嵌入式应用程序设计综合教程

作者：华清远见

第 1 章 Linux 标准 I/O 编程

本章目标

在应用开发中经常要访问文件。Linux 下读写文件的方式有两大类：标准 I/O 和文件 I/O。其中标准 I/O 是最常用也是最基本的内容，希望读者好好掌握。

本章主要内容：

- Linux 系统调用和用户编程接口（API）；
- Linux 标准 I/O 概述；
- 标准 I/O 操作。

1.1 Linux 系统调用和用户编程接口

1.1.1 系统调用

操作系统负责管理和分配所有的计算机资源。为了更好地服务于应用程序，操作系统提供了一组特殊接口——系统调用。通过这组接口用户程序可以使用操作系统内核提供的各种功能。例如分配内存、创建进程、实现进程之间的通信等。

为什么不允许程序直接访问计算机资源？答案是不安全。单片机开发中，由于不需要操作系统，所以开发人员可以编写代码直接访问硬件。而在 32 位嵌入式系统中通常都要运行操作系统，程序访问资源的方式就发生了改变。操作系统基本上都支持多任务，即同时可以运行多个程序。如果允许程序直接访问系统资源，肯定会带来很多问题。因此，所有软硬件资源的管理和分配都由操作系统负责。程序要获取资源（如分配内存，读写串口）必须通过操作系统来完成，即用户程序向操作系统发出服务请求，操作系统收到请求后执行相关的代码来处理。

用户程序向操作系统提出请求的接口就是系统调用。所有的操作系统都会提供系统调用接口，只不过不同的操作系统提供的系统调用接口各不相同。Linux 系统调用接口非常精简，它继承了 UNIX 系统调用中最基本和最有用的部分。这些系统调用按照功能大致可分为进程控制、进程间通信、文件系统控制、存储管理、网络管理、套接字控制、用户管理等几类。

1.1.2 用户编程接口

前面提到利用系统调用接口程序可以访问各种资源，但在实际开发中程序并不直接使用系统调用接口，而是使用用户编程接口（API）。为什么不直接使用系统调用接口呢？原因如下。

（1）系统调用接口功能非常简单，无法满足程序的需求。

（2）不同操作系统的系统调用接口不兼容，程序移植时工作量大。

用户编程接口通俗的解释就是各种库（最重要的就是 C 库）中的函数。为了提高开发效率，C 库中实现了很多函数。这些函数实现了常用的功能，供程序员调用。这样一来，程序员不需要自己编写这些代码，直接调用库函数就可以实现基本功能，提高了代码的复用率。使用用户编程接口还有一个好处：程序具有良好的可移植行。几乎所有的操作系统上都实现了 C 库，所以程序通常只需要重新编译一下就可以在其他操作系统下运行。

用户编程接口（API）在实现时，通常都要依赖系统调用接口。例如，创建进程的 API 函数 fork()对应于内核空间的 sys_fork()系统调用。很多 API 函数需要通过多个系统调用来完成其功能。还有一些 API 函数不需要调用任何系统调用。

在 Linux 中用户编程接口(API)遵循了在 UNIX 中最流行的应用编程界面标准——POSIX 标准。POSIX 标准是由 IEEE 和 ISO/IEC 共同开发的标准系统。该标准基于当时现有的 UNIX 实践和经验，描述了操作系统的系统调用编程接口（实际上就是 API），用于保证应用程序可以在源代码一级上在多种操作系统上移植运行。这些系统调用编程接口主要是通过 C 库（libc）实现的。

1.2 Linux 标准 I/O 概述

1.2.1 标准 I/O 的由来

标准 I/O 指的是 ANSI C 中定义的用于 I/O 操作的一系列函数。

只要操作系统中安装了 C 库，标准 I/O 函数就可以调用。换句话说，如果程序中使用的是标准 I/O 函数，那么源代码不需要修改就可以在其他操作系统下编译运行，具有更好的可移植性。

除此之外，使用标准 I/O 可以减少系统调用的次数，提高系统效率。标准 I/O 函数在执行时也会用到系统调用。在执行系统调用时，Linux 必须从用户态切换到内核态，处理相应的请求，然后再返回到用户态。如果频繁地执行系统调用会增加系统的开销。为了避免这种情况，标准 I/O 使用时在用户空间创建缓冲区，读写时先操作缓冲区，在合适的时机再通过系统调用访问实际的文件，从而减少了使用系统调用的次数。

1.2.2 流的含义

标准 I/O 的核心对象就是流。当用标准 I/O 打开一个文件时，就会创建一个 FILE 结构体描述该文件（或者理解为创建一个 FILE 结构体和实际打开的文件关联起来）。我们把这个 FILE 结构体形象地称为流。标准 I/O 函数都基于流进行各种操作。

标准 I/O 中的流的缓冲类型有以下三种。

（1）全缓冲：在这种情况下，当填满标准 I/O 缓冲区后才进行实际 I/O 操作。对于存放在磁盘上的普通文件用标准 I/O 打开时默认是全缓冲的。当缓冲区已满或执行 flush 操作时才会进行磁盘操作。

（2）行缓冲：在这种情况下，当在输入和输出中遇到换行符时执行 I/O 操作。标准输入流和标准输出流就是使用行缓冲的典型例子。

（3）无缓冲：不对 I/O 操作进行缓冲，即在对流的读写时会立刻操作实际的文件。标准出错流是不带缓冲的，这就使得出错信息可以立刻显示在终端上，而不管输出的内容是否包　含换行符。

在下面讨论具体函数时，请读者注意区分以上的 3 种不同情况。

1.3 标准 I/O 编程

本节所要讨论的 I/O 操作都是基于流的，它符合 ANSI C 的标准。有一些函数读者已经非常熟悉了（如 printf()、scantf()函数等），因此本节中仅介绍最常用的函数。

1.3.1 流的打开

使用标准 I/O 打开文件的函数有 fopen()、fdopen()和 freopen()。它们可以以不同的模式打开文件，都返回一个指向 FILE 的指针，该指针指向对应的 I/O 流。此后，对文件的读写都是通过这个 FILE 指针来进行。其中 fopen()可以指定打开文件的路径和模式，fdopen()可以指定打开的文件描述符和模式，而 freopen()除可指定打开的文件、模式外，还可指定特定的 I/O 流。

fopen()函数格式如表 1.1 所示。

表 1.1　　fopen()函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * fopen (const char * path, const char * mode) ;
函数参数	path：包含要打开的文件路径及文件名
	mode：文件打开方式，详细信息参考表 1.2
函数返回值	成功：指向 FILE 的指针
	失败：NULL

其中，mode 用于指定打开文件的方式。表 1.2 说明了 fopen()中 mode 的各种取值。

表 1.2　　mode 取值说明

r 或 rb	打开只读文件，该文件必须存在
r＋或 r＋b	打开可读写的文件，该文件必须存在
w 或 wb	打开只写文件，若文件存在则文件长度为 0，即会擦写文件以前的内容；若文件不存在则建立该文件
w+或 w＋b	打开可读写文件，若文件存在则文件长度为 0，即会擦写文件以前的内容；若文件不存在则建立该文件
a 或 ab	以附加的方式打开只写文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留
a+或 a＋b	以附加方式打开可读写的文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留

注意：在每个选项中加入 b 字符用来告诉函数库打开的文件为二进制文件，而非纯文本文件。不过在 Linux 系统中会忽略该符号。

当用户程序运行时，系统自动打开了三个流：标准输入流 stdin、标准输出流 stdout 和标准错误流 stderr。stdin 用来从标准输入设备（默认是键盘）中读取输入内容；stdout 用来向标准输出设备（默认是当前终端）输出内容；stderr 用来向标准错误设备（默认是当前终端）输出错误信息。

1.3.2　流的关闭

关闭流的函数为 fclose()，该函数将流的缓冲区内的数据全部写入文件中，并释放相关资源。fclose()函数格式如表 1.3 所示。

表 1.3　　fclose()函数语法要点

所需头文件	#include <stdio.h>
函数原型	int fclose（FILE * stream）;
函数参数	stream：已打开的流指针
函数返回值	成功：0
	失败：EOF

程序结束时会自动关闭所有打开的流。

1.3.3　错误处理

标准 I/O 函数执行时如果出现错误，会把错误码保存在 errno 中。程序员可以通过相应的函数打印错误信息。

错误处理相关函数 perror 如表 1.4 所示。

表 1.4　　perror()函数语法要点

所需头文件	#include <stdio.h>

函数原型	void perror（const char* s）;
函数参数	s：在标准错误流上输出的信息
函数返回值	无

```
#include <stdio.h>
int main()
{
    FILE *fp;  //  定义流指针
    if  ((fp = fopen ( "1.txt",  "r" ))  == NULL)   //  NULL 是系统定义的宏，其值为 0
    {
        perror ( "fail to fopen" ) ;   //  输出错误信息
        return -1;
    }
    fclose (fp) ;
    return 0;
}
```

如果文件 1.txt 不存在，程序执行时会打印如下信息：

```
fail to fopen: No such file or directory
```

错误处理相关函数 strerror 如表 1.5 所示。

表 1.5　　strerror()函数语法要点

所需头文件	#include <string.h> #include <errno.h>
函数原型	char *strerror（int errnum）;
函数参数	错误码
函数返回值	错误码对应的错误信息

```
#include <stdio.h>
int main()
{
    FILE *fp;
    if  ((fp = fopen ( "1.txt",  "r" ))  == NULL)
    {
        printf ( "fail to fopen: %s\n", strerror (errno)) ;
        return -1;
    }
    fclose (fp) ;
    return 0;
}
```

如果文件 1.txt 不存在，程序执行时会打印如下信息：

```
fail to fopen: No such file or directory
```

1.3.4　流的读写

1. 按字符（字节）输入/输出

字符输入/输出函数一次仅读写一个字符。其中字符输入/输出函数如表 1.6 和表 1.7 所示。

表 1.6　　字符输入函数语法要点

所需头文件	#include <stdio.h>
函数原型	int getc (FILE * stream) ; int fgetc (FILE * stream) ; int getchar (void) ;
函数参数	stream：要输入的文件流
函数返回值	成功：读取的字符
	失败：EOF

getc()和 fgetc ()从指定的流中读取一个字符（节），getchar()从 stdin 中读取一个字符（节）。

表 1.7　　字符输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	int putc (int c, FILE * stream) ; int fputc (int c, FILE * stream) ; int putchar (int c) ;
函数返回值	成功：输出的字符 c
	失败：EOF

putc()和 fputc()向指定的流输出一个字符（节），putchar()向 stdout 输出一个字符（节）。

下面这个实例结合 fputc()和 fgetc()，循环从标准输入读取任意个字符并将其中的数字输出到标准输出。

```
/*fput.c*/
#include <stdio.h>
int main()
{
    int c;
    while ( 1 )
    {
       c = fgetc (stdin) ;  //  从键盘读取一个字符
       if  ((c >= '0')  &&  (c <= '9'))  fputc (c, stdout) ;  //  若输入的是数字，输出
       if  (c == '\n')  break;  //  若遇到换行符，跳出循环
    }
    return 0;
}
```

运行结果如下。

```
$ ./a.out
abc98io#4/wm
984
```

2. 按行输入/输出

行输入/输出函数一次操作一行。其中行输入/输出函数如表 1.8 和表 1.9 所示。

表 1.8 行输入函数语法要点

所需头文件	#include <stdio.h>
函数原型	char * gets (char *s) char * fgets (char * s, int size, FILE * stream)
函数参数	s：存放输入字符串的缓冲区首地址
	size：输入的字符串长度
	stream：对应的流
函数返回值	成功：s
	失败或到达文件末尾：NULL

gets 函数容易造成缓冲区溢出，不推荐大家使用。

fgets 从指定的流中读取一个字符串，当遇到\n 或读取了 size-1 个字符后返回。注意，fgets 不能保证每次都能读出一行。

表 1.9 行输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	int puts (const char *s) int fputs (const char * s, FILE * stream)
函数参数	s：存放输出字符串的缓冲区首地址
	stream：对应的流
函数返回值	成功：s
	失败：NULL

下面以 fgets()为例计算一个文本文件的行数。

```
/*fgets.c*/
#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[])
{
     int line = 0;
     char buf[128];
     FILE *fp;

     if  (argc < 2)
     {
         printf ( " Usage : %s <file>\n ", argv[0]) ;
```

```
        return -1;
    }
    if ((fp = fopen(argv[1], "r")) == NULL)
    {
        perror("fail to fopen");
        return -1;
    }
    while(fgets(buf, 128, fp) != NULL)
    {
        if (buf[strlen(buf)-1] == '\n') line++;
    }
    printf("The line of %s is %d\n", argv[1], line);
    return 0;
}
```

运行该程序，结果如下。

```
$ ./a.out test.txt
The line of test.txt is 64
```

3．以指定大小为单位读写文件

在文件流被打开之后，可对文件流按指定大小为单位进行读写操作。
fread()函数格式如表 1.10 所示。

表 1.10　　fread()函数语法要点

所需头文件	#include <stdio.h>
函数原型	size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream);
函数参数	ptr：存放读入记录的缓冲区
	size：读取的每个记录的大小
	nmemb：读取的记录数
	stream：要读取的文件流
函数返回值	成功：返回实际读取到的 nmemb 数目
	失败：EOF

fwrite()函数格式如表 1.11 所示。

表 1.11　　fwrite()函数语法要点

所需头文件	#include <stdio.h>
函数原型	size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream);
函数参数	ptr：存放写入记录的缓冲区
	size：写入的每个记录的大小
	nmemb：写入的记录数
	stream：要写入的文件流

函数返回值	成功：返回实际写入的 nmemb 数目
	失败：EOF

1.3.5 流的定位

每个打开的流内部都有一个当前读写位置。流在打开时，当前读写位置为 0，表示文件的开始位置。每读写一次后，当前读写位置自动增加实际读写的大小。在读写流之间可先对流进行定位，即移动到指定的位置再操作。

流的定位相关函数如表 1.12 和表 1.13 所示。

表 1.12 fseek 函数语法要点

所需头文件	#include <stdio.h>
函数原型	int fseek (FILE * stream, long offset, int whence) ;
函数参数	stream：要定位的文件流
	offset ：相对于基准值的偏移量
	whence：基准值 SEEK_SET 代表文件起始位置 SEEK_END 代表文件结束位置 SEEK_CUR 代表文件当前读写位置
函数返回值	成功：0
	失败：EOF

表 1.13 ftell()函数语法要点

所需头文件	#include <stdio.h>
函数原型	long ftell (FILE * stream) ;
函数参数	stream：要定位的文件流
函数返回值	成功：返回当前读写位置
	失败：EOF

下面的例子获取一个文件的大小。

```
/*ftell.c*/
#include <stdio.h>
int main (int argc, char *argv[])
{
    FILE *fp;

    if (argc < 2)
    {
        printf ( "Usage : %s <file>\n", argv[0]) ;
```

```
        return -1;
    }
    if ((fp = fopen(argv[1], "r")) == NULL)
    {
        perror("fail to fopen");
        return -1;
    }
    fseek(fp, 0, SEEK_END);
    printf("The size of %s is %ld\n", argv[1], ftell(fp));
    return 0;
}
```

运行该程序，结果如下。

```
$ ./a.out test.txt
The size of test.txt is 305
```

1.3.6 格式化输入输出

格式化输入/输出函数可以指定输入/输出的具体格式，包括读者已经非常熟悉的 printf()、scanf()等函数。以下简要介绍它们的格式，如表 1.14～表 1.15 所示。

表 1.14 格式化输入函数

所需头文件	#include <stdio.h>
函数原型	int scanf(const char *format, …); int fscanf(FILE *fp, const char *format, …); int sscanf(char *buf, const char *format, …);
函数传入值	format：输入的格式
	fp：作为输入的流
	buf：作为输入的缓冲区
函数返回值	成功：输出字符数（sprintf 返回存入数组中的字符数）
	失败：EOF

表 1.15 格式化输出函数

所需头文件	#include <stdio.h>
函数原型	int printf(const char *format,…); int fprintf(FILE *fp, const char *format,…); int sprintf(char *buf, const char *format,…);
函数参值	format：输出的格式
	fp：接收输出的流
	buf：接收输出的缓冲区
函数返回值	成功：输出字符数（sprintf 返回存入数组中的字符数）
	失败：EOF

fprintf 和 sprintf 在应用开发中经常会使用，建议读者查看其帮助信息掌握用法。

1.4 实验内容

1.4.1 文件的复制

1. 实验目的

通过实现文件的复制，掌握流的基本操作。

2. 实验内容

在程序中分别打开源文件和目标文件。循环从源文件中读取内容并写入目标文件。

3. 实验步骤

（1）设计流程。

检查参数 → 打开源文件 → 打开目标文件 → 循环读写文件 → 关闭文件

（2）编写代码。

```
/*mycopy.c*/
#include <stdio.h>
#include <errno.h>
#define  N  64
int main (int argc, char *argv[])
{
     int n;
     char buf[N];
     FILE *fps, *fpd;

     if  (argc < 3)
     {
         printf ( "Usage : %s <src_file> <dst_file>\n", argv[0]) ;
         return -1;
     }

     if  ((fps = fopen (argv[1],  "r" ))  == NULL)
     {
         fprintf (stderr,  "fail to fopen %s : %s\n", argv[1], strerror (errno)) ;
         return -1;
     }

     if  ((fpd = fopen (argv[2],  "w" ))  == NULL)
     {
         fprintf (stderr,  "fail to fopen %s : %s\n", argv[2], strerror (errno)) ;
         fclose (fps) ;
         return -1;
```

```
    }

    while ((n = fread (buf, 1, N, fps)) >= 0)
    {
        fwrite (buf, 1, N, fpd) ;
    }
    fclose (fps) ;
    fclose (fpd) ;
    return 0;
}
```

1.4.2 循环记录系统时间

1. 实验目的

获取系统时间。
在程序中延时。
流的格式化输出。

2. 实验内容

程序中每隔一秒读取一次系统时间并写入文件。

3. 实验步骤

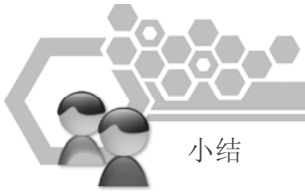
（1）设计流程。

打开文件 → 获取系统时间 → 写入文件 → 延时 1s → 返回第二步（获取系统时间）

（2）编写代码。

```
/*mycopy.c*/
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#define  N  64
int main (int argc, char *argv[])
{
    int n;
    char buf[N];
    FILE *fps;
    time_t t;
    if (argc < 2)
    {
        printf ( "Usage : %s <file>\n", argv[0]) ;
        return -1;
    }
    if ((fp = fopen (argv[1], "w")) == NULL)
    {
        perror ( "fail to fopen" ) ;
        return -1;
```

```
        }
        while ( 1 )
        {
            time (&t) ;  // 获取系统时间
            fprintf (fp,  "%s\n", ctime (&t)) ;  // 将秒数转换成本地时间并写入指定的流
            sleep (1) ;  // 延时 1s
        }
        fclose (fp) ;
        return 0;
    }
```

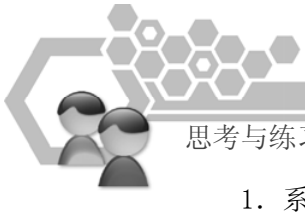


小结

本章首先讲解了系统调用、用户函数接口和系统命令之间的联系和区别。

接下来本章重点介绍了标准 I/O 的相关函数，建议读者以流的概念为出发点理解标准 I/O 的特点，并通过练习掌握标准 I/O 常用函数的用法。

最后，本章安排了两个实验，分别是文件复制和记录系统时间，希望读者认真分析代码。



思考与练习

1. 系统调用和用户编程接口的联系和区别是什么？
2. 标准 I/O 有哪些特点？
3. 分别用字符方式和按行访问方式实现文件的复制。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

全国免费咨询电话：400-706-1880

双休日及节假日请致电值班手机：15010390966

在线咨询：张老师 QQ（619366077），王老师 QQ（2814652411），杨老师 QQ（1462495461）

企业培训洽谈专线：010-82600901

院校合作洽谈专线：010-82600350，在线咨询：QQ（248856300）