如程序清单 2.4 所示，对应已编译出来的程序文件为附件中的“e4”。示例程序通过 unlink 函数删除./test 文件。

程序清单 2.4　unlink 函数应用示例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    unlink("./test");
}
```

2.3　ANSI C文件操作

ANSI C 文件操作方法是所有操作系统通用的文件操作方法，它的操作是被缓冲过的，

被修改的文件并不会立即反应到磁盘中，它在内存中开辟一个“缓冲区”，为程序中的每一个文件操作所使用，当执行读文件的操作时，从磁盘文件中将数据先读入内存“缓冲区”，装满后再从内存“缓冲区”依次读入接收的数据。这种文件操作方式又被称作流式文件操作。

ANSI C 文件操作常用的函数有 fopen、fclose、fread、fwrite、fseek、ftell、rewind、fgetc、fputc、fgets、fputs 及 remove 等，这些函数的作用介绍如表 2.3 所示。

表 2.3　ANSI C 文件操作函数

函数	作用
fopen	打开或创建文件
fclose	关闭文件
fread	从文件中读取一个字节
fwrite	将数据成块写入文件流
fseek	移动文件流的读写位置
ftell	查询文件流当前的读写位置
rewind	把文件的读写位置设置在文件头
fgetc	从文本文件中读取一个字符
fputc	向文本文件中写入一个字符
fgets	从文本文件中读取一个字符串(一行数据，以\n 结尾)
fputs	向文本文件中写入一个字符串
remove	删除文件

2.3.1 fopen函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
FILE * fopen(const char* path, const char * mode);
```

参数说明：

- path：带文件路径的文件名；
- mode：文件打开状态，其具体取值及对应的含义如表 2.4 所示。

表 2.4　fopen 函数的 mode 取值表

参数	作用
r	打开只读文件，文件必须存在
r+	打开可读写文件，文件必须存在
w	打开只写文件，若文件存在则清除内容，不存在则新建该文件
w+	打开可读写文件，若文件存在则清除内容，不存在则新建该文件
a	以附加方式打开只写文件，若文件不存在则建立该文件；若存在则写入的数据被加到文件尾
a+	以附加方式打开可读写文件，若文件不存在则建立该文件；若存在则写入的数据被加到文件尾
上面的参数后面可以再加上一个 b，表示打开的二进制文件，而不是纯文本文件，如 rb、w+b、ab+等	

返回值：

- 成功则返回指向该流（流式文件操作）的文件指针（即文件句柄）fp；

- 失败则返回 NULL。

说明：文件句柄 fp 是一个指向 FILE 结构的指针，FILE 是由系统定义的一个结构，该结构中含有文件名、文件状态和文件当前位置等信息。在编写应用程序时通常不必关心 FILE 结构的细节，但调用 fclose、fread、fwrite、fseek、ftell、rewind、fgetc、fputc、fgets 及 fputs 等函数前均需要先调用 fopen 函数打开文件，以获取操作该文件的句柄 fp。

2.3.2 fclose函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int    fclose(FILE * stream);
```

参数说明：

- stream：文件句柄。

返回值：

- 成功返回 0；
- 出错返回 EOF。

说明：与前面介绍的“系统调用”相类似，每一个通过 fopen 打开的文件，在文件操作完后均需要调用 fclose 函数进行关闭，否则这个文件资源将长期被占用，影响其他程序对文件的读写操作。对于流式文件写操作，调用 fclose 还具有重要作用——让内存“缓冲区”中未及时写入存储介质的数据最终写入存储介质。

2.3.3 fread函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

参数说明：

- ptr：指向一块存储空间，用来存放本次读取到的数据；
- size：读取文件一条记录的字节数大小；
- nmemb：本次读取文件记录的数目；
- stream：将要读取的文件流句柄。

返回值：

- 成功则返回实际读取到的数目 nmemb；
- 出错则返回 EOF。

应用示例：

fread 函数应用示例如程序清单 2.5 所示，对应已编译出来的程序文件为附件中的“e5”。示例程序以只读方式打开系统配置文件“/etc/passwd”，并读取打印出前 100 个字符。

程序清单 2.5　fread 函数应用示例

```
#include <stdio.h>
```

```
int main(void)
{
    FILE *fp;
    int iCount;
    char buff[101];
    buff[100] = '\0';
    fp = fopen("/etc/passwd", "r");                    // 打开文件以获得操作该文件的句柄
    iCount = fread(buff, 50, 2, fp);
    printf("%d blocks(50 bytes per block) read, as bellow:\n", iCount);
    printf("%s\n", buff);
    fclose(fp);
}
```

2.3.4 fwrite函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

参数说明：

- ptr: 需写入的数据地址；
- size: 写入文件一条记录的字节数大小；
- nmemb: 写入文件记录的数目；
- stream: 将要写入数据的文件流句柄。

返回值：

- 成功则返回实际写入到的数目 nmemb；
- 出错返回 EOF。

应用示例：

fwrite 函数应用示例代码如程序清单 2.6 所示，对应已编译出来的程序文件为附件中的“e6”。示例程序打开或创建当前目录下的“./test2”文件并写入“Hello,world!”字符串。

程序清单 2.6 fwrite 函数应用示例

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int iCount;
    char *str = "Hello world!";

    fp = fopen("./test2", "w");                        // 打开文件以获得操作该文件的句柄
    iCount = fwrite(str, 12, 1, fp);
    printf("%d block(12 bytes per block) read, check it out!\n", iCount);
    fclose(fp);
```

```
}
```

注意：若用户需要将 test 文件存放到指定的存储介质，则可以修改示例程序，将 fopen 函数的路径设置为对应存储介质的绝对路径。

2.3.5 fseek函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int fseek(FILE *stream, long offset, int whence);
```

参数说明：

- stream：文件句柄；
- offset：偏移量（可正可负）；
- whence：取值为 SEEK_SET 时表示“文件开头+offset”为新读写位置，取值为 SEEK_CUR 时表示“当前读写位置+offset”为新位置，取值为 SEEK_END 时表示“文件结尾+offset”为新位置。

返回值：

- 成功返回 0；
- 出错返回-1。

2.3.6 ftell函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int ftell(FILE *stream);
```

函数说明：简单地返回当前位置。

参数说明：

- stream：文件句柄。

返回值：

- >0：文件流当前操作位置相对于文件头的偏移量；
- <0：出错。

2.3.7 rewind函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
void rewind(FILE *stream);
```

函数说明：把文件指针位置设置为 0，即把文件指针设置到文件的起始位置。

参数说明：

- stream：文件句柄。

返回值：

- 无。

应用示例：

ftell、rewind 函数应用示例代码如

程序清单 2.7 所示，对应已编译出来的程序文件为附件中的“e7”。示例程序打开“/etc/passwd”文件并计算文件长度，然后读取并打印文件所有内容。

程序清单 2.7　ftell、rewind 函数应用示例

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int iCount;
    char buff[1];

    fp = fopen("/etc/passwd", "r");                    // 打开文件以获得操作该文件的句柄
    fseek(fp, 0, SEEK_END);
    iCount = ftell(fp);
    printf("%d bytes total!\n", iCount);
    rewind(fp);
    while(fread(buff, 1, 1, fp) == 1){
        printf("%c", buff[0]);
    }
    fclose(fp);
}
```

2.3.8 fgetc函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int fgetc(FILE *stream);
```

函数说明：从文件中读取一个字符。

参数说明：

- stream：文件句柄。

返回值：

- >0：成功读取字符内容；
- EOF：出错。

2.3.9 fputc函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int fputc(int ch, FILE *stream);
```

函数说明：向文件中写入一个字符。

参数说明：

- ch：需要写入文件的字符内容；
- stream：文件句柄。

返回值：

- >0：成功写入的字符值；
- EOF：出错。

2.3.10 fgets函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
char * fgets(char *s, int n, FILE *stream);
```

函数说明：从文件中读取一行字符串。

参数说明：

- s：读取字符串的存储内存指针；
- n：读取字符串内存的大小；
- stream：文件句柄。

返回值：

- 非空：成功；
- NULL：出错。

应用示例：

fgets 函数的应用示例代码如程序清单 2.8 所示，对应已编译出来的程序文件为附件中的“e8”。示例程序打开“/etc/passwd”文件，依次读取并打印每行内容。

程序清单 2.8 fgets 函数应用示例

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char buff[101];
    buff[100] = '\0';

    fp = fopen("/etc/passwd", "r");                    // 打开文件以获得操作该文件的句柄
    while(fgets(buff, 100, fp) != NULL){
        printf("%s", buff);
    }
    fclose(fp);
}
```

2.3.11 fputs函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int fputs(char *string, FILE *stream);
```

函数说明：向文件中写入一个字符串。

参数说明：

- string：需要写入文件的字符串内容；
- stream：文件句柄。

返回值：

- 0：成功
- <0：出错

2.3.12 remove函数

函数原型：

```
int remove(const char *path);
```

参数说明：

- path：需要删除的包含文件名的文件路径。

返回值：

- 成功返回 0；
- 失败返回-1。

应用示例：

remove 函数应用示例代码如程序清单 2.9 所示，对应已编译出来的程序文件为附件中的“e9”。示例程序使用 remove 函数删除一个文件。

程序清单 2.9 remove 函数应用示例

```
#include <stdio.h>

int main(void)
{
    remove("./test");
}
```

2.3.13 fflush函数

由于缓冲区的存在，因此流中的数据与对应文件的数据可能不一致，当系统掉电时，文件若没有及时 close()，缓冲区的数据就会丢失，为了同步缓冲区，此时可以调用 fflush()函数实现。

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int fflush(FILE *stream);
```

参数说明：

- stream：文件句柄。

返回值：

- 成功返回 0；
- 失败返回 EOF。

应用示例：

fflush 函数应用示例代码如程序清单 2.90 所示，对应已编译出来的程序文件为附件中的“e10”。示例程序用来用 while 死循环模拟掉电状态，打开或创建当前目录下的“./test_2”文件并写入“Hello,world!”字符串。当调用 fflush 函数后，文件成功写入“Hello,world!”字符串，否则将未写入该字符串。

程序清单 2.10 fflush 函数应用示例

```
#include <stdio.h>
#define Sync_Test_On   0

int main(void)
{
    FILE *fp;
    int iCount;
    char *str = "Hello world!";
    fp = fopen("./test_2", "w");                    // 打开文件以获得操作该文件的句柄
    iCount = fwrite(str, 12, 1, fp);
    printf("%d block(12 bytes per block) read, check it out!\n", iCount);
    if (Sync_Test_On) {
        fflush(fp);
        while(1);
    } else {
        while(1);
    }
    fclose(fp);
}
```

说明：Sync_Test_On 为 0 时，程序没有调用 fflush(),运行 e10 时，程序进入 while 循环，用“Ctrl”+“C”强制退出程序（模拟掉电状态），用 cat test_2 查看文件，数据没有写入；Sync_Test_On 为 1 时，程序调用 fflush(),运行 e10，程序进入 while，用“Ctrl”+“C”强制退出程序，然后用 cat test_2 查看文件，数据同步。

2.3.14 setvbuf函数

由于缓冲区的存在，为了同步缓冲区，可以调用 fflush()实现。但是当掉电时，未调用 fflush()时，此时缓冲区的数据仍旧会丢失。而 setvbuf()函数可以修改缓冲区的工作模式，可以很好的解决这一问题。

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int setvbuf(FILE *stream,char *buf,int mode,size_t size);
```

参数说明：

- stream：文件句柄。
- buf：如果 buf 未 NULL，由编译器决定如何建立流的缓冲区，否则其应该指向一段大小为 size 的内存。
- mode：指定了缓冲区的类型，_IOFBF(全缓冲)，_IOLBJ(行缓冲)，_IONBF(无缓冲)。
- size：指定了缓冲区的大小。
  如果指定一个不带缓冲区的流，则忽略 buf 和 size 参数。

返回值：

- 成功返回 0；
- 失败返回非零值。

应用示例：

setvbuf 函数应用示例代码如程序清单 2.91 所示，对应已编译出来的程序文件为附件中的“e11”。示例程序用来用 while 死循环模拟掉电状态，打开或创建当前目录下的“./test_3”文件并写入“Hello,world!”字符串。当调用 setvbuf 函数将缓冲区的类型设置为_IONBF 后，文件可以正常写入“Hello,world!”字符串，否则将未写入该字符串。

程序清单 2.12 setvbuf 函数应用示例

```
#include <stdio.h>
#define Sycn_Test_On 1
int main(void)
{
    FILE *fp;
    int iCount;
    char *str = "Hello world!";
    fp = fopen("./test_3", "w");                    // 打开文件以获得操作该文件的句柄
    if(Sycn_Test_On){
        setvbuf(fp,NULL,_IONBF,0);
    }
    iCount = fwrite(str, 12, 1, fp);
    printf("%d block(12 bytes per block) read, check it out!\n", iCount);
    while(1);
    fclose(fp);
}
```

说明：测试方法同 fflush()一致；setvbuf()函数如果要设置流的缓冲区，则函数必须在打开文件后立即调用，一旦操作了流，就不能再调用此函数了，否则结果不可预知。非缓冲的文件操作访问方式，每次对文件进行一次读写操作时，都需要使用 linux 系统调用来处理此操作，执行一次 linux 系统调用将涉及到 CPU 状态的切换，即从用户空间切换到内核空间，实现进程上下文的切换，这将损耗一定的 CPU 时间，频繁的磁盘访问对程序的执行效率会造成较大的影响。

2.3.15 setbuf函数

需要的头文件：

```
#include <stdio.h>
```

函数原型：

```
int setbuf(FILE *stream,char *buf);
```

参数说明：

- stream：文件句柄。
- buf：参数 buf 须指向一个长度为 BUFSIZ 的缓冲区，如果将 buf 设置为 NULL，则关闭缓冲区。

返回值：

- 成功返回 0；
- 失败返回非零值。

说明：setbuf 用法和 setvbuf 用法类似，不再赘述。

2.4 Linux系统调用和ANSI C文件操作的区别

通过以上内容，我们知道在 Linux 下对文件操作有两种方式：Linux 系统调用和 ANSI C 文件操作。Linux 系统调用实际上就是指最底层的一个调用，在 linux 程序设计里面就是底层调用，面向的是硬件。而 ANSI C 则是库函数调用，是面向的是应用开发的，相当于应用程序的 API（应用程序接口）。

采用这样的方式有很多种原因：第一，双缓冲技术的实现；第二，可移植性的考虑；第三，底层调用本身的一些性能缺陷（如频繁擦写影响文件存储介质的寿命）；第四，让 API 也可以有具体分层和专门的应用方向。

2.4.1 Linux系统调用

Linux 系统调用是操作系统相关的，因此一般没有跨操作系统的可移植性。

Linux 系统调用发生在内核空间，因此如果在用户空间的一般应用程序中使用系统调用来进行文件操作，会有用户空间到内核空间切换的开销。事实上，即使在用户空间使用 ANSI C 文件操作，因为文件总是存在于存储介质上，因此不管是读写操作，都是对硬件(存储器)的操作，都必然会引起 Linux 系统调用。也就是说，ANSI C 文件操作实际上是通过系统调用来实现的。例如 C 库函数 fwrite()就是通过 write()系统调用来实现的。

这样的话，使用库函数也有系统调用的开销，为什么不直接使用系统调用呢？这是因为读写文件通常是大量的数据(这种大量是相对于底层驱动的系统调用所实现的数据操作单位而言)，这时，使用 ANSI C 文件操作就可以大大减少系统调用的次数。这一结果又缘于缓冲区技术。在用户空间和内核空间，对文件操作都使用了缓冲区，例如用 fwrite 写文件，都是先将内容写到用户空间缓冲区，当用户空间缓冲区满或者写操作结束时，才将用户缓冲区的内容写到内核缓冲区，同样的道理，当内核缓冲区满或写结束时才将内核缓冲区内容写到文件对应的硬件媒介。

2.4.2 ANSI C文件操作

ANSI C 文件操作通常用于应用程序中对一般文件的访问。

ANSI C 文件操作是系统无关的，因此可移植性好，由于 ANSI C 文件操作是基于 C 库的，因此也就不可能用于内核空间的驱动程序中对设备的操作。

3. 免责声明

广州周立功单片机科技有限公司所提供的所有服务内容旨在协助客户加速产品的研发进度，在服务过程中所提供的任何程序、文档、测试结果、方案、支持等资料和信息，都仅供参考，客户有权不使用或自行参考修改，本公司不提供任何的完整性、可靠性等保证，若在客户使用过程中因任何原因造成的特别的、偶然的或间接的损失，本公司不承担任何责任。

销售与服务网络

广州周立功单片机科技有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4
邮编：510630
传真：(020)38730925
网址：www.zlgmcu.com
电话：(020)38730916　38730917　38730972　38730976　38730977

广州专卖店

地址：广州市天河区新赛格电子城 203-204 室
电话：(020)87578634　87569917
传真：(020)87578842

南京周立功

地址：南京市珠江路 280 号珠江大厦 1501 室
电话：(025) 68123920　68123923　68123901
传真：(025) 68123900

北京周立功

地址：北京市海淀区知春路 108 号豪景大厦 A 座 19 层
电话：(010)62536178　62536179　82628073
传真：(010)82614433

重庆周立功

地址：重庆市九龙坡区石桥铺科园一路二号大西洋国际大厦（赛格电子市场）2705 室
电话：(023)68796438　68796439
传真：(023)68796439

杭州周立功

地址：杭州市天目山路 217 号江南电子大厦 502 室
电话：(0571)89719480　89719481　89719482
　　89719483　89719484　89719485
传真：(0571)89719494

成都周立功

地址：成都市一环路南二段 1 号数码科技大厦 403 室
电话：(028)85439836　85437446
传真：(028)85437896

深圳周立功

地址：深圳市福田区深南中路 2072 号电子大厦 12 楼 1203
电话：(0755)83781788（5 线）　83782922　83273683
传真：(0755)83793285

武汉周立功

地址：武汉市洪山区广埠屯珞瑜路 158 号 12128 室（华中电脑数码市场）
电话：(027)87168497　87168297　87168397
传真：(027)87163755

上海周立功

地址：上海市北京东路 668 号科技京城东座 12E 室
电话：(021)53083452　53083453　53083496
传真：(021)53083491

西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室
电话：(029)87881296　83063000　87881295
传真：(029)87880865

厦门办事处

E-mail：sales.xiamen@zlgmcu.com

沈阳办事处

E-mail：sales.shenyang@zlgmcu.com