

嵌入式与移动开发系列

NITE 国家信息技术紧缺人才培养工程
National Information Technology Education Project
国家信息技术紧缺人才培养工程系列丛书

众多专家、厂商联合推荐 • 业界权威培训机构的经验总结

嵌入式Linux应用程序开发 标准教程 (第2版)

华清远见嵌入式培训中心 编著

提供36小时嵌入式专家讲座视频和教学课件

Embedded Linux Application Development




光盘内容
本书源代码
本书配套PPT
嵌入式专家讲座视频

 **人民邮电出版社**
POSTS & TELECOM PRESS



第 6 章 文件 I/O 编程

本章目标

在搭建起嵌入式开发环境之后，从本章开始，读者将真正开始学习嵌入式 Linux 的应用开发。由于嵌入式 Linux 是经 Linux 裁减而来的，它的系统调用及用户编程接口 API 与 Linux 基本是一致的，因此，在以后的章节中，笔者将首先介绍 Linux 中相关内容的基本编程开发，主要讲解与嵌入式 Linux 中一致的部分，然后再将程序移植到嵌入式的开发板上运行。因此，没有开发板的读者也可以先在 Linux 上开发相关应用程序，这对以后进入嵌入式 Linux 的实际开发是十分有帮助的。本章主要讲解文件 I/O 相关开发，经过本章的学习，读者将会掌握以下内容。

- 掌握 Linux 中系统调用的基本概念
- 掌握 Linux 中用户编程接口（API）及系统命令的相互关系
- 掌握文件描述符的概念
- 掌握 Linux 下文件相关的不带缓存 I/O 函数的使用
- 掌握 Linux 下设备文件读写方法
- 掌握 Linux 中对串口的操作
- 熟悉 Linux 中标准文件 I/O 函数的使用

6.1 Linux 系统调用及用户编程接口（API）

由于本章是讲解 Linux 编程开发的第 1 章，因此希望读者更加明确 Linux 系统调用和用户编程接口（API）的概念。在了解了这些之后，会对 Linux 以及 Linux 的应用编程有更深入的理解。

6.1.1 系统调用

所谓系统调用是指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。例如用户可以通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

在这里，为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在 Linux 中，为了更好地保护内核空间，将程序的运行空间分为内核空间和用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，在逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回用户空间。

Linux 系统调用部分是非常精简的系统调用（只有 250 个左右），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逻辑大致可分为进程控制、进程间通信、文件系统控制、系统控制、存储管理、网络管理、socket 控制、用户管理等几类。

6.1.2 用户编程接口（API）

前面讲到的系统调用并不是直接与程序员进行交互的，它仅仅是一个通过软中断机制向内核提交请求，以获取内核服务的接口。在实际使用中程序员调用的通常是用户编程接口——API，也就是本书后面要讲到的 API 函数。但并不是所有的函数都一一对应一个系统调用，有时，一个 API 函数会需要几个系统调用来共同完成函数的功能，甚至还有一些 API 函数不需要调用相应的系统调用（因此它所完成的不是内核提供的服务）。

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。

6.1.3 系统命令

以上讲解了系统调用、用户编程接口（API）的概念，分析了它们之间的相互关系，那么，读者在第 2 章中学到的那么多的 Shell 系统命令与它们之间又是怎样的关系呢？

系统命令相对 API 更高了一层，它实际上是一个可执行程序，它的内部引用了用户编程接口（API）来实现相应的功能。它们之间的关系如图 6.1 所示。

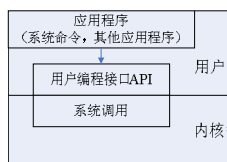


图 6.1 系统调用、API 与系统命令之间的关系

6.2 Linux 中文件及文件描述符概述

在 Linux 中对目录和设备的操作都等同于文件的操作，因此，大大简化了系统对不同设备的处理，提高了效率。Linux 中的文件主要分为 4 种：普通文件、目录文件、链接文件和设备文件。

那么，内核如何区分和引用特定的文件呢？这里用到了一个重要的概念——文件描述符。对于 Linux 而言，所有对设备和文件的操作都是使用文件描述符来进行的。文件描述符是一个非负的整数，它是一个索引值，并指向在内核中每个进程打开文件的记录表。当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。

通常，一个进程启动时，都会打开 3 个文件：标准输入、标准输出和标准出错处理。这 3 个文件分别对应文件描述符为 0、1 和 2（也就是宏替换 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`，鼓励读者使用这些宏替换）。

基于文件描述符的 I/O 操作虽然不能移植到类 Linux 以外的系统上去（如 Windows），但它往往是实现某些 I/O 操作的惟一途径，如 Linux 中低级文件操作函数、多路 I/O、TCP/IP 套接字编程接口等。同时，它们也很好地兼容 POSIX 标准，因此，可以很方便地移植到任何 POSIX 平台上。基于文件描述符的 I/O 操作是 Linux 中最常用的操作之一，希望读者能够很好地掌握。

6.3 底层文件 I/O 操作

本节主要介绍文件 I/O 操作的系统调用，主要用到 5 个函数：`open()`、`read()`、`write()`、`lseek()` 和 `close()`。这些函数的特点是不带缓存，直接对文件（包括设备）进行读写操作。这些函数虽然不是 ANSI C 的组成部分，但是是 POSIX 的组成部分。

6.3.1 基本文件操作

（1）函数说明。

`open()` 函数用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

`close()` 函数用于关闭一个被打开的文件。当一个进程终止时，所有被它打开的文

件都由内核自动关闭，很多程序都使用这一功能而不显示地关闭一个文件。

`read()`函数用于将从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。若返回 0，则表示没有数据可读，即已达到文件尾。读操作从文件的当前指针位置开始。当从终端设备文件中读出数据时，通常一次最多读一行。

`write()`函数用于向打开的文件写数据，写操作从文件的当前指针位置开始。对磁盘文件进行写操作，若磁盘已满或超出该文件的长度，则 `write()`函数返回失败。

`lseek()`函数用于在指定的文件描述符中将文件指针定位到相应的位置。它只能用在可定位（可随机访问）文件操作中。管道、套接字和大部分字符设备文件是不可定位的，所以在这些文件的操作中无法使用 `lseek()`调用。

（2）函数格式。

`open()`函数的语法格式如表 6.1 所示。

表 6.1 `open()` 函数语法要点

所需头文件	#include <sys/types.h> /* 提供类型 pid_t 的定义 */ #include <sys/stat.h> #include <fcntl.h>	
函数原型	int open(const char *pathname, int flags, int perms)	
函数传入值	pathname	被打开的文件名（可包括路径名）
	flag: 文件打开的方式	O_RDONLY: 以只读方式打开文件
		O_WRONLY: 以只写方式打开文件
		O_RDWR: 以读写方式打开文件
		O_CREAT: 如果该文件不存在，就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL: 如果使用 O_CREAT 时文件存在，则可返回错误消息。这一参数可测试文件是否存在。此时 open 是原子操作，防止多个进程同时创建同一个文件
		O_NOCTTY: 使用本参数时，若文件为终端，那么该终端不会成为调用 open()的那个进程的控制终端
		O_TRUNC: 若文件已经存在，那么会删除文件中的全部原有数据，并且设置文件大小为 0。
		O_APPEND: 以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾，即将写入的数据添加到文件的末尾
函数返回值	perms	被打开文件的存取权限 可以用一组宏定义: S_I(R/W/X)(USR/GRP/OTH) 其中 R/W/X 分别表示读/写/执行权限 USR/GRP/OTH 分别表示文件所有者/文件所属组/其他用户 例如, S_IRUSR S_IWUSR 表示设置文件所有者的可读可写属性。八进制表示法中 600 也表示同样的权限
	成功: 返回文件描述符 失败: -1	

在 open()函数中, flag 参数可通过“|”组合构成, 但前 3 个标志常量(O_RDONLY、O_WRONLY 以及 O_RDWR) 不能相互组合。perms 是文件的存取权限, 既可以用宏定义表示法, 也可以用八进制表示法。

close()函数的语法格式表 6.2 所示。

表 6.2 close()函数语法要点

所需头文件	#include <unistd.h>
函数原型	int close(int fd)
函数输入值	fd: 文件描述符
函数返回值	0: 成功 -1: 出错

read()函数的语法格式如表 6.3 所示。

表 6.3 read()函数语法要点

所需头文件	#include <unistd.h>
函数原型	ssize_t read(int fd, void *buf, size_t count)
函数传入值	fd: 文件描述符
	buf: 指定存储器读出数据的缓冲区
	count: 指定读出的字节数
函数返回值	成功: 读到的字节数 0: 已到达文件尾 -1: 出错

在读普通文件时, 若读到要求的字节数之前已到达文件的尾部, 则返回的字节数会小于希望读出的字节数。

write()函数的语法格式如表 6.4 所示。

表 6.4 write()函数语法要点

所需头文件	#include <unistd.h>
函数原型	ssize_t write(int fd, void *buf, size_t count)
函数传入值	fd: 文件描述符
	buf: 指定存储器写入数据的缓冲区
	count: 指定读出的字节数
函数返回值	成功: 已写的字节数 -1: 出错

在写普通文件时, 写操作从文件的当前指针位置开始。

lseek()函数的语法格式如表 6.5 所示。

表 6.5 lseek()函数语法要点

所需头文件	#include <unistd.h> #include <sys/types.h>	
函数原型	off_t lseek(int fd, off_t offset, int whence)	
函数传入值	fd: 文件描述符	
	offset: 偏移量, 每一读写操作所需要移动的距离, 单位是字节, 可正可负 (向前移, 向后移)	
	whence: 当前位置 的基点	SEEK_SET: 当前位置为文件的开头, 新位置为偏移量的大小
		SEEK_CUR: 当前位置为文件指针的位置, 新位置为当前位置加上偏移量
		SEEK_END: 当前位置为文件的结尾, 新位置为文件的大小加上偏移量的大小
函数返回值	成功: 文件的当前位移 -1: 出错	

(3) 函数使用实例。

下面实例中的 `open()` 函数带有 3 个 flag 参数: `O_CREAT`、`O_TRUNC` 和 `O_WRONLY`, 这样就可以对不同的情况指定相应的处理方法。另外, 这里对该文件的权限设置为 `0600`。其源码如下所示:

下面列出文件基本操作的实例, 基本功能是从一个文件 (源文件) 中读取最后 10KB 数据并到另一个文件 (目标文件)。在实例中源文件是以只读方式打开, 目标文件是以只写方式打开 (可以是读写方式)。若目标文件不存在, 可以创建并设置权限的初始值为 `644`, 即文件所有者可读可写, 文件所属组和其他用户只能读。

读者需要留意的地方是改变每次读写的缓存大小 (实例中为 `1KB`) 会怎样影响运行效率。

```
/* copy_file.c */
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

#define BUFFER_SIZE    1024                /* 每次读写缓存大小, 影响运行效率 */
#define SRC_FILE_NAME  "src_file" /* 源文件名 */
#define DEST_FILE_NAME "dest_file" /* 目标文件名 */
#define OFFSE         10240         /* 复制的数据大小 */

int main()
{
    int src_file, dest_file;
```

```

unsigned char buff[BUFFER_SIZE];
int real_read_len;

/* 以只读方式打开源文件 */
src_file = open(SRC_FILE_NAME, O_RDONLY);

/* 以只写方式打开目标文件，若此文件不存在则创建该文件，访问权限值为 644 */
dest_file = open(DEST_FILE_NAME,
                 O_WRONLY|O_CREAT,
S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
if (src_file < 0 || dest_file < 0)
{
    printf("Open file error\n");
    exit(1);
}

/* 将源文件的读写指针移到最后 10KB 的起始位置*/
lseek(src_file, -OFFSET, SEEK_END);

/* 读取源文件的最后 10KB 数据并写到目标文件中，每次读写 1KB */
while ((real_read_len = read(src_file, buff, sizeof(buff))) > 0)
{
    write(dest_file, buff, real_read_len);
}
close(dest_file);
close(src_file);
return 0;
}

$ ./copy_file
$ ls -lh dest_file
-rw-r--r-- 1 david root 10K 14:06 dest_file

```

open()函数返回的文件描述符一定是最小的未用文件描述符。由于一个进程在启动时自动打开了 0、1、2 三个文件描述符，因此，该文件运行结果中返回的文件描述符为 3。读者可以尝试在调用 open()函数之前，加一句 close(0)，则此后在调用 open()函数时返回的文件描述符为 0（若关闭文件描述符 1，则在程序执行时会由于没有标准输出文件而无法输出）。



注意

6.3.2 文件锁


(1) fcntl()函数说明。

前面的这 5 个基本函数实现了文件的打开、读写等基本操作，本小节将讨论的是，在文件已经共享的情况下如何操作，也就是当多个用户共同使用、操作一个文件的情况，这时，Linux 通常采用的方法是给文件上锁，来避免共享的资源产生竞争的状态。

文件锁包括建议性锁和强制锁。建议性锁要求每个上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。在一般情况下，内核和系统都不使用建议性锁。强制锁是由内核执行的锁，当一个文件被上锁进行写入操作的时候，内核将阻止其他任何文件对其进行读写操作。采用强制锁对性能的影响很大，每次读写操作都必须检查是否有锁存在。

在 Linux 中，实现文件上锁的函数有 lockf()和 fcntl()，其中 lockf()用于对文件施加建议性锁，而 fcntl()不仅可以施加建议性锁，还可以施加强制锁。同时，fcntl()还能对文件的某一记录上锁，也就是记录锁。

记录锁又可分为读取锁和写入锁，其中读取锁又称为共享锁，它能够使多个进程都能在文件的同一部分建立读取锁。而写入锁又称为排斥锁，在任何时刻只能有一个进程在文件的某个部分上建立写入锁。当然，在文件的同一部分不能同时建立读取锁和写入锁。

fcntl()是一个非常通用的函数，它可以对已打开的文件描述符进行各种操作，
 **注意** 不仅包括管理文件锁，还包括获得和设置文件描述符和文件描述符标志、文件描述符的复制等很多功能。在本节中，主要介绍建立记录锁的方法。

(2) fcntl()函数格式。

用于建立记录锁的 fcntl()函数格式如表 6.6 所示。

表 6.6 fcntl()函数语法要点

所需头文件	<pre>#include <sys/types.h> #include <unistd.h> #include <fcntl.h></pre>
函数原型	<pre>int fcntl(int fd, int cmd, struct flock *lock)</pre>
函数传入值	fd: 文件描述符

cmd	F_DUPFD: 复制文件描述符	
	F_GETFD: 获得 fd 的 close-on-exec 标志, 若标志未设置, 则文件经过 exec() 函数之后仍保持打开状态	
	F_SETFD: 设置 close-on-exec 标志, 该标志由参数 arg 的 FD_CLOEXEC 位决定	
	F_GETFL: 得到 open 设置的标志	
	F_SETFL: 改变 open 设置的标志	
	F_GETLK: 根据 lock 参数值, 决定是否上文件锁	
	F_SETLK: 设置 lock 参数值的文件锁	
	F_SETLKW: 这是 F_SETLK 的阻塞版本(命令名中的 W 表示等待(wait))。在无法获取锁时, 会进入睡眠状态; 如果可以获取锁或者捕捉到信号则会返回	
lock: 结构为 flock, 设置记录锁的具体状态		
函数返回值	0: 成功 -1: 出错	


这里, lock 的结构如下所示:

```
struct flock
{
    short l_type;
    off_t l_start;
    short l_whence;
    off_t l_len;
    pid_t l_pid;
}
```

lock 结构中每个变量的取值含义如表 6.7 所示。

表 6.7 lock 结构变量取值

l_type	F_RDLCK: 读取锁 (共享锁)
	F_WRLCK: 写入锁 (排斥锁)
	F_UNLCK: 解锁
l_stat	相对位移量 (字节)
l_whence: 相对位移量的起点 (同 lseek 的 whence)	SEEK_SET: 当前位置为文件的开头, 新位置为偏移量的大小
	SEEK_CUR: 当前位置为文件指针的位置, 新位置为当前位置加上偏移量
	SEEK_END: 当前位置为文件的结尾, 新位置为文件的大小加上偏移量的大小
l_len	加锁区域的长度

 小技巧 为加锁整个文件, 通常的方法是将 l_start 设置为 0, l_whence 设置为

SEEK_SET, l_len 设置为 0。

(3) fcntl()使用实例

下面首先给出了使用 fcntl()函数的文件记录锁功能的代码实现。在该代码中，首先给 flock 结构体的对应位赋予相应的值。接着使用两次 fcntl()函数，分别用于判断文件是否可以上锁和给相关文件上锁，这里用到的 cmd 值分别为 F_GETLK 和 F_SETLK（或 F_SETLKW）。

用 F_GETLK 命令判断是否可以进行 flock 结构所描述的锁操作：若可以进行，则 flock 结构的 l_type 会被设置为 F_UNLCK，其他域不变；若不可行，则 l_pid 被设置为拥有文件锁的进程号，其他域不变。

用 F_SETLK 和 F_SETLKW 命令设置 flock 结构所描述的锁操作，后者是前者的阻塞版。

文件记录锁功能的源代码如下所示：

```
/* lock_set.c */
int lock_set(int fd, int type)
{
    struct flock old_lock, lock;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;
    lock.l_type = type;
    lock.l_pid = -1;

    /* 判断文件是否可以上锁 */
    fcntl(fd, F_GETLK, &lock);
    if (lock.l_type != F_UNLCK)
    {
        /* 判断文件不能上锁的原因 */
        if (lock.l_type == F_RDLCK) /* 该文件已有读取锁 */
        {
            printf("Read lock already set by %d\n", lock.l_pid);
        }
        else if (lock.l_type == F_WRLCK) /* 该文件已有写入锁 */
        {
            printf("Write lock already set by %d\n", lock.l_pid);
        }
    }

    /* l_type 可能已被 F_GETLK 修改过 */
}
```

```

lock.l_type = type;

/* 根据不同的 type 值进行阻塞式上锁或解锁 */
if ((fcntl(fd, F_SETLK, &lock)) < 0)
{
    printf("Lock failed:type = %d\n", lock.l_type);
    return 1;
}

switch(lock.l_type)
{
    case F_RDLCK:
    {
        printf("Read lock set by %d\n", getpid());
    }
    break;

    case F_WRLCK:
    {
        printf("Write lock set by %d\n", getpid());
    }
    break;

    case F_UNLCK:
    {
        printf("Release lock by %d\n", getpid());
        return 1;
    }
    break;

    default:
    break;
}/* end of switch */
return 0;
}

```

下面的实例是文件写入锁的测试用例，这里首先创建了一个 hello 文件，之后对其上写入锁，最后释放写入锁，代码如下所示：

```

/* write_lock.c */
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

```



```
#include "lock_set.c"

int main(void)
{
    int fd;

    /* 首先打开文件 */
    fd = open("hello", O_RDWR | O_CREAT, 0644);
    if (fd < 0)
    {
        printf("Open file error\n");
        exit(1);
    }

    /* 给文件上写入锁 */
    lock_set(fd, F_WRLCK);
    getchar();
    /* 给文件解锁 */
    lock_set(fd, F_UNLCK);
    getchar();
    close(fd);
    exit(0);
}
```

为了能够使用多个终端，更好地显示写入锁的作用，本实例主要在 PC 机上测试，读者可将其交叉编译，下载到目标板上运行。下面是在 PC 机上的运行结果。为了使程序有较大的灵活性，笔者采用文件上锁后由用户键入一任意键使程序继续运行。建议读者开启两个终端，并且在两个终端上同时运行该程序，以达到多个进程操作一个文件的效果。在这里，笔者首先运行终端一，请读者注意终端二中的第一句。

终端一：

```
$ ./write_lock
write lock set by 4994
release lock by 4994
```

终端二：

```
$ ./write_lock
write lock already set by 4994
write lock set by 4997
release lock by 4997
```

由此可见，写入锁为互斥锁，同一时刻只能有一个写入锁存在。接下来的程序是文件读取锁的测试用例，原理和上面的程序一样。

```
/* fcntl_read.c */
#include <unistd.h>
```

```

#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include "lock_set.c"

int main(void)
{
    int fd;
    fd = open("hello", O_RDWR | O_CREAT, 0644);
    if(fd < 0)
    {
        printf("Open file error\n");
        exit(1);
    }

    /* 给文件上读取锁 */
    lock_set(fd, F_RDLCK);
    getchar();
    /* 给文件解锁 */
    lock_set(fd, F_UNLCK);
    getchar();
    close(fd);
    exit(0);
}

```

同样开启两个终端，并首先启动终端一上的程序，其运行结果如下所示：
终端一：

```

$ ./read_lock
read lock set by 5009
release lock by 5009

```

终端二：

```

$ ./read_lock
read lock set by 5010
release lock by 5010

```

读者可以将此结果与写入锁的运行结果相比较，可以看出，读取锁为共享锁，当进程 5009 已设置读取锁后，进程 5010 仍然可以设置读取锁。



思考 如果在一个终端上运行设置读取锁的程序，则在另一个终端上运行设置写入锁的程序，会有什么结果呢？

6.3.3 多路复用

(1) 函数说明。

前面的 `fcntl()` 函数解决了文件的共享问题，接下来该处理 I/O 复用的情况了。

总的来说，I/O 处理的模型有 5 种。

- n **阻塞 I/O 模型**：在这种模型下，若所调用的 I/O 函数没有完成相关的功能，则会使进程挂起，直到相关数据到达才会返回。对管道设备、终端设备和网络设备进行读写时经常会出现这种情况。
- n **非阻塞模型**：在这种模型下，当请求的 I/O 操作不能完成时，则不让进程睡眠，而且立即返回。非阻塞 I/O 使用户可以调用不会阻塞的 I/O 操作，如 `open()`、`write()` 和 `read()`。如果该操作不能完成，则会立即返回出错（例如：打不开文件）或者返回 0（例如：在缓冲区中没有数据可以读取或者没有空间可以写入数据）。
- n **I/O 多路转接模型**：在这种模型下，如果请求的 I/O 操作阻塞，且它不是真正阻塞 I/O，而是让其中的一个函数等待，在这期间，I/O 还能进行其他操作。本节要介绍的 `select()` 和 `poll` 函数()就是属于这种模型。
- n **信号驱动 I/O 模型**：在这种模型下，通过安装一个信号处理程序，系统可以自动捕获特定信号的到来，从而启动 I/O。这是由内核通知用户何时可以启动一个 I/O 操作决定的。
- n **异步 I/O 模型**：在这种模型下，当一个描述符已准备好，可以启动 I/O 时，进程会通知内核。现在，并不是所有的系统都支持这种模型。

可以看到，`select()` 和 `poll()` 的 I/O 多路转接模型是处理 I/O 复用的一个高效的方法。它可以具体设置程序中每一个所关心的文件描述符的条件、希望等待的时间等，从 `select()` 和 `poll()` 函数返回时，内核会通知用户已准备好的文件描述符的数量、已准备好的条件等。通过使用 `select()` 和 `poll()` 函数的返回结果，就可以调用相应的 I/O 处理函数。


(2) 函数格式。

`select()` 函数的语法格式如表 6.8 所示。

表 6.8 `select()` 函数语法要点

所需头文件	<code>#include <sys/types.h></code> <code>#include <sys/time.h></code> <code>#include <unistd.h></code>	
函数原型	<code>int select(int numfds, fd_set *readfds, fd_set *writefds,</code> <code>fd_set *exeptfds, struct timeval *timeout)</code>	
函数传入值	<code>numfds</code>	该参数值为需要监视的文件描述符的最大值加 1
	<code>readfds</code>	由 <code>select()</code> 监视的读文件描述符集合
	<code>writefds</code>	由 <code>select()</code> 监视的写文件描述符集合
	<code>exeptfds</code>	由 <code>select()</code> 监视的异常处理文件描述符集合
	<code>timeout</code>	<code>NULL</code> ：永远等待，直到捕捉到信号或文件描述符已准备好为止

		具体值： struct timeval 类型的指针，若等待了 timeout 时间还没有检测到任何文件描述符准备好，就立即返回
		0 ：从不等待，测试所有指定的描述符并立即返回
函数返回值	大于 0 ：成功，返回准备好的文件描述符的数目 0：超时；-1：出错	

 **思考** 请读者考虑一下如何确定被监视的文件描述符的最大值？

可以看到，**select()**函数根据希望进行的文件操作对文件描述符进行了分类处理，这里，对文件描述符的处理主要涉及 4 个宏函数，如表 6.9 所示。

表 6.9 **select()** 文件描述符处理函数

FD_ZERO(fd_set *set)	清除一个文件描述符集
FD_SET(int fd, fd_set *set)	将一个文件描述符加入文件描述符集中
FD_CLR(int fd, fd_set *set)	将一个文件描述符从文件描述符集中清除
FD_ISSET(int fd, fd_set *set)	如果文件描述符 fd 为 fd_set 集中的一个元素，则返回非零值，可以用于调用 select() 之后测试文件描述符集中的文件描述符是否有变化

一般来说，在使用 **select()**函数之前，首先使用 **FD_ZERO()**和 **FD_SET()**来初始化文件描述符集，在使用了 **select()**函数时，可循环使用 **FD_ISSET()**来测试描述符集，在执行完对相关文件描述符的操作之后，使用 **FD_CLR()**来清除描述符集。

另外，**select()**函数中的 **timeout** 是一个 **struct timeval** 类型的指针，该结构体如下所示：

```
struct timeval
{
    long tv_sec; /* 秒 */
    long tv_unsec; /* 微秒 */
}
```

可以看到，这个时间结构体的精确度可以设置到微秒级，这对于大多数的应用而言已经足够了。

poll()函数的语法格式如表 6.10 所示。

表 6.10 **poll()** 函数语法要点

所需头文件	#include <sys/types.h> #include <poll.h>
函数原型	int poll(struct pollfd *fds, int numfds, int timeout)

函数传入值	<p>fds: struct pollfd 结构的指针, 用于描述需要对哪些文件的哪种类型的操作进行监控。</p> <pre>struct pollfd { int fd; /* 需要监听的文件描述符 */ short events; /* 需要监听的事件 */ short revents; /* 已发生的事件 */ }</pre> <p>events 成员描述需要监听哪些类型的事件, 可以用以下几种标志来描述。</p> <p>POLLIN: 文件中有数据可读, 下面实例中使用到了这个标志</p> <p>POLLPRI: 文件中有紧急数据可读</p> <p>POLLOUT: 可以向文件写入数据</p> <p>POLLERR: 文件中出现错误, 只限于输出</p> <p>POLLHUP: 与文件的连接被断开了, 只限于输出</p> <p>POLLNVAL: 文件描述符是不合法的, 即它并没有指向一个成功打开的文件</p>
	<p>numfds: 需要监听的文件个数, 即第一个参数所指向的数组中的元素数目</p>
	<p>timeout: 表示 poll 阻塞的超时时间 (毫秒)。如果该值小于等于 0, 则表示无限等待</p>
函数返回值	<p>成功: 返回大于 0 的值, 表示事件发生的 pollfd 结构的个数</p> <p>0: 超时; -1: 出错</p>

(3) 使用实例。

由于多路复用通常用于 I/O 操作可能会被阻塞的情况, 而对于可能会有阻塞 I/O 的管道、网络编程, 本书到现在为止还没有涉及。这里通过手动创建 (用 `mknod` 命令) 两个管道文件, 重点说明如何使用两个多路复用函数。

在本实例中, 分别用 `select()` 函数和 `poll()` 函数实现同一个功能, 以下功能说明是以 `select()` 函数为例描述的。

本实例中主要通过调用 `select()` 函数来监听 3 个终端的输入 (分别重定向到两个管道文件的虚拟终端以及主程序所运行的虚拟终端), 并分别进行相应的处理。在这里我们建立了一个 `select()` 函数监视的读文件描述符集, 其中包含 3 个文件描述符, 分别为一个标准输入文件描述符和两个管道文件描述符。通过监视主程序的虚拟终端标准输入来实现程序的控制 (例如: 程序结束); 以两个管道作为数据输入, 主程序将从两个管道读取的输入字符串写入到标准输出文件 (屏幕)。

为了充分表现 `select()` 调用的功能, 在运行主程序的时候, 需要打开 3 个虚拟终端: 首先用 `mknod` 命令创建两个管道 `in1` 和 `in2`。接下来, 在两个虚拟终端上分别运行 `cat>in1` 和 `cat>in2`。同时在第三个虚拟终端上运行主程序。在程序运行之后, 如果在两个管道终端上输入字符串, 则可以观察到同样的内容将在主程序的虚拟终端上逐行显示。如果想结束主程序, 只要在主程序的虚拟终端下输入以 'q' 或 'Q' 字符开头的字符串即可。如果三个文件一直在无输入状态中, 则主程序一直处于阻塞状态。为了防止无限期的阻塞, 在 `select` 程序中设置超时值 (本实例中设置为 60s), 当无输入状态持续到超时值时, 主程序主动结束运行并退出。而 `poll` 程序中依然无限等待, 当然 `poll()` 函数也可以设置超时参数。

该程序的流程图如图 6.2 所示。

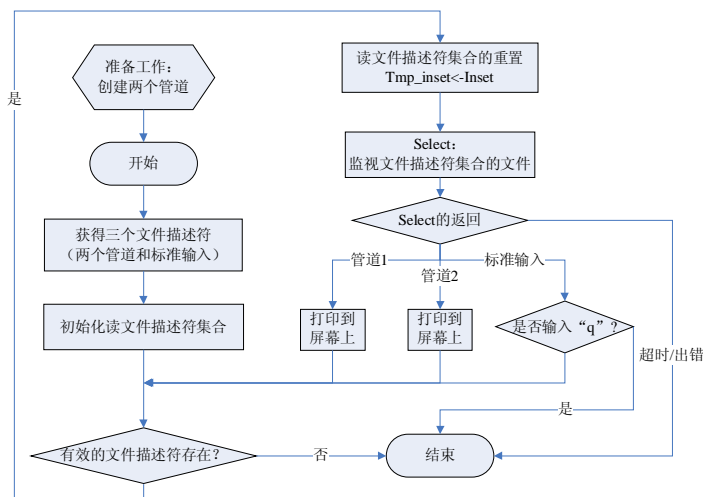


图 6.2 select 实例流程图

使用 select() 函数实现的代码如下所示:

```

/* multiplex_select */
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>

#define MAX_BUFFER_SIZE    1024           /* 缓冲区大小 */
#define IN_FILES           3              /* 多路复用输入文件数目 */
#define TIME_DELAY         60            /* 超时值秒数 */
#define MAX(a, b)          ((a > b) ? (a) : (b))

int main(void)
{
    int fds[IN_FILES];
    char buf[MAX_BUFFER_SIZE];
    int i, res, real_read, maxfd;
    struct timeval tv;
    fd_set inset, tmp_inset;

    /* 首先以只读非阻塞方式打开两个管道文件 */
    fds[0] = 0;

    if((fds[1] = open("in1", O_RDONLY|O_NONBLOCK)) < 0)

```



```

{
    printf("Open in1 error\n");
    return 1;
}

if((fds[2] = open ("in2", O_RDONLY|O_NONBLOCK)) < 0)
{
    printf("Open in2 error\n");
    return 1;
}

/*取出两个文件描述符中的较大者*/
maxfd = MAX(MAX(fds[0], fds[1]), fds[2]);
/*初始化读集合 inset, 并在读集合中加入相应的描述集*/
FD_ZERO(&inset);
for (i = 0; i < IN_FILES; i++)
{
    FD_SET(fds[i], &inset);
}
FD_SET(0, &inset);
tv.tv_sec = TIME_DELAY;
tv.tv_usec = 0;

/*循环测试该文件描述符是否准备就绪, 并调用 select 函数对相关文件描述符做对应操作
*/
while(FD_ISSET(fds[0], &inset)
      || FD_ISSET(fds[1], &inset) || FD_ISSET(fds[2], &inset))
{
    /* 文件描述符集合的备份, 这样可以避免每次进行初始化 */
    tmp_inset = inset;
    res = select(maxfd + 1, &tmp_inset, NULL, NULL, &tv);

    switch(res)
    {
        case -1:
            {
                printf("Select error\n");
                return 1;
            }
        break;
    }
}

```



```

        case 0: /* Timeout */
        {
            printf("Time out\n");
            return 1;
        }
        break;

        default:
        {
            for (i = 0; i < IN_FILES; i++)
            {
                f (FD_ISSET(fds[i], &tmp_inset))
                {
                    memset(buf, 0, MAX_BUFFER_SIZE);
                    real_read      =      read(fds[i],      buf,
MAX_BUFFER_SIZE);

                    if (real_read < 0)
                    {
                        if (errno != EAGAIN)
                        {
                            return 1;
                        }
                    }
                    else if (!real_read)
                    {
                        close(fds[i]);
                        FD_CLR(fds[i], &inset);
                    }
                    else
                    {
                        if (i == 0)
                        {
                            /* 主程序终端控制 */
                            if ((buf[0] == 'q') || (buf[0] == 'Q'))
                            {
                                return 1;
                            }
                        }
                    }
                    else

```



```
        { /* 显示管道输入字符串 */
            buf[real_read] = '\0';
            printf("%s", buf);
        }
    }
} /* end of if */
} /* end of for */
}
break;

} /* end of switch */
} /*end of while */

return 0;
}
```

读者可以将以上程序交叉编译，并下载到开发板上运行。以下是运行结果：

```
$ mknod in1 p
$ mknod in2 p
$ cat > in1
SELECT CALL
TEST PROGRAMME
END
$ cat > in2
select call
test programme
end
$ ./multiplex_select
SELECT CALL
select call
TEST PROGRAMME
test programme
END
end
q /* 在终端上输入 'q' 或 'Q' 则立刻结束程序运行 */
```

程序的超时结束结果如下：

```
$ ./multiplex_select
.....
Time out
```

可以看到，使用 `select()` 可以很好地实现 I/O 多路复用。

但是当使用 `select()` 函数时，存在一系列的问题，例如：内核必须检查多余的文件描述符，每次调用 `select()` 之后必须重置被监听的文件描述符集，而且可监听的文件个数受限制（使用 `FD_SETSIZE` 宏来表示 `fd_set` 结构能够容纳的文件描述符的最大数目）等。实际上，`poll` 机制与 `select` 机制相比效率更高，使用范围更广，下面给出用 `poll()` 函数实现同样功能的代码。

```
/* multiplex_poll.c */
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <poll.h>

#define MAX_BUFFER_SIZE      1024    /* 缓冲区大小 */
#define IN_FILES              3       /* 多路复用输入文件数目 */
#define TIME_DELAY           60      /* 超时时间秒数 */
#define MAX(a, b)             ((a > b)?(a):(b))

int main(void)
{
    struct pollfd fds[IN_FILES];
    char buf[MAX_BUFFER_SIZE];
    int i, res, real_read, maxfd;

    /* 首先按一定的权限打开两个源文件 */
    fds[0].fd = 0;
    if((fds[1].fd = open("in1", O_RDONLY|O_NONBLOCK)) < 0)
    {
        printf("Open in1 error\n");
        return 1;
    }

    if((fds[2].fd = open("in2", O_RDONLY|O_NONBLOCK)) < 0)
    {
        printf("Open in2 error\n");
        return 1;
    }
}
```

```

/*取出两个文件描述符中的较大者*/
for (i = 0; i < IN_FILES; i++)
{
    fds[i].events = POLLIN;
}

/*循环测试该文件描述符是否准备就绪，并调用 select 函数对相关文件描述符做对应
操作*/
while(fds[0].events || fds[1].events || fds[2].events)
{
    if (poll(fds, IN_FILES, 0) < 0)
    {
        printf("Poll error\n");
        return 1;
    }

    for (i = 0; i < IN_FILES; i++)
    {
        if (fds[i].revents)
        {
            memset(buf, 0, MAX_BUFFER_SIZE);
            real_read = read(fds[i].fd, buf, MAX_BUFFER_SIZE);

            if (real_read < 0)
            {
                if (errno != EAGAIN)
                {
                    return 1;
                }
            }
            else if (!real_read)
            {
                close(fds[i].fd);
                fds[i].events = 0;
            }
            else
            {
                if (i == 0)
                {
                    if ((buf[0] == 'q') || (buf[0] == 'Q'))
                    {

```

```

        return 1;
    }
}
else
{
    buf[real_read] = '\0';
    printf("%s", buf);
}
} /* end of if real_read*/
} /* end of if revents */
} /* end of for */
} /*end of while */
exit(0);
}

```

运行结果与 `select` 程序类似。请读者比较使用 `select()` 和 `poll()` 函数实现的代码的运行效率（提示：使用获得时间的函数计算程序运行时间，可以证明 `poll()` 函数的效率更高）。

6.4 嵌入式 Linux 串口应用编程

6.4.1 串口概述

常见的数据通信的基本方式可分为并行通信与串行通信两种。

- 并行通信是指利用多条数据传输线将一个数据的各比特位同时传送。它的特点是传输速度快，适用于传输距离短且传输速度较高的通信。
- 串行通信是指利用一条传输线将数据以比特位为单位顺序传送。特点是通信线路简单，利用简单的线缆就可实现通信，降低成本，适用于传输距离长且传输速度较慢的通信。

串口是计算机一种常用的接口，常用的串口有 RS-232-C 接口。它是于 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通信的标准，它的全称是“数据终端设备（DTE）和数据通信设备（DCE）之间串行二进制数据交换接口技术标准”。该标准规定采用一个 DB25 芯引脚的连接器或 9 芯引脚的连接器，其中 25 芯引脚的连接器如图 6.3 所示。

S3C2410X 内部具有两个独立的 UART 控制器,每个控制器都可以工作在 Interrupt(中断)模式或者 DMA(直接存储访问)模式。同时,每个 UART 均具有 16 字节的 FIFO(先入先出寄存器),支持的最高波特率可达到 230.4Kbps。UART 的操作主要可分为以下几个部

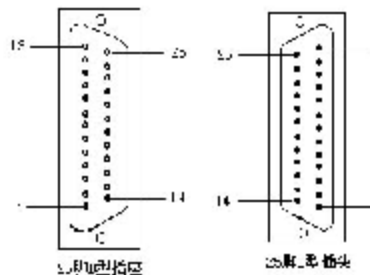


图 6.3 25 引脚串行接口图

分：数据发送、数据接收、产生中断、设置波特率、Loopback 模式、红外模式以及硬流控模式。

串口参数的配置读者在配置超级终端和 minicom 时也已经接触过，一般包括波特率、起始位比特数、数据位比特数、停止位比特数和流控模式。在此，可以将其配置为波特率 115200、起始位 1b、数据位 8b、停止位 1b 和无流控模式。

在 Linux 中，所有的设备文件一般都位于“/dev”下，其中串口 1 和串口 2 对应的设备名依次为“/dev/ttyS0”和“/dev/ttyS1”，而且 USB 转串口的设备名通常为“/dev/ttyUSB0”和“/dev/ttyUSB1”（因版本不同该设备名会有所不同），可以查看在“/dev”下的文件以确认。在本章中已经提到过，在 Linux 下对设备的操作方法与对文件的操作方法是一样的，因此，对串口的读写就可以使用简单的 read()、write() 函数来完成，所不同的只是需要对串口的其他参数另做配置，下面就来详细讲解串口应用开发的步骤。

6.4.2 串口设置详解

串口的设置主要是设置 struct termios 结构体的各成员值，如下所示：

```
#include<termios.h>
struct termios
{
    unsigned short  c_iflag;        /* 输入模式标志 */
    unsigned short  c_oflag;        /* 输出模式标志 */
    unsigned short  c_cflag;        /* 控制模式标志 */
    unsigned short  c_lflag;        /* 本地模式标志 */
    unsigned char   c_line;         /* 线路规程 */
    unsigned char   c_cc[NCC];     /* 控制特性 */
    speed_t         c_ispeed;       /* 输入速度 */
    speed_t         c_ospeed;       /* 输出速度 */
};
```

termios 是在 POSIX 规范中定义的标准接口，表示终端设备（包括虚拟终端、串口等）。口是一种终端设备，一般通过终端编程接口对其进行配置和控制。在具体讲解串口相关编程之前，先了解一下终端相关知识。

终端有 3 种工作模式，分别为规范模式（canonical mode）、非规范模式（non-canonical mode）和原始模式（raw mode）。

通过在 termios 结构的 c_lflag 中设置 ICANNON 标志来定义终端是以规范模式（设置 ICANNON 标志）还是以非规范模式（清除 ICANNON 标志）工作，默认情况为规范模式。

在规范模式下，所有的输入是基于行进行处理。在用户输入一个行结束符（回车符、EOF 等）之前，系统调用 read() 函数读不到用户输入的任何字符。除了 EOF 之外的行结束符（回车符等）与普通字符一样会被 read() 函数读取到缓冲区之中。在规范模式中，行编辑是可行的，而且一次调用 read() 函数最多只能读取一行数据。如果在 read() 函数中被请求读取的数据字节数小于当前行可读取的字节数，则 read() 函数只会读取被请求的字节

数，剩下的字节下次再被读取。

在非规范模式下，所有的输入是即时有效的，不需要用户另外输入行结束符，而且不可进行行编辑。在非规范模式下，对参数 MIN (c_cc[VMIN]) 和 TIME (c_cc[VTIME]) 的设置决定 read()函数的调用方式。设置可以有 4 种不同的情况。

- n MIN = 0 和 TIME = 0: read()函数立即返回。若有可读数据，则读取数据并返回被读取的字节数，否则读取失败并返回 0。
- n MIN > 0 和 TIME = 0: read()函数会被阻塞直到 MIN 个字节数据可被读取。
- n MIN = 0 和 TIME > 0: 只要有数据可读或者经过 TIME 个十分之一秒的时间，read()函数则立即返回，返回值为被读取的字节数。如果超时并且未读到数据，则 read()函数返回 0。
- n MIN > 0 和 TIME > 0: 当有 MIN 个字节可读或者两个输入字符之间的时间间隔超过 TIME 个十分之一秒时，read()函数才返回。因为在输入第一个字符之后系统才会启动定时器，所以在这种情况下，read()函数至少读取一个字节之后才返回。

按照严格意义来讲，原始模式是一种特殊的非规范模式。在原始模式下，所有的输入数据以字节为单位被处理。在这个模式下，终端是不可回显的，而且所有特定的终端输入/输出控制处理不可用。通过调用 cfmakeraw()函数可以将终端设置为原始模式，而且该函数通过以下代码可以得到实现。

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                        | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

下面讲解设置串口的的基本方法。设置串口中最基本的包括波特率设置，校验位和停止位设置。在这个结构中最为重要的是 c_cflag，通过对它的赋值，用户可以设置波特率、字符大小、数据位、停止位、奇偶校验位和硬软流控等。另外 c_iflag 和 c_cc 也是比较常用的标志。在此主要对这 3 个成员进行详细说明。c_cflag 支持的常量名称如表 6.11 所示。其中设置波特率宏名为相应的波特率数值前加上 ‘B’，由于数值较多，本表没有全部列出。

表 6.11 c_cflag 支持的常量名称

CBAUD	波特率的位掩码
B0	0 波特率（放弃 DTR）
...	...
B1800	1800 波特率
B2400	2400 波特率

续表

B4800	4800 波特率
B9600	9600 波特率
B19200	19200 波特率
B38400	38400 波特率
B57600	57600 波特率
B115200	115200 波特率
EXTA	外部时钟率
EXTB	外部时钟率
CSIZE	数据位的位掩码
CS5	5 个数据位
CS6	6 个数据位
CS7	7 个数据位
CS8	8 个数据位
CSTOPB	2 个停止位（不设则是 1 个停止位）
CREAD	接收使能
PARENB	校验位使能
PARODD	使用奇校验而不使用偶校验
HUPCL	最后关闭时挂线（放弃 DTR）
CLOCAL	本地连接（不改变端口所有者）
CRTSCTS	硬件流控

在这里，不能直接对 `c_cflag` 成员初始化，而要将其通过“与”、“或”操作使用其中的某些选项。输入模式标志 `c_iflag` 用于控制端口接收端的字符输入处理。`c_iflag` 支持的常量名称如表 6.12 所示。

表 6.12 `c_iflag` 支持的常量名称

INPCK	奇偶校验使能
IGNPAR	忽略奇偶校验错误
PARMRK	奇偶校验错误掩码
ISTRIP	裁减掉第 8 位比特
IXON	启动输出软件流控
IXOFF	启动输入软件流控
IXANY	输入任意字符可以重新启动输出（默认为输入起始字符才重启输出）
IGNBRK	忽略输入终止条件
BRKINT	当检测到输入终止条件时发送 SIGINT 信号
INLCR	将接收到的 NL（换行符）转换为 CR（回车符）
IGNCR	忽略接收到的 CR（回车符）
ICRNL	将接收到的 CR（回车符）转换为 NL（换行符）
IUCLC	将接收到的大写字符映射为小写字符
IMAXBEL	当输入队列满时响铃

`c_oflag` 用于控制终端端口发送出去的字符处理，`c_oflag` 支持的常量名称如表

6.12 所示。因为现在终端的速度比以前快得多，所以大部分延时掩码几乎没什么用途。

表 6.13 c_oflag 支持的常量名称

OPOST	启用输出处理功能，如果不设置该标志，则其他标志都被忽略
OLCUC	将输出中的大写字符转换成小写字符
ONLCR	将输出中的换行符（‘\n’）转换成回车符（‘\r’）
ONOCR	如果当前列号为 0，则不输出回车符
OCRNL	将输出中的回车符（‘\r’）转换成换行符（‘\n’）
ONLRET	不输出回车符
OFILL	发送填充字符以提供延时
OFDEL	如果设置该标志，则表示填充字符为 DEL 字符，否则为 NUL 字符
NLDLY	换行延时掩码
CRDLY	回车延时掩码
TABDLY	制表符延时掩码
BSDLY	水平退格符延时掩码
VTDLY	垂直退格符延时掩码
FFLDY	换页符延时掩码

c_lflag 用于控制控制终端的本地数据处理和工作模式，c_lflag 所支持的常量名称如表 6.14 所示。

表 6.14 c_lflag 支持的常量名称

ISIG	若收到信号字符（INTR、QUIT 等），则会产生相应的信号
ICANON	启用规范模式
ECHO	启用本地回显功能
ECHOE	若设置 ICANON，则允许退格操作
ECHOK	若设置 ICANON，则 KILL 字符会删除当前行
ECHONL	若设置 ICANON，则允许回显换行符
ECHOCTL	若设置 ECHO，则控制字符（制表符、换行符等）会显示成“^X”，其中 X 的 ASCII 码等于给相应控制字符的 ASCII 码加上 0x40。例如：退格字符（0x08）会显示成“^H”（‘H’的 ASCII 码为 0x48）
ECHOPRT	若设置 ICANON 和 IECHO，则删除字符（退格符等）和被删除的字符都会被显示
ECHOKE	若设置 ICANON，则允许回显在 ECHOE 和 ECHOPRT 中设定的 KILL 字符
NOFLSH	在通常情况下，当接收到 INTR、QUIT 和 SUSP 控制字符时，会清空输入和输出队列。如果设置该标志，则所有的队列不会被清空
TOSTOP	若一个后台进程试图向它的控制终端进行写操作，则系统向该后台进程的进程发送 SIGTTOU 信号。该信号通常终止进程的执行
IEXTEN	启用输入处理功能

c_cc 定义特殊控制特性。c_cc 所支持的常量名称如表 6.13 所示。

表 6.13 c_cc 支持的常量名称

VINTR	中断控制字符，对应键为 CTRL+C
VQUIT	退出操作符，对应键为 CTRL+Z
VERASE	删除操作符，对应键为 Backspace (BS)
VKILL	删除行符，对应键为 CTRL+U
VEOF	文件结尾符，对应键为 CTRL+D
VEOL	附加行结尾符，对应键为 Carriage return (CR)
VEOL2	第二行结尾符，对应键为 Line feed (LF)
VMIN	指定最少读取的字符数
VTIME	指定读取的每个字符之间的超时时间

下面就详细讲解设置串口属性的基本流程。

1. 保存原先串口配置

首先，为了安全起见和以后调试程序方便，可以先保存原先串口的配置，在这里可以使用函数 `tcgetattr(fd, &old_cfg)`。该函数得到 `fd` 指向的终端的配置参数，并将它们保存于 `termios` 结构变量 `old_cfg` 中。该函数还可以测试配置是否正确、该串口是否可用等。若调用成功，函数返回值为 0，若调用失败，函数返回值为 -1，其使用如下所示：

```
if (tcgetattr(fd, &old_cfg) != 0)
{
    perror("tcgetattr");
    return -1;
}
```

2. 激活选项

CLOCAL 和 CREAD 分别用于本地连接和接受使能，因此，首先要通过位掩码的方式激活这两个选项。

```
newtio.c_cflag |= CLOCAL | CREAD;
```

调用 `cfmakeraw()` 函数可以将终端设置为原始模式，在后面的实例中，采用原始模式进行串口数据通信。

```
cfmakeraw(&new_cfg);
```

3. 设置波特率

设置波特率有专门的函数，用户不能直接通过位掩码来操作。设置波特率的主要函数有：`cfsetispeed()`和`cfsetospeed()`。这两个函数的使用很简单，如下所示：

```
cfsetispeed(&new_cfg, B115200);
cfsetospeed(&new_cfg, B115200);
```

一般地，用户需将终端的输入和输出波特率设置成一样的。这几个函数在成功时返回 0，失败时返回-1。

4. 设置字符大小

与设置波特率不同，设置字符大小并没有现成可用的函数，需要用位掩码。一般首先去除数据位中的位掩码，再重新按要求设置。如下所示：

```
new_cfg.c_cflag &= ~CSIZE; /* 用数据位掩码清空数据位设置 */
new_cfg.c_cflag |= CS8;
```

5. 设置奇偶校验位

设置奇偶校验位需要用到 `termios` 中的两个成员：`c_cflag` 和 `c_iflag`。首先要激活 `c_cflag` 中的校验位使能标志 `PARENB` 和是否要进行偶校验，同时还要激活 `c_iflag` 中的对于输入数据的奇偶校验使能 (`INPCK`)。如使能奇校验时，代码如下所示：

```
new_cfg.c_cflag |= (PARODD | PARENB);
new_cfg.c_iflag |= INPCK;
```

而使能偶校验时，代码如下所示：

```
new_cfg.c_cflag |= PARENB;
new_cfg.c_cflag &= ~PARODD; /* 清除偶校验标志，则配置为奇校验 */
new_cfg.c_iflag |= INPCK;
```

6. 设置停止位

设置停止位是通过激活 `c_cflag` 中的 `CSTOPB` 而实现的。若停止位为一个，则清除 `CSTOPB`，若停止位为两个，则激活 `CSTOPB`。以下分别是停止位为一个和两个比特时的代码：

```
new_cfg.c_cflag &= ~CSTOPB; /* 将停止位设置为一个比特 */
new_cfg.c_cflag |= CSTOPB; /* 将停止位设置为两个比特 */
```

7. 设置最少字符和等待时间

在对接收字符和等待时间没有特别要求的情况下，可以将其设置为 0，则在任何情况下 `read()`函数立即返回，如下所示：

```
new_cfg.c_cc[VTIME] = 0;
new_cfg.c_cc[VMIN] = 0;
```

8. 清除串口缓冲

由于串口在重新设置之后，需要对当前的串口设备进行适当的处理，这时就可调用在<termios.h>中声明的 `tcdrain()`、`tcflow()`、`tcflush()`等函数来处理目前串口缓冲中的数据，它们的格式如下所示。

```
int tcdrain(int fd); /* 使程序阻塞，直到输出缓冲区的数据全部发送完毕*/
int tcflow(int fd, int action); /* 用于暂停或重新开始输出 */
int tcflush(int fd, int queue_selector); /* 用于清空输入/输出缓冲区*/
```

在本实例中使用 `tcflush()`函数，对于在缓冲区中的尚未传输的数据，或者收到的但是尚未读取的数据，其处理方法取决于 `queue_selector` 的值，它可能的取值有以下几种。

- n **TCIFLUSH**: 对接收到而未被读取的数据进行清空处理。
- n **TCOFLUSH**: 对尚未传送成功的输出数据进行清空处理。
- n **TCIOFLUSH**: 包括前两种功能，即对尚未处理的输入输出数据进行清空处理。

如在本例中所采用的是第一种方法：

```
tcflush(fd, TCIFLUSH);
```

9. 激活配置

在完成全部串口配置之后，要激活刚才的配置并使配置生效。这里用到的函数是 `tcsetattr()`，它的函数原型是：

```
tcsetattr(int fd, int optional_actions, const struct termios
*termios_p);
```

其中参数 `termios_p` 是 `termios` 类型的新配置变量。

参数 `optional_actions` 可能的取值有以下 3 种：

- n **TCSANOW**: 配置的修改立即生效。
- n **TCSADRAIN**: 配置的修改在所有写入 `fd` 的输出都传输完毕之后生效。
- n **TCSAFLUSH**: 所有已接受但未读入的输入都将在修改生效之前被丢弃。

该函数若调用成功则返回 0，若失败则返回-1，代码如下所示：

```
if ((tcsetattr(fd, TCSANOW, &new_cfg)) != 0)
{
    perror("tcsetattr");
    return -1;
}
```

下面给出了串口配置的完整函数。通常，为了函数的通用性，通常将常用的选项都在函数中列出，这样可以大大方便以后用户的调试使用。该设置函数如下所示：

```
int set_com_config(int fd,int baud_rate,
                   int data_bits, char parity, int stop_bits)
{
    struct termios new_cfg,old_cfg;
    int speed;

    /*保存并测试现有串口参数设置，在这里如果串口号等出错，会有相关的出错信息*/
    if (tcgetattr(fd, &old_cfg) != 0)
    {
        perror("tcgetattr");
        return -1;
    }

    /* 设置字符大小*/
    new_cfg = old_cfg;
    cfmakeraw(&new_cfg); /* 配置为原始模式 */
    new_cfg.c_cflag &= ~CSIZE;

    /*设置波特率*/
    switch (baud_rate)
    {
        case 2400:
        {
            speed = B2400;
        }
        break;
        case 4800:
        {
            speed = B4800;
        }
        break;
        case 9600:
        {
            speed = B9600;
        }
        break;
        case 19200:
        {
            speed = B19200;
        }
    }
}
```



```
}

    break;
    case 38400:
    {
        speed = B38400;
    }
    break;

    default:
    case 115200:
    {
        speed = B115200;
    }
    break;
}

cfsetispeed(&new_cfg, speed);
cfsetospeed(&new_cfg, speed);

/*设置停止位*/
switch (data_bits)
{
    case 7:
    {
        new_cfg.c_cflag |= CS7;
    }
    break;

    default:
    case 8:
    {
        new_cfg.c_cflag |= CS8;
    }
    break;
}

/*设置奇偶校验位*/
switch (parity)
{
    default:
    case 'n':
    case 'N':
```



```

{
    new_cfg.c_cflag &= ~PARENB;
    new_cfg.c_iflag &= ~INPCK;
}
break;

case 'o':
case 'O':
{
    new_cfg.c_cflag |= (PARODD | PARENB);
    new_cfg.c_iflag |= INPCK;
}
break;

case 'e':
case 'E':
{
    new_cfg.c_cflag |= PARENB;
    new_cfg.c_cflag &= ~PARODD;
    new_cfg.c_iflag |= INPCK;
}
break;

case 's': /*as no parity*/
case 'S':
{
    new_cfg.c_cflag &= ~PARENB;
    new_cfg.c_cflag &= ~CSTOPB;
}
break;
}

/*设置停止位*/
switch (stop_bits)
{
    default:
    case 1:
    {
        new_cfg.c_cflag &= ~CSTOPB;
    }
}

```

```

        break;

    case 2:
    {
        new_cfg.c_cflag |= CSTOPB;
    }
}

/*设置等待时间和最小接收字符*/
new_cfg.c_cc[VTIME] = 0;
new_cfg.c_cc[VMIN] = 1;

/*处理未接收字符*/
tcflush(fd, TCIFLUSH);
/*激活新配置*/
if ((tcsetattr(fd, TCSANOW, &new_cfg)) != 0)
{
    perror("tcsetattr");
    return -1;
}
return 0;
}

```

6.4.3 串口使用详解

在配置完串口的相关属性后，就可以对串口进行打开和读写操作了。它所使用的函数和普通文件的读写函数一样，都是 `open()`、`write()` 和 `read()`。它们之间的区别的只是串口是一个终端设备，因此在选择函数的具体参数时会有一些区别。另外，这里会用到一些附加的函数，用于测试终端设备的连接情况等。下面将对其进行具体讲解。

1. 打开串口

打开串口和打开普通文件一样，都是使用 `open()` 函数，如下所示：

```
fd = open( "/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
```

可以看到，这里除了普通的读写参数外，还有两个参数 `O_NOCTTY` 和 `O_NDELAY`。

- n `O_NOCTTY` 标志用于通知 Linux 系统，该参数不会使打开的文件成为这个进程的控制终端。如果没有指定这个标志，那么任何一个输入（诸如键盘中止信号等）都将会影响用户的进程。
- n `O_NDELAY` 标志通知 Linux 系统，这个程序不关心 DCD 信号线所处的状态（端口的另一端是否激活或者停止）。如果用户指定了这个标志，则进程将会

一直处在睡眠状态，直到 DCD 信号线被激活。

接下来可恢复串口的状态为阻塞状态，用于等待串口数据的读入，可用 `fcntl()` 函数实现，如下所示：

```
fcntl(fd, F_SETFL, 0);
```

再接着可以测试打开文件描述符是否连接到一个终端设备，以进一步确认串口是否正确打开，如下所示：

```
isatty(STDIN_FILENO);
```

该函数调用成功则返回 0，若失败则返回-1。

这时，一个串口就已经成功打开了。接下来就可以对这个串口进行读和写操作。下面给出了一个完整的打开串口的函数，同样考虑到了各种不同的情况。程序如下所示：

```
/*打开串口函数*/
int open_port(int com_port)
{
    int fd;
    #if (COM_TYPE == GNR_COM) /* 使用普通串口 */
        char *dev[] = {"/dev/ttyS0", "/dev/ttyS1", "/dev/ttyS2"};
    #else /* 使用 USB 转串口 */
        char *dev[] = {"/dev/ttyUSB0", "/dev/ttyUSB1", "/dev/ttyUSB2"};
    #endif
    if ((com_port < 0) || (com_port > MAX_COM_NUM))
    {
        return -1;
    }
    /* 打开串口 */
    fd = open(dev[com_port - 1], O_RDWR|O_NOCTTY|O_NDELAY);
    if (fd < 0)
    {
        perror("open serial port");
        return(-1);
    }

    /*恢复串口为阻塞状态*/
    if (fcntl(fd, F_SETFL, 0) < 0)
    {
        perror("fcntl F_SETFL\n");
    }
}
```

```
/*测试是否为终端设备*/
if (isatty(STDIN_FILENO) == 0)
{
    perror("standard input is not a terminal device");
}
return fd;
}
```

2. 读写串口

读写串口操作和读写普通文件一样，使用 `read()`和 `write()`函数即可，如下所示：

```
write(fd, buff, strlen(buff));
read(fd, buff, BUFFER_SIZE);
```

下面两个实例给出了串口读和写的两个程序，其中用到前面所讲述的 `open_port()`和 `set_com_config ()`函数。写串口的程序将在宿主机上运行，读串口的程序将在目标板上运行。

写串口的程序如下所示。

```
/* com_writer.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include "uart_api.h"

int main(void)
{
    int fd;
    char buff[BUFFER_SIZE];
    if((fd = open_port(HOST_COM_PORT)) < 0) /* 打开串口 */
    {
        perror("open_port");
        return 1;
    }

    if(set_com_config(fd, 115200, 8, 'N', 1) < 0) /* 配置串口 */
    {
        perror("set_com_config");
    }
}
```

```

        return 1;
    }

    do
    {
        printf("Input some words(enter 'quit' to exit):");
        memset(buff, 0, BUFFER_SIZE);
        if (fgets(buff, BUFFER_SIZE, stdin) == NULL)
        {
            perror("fgets");
            break;
        }
        write(fd, buff, strlen(buff));
    } while(strncmp(buff, "quit", 4));
    close(fd);
    return 0;
}

```

读串口的程序如下所示:

```

/* com_reader.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include "uart_api.h"

int main(void)
{
    int fd;
    char buff[BUFFER_SIZE];

    if((fd = open_port(TARGET_COM_PORT)) < 0) /* 打开串口 */
    {
        perror("open_port");
        return 1;
    }

    if(set_com_config(fd, 115200, 8, 'N', 1) < 0) /* 配置串口 */
    {

```

```

        perror("set_com_config");
        return 1;
    }

    do
    {
        memset(buff, 0, BUFFER_SIZE);
        if (read(fd, buff, BUFFER_SIZE) > 0)
        {
            printf("The received words are : %s", buff);
        }
    } while(strncmp(buff, "quit", 4));
    close(fd);
    return 0;
}

```

在宿主机上运行写串口的程序，而在目标板上运行读串口的程序，运行结果如下所示。

```

/* 宿主机，写串口*/
$ ./com_writer
Input some words(enter 'quit' to exit):hello, Reader!
Input some words(enter 'quit' to exit):I'm Writer!
Input some words(enter 'quit' to exit):This is a serial port testing
program.
Input some words(enter 'quit' to exit):quit
/* 目标板，读串口*/
$ ./com_reader
The received words are : hello, Reader!
The received words are : I'm Writer!
The received words are : This is a serial port testing program.
The received words are : quit

```

另外，读者还可以考虑一下如何使用 `select()` 函数实现串口的非阻塞读写，具体实例会在本章的后面的实验中给出。

6.5 标准 I/O 编程

本章前面几节所述的文件及 I/O 读写都是基于文件描述符的。这些都是基本的 I/O 控制，是不带缓存的。而本节所要讨论的 I/O 操作都是基于流缓冲的，它是符合 ANSI C 的标准 I/O 处理，这里有很多函数读者已经非常熟悉了（如 `printf()`、`scanf()` 函数等），因此本节中仅简要介绍最主要的函数。

前面讲述的系统调用是操作系统直接提供的函数接口。因为运行系统调用时，Linux 必须从用户态切换到内核态，执行相应的请求，然后再返回到用户态，所以应该尽量减少系统调用的次数，从而提高程序的效率。

标准 I/O 提供流缓冲的目的是尽可能减少使用 read()和 write()等系统调用的数量。标准 I/O 提供了 3 种类型的缓冲存储。

- n 全缓冲：在这种情况下，当填满标准 I/O 缓存后才进行实际 I/O 操作。存放在磁盘上的文件通常是由标准 I/O 库实施全缓冲的。在一个流上执行第一次 I/O 操作时，通常调用 malloc()就是使用全缓冲。
- n 行缓冲：在这种情况下，当在输入和输出中遇到行结束符时，标准 I/O 库执行 I/O 操作。这允许我们一次输出一个字符（如 fputc()函数），但只有写了一行之后才进行实际 I/O 操作。标准输入和标准输出就是使用行缓冲的典型例子。
- n 不带缓冲：标准 I/O 库不对字符进行缓冲。如果用标准 I/O 函数写若干字符到不带缓冲的流中，则相当于用系统调用 write()函数将这些字符全写到被打开的文件上。标准出错 stderr 通常是不带缓存的，这就使得出错信息可以尽快显示出来，而不管它们是否含有一个行结束符。

在下面讨论具体函数时，请读者注意区分以上的三种不同情况。

6.5.1 基本操作

1. 打开文件

(1) 函数说明。

打开文件有三个标准函数，分别为：fopen()、fdopen()和 freopen()。它们可以以不同的模式打开，但都返回一个指向 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。

(2) 函数格式定义。

fopen()函数格式如表 6.14 所示。

表 6.14 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)
函数传入值	Path: 包含要打开的文件路径及文件名
	mode: 文件打开状态（后面会具体说明）
函数返回值	成功：指向 FILE 的指针 失败：NULL

其中，mode 类似于 open()函数中的 flag，可以定义打开文件的访问权限等，表 6.15 说明了 fopen()中 mode 的各种取值。

表 6.15 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清为 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清为 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 系统中会自动识别不同类型的文件而将此符号忽略。

fdopen()函数格式如表 6.16 所示。

表 6.16 fdopen()函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * fdopen(int fd, const char * mode)
函数传入值	fd: 要打开的文件描述符
	mode: 文件打开状态（后面会具体说明）
函数返回值	成功: 指向 FILE 的指针 失败: NULL

freopen()函数格式如表 6.17 所示。

表 6.17 freopen()函数语法要点

所需头文件	#include <stdio.h>
函数原型	FILE * freopen(const char *path, const char * mode, FILE * stream)
函数传入值	path: 包含要打开的文件路径及文件名
	mode: 文件打开状态（后面会具体说明）
	stream: 已打开的文件指针
函数返回值	成功: 指向 FILE 的指针 失败: NULL

2. 关闭文件

(1) 函数说明。

关闭标准流文件的函数为 fclose(), 该函数将缓冲区内的数据全部写入到文件中，并释放系统所提供的文件资源。

(2) 函数格式说明。

fclose()函数格式如表 6.18 所示。

表 6.18 fclose()函数语法要点

所需头文件	#include <stdio.h>
函数原型	int fclose(FILE * stream)
函数传入值	stream: 已打开的文件指针

函数返回值	成功：0 失败：EOF
-------	----------------

3. 读文件

- (1) fread()函数说明。
在文件流被打开之后，可对文件流进行读写等操作，其中读操作的函数为 fread()。
- (2) fread()函数格式。
fread()函数格式如表 6.19 所示。

表 6.19 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)
函数传入值	ptr: 存放读入记录的缓冲区
	size: 读取的记录大小
	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

4. 写文件

- (1) fwrite()函数说明。
fwrite()函数用于对指定的文件流进行写操作。
- (2) fwrite()函数格式。
fwrite()函数格式如表 6.20 所示。

表 6.20 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)

函数传入值	ptr: 存放写入记录的缓冲区
	size: 写入的记录大小
	nmemb: 写入的记录数
	stream: 要写入的文件流
函数返回值	成功: 返回实际写入的记录数目 失败: EOF

5. 使用实例

下面实例的功能跟底层 I/O 操作的实例基本相同，运行结果也相同（请参考 6.3.1 节的实例），只是用标准 I/O 库的文件操作来替代原先的底层文件系统调用而已。

读者可以观察哪种方法的效率更高，其原因又是什么。

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

#define BUFFER_SIZE    1024          /* 每次读写缓存大小 */
#define SRC_FILE_NAME  "src_file"    /* 源文件名 */
#define DEST_FILE_NAME "dest_file"   /* 目标文件名 */
#define OFFSET         10240        /* 复制的数据大小 */

int main()
{
    FILE *src_file, *dest_file;
    unsigned char buff[BUFFER_SIZE];
    int real_read_len;

    /* 以只读方式打开源文件 */
    src_file = fopen(SRC_FILE_NAME, "r");

    /* 以写方式打开目标文件，若此文件不存在则创建 */
    dest_file = fopen(DEST_FILE_NAME, "w");

    if (!src_file || !dest_file)
    {
        printf("Open file error\n");
        exit(1);
    }

    /* 将源文件的读写指针移到最后 10KB 的起始位置 */
    fseek(src_file, -OFFSET, SEEK_END);

    /* 读取源文件的最后 10KB 数据并写到目标文件中，每次读写 1KB */
    while ((real_read_len = fread(buff, 1, sizeof(buff), src_file)) >
0)
```

```
{
    fwrite(buff, 1, real_read_len, dest_file);
}
fclose(dest_file);
fclose(src_file);
return 0;
}
```

读者可以尝试用其他文件打开函数进行练习。

6.5.2 其他操作

文件打开之后，根据一次读写文件中字符的数目可分为字符输入输出、行输入输出和格式化输入输出，下面分别对这 3 种不同的方式进行讲解。

1. 字符输入输出

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 6.21 和表 6.22 所示。

表 6.21 字符输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	int getc(FILE * stream) int fgetc(FILE * stream) int getchar(void)
函数传入值	stream: 要输入的文件流
函数返回值	成功: 下一个字符 失败: EOF

表 6.22 字符输入函数语法要点

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: 字符 c 失败: EOF

这几个函数功能类似，其区别仅在于 getc()和 putc()通常被实现为宏，而 fgetc()和 fputc()不能实现为宏，因此，函数的实现时间会有所差别。

下面这个实例结合 fputc()和 fgetc()将标准输入复制到标准输出中去。

```
/*fput.c*/
#include<stdio.h>
main()
{
    int c;
    /*把 fgetc()的结果作为 fputc()的输入*/
    fputc(fgetc(stdin), stdout);
}
```

```
}
```

运行结果如下所示：

```
$ ./fput
w (用户输入)
w (屏幕输出)
```

2. 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 6.23 和表 6.24 所示。

表 6.23 行输出函数语法要点

所需头文件	#include <stdio.h>
函数原型	char * gets(char *s) char fgets(char * s, int size, FILE * stream)
函数传入值	s: 要输入的字符串 size: 输入的字符串长度 stream: 对应的文件流
函数返回值	成功: s 失败: NULL

表 6.24 行输入函数语法要点

所需头文件	#include <stdio.h>
函数原型	int puts(const char *s) int fputs(const char * s, FILE * stream)
函数传入值	s: 要输出的字符串 stream: 对应的文件流
函数返回值	成功: s 失败: NULL

这里以 gets()和 puts()为例进行说明，本实例将标准输入复制到标准输出，如下所示：

```
/*gets.c*/
#include<stdio.h>
main()
{
    char s[80];
    /*同上例，把 fgets()的结果作为 fputs()的输入*/
    fputs(fgets(s, 80, stdin), stdout);
}
```

运行该程序，结果如下所示：

```
$ ./gets
This is stdin (用户输入)
This is stdin (屏幕输出)
```

3. 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，这里有读者已经非常熟悉的 printf()、scanf()等函数，这里就简要介绍一下它们的格式，如表 6.25～表 6.27 所示。

表 6.25 格式化输出函数 1

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,...)
函数传入值	format: 记录输出格式 fp: 文件描述符 buf: 记录输出缓冲区
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: NULL

表 6.26 格式化输出函数 2

所需头文件	#include <stdarg.h> #include <stdio.h>
函数原型	int vprintf(const char *format, va_list arg) int vfprintf(FILE *fp, const char *format, va_list arg) int vsprintf(char *buf, const char *format, va_list arg)
函数传入值	format: 记录输出格式 fp: 文件描述符 arg: 相关命令参数
函数返回值	成功: 存入数组的字符数 失败: NULL

表 6.27 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: 记录输出格式 fp: 文件描述符 buf: 记录输入缓冲区
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: NULL

由于本节的函数用法比较简单，并且比较常用，因此就不再举例了，请读者需要用到时自行查找其用法。

6.6 实验内容

6.6.1 文件读写及上锁

1. 实验目的

通过编写文件读写及上锁的程序，进一步熟悉 Linux 中文件 I/O 相关的应用开发，并且熟练掌握 `open()`、`read()`、`write()`、`fcntl()`等函数的使用。

2. 实验内容

在 Linux 中 FIFO 是一种进程之间的管道通信机制。Linux 支持完整的 FIFO 通信机制。

本实验内容比较有趣，通过使用文件操作，仿真 FIFO（先进先出）结构以及生产者-消费者运行模型。

本实验中需要打开两个虚拟终端，分别运行生产者程序（producer）和消费者程序（customer）。此时两个进程同时对同一个文件进行读写操作。因为这个文件是临界资源，所以可以使用文件锁机制来保证两个进程对文件的访问都是原子操作。

先启动生产者进程，它负责创建仿真 FIFO 结构的文件（其实是一个普通文件）并投入生产，就是按照给定的时间间隔，向 FIFO 文件写入自动生成的字符（在程序中用宏定义选择使用数字还是使用英文字符），生产周期以及要生产的资源数通过参数传递给进程（默认生产周期为 1s，要生产的资源数为 10 个字符）。

后启动的消费者进程按照给定的数目进行消费，首先从文件中读取相应数目的字符并在屏幕上显示，然后从文件中删除刚才消费过的数据。为了仿真 FIFO 结构，此时需要使用两次复制来实现文件内容的偏移。每次消费的资源数通过参数传递给进程，默认值为 10 个字符。

3. 实验步骤

（1）画出实验流程图。

本实验的两个程序的流程图如图 6.4 所示。

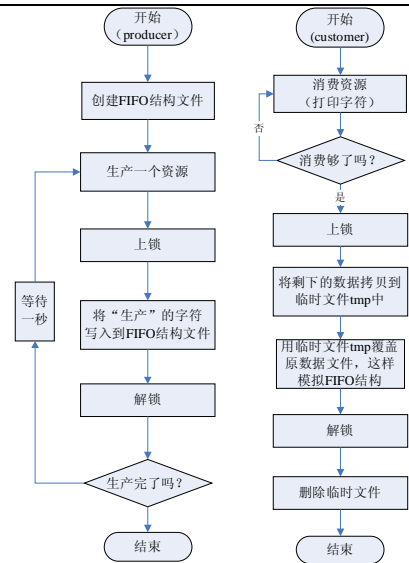


图 6.4 节流程图

(2) 编写代码。

本实验中的生产者程序的源代码如下所示，其中用到的 lock_set()函数可参见第 6.3.2 节。

```
/* producer.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "mylock.h"

#define MAXLEN      10      /* 缓冲区大小最大值*/

#define ALPHABET    1      /* 表示使用英文字符 */
#define ALPHABET_START 'a' /* 头一个字符，可以用 'A'*/
#define COUNT_OF_ALPHABET 26 /* 字母字符的个数 */

#define DIGIT       2      /* 表示使用数字字符 */
#define DIGIT_START '0'    /* 头一个字符 */
#define COUNT_OF_DIGIT 10  /* 数字字符的个数 */

#define SIGN_TYPE ALPHABET /* 本实例选用英文字符 */
const char *fifo_file = "./myfifo"; /* 仿真 FIFO 文件名 */
char buff[MAXLEN]; /* 缓冲区 */
```



```
/* 功能: 生产一个字符并写入仿真 FIFO 文件中 */
int product(void)
{
    int fd;
    unsigned int sign_type, sign_start, sign_count, size;
    static unsigned int counter = 0;

    /* 打开仿真 FIFO 文件 */
    if ((fd = open(fifo_file, O_CREAT|O_RDWR|O_APPEND, 0644)) < 0)
    {
        printf("Open fifo file error\n");
        exit(1);
    }

    sign_type = SIGN_TYPE;
    switch(sign_type)
    {
        case ALPHABET: /* 英文字符 */
        {
            sign_start = ALPHABET_START;
            sign_count = COUNT_OF_ALPHABET;
        }
        break;

        case DIGIT: /* 数字字符 */
        {
            sign_start = DIGIT_START;
            sign_count = COUNT_OF_DIGIT;
        }
        break;

        default:
        {
            return -1;
        }
    } /*end of switch*/

    sprintf(buff, "%c", (sign_start + counter));
    counter = (counter + 1) % sign_count;
}
```



```

lock_set(fd, F_WRLCK); /* 上写锁 */
if ((size = write(fd, buff, strlen(buff))) < 0)
{
    printf("Producer: write error\n");
    return -1;
}
lock_set(fd, F_UNLCK); /* 解锁 */

close(fd);
return 0;
}

int main(int argc, char *argv[])
{
    int time_step = 1; /* 生产周期 */
    int time_life = 10; /* 需要生产的资源数 */

    if (argc > 1)
    { /* 第一个参数表示生产周期 */
        sscanf(argv[1], "%d", &time_step);
    }

    if (argc > 2)
    { /* 第二个参数表示需要生产的资源数 */
        sscanf(argv[2], "%d", &time_life);
    }
    while (time_life-- > 0)
    {
        if (product() < 0)
        {
            break;
        }
        sleep(time_step);
    }

    exit(EXIT_SUCCESS);
}

```

本实验中的消费者程序的源代码如下所示。

```

/* customer.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

```

```
#include <fcntl.h>

#define MAX_FILE_SIZE      100 * 1024 * 1024 /* 100M*/

const char *fifo_file = "./myfifo";          /* 仿真 FIFO 文件名 */
const char *tmp_file = "./tmp";              /* 临时文件名 */

/* 资源消费函数 */
int customing(const char *myfifo, int need)
{
    int fd;
    char buff;
    int counter = 0;

    if ((fd = open(myfifo, O_RDONLY)) < 0)
    {
        printf("Function customing error\n");
        return -1;
    }

    printf("Enjoy:");
    lseek(fd, SEEK_SET, 0);
    while (counter < need)
    {
        while ((read(fd, &buff, 1) == 1) && (counter < need))
        {
            fputc(buff, stdout); /* 消费就是在屏幕上简单的显示 */
            counter++;
        }

        fputs("\n", stdout);
        close(fd);
        return 0;
    }

    /* 功能:从 sour_file 文件的 offset 偏移处开始
    将 count 个字节数据复制到 dest_file 文件 */
    int myfilecopy(const char *sour_file,
        const char *dest_file, int offset, int count, int copy_mode)
    {
        int in_file, out_file;
        int counter = 0;
```



```

char buff_unit;

if ((in_file = open(sour_file, O_RDONLY|O_NONBLOCK)) < 0)
{
    printf("Function myfilecopy error in source file\n");
    return -1;
}

if ((out_file = open(dest_file,
                     O_CREAT|O_RDWR|O_TRUNC|O_NONBLOCK, 0644)) < 0)
{
    printf("Function myfilecopy error in destination file:");
    return -1;
}

lseek(in_file, offset, SEEK_SET);
while ((read(in_file, &buff_unit, 1) == 1) && (counter < count))
{
    write(out_file, &buff_unit, 1);
    counter++;
}

close(in_file);
close(out_file);
return 0;
}

/* 功能: 实现 FIFO 消费者 */
int custom(int need)
{
    int fd;

    /* 对资源进行消费, need 表示该消费的资源数目 */
    customing(fifo_file, need);

    if ((fd = open(fifo_file, O_RDWR)) < 0)
    {
        printf("Function myfilecopy error in source_file:");
        return -1;
    }
}

```

```

/* 为了模拟 FIFO 结构，对整个文件内容进行平行移动 */
lock_set(fd, F_WRLCK);
myfilecopy(fifo_file, tmp_file, need, MAX_FILE_SIZE, 0);
myfilecopy(tmp_file, fifo_file, 0, MAX_FILE_SIZE, 0);
lock_set(fd, F_UNLCK);
unlink(tmp_file);
close(fd);
return 0;
}

int main(int argc ,char *argv[])
{
    int customer_capacity = 10;

    if (argc > 1) /* 第一个参数指定需要消费的资源数目，默认值为 10 */
    {
        sscanf(argv[1], "%d", &customer_capacity);
    }
    if (customer_capacity > 0)
    {
        custom(customer_capacity);
    }
    exit(EXIT_SUCCESS);
}

```

(3) 先在宿主机上编译该程序，如下所示：

```
$ make clean; make
```

(4) 在确保没有编译错误后，交叉编译该程序，此时需要修改 Makefile 中的变量

```
CC = arm-linux-gcc /* 修改 Makefile 中的编译器 */
$ make clean; make
```

(5) 将生成的可执行程序下载到目标板上运行。

4. 实验结果

此实验在目标板上的运行结果如下所示。实验结果会和这两个进程运行的具体过程相关，希望读者能具体分析每种情况。下面列出其中一种情况：

终端一：

```
$ ./producer 1 20 /* 生产周期为 1s，需要生产的资源数为 20 个 */
Write lock set by 21867
```

```
Release lock by 21867
Write lock set by 21867
Release lock by 21867
.....
```

终端二：

```
$ ./customer 5 /* 需要消费的资源数为 5 个 */
Enjoy:abcde /* 消费资源，即打印到屏幕上 */
Write lock set by 21872 /* 为了仿真 FIFO 结构，进行两次复制 */
Release lock by 21872
```

在两个进程结束之后，仿真 FIFO 文件的内容如下：

```
$ cat myfifo
fghijklmnopqr /* a~e 的 5 个字符已经被消费，就剩下后面 15 个字符 */
```

6.6.2 多路复用式串口操作

1. 实验目的

通过编写多路复用式串口读写，进一步理解多路复用函数的用法，同时更加熟练掌握 Linux 设备文件的读写方法。

2. 实验内容

本实验主要实现两台机器（宿主机和目标板）之间的串口通信，每台机器都可以发送和接收数据。除了串口设备名称不同（宿主机上使用串口 1：/dev/ttyS0，而在目标板上使用串口 2：/dev/ttyS1），两台机器上的程序基本相同。

3. 实验步骤

（1）画出流程图

如图 6.5 所示为程序流程图，两台机器上的程序使用同样的流程图。

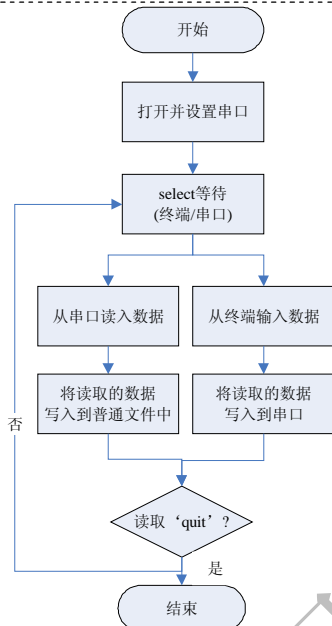


图 6.5 宿主机/目标板程序的流程图

(2) 编写代码。

编写宿主机和目标板上的代码，在这些程序中用到的 `open_port()` 和 `set_com_config()` 函数请参照 6.4 节。这里只列出宿主机上的代码。

```

/* com_host.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include "uart_api.h"

int main(void)
{
    int fds[SEL_FILE_NUM], recv_fd, maxfd;
    char buff[BUFFER_SIZE];
    fd_set inset, tmp_inset;
    struct timeval tv;
    unsigned loop = 1;
    int res, real_read, i;
    /* 将从串口读取的数据写入这个文件中 */

```

```

if ((recv_fd = open(RECV_FILE_NAME, O_CREAT | O_WRONLY, 0644)) < 0)
{
    perror("open");
    return 1;
}

fds[0] = STDIN_FILENO; /* 标准输入 */
if ((fds[1] = open_port(HOST_COM_PORT)) < 0) /* 打开串口 */
{
    perror("open_port");
    return 1;
}

if (set_com_config(fds[1], 115200, 8, 'N', 1) < 0) /* 配置串口 */
{
    perror("set_com_config");
    return 1;
}

FD_ZERO(&inset);
FD_SET(fds[0], &inset);
FD_SET(fds[1], &inset);
maxfd = (fds[0] > fds[1])?fds[0]:fds[1];
tv.tv_sec = TIME_DELAY;
tv.tv_usec = 0;
printf("Input some words(enter 'quit' to exit):\n");
while (loop && (FD_ISSET(fds[0], &inset) || FD_ISSET(fds[1],
&inset)))
{
    tmp_inset = inset;
    res = select(maxfd + 1, &tmp_inset, NULL, NULL, &tv);
    switch(res)
    {
        case -1:
            {
                perror("select");
                loop = 0;
            }
        break;
    }
}

```



```
case 0: /* Timeout */
{
    perror("select time out");
    loop = 0;
}
break;

default:
{
    for (i = 0; i < SEL_FILE_NUM; i++)
    {
        if (FD_ISSET(fds[i], &tmp_inset))
        {
            memset(buff, 0, BUFFER_SIZE);
            /* 读取标准输入或者串口设备文件 */
            real_read = read(fds[i], buff, BUFFER_SIZE);
            if ((real_read < 0) && (errno != EAGAIN))
            {
                loop = 0;
            }
            else if (!real_read)
            {
                close(fds[i]);
                FD_CLR(fds[i], &inset);
            }
            else
            {
                buff[real_read] = '\0';
                if (i == 0)
                { /* 将从终端读取的数据写入串口*/
                    write(fds[1], buff, strlen(buff));
                    printf("Input some words\n\n");
                }
                else if (i == 1)
                { /* 将从串口读取的数据写入普通文件中*/
                    write(recv_fd, buff, real_read);
                }
                if (strcmp(buff, "quit", 4) == 0)
                { /* 如果读取为 'quit' 则退出*/
```



```

        loop = 0;
    }
}
} /* end of if FD_ISSET */
} /* for i */
}
} /* end of switch */
} /* end of while */
close(recv_fd);
return 0;
}

```

(3) 接下来，将目标板的串口程序交叉编译，再将宿主机的串口程序在 PC 机上编译。

(4) 连接 PC 的串口 1 和开发板的串口 2。然后将目标板串口程序下载到开发板上，分别在两台机器上运行串口程序。

4. 实验结果

宿主机上的运行结果如下所示：

```

$ ./com_host
Input some words(enter 'quit' to exit):
Hello, Target!
Input some words(enter 'quit' to exit):
I'm host program!
Input some words(enter 'quit' to exit):
Byebye!
Input some words(enter 'quit' to exit):
quit    /* 这个输入使双方的程序都结束*/

```

从串口读取的数据（即目标板中发送过来的数据）写入同目录下的 `recv.dat` 文件中。

```

$ cat recv.dat
Hello, Host!
I'm target program!
Byebye!

```

目标板上的运行结果如下所示：

```

$ ./com_target
Input some words(enter 'quit' to exit):
Hello, Host!

```

```
Input some words(enter 'quit' to exit):  
I'm target program!  
Input some words(enter 'quit' to exit):  
Byebye!
```

与宿主机上的代码相同，从串口读取的数据（即目标板中发送过来的数据）写入同目录下的 recv.dat 文件中。

```
$ cat recv.dat  
Hello, Target!  
I'm host program!  
Byebye!  
Quit
```

请读者用 poll() 函数实现具有以上功能的代码。

6.7 本章小结

本章首先讲解了系统调用（System Call）、用户函数接口（API）和系统命令之间的联系和区别，这也是贯穿本书的一条主线，本书就是按照系统命令、用户函数接口（API）系统调用的顺序逐层深入讲解，希望读者能有一个较为深刻的认识。

接着，本章讲解了嵌入式 Linux 中文件 I/O 相关的开发，在这里主要讲解了不带缓存的 I/O 系统调用函数的使用，这也是本章的重点，其中主要讲解了 open()、close()、read()、write()、lseek()、fcntl()、select() 以及 poll() 等函数。

接下来，本章讲解了嵌入式 Linux 串口编程。这其实是 Linux 中设备文件读写的实例，由于它能很好地体现前面所介绍的内容，而且在嵌入式开发中也较为常见，因此对它进行了比较详细的讲解。

之后，本章简单介绍了标准 I/O 的相关函数，希望读者也能对它有一个总体的认识。

最后，本章安排了两个实验，分别是文件使用及上锁和多路复用串口操作。希望读者能够认真完成。

6.8 思考与练习

使用多路复用函数实现 3 个串口的通信：串口 1 接收数据，串口 2 和串口 3 向串口 1 发送数据。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章: <http://www.embedu.org/courses/index.htm>
- 课程内容: <http://www.embedu.org/courses/course1.htm>
- 项目实战: <http://www.embedu.org/courses/project.htm>
- 出版教材: <http://www.embedu.org/courses/course3.htm>
- 实验设备: <http://www.embedu.org/courses/course5.htm>

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班:
<http://www.farsight.com.cn/courses/TS-LinuxApp.htm>
- 嵌入式 Linux 系统开发班:
<http://www.farsight.com.cn/courses/TS-LinuxEMB.htm>
- 嵌入式 Linux 驱动开发班:
<http://www.farsight.com.cn/courses/TS-LinuxDriver.htm>