Connection Control

Studying `stty`

Objectives

- **Ideas and Skills**
  - Similarities between files and devices
  - Differences between files and devices
  - Attributes of connections
  - Race conditions and atomic operations
  - Controlling device drivers
  - Streams
- **System Calls and Functions**
  - `fcntl, ioctl`
  - `tcsetattr, tcgetattr`
- **Commands**
  - `stty`
  - `write`

Connection Control

5.2 Devices Are Just Like Files

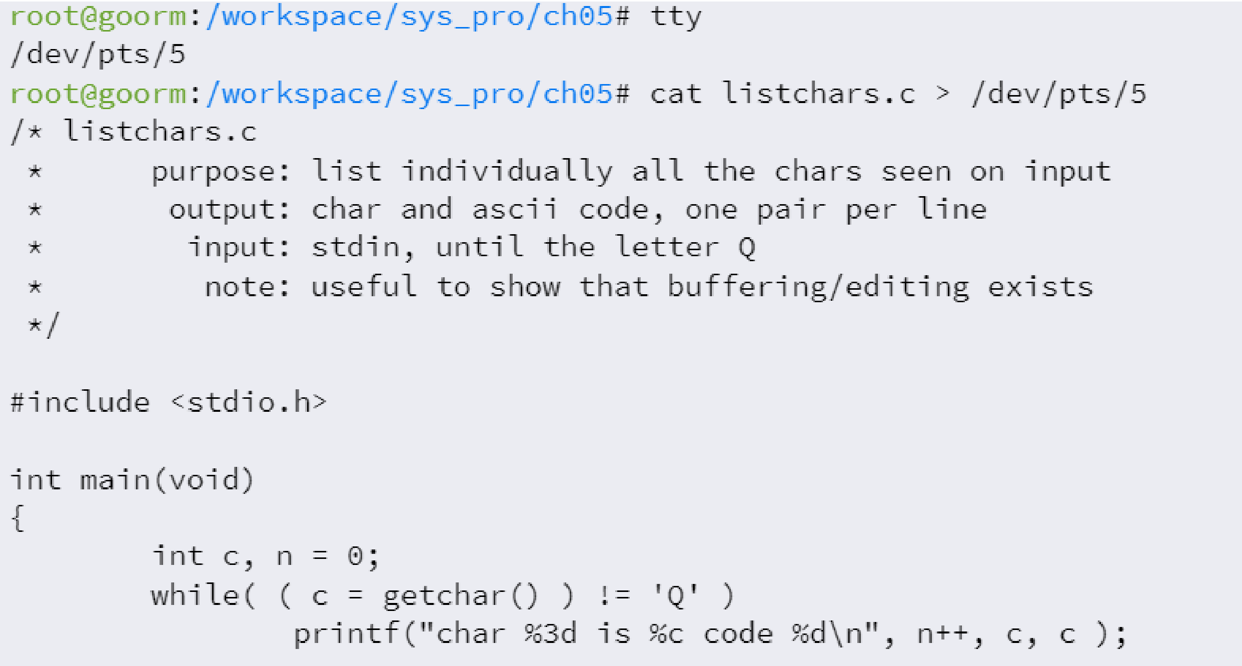
5.3 Devices Are Not Like files

5.4 Attributes of Disk Connections

5.5 Attributes of Terminal Connections

Devices Are Just Like Files

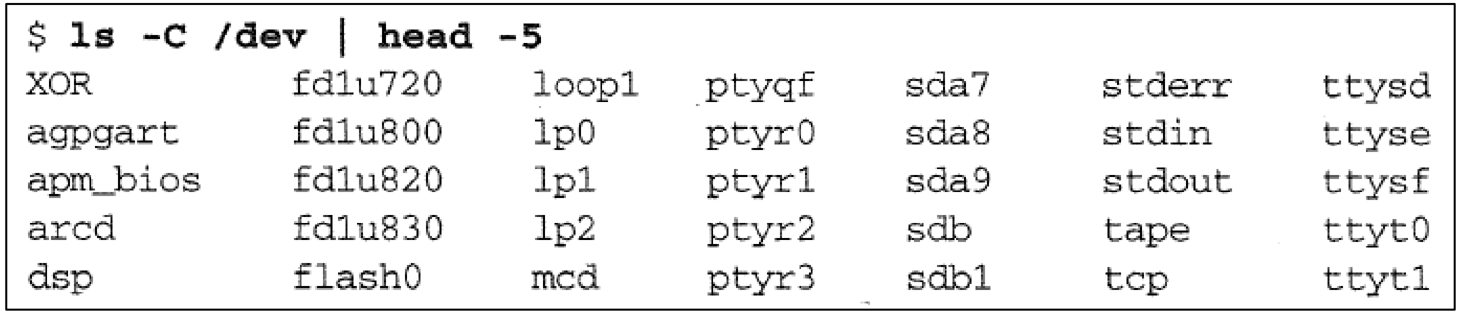
- **In Unix, every device is treated as a file**
- **Each device has:**
  - a filename
  - an inode number
  - an owner
  - permission bits
  - a last-modified time
- **File operations apply the same way to terminals and other devices**



```
root@goorm:/workspace/sys_pro/ch05# tty
/dev/pts/5
root@goorm:/workspace/sys_pro/ch05# cat listchars.c > /dev/pts/5
/* listchars.c
 *      purpose: list individually all the chars seen on input
 *       output: char and ascii code, one pair per line
 *        input: stdin, until the letter Q
 *         note: useful to show that buffering/editing exists
 */

#include <stdio.h>

int main(void)
{
        int c, n = 0;
        while( ( c = getchar() ) != 'Q' )
                printf("char %3d is %c code %d\n", n++, c, c );
```

Devices Have Filenames

- **By tradition, device files are in the `/dev` directory :**
  - `lp*` files : printers
  - `fd*` files : floppy-disk drives
  - `sd*` files : partitions on SCSI drives
  - `tape` file : the backup tape drive
  - **`tty*` files : terminals**
  - `dsp` file : a connection to a sound card

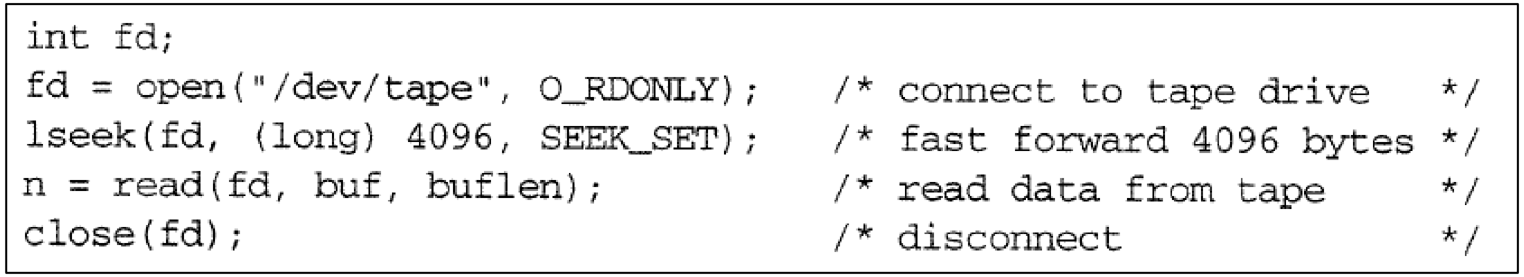


```
$ ls -C /dev | head -5
XOR          fd1u720   loop1   ptyqf   sda7    stderr   ttysd
agpgart      fd1u800   lp0     ptyr0   sda8    stdin    ttyse
apm_bios     fd1u820   lp1     ptyr1   sda9    stdout   ttysf
arcd         fd1u830   lp2     ptyr2   sdb     tape     ttyt0
dsp          flash0    mcd     ptyr3   sdb1    tcp      ttyt1
```

Devices and System Calls

- **Devices support standard file system calls:**
  - `open` – to access the device
  - `read` – to get data from it
  - `write` – to send data to it
  - `lseek` – to move the read/write position (if supported)
  - `close` – to release the device
  - `stat` – to get device metadata

- **Ex) Code to read from a magnetic tape**



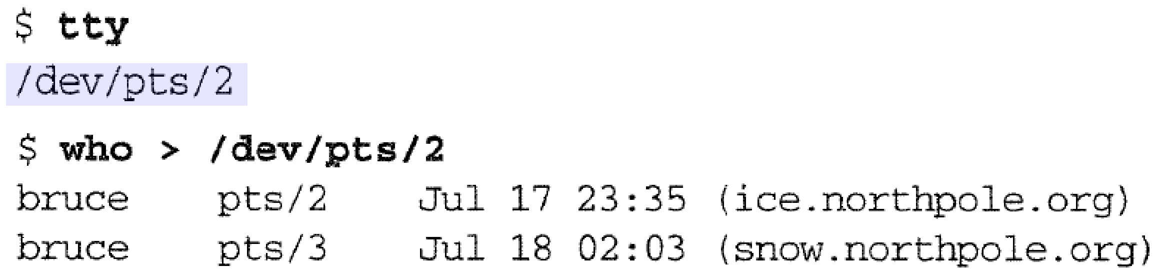
```
int fd;
fd = open("/dev/tape", O_RDONLY);   /* connect to tape drive    */
lseek(fd, (long) 4096, SEEK_SET);   /* fast forward 4096 bytes */
n = read(fd, buf, buflen);          /* read data from tape      */
close(fd);                          /* disconnect               */
```

Ex: Terminals Are Just Like Files

- **A terminal** is any device or program that behaves like a **classic keyboard + display unit**
  - `telnet`
  - `ssh`

- Command **`tty`**
  - Prints the file name of your terminal (e.g., /dev/pts/0)

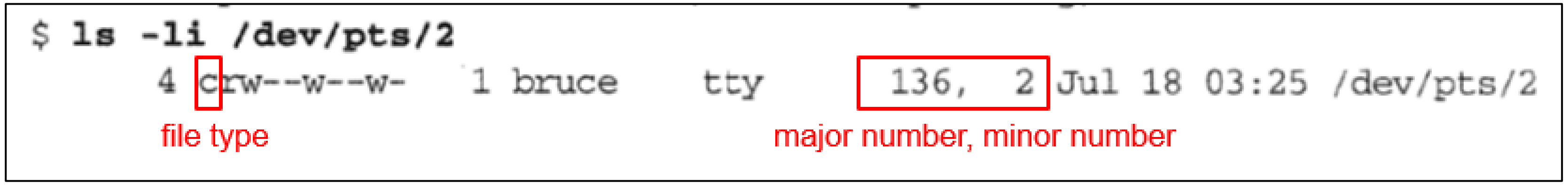
- We can use **regular file commands** on terminal files:
  - `cp, >, mv, ln, rm, cat, ls`



```
$ tty
/dev/pts/2
$ who > /dev/pts/2
bruce      pts/2     Jul 17 23:35 (ice.northpole.org)
bruce      pts/3     Jul 18 02:03 (snow.northpole.org)
```

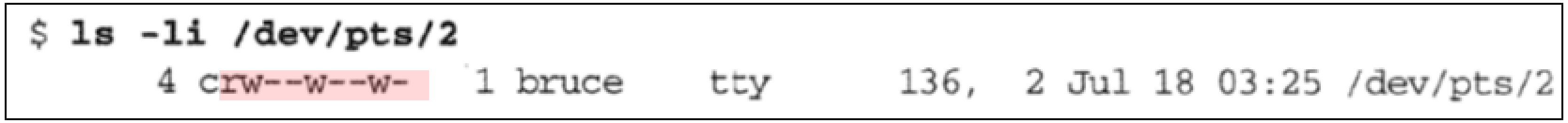
Properties of Device Files

- **Device files** have most of the **properties** with disk files
  - **inode :** a pointer to **a device driver** in the kernel
  - **Major number**: identifies **which driver** handles the device
  - **Minor number**: passed to the driver to identify the **specific device instance**



```
$ ls -li /dev/pts/2
       4 crw--w--w-   1 bruce    tty       136,   2 Jul 18 03:25 /dev/pts/2
```

- **Device files and Permission Bits:** same concept as with regular files
  - **Write permission**: …
    - e.g., writing audio to `/dev/dsp` plays sound
  - **Read permission**: …
    - e.g., reading from `/dev/input/mice` gets mouse input

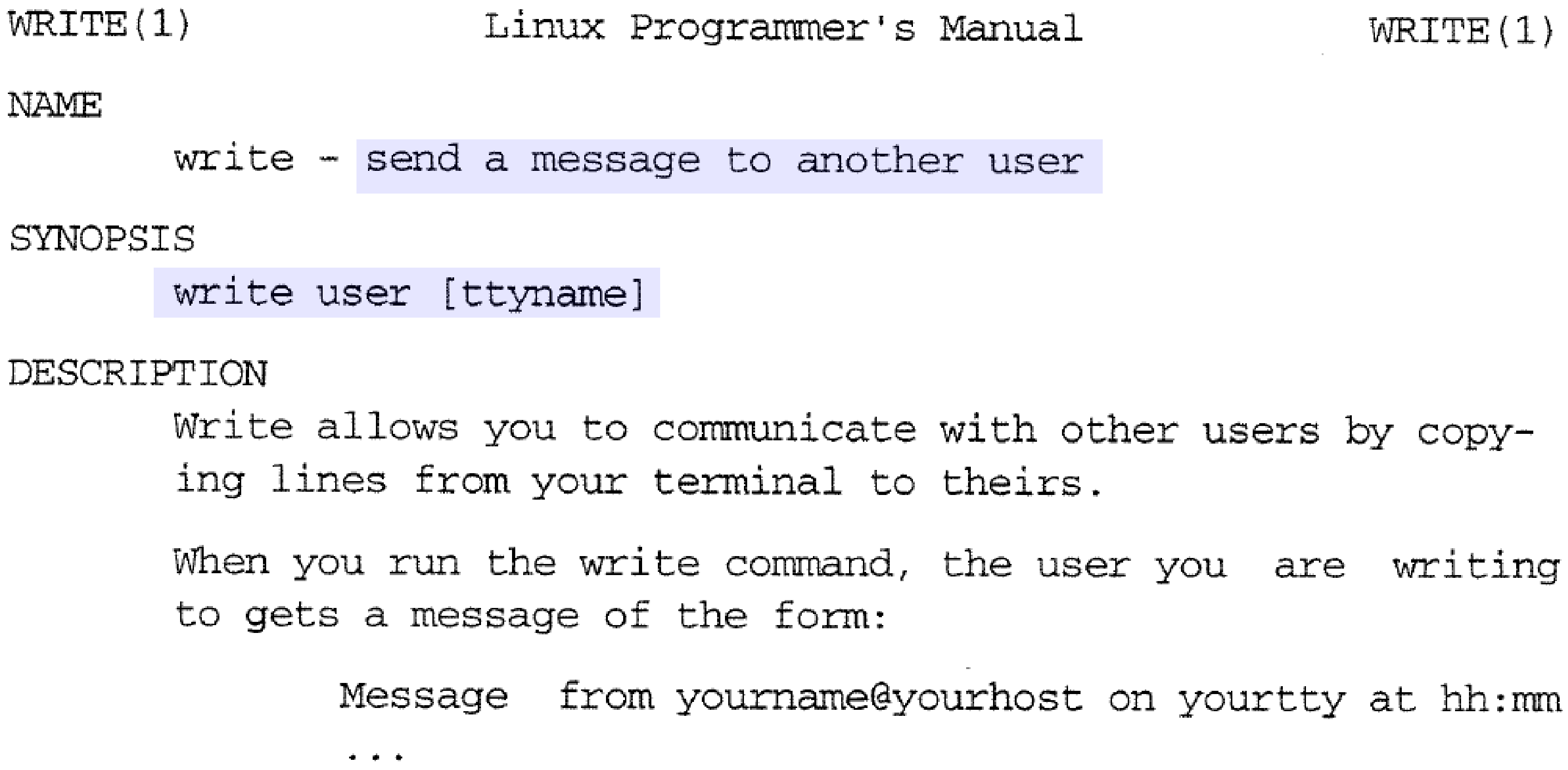


```
$ ls -li /dev/pts/2
      4 crw--w--w-   1 bruce    tty       136,  2 Jul 18 03:25 /dev/pts/2
```

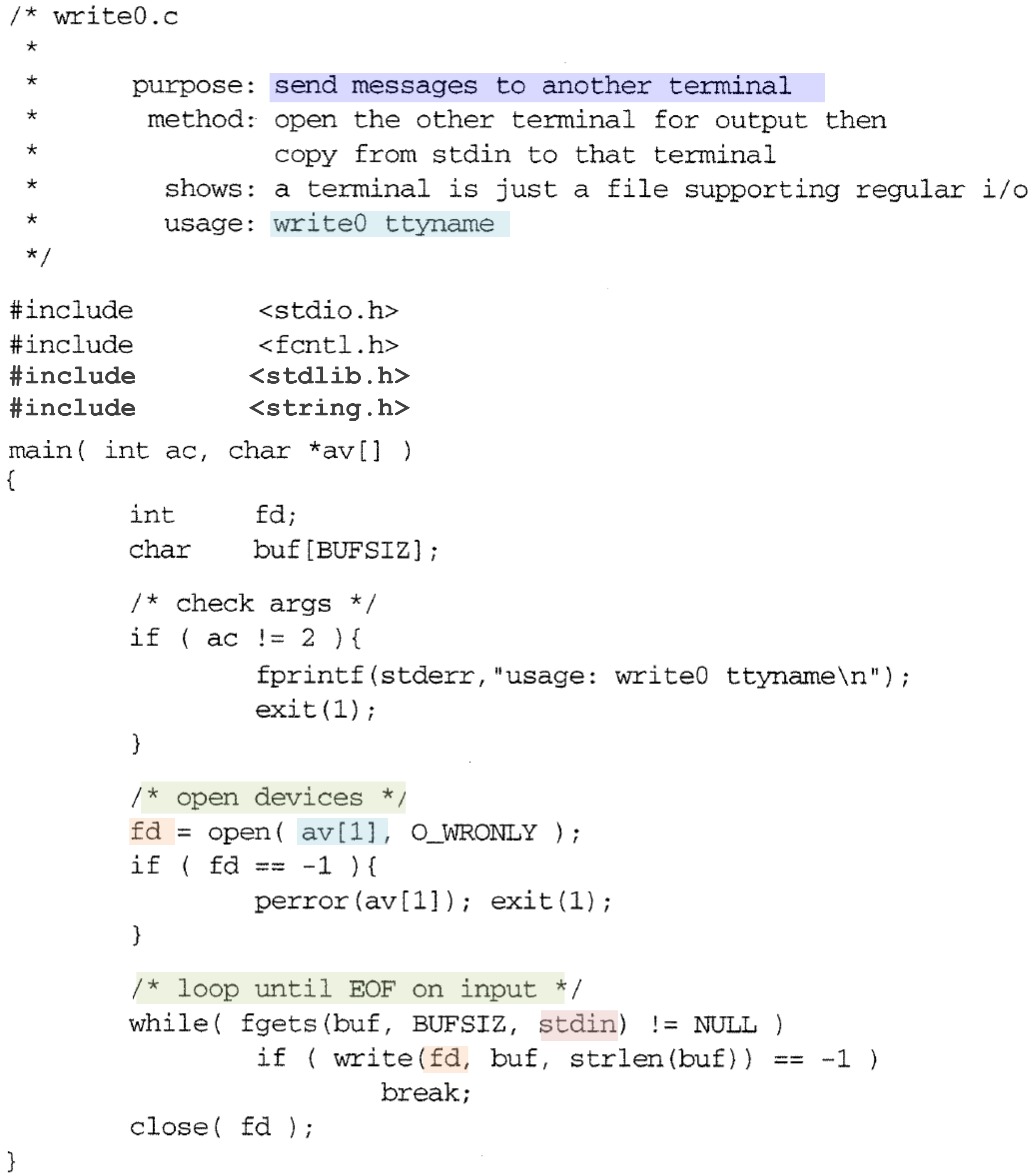
Writing `write`

```
Ex) chatting
$ write root /dev/pts/6
Hello! ^^
…
ctrl+d
```

- **`$ man 1 write`**

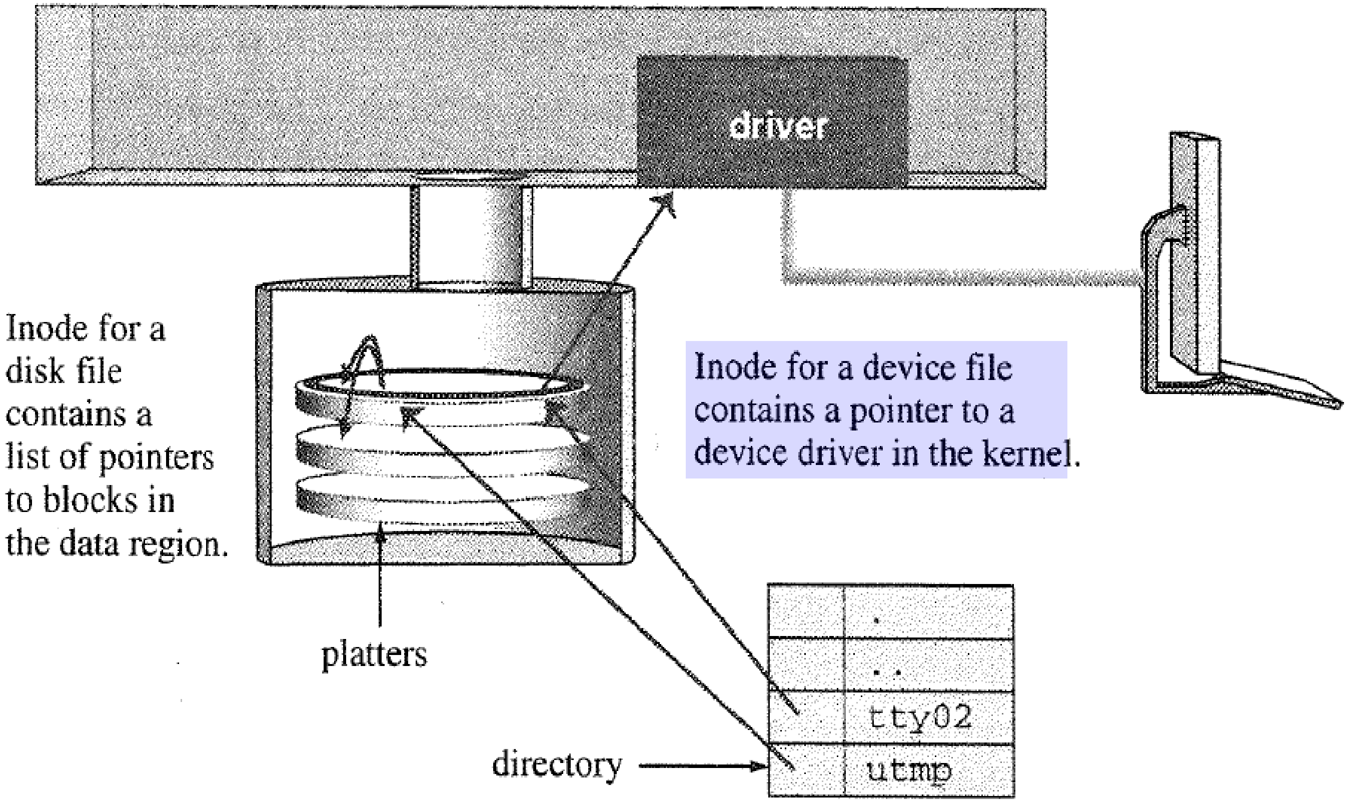


```
WRITE(1)            Linux Programmer's Manual            WRITE(1)

NAME
       write - send a message to another user

SYNOPSIS
       write user [ttyname]

DESCRIPTION
       Write allows you to communicate with other users by copy-
       ing lines from your terminal to theirs.

       When you run the write command, the user you  are  writing
       to gets a message of the form:

              Message  from yourname@yourhost on yourtty at hh:mm
              ...
```



```
/* write0.c
 *
 *      purpose: send messages to another terminal
 *       method: open the other terminal for output then
 *               copy from stdin to that terminal
 *        shows: a terminal is just a file supporting regular i/o
 *        usage: write0 ttyname
 */

#include        <stdio.h>
#include        <fcntl.h>
#include        <stdlib.h>
#include        <string.h>

main( int ac, char *av[] )
{
        int     fd;
        char    buf[BUFSIZ];

        /* check args */
        if ( ac != 2 ){
                fprintf(stderr,"usage: write0 ttyname\n");
                exit(1);
        }

        /* open devices */
        fd = open( av[1], O_WRONLY );
        if ( fd == -1 ){
                perror(av[1]); exit(1);
        }

        /* loop until EOF on input */
        while( fgets(buf, BUFSIZ, stdin) != NULL )
                if ( write(fd, buf, strlen(buf)) == -1 )
                        break;
        close( fd );
}
```

```
$ make write0
$ ./write0 /dev/pts/1
Is it working?...
Bye
^c
```



Inode for a disk file contains a list of pointers to blocks in the data region.

Inode for a device file contains a pointer to a device driver in the kernel.

FIGURE 5.1

Inode points to data blocks or to driver code.

Device Files and Inodes

- How `read` Works
  - **1. Kernel finds the inode** for the given file descriptor
  - **2. Inode tells the kernel** what kind of file it is:
    - **Disk file** → read data from the file system
    - **Device file** → kernel calls the `read` function in the device driver

- Other Operations Work Similarly
  - `open`, `write`, `lseek`, `close`

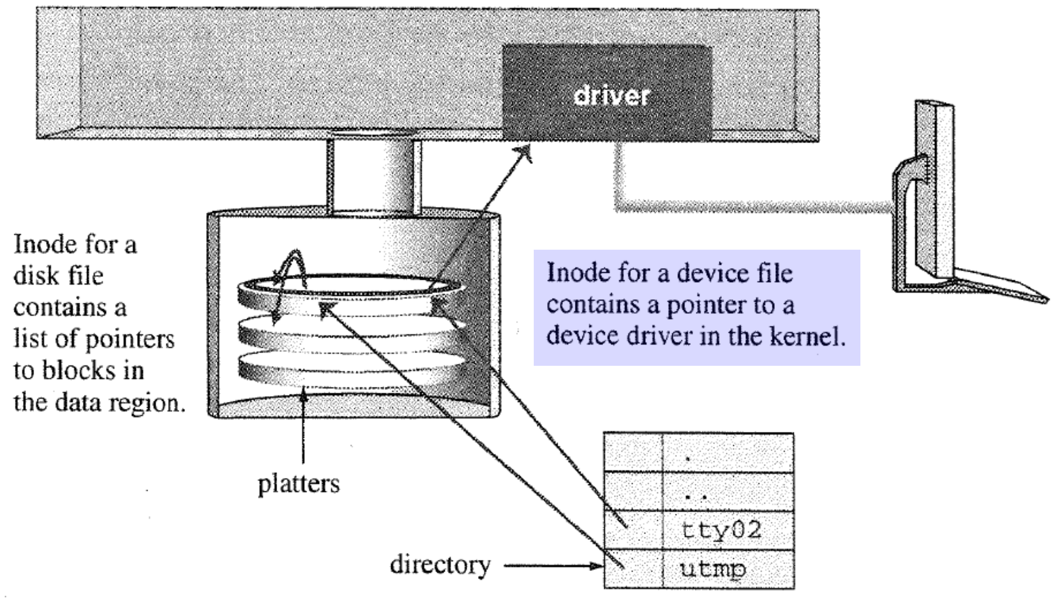

driver
Inode for a disk file contains a list of pointers to blocks in the data region.
Inode for a device file contains a pointer to a device driver in the kernel.
platters
directory
.
..
tty02
utmp

FIGURE 5.1

Inode points to data blocks or to driver code.

Connection Control

5.2 Devices Are Just Like Files

5.3 Devices Are Not Like files

5.4 Attributes of Disk Connections

5.5 Attributes of Terminal Connections

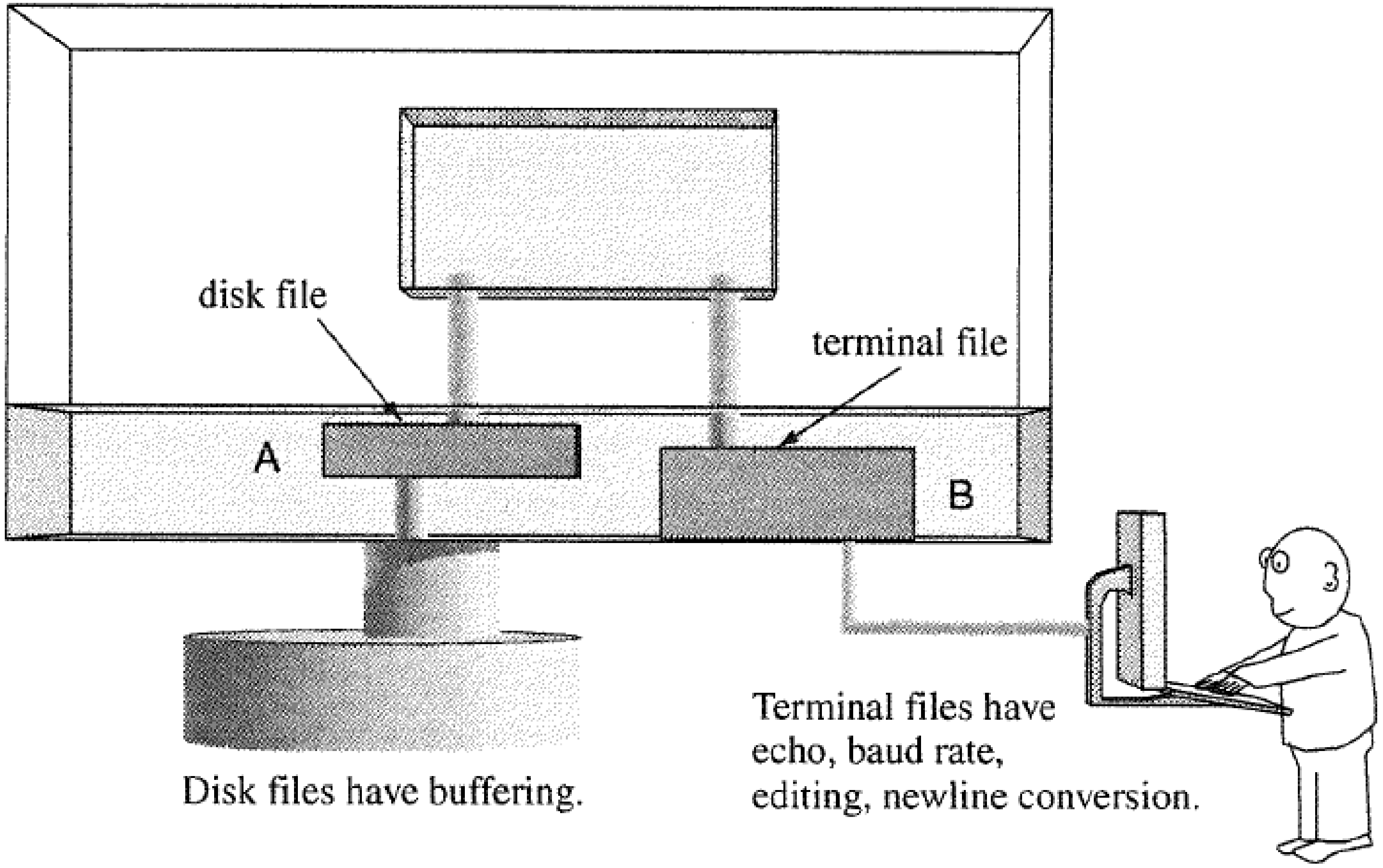


FIGURE 5.2

A process with two file descriptors.

Devices Are Not Like Files

- **Disk File vs Terminal File**
  - Both have filenames and properties (e.g., inode)
  - `open()` creates a **connection** to a file or device

- **Connection Differences**
  - Disk File
    - Has **buffering**
  - Terminal File
    - Has attributes like:
      - **echo**
      - **baud rate**
      - **line editing**
      - **newline conversion**

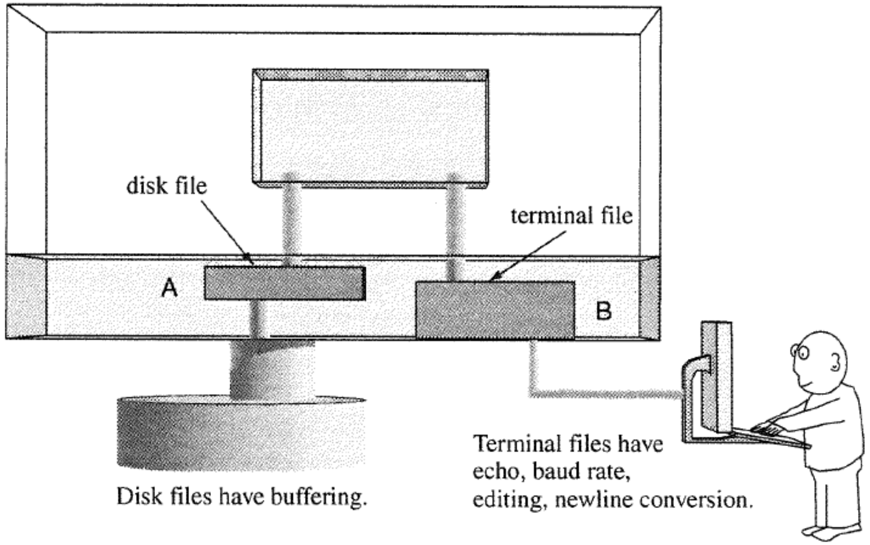

disk file
terminal file
A
B
Disk files have buffering.
Terminal files have echo, baud rate, editing, newline conversion.

FIGURE 5.2

A process with two file descriptors.

Connection Attributes and Control

- **The attributes of connections:**
  1. What attributes can a connection have?
  2. How can you examine the current attributes?
  3. How can you change the current attributes?

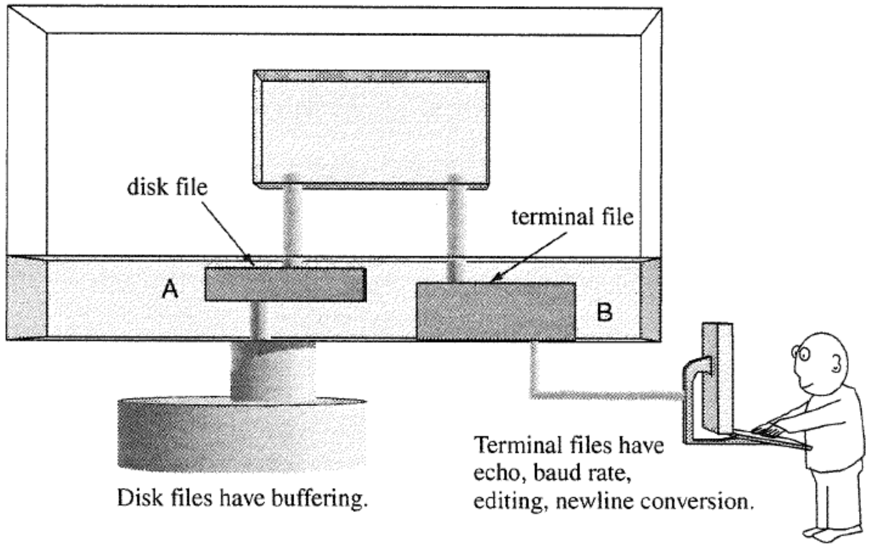

disk file
terminal file
A
B
Disk files have buffering.
Terminal files have echo, baud rate, editing, newline conversion.

FIGURE 5.2

A process with two file descriptors.

Connection Control

5.2 Devices Are Just Like Files

5.3 Devices Are Not Like files

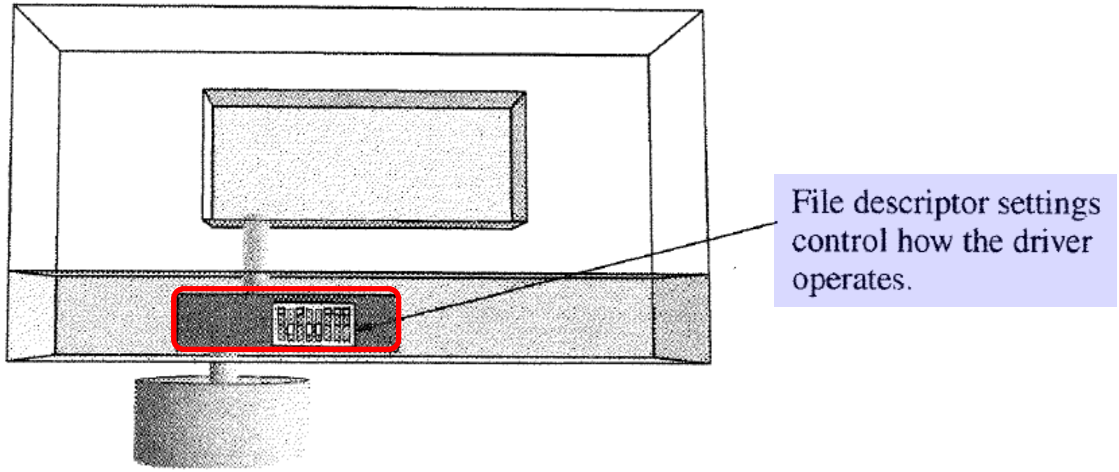
5.4 Attributes of Disk Connections

5.5 Attributes of Terminal Connections

5.6 Programming Other Devices: ioctl

Attributes of Disk Connections : Buffering

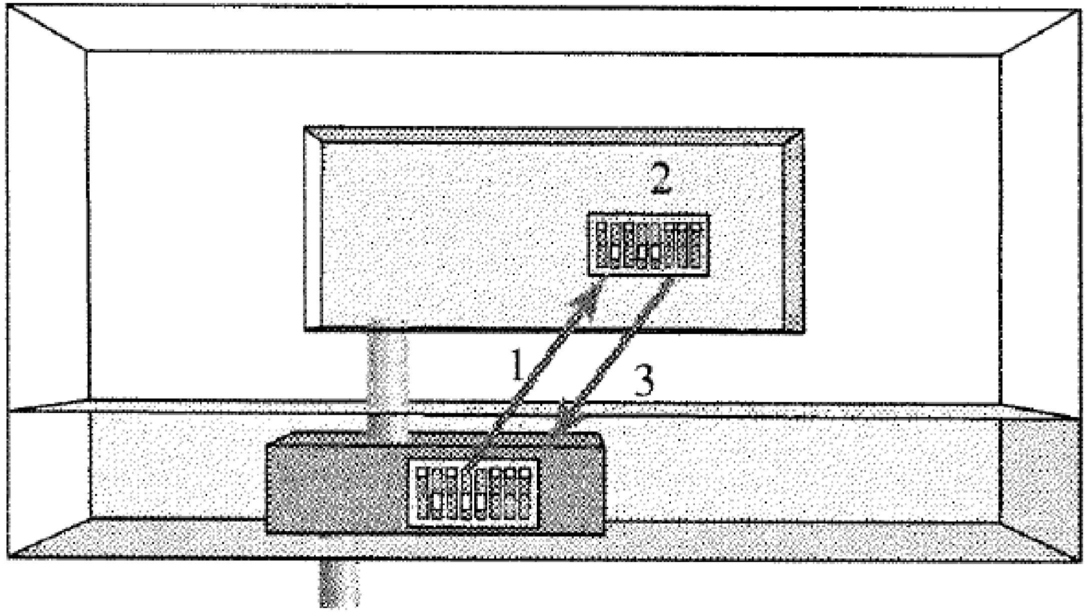
- The **processing unit** is kernel code
  - It handles:
    - **Buffering**
    - **Other I/O processing tasks**
  - Inside this unit are **control variables**
    - we can **change the behavior** (e.g., echo, buffering) by modifying these variables



File descriptor settings control how the driver operates.

FIGURE 5.3

A processing unit in a data stream.



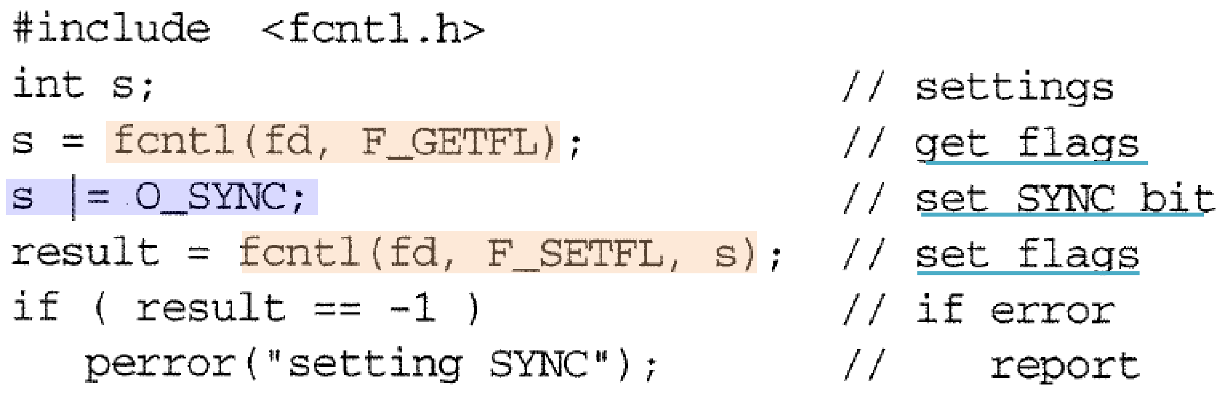
To change driver settings:

1. Get settings,
2. modify them
3. send them back.

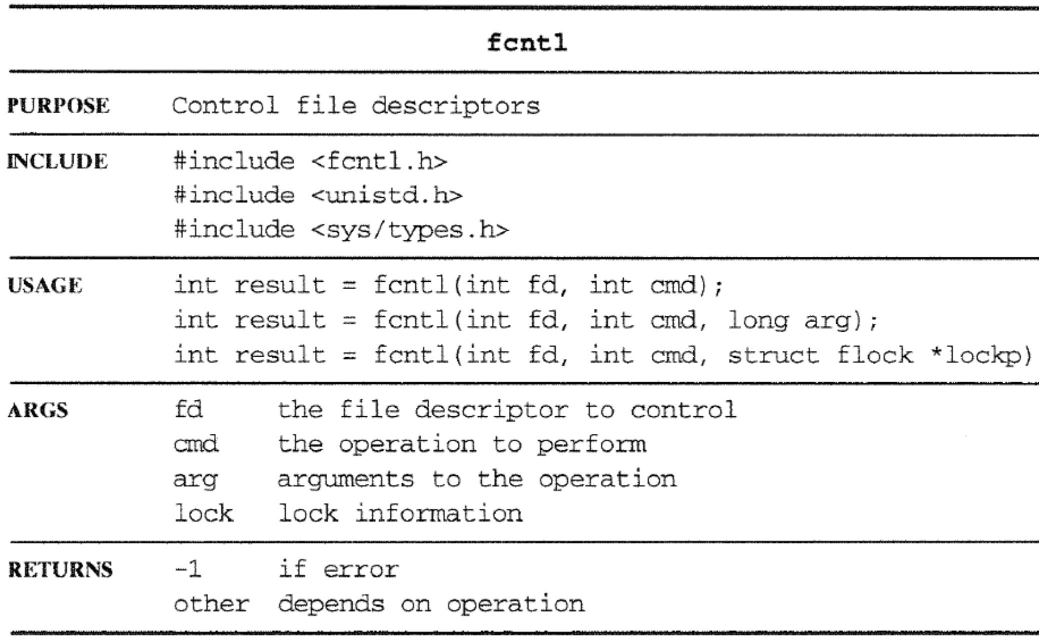
FIGURE 5.4

Modifying the operation of a file descriptor.

- **Ex: Turning off disk buffering**



```
#include  <fcntl.h>
int s;                                 // settings
s = fcntl(fd, F_GETFL);                // get flags
s |= O_SYNC;                           // set SYNC bit
result = fcntl(fd, F_SETFL, s);        // set flags
if ( result == -1 )                    // if error
   perror("setting SYNC");             //    report
```



	fcntl
PURPOSE	Control file descriptors

INCLUDE	#include <fcntl.h>
	#include <unistd.h>
	#include <sys/types.h>

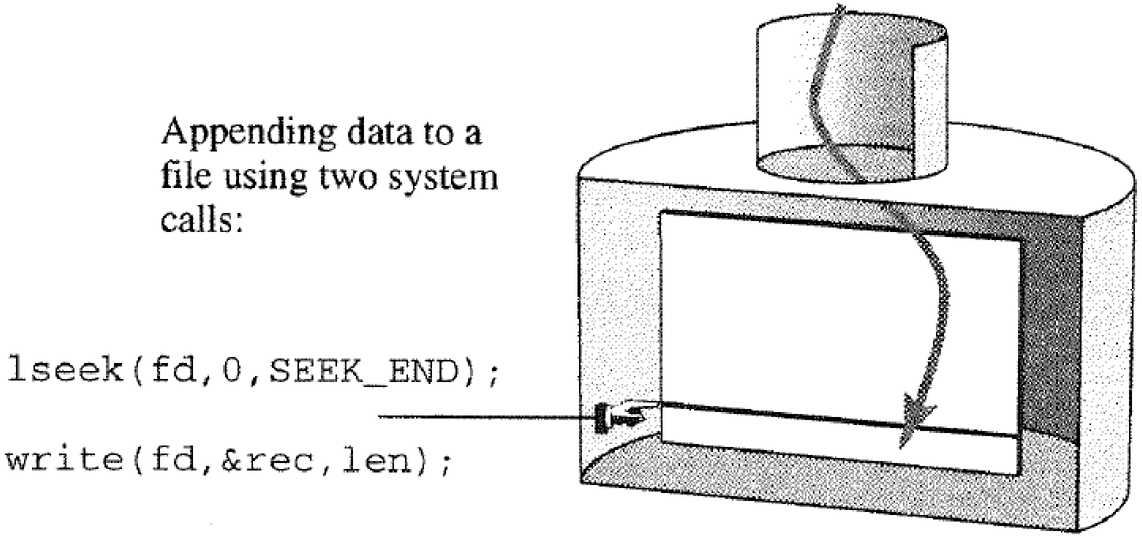
USAGE	int result = fcntl(int fd, int cmd);
	int result = fcntl(int fd, int cmd, long arg);
	int result = fcntl(int fd, int cmd, struct flock *lockp)

ARGS	fd the file descriptor to control cmd the operation to perform arg arguments to the operation lock lock information
RETURNS	-1 if error other depends on operation

Attributes of Disk Connections : Auto-Append Mode

- **Auto-append**
  - Ensures **data is always added to the end** of a file
  - **Useful when multiple processes** write to the same file

- Ex) `wtmp` Logfile
  - Stores **login/logout history**
  - When a user logs in → login record is **appended**
  - When a user logs out → logout record is **appended**
  - Each record **must be added to the end** of the file to keep the log consistent



Appending data to a file using two system calls:

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

write(fd,&rec,len);
```

FIGURE 5.5

Appending with `lseek` and `write`.

What If Two People Log In at the Same Time?

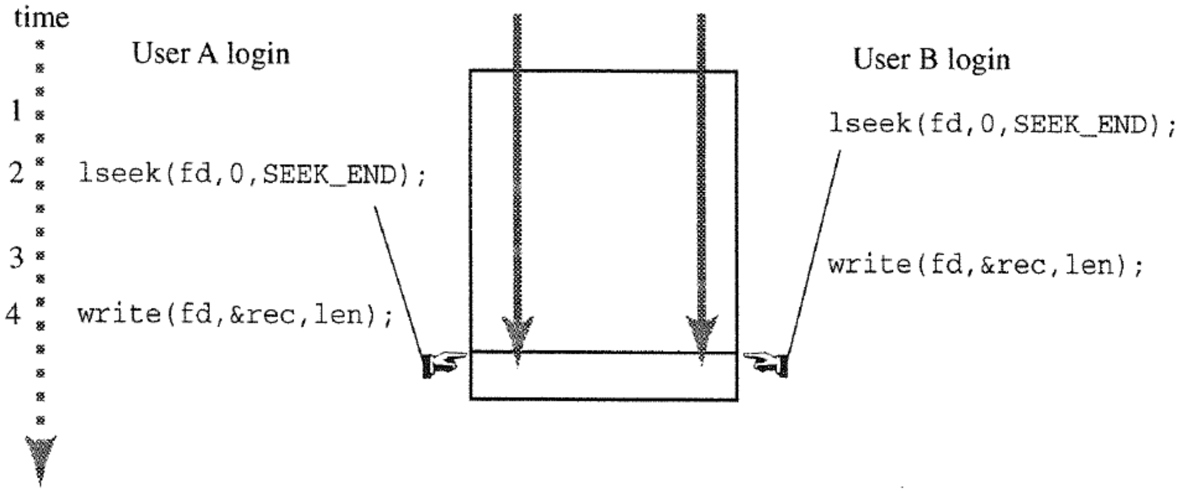


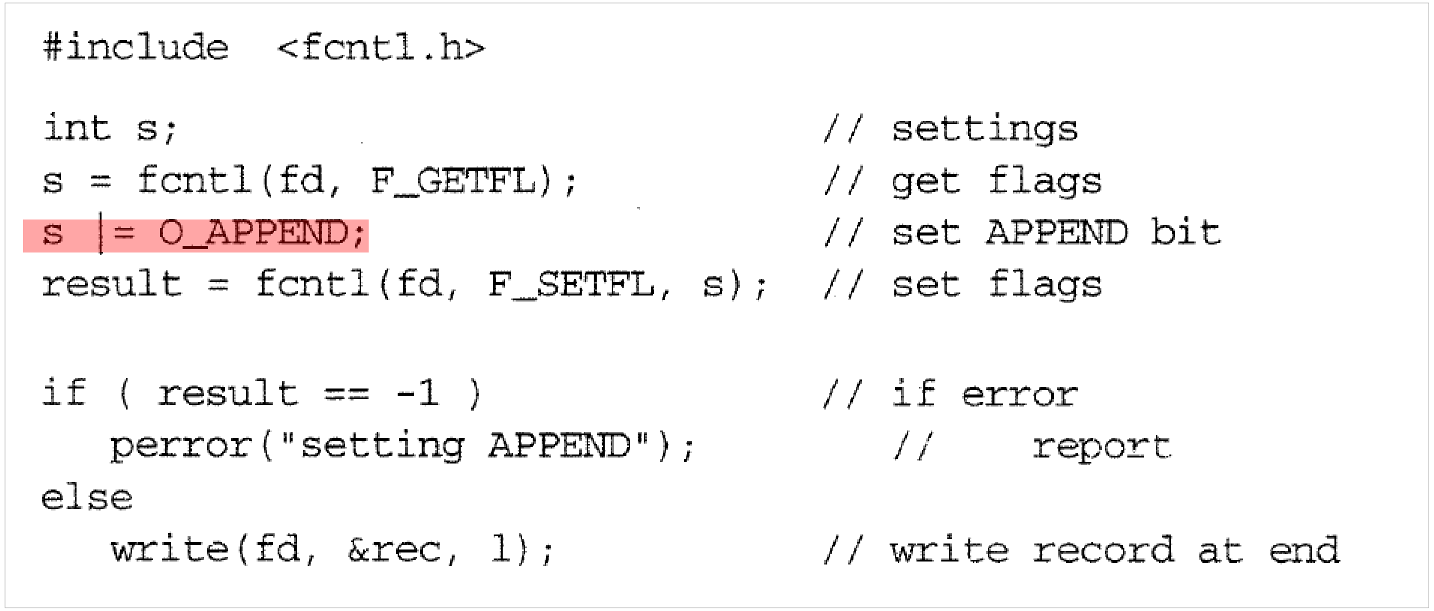
FIGURE 5.6

Interleaved lseek and write = chaos.

time 1—B's login process seeks to end of file
time 2—B's time slice is up, A's login process seeks to end of file
time 3—A's time slice is up, B's login process writes record
time 4—B's time slice is up, A's login process writes record

→ What problem …

- **How Can This “Race Condition” Be Avoided?**
  - Kernel provides a solution **auto-append mode: `O_APPEND`** bit



```
#include  <fcntl.h>

int s;                                 // settings
s = fcntl(fd, F_GETFL);                // get flags
s |= O_APPEND;                         // set APPEND bit
result = fcntl(fd, F_SETFL, s);        // set flags

if ( result == -1 )                    // if error
   perror("setting APPEND");              //    report
else
   write(fd, &rec, 1);                 // write record at end
```

  - With O_APPEND, the kernel performs:
    `lseek + write` → **as one atomic step**

Controlling File Descriptors with `open`

- `open()` lets you set ***fd*** **attribute bits** :

```
fd = open(WTMP_FILE, O_WRONLY | O_APPEND | O_SYNC);
```

- **Equivalent Calls:**

```
fd = creat(filename, permission_bits);
fd = open(filename, O_CREAT | O_TRUNC | O_WRONLY, permission_bits);
```

- Other Flags
  - **O_CREAT** – Create file if it doesn't exist
  - **O_TRUNC** – If file exists, truncate (empty) it
  - **O_EXCL** – Used with O_CREAT; if file exists, fail (prevents duplicates)

Summary of Disk Connections

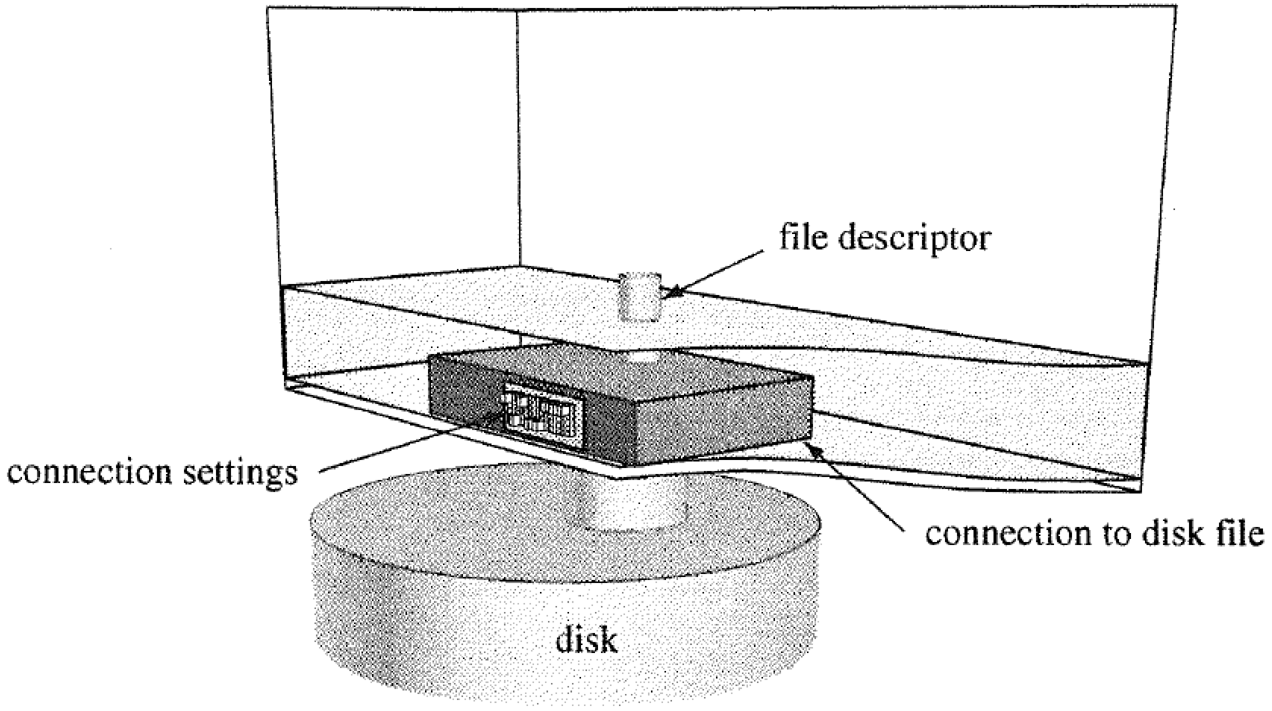


FIGURE 5.7

Connections to files have settings.

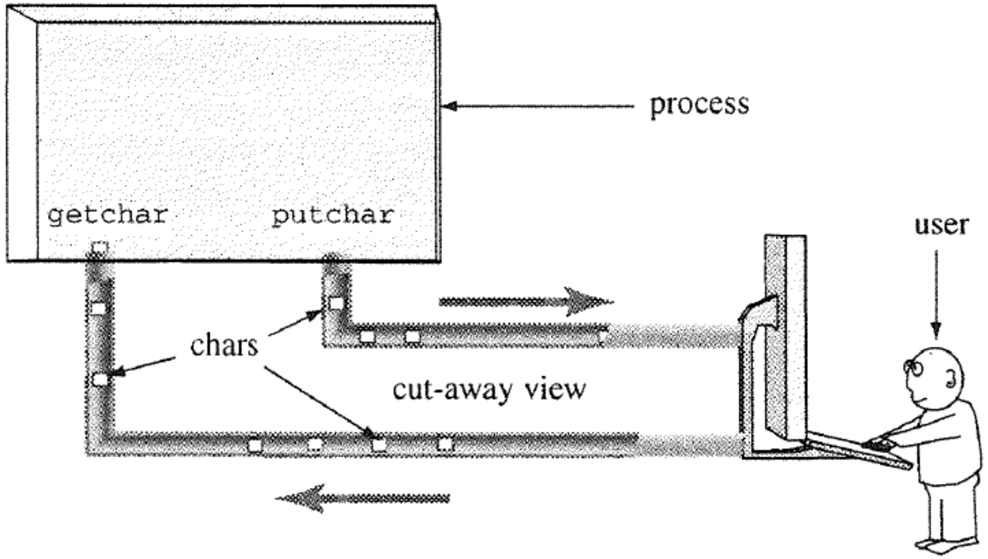
Connection Control

5.2 Devices Are Just Like Files

5.3 Devices Are Not Like files

5.4 Attributes of Disk Connections

5.5 Attributes of Terminal Connections



Terminal I/O Is Not as Simple as It Appears

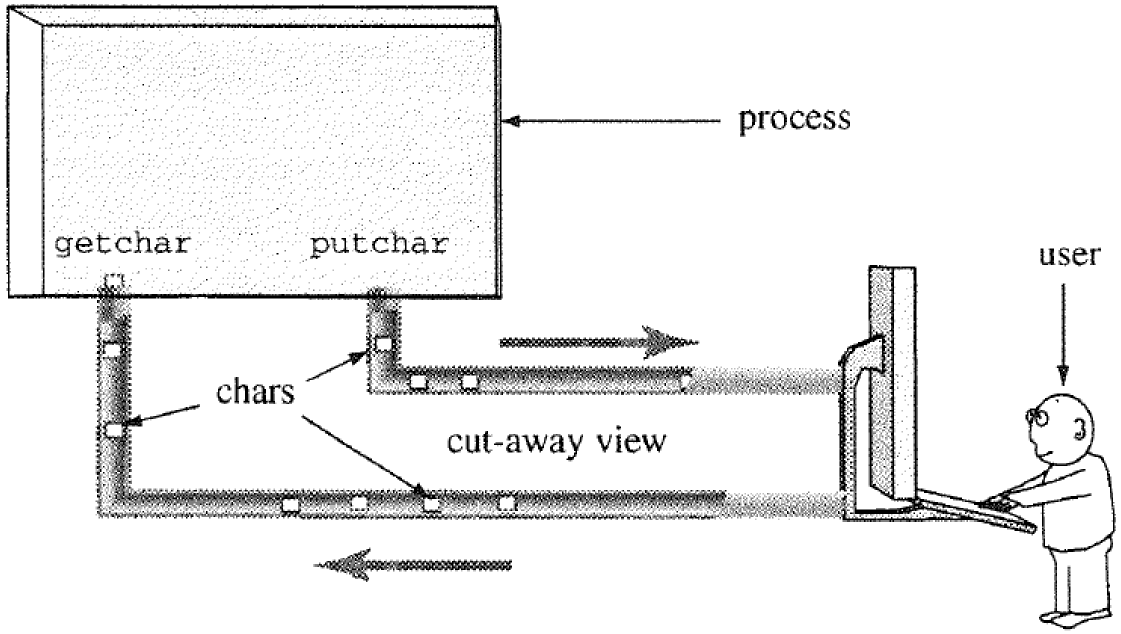
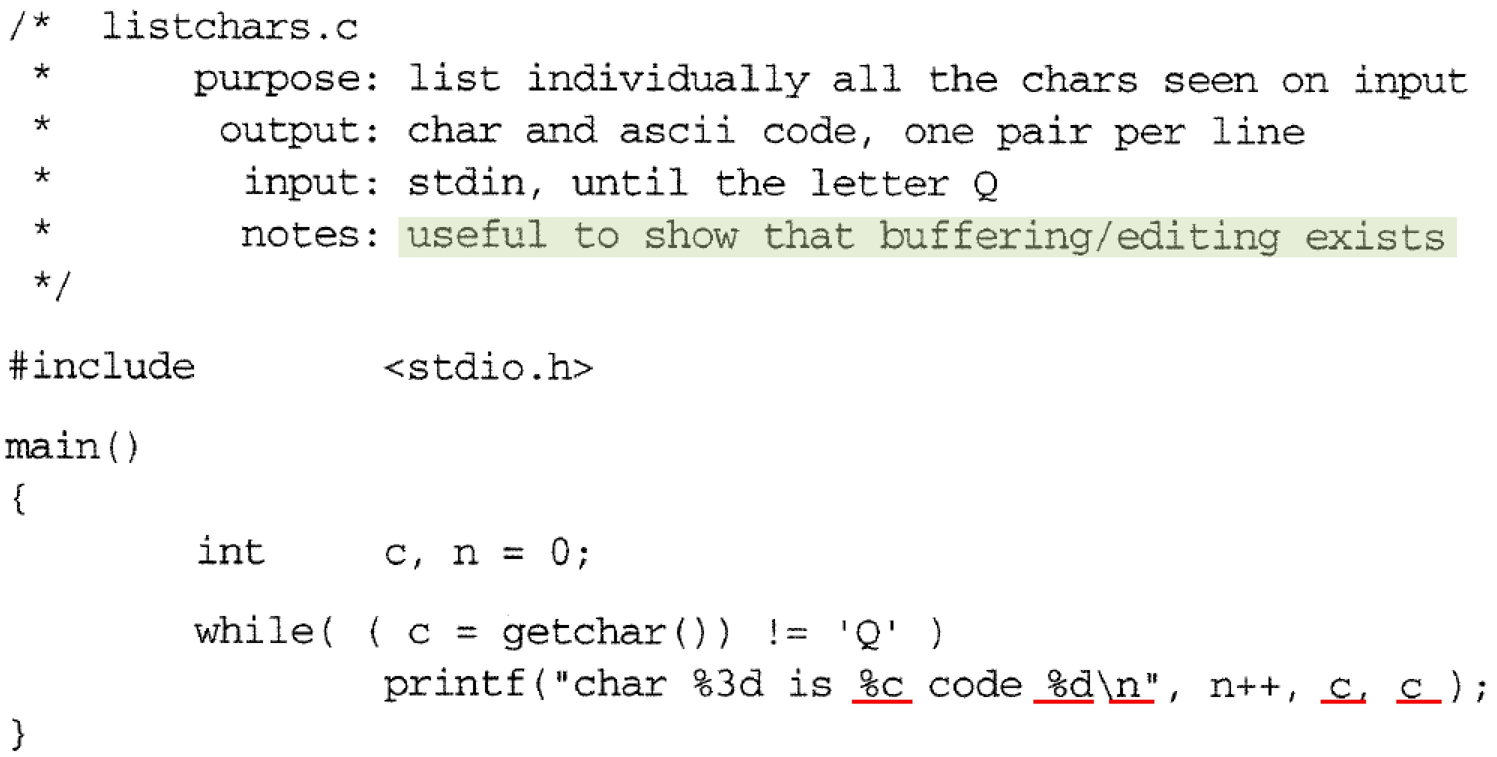
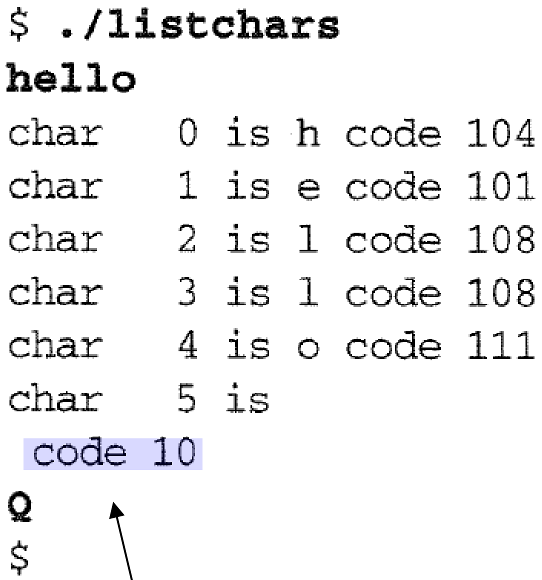


FIGURE 5.8

The illusion of a simple, direct connection.



```
/*  listchars.c
 *      purpose: list individually all the chars seen on input
 *       output: char and ascii code, one pair per line
 *        input: stdin, until the letter Q
 *        notes: useful to show that buffering/editing exists
 */

#include        <stdio.h>

main()
{
        int     c, n = 0;

        while( ( c = getchar()) != 'Q' )
                printf("char %3d is %c code %d\n", n++, c, c );
}
```



```
$ ./listchars
hello
char   0 is h code 104
char   1 is e code 101
char   2 is l code 108
char   3 is l code 108
char   4 is o code 111
char   5 is
 code 10
Q
$
```

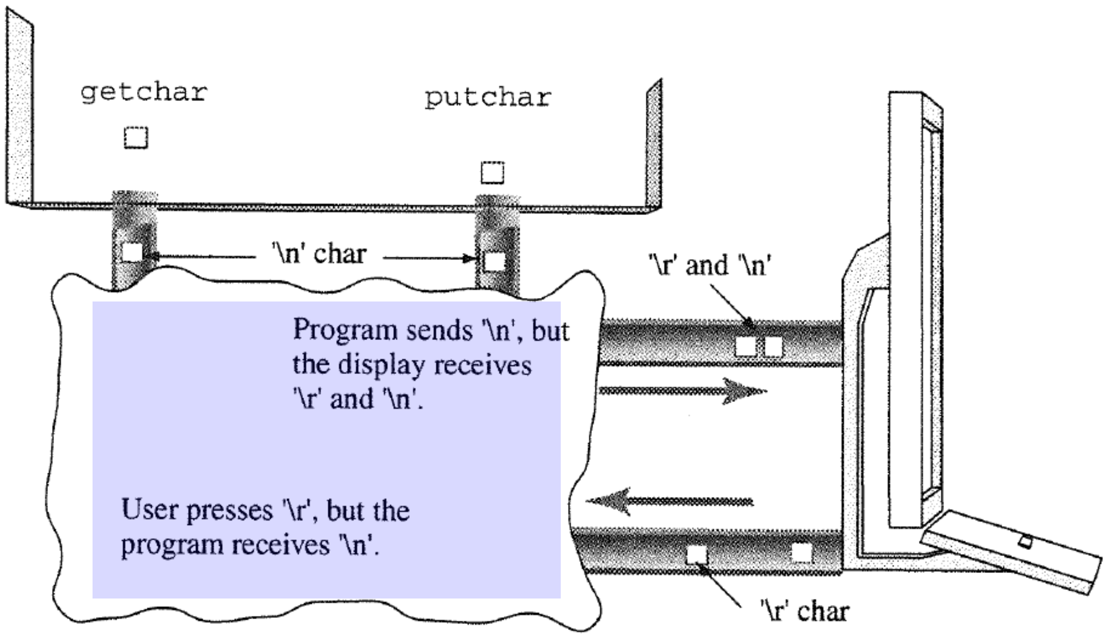

getchar
putchar
'\n' char
'\r' and '\n'
Program sends '\n', but the display receives '\r' and '\n'.
User presses '\r', but the program receives '\n'.
'\r' char

FIGURE 5.9

Kernel processes terminal data.

1. The process receives no data until user presses Return
2. User presses Return (ASCII 13), process sees newline (ASCII 10)
3. Process sends newline, terminal receives Return-Newline pair

코드	이름	의미
13	**Carriage Return (CR)**	커서(쓰기 위치)를 줄 맨 앞으로 이동
10	**Line Feed (LF)**	한 줄 아래로 이동

Terminal (TTY) Driver

- A set of **kernel subroutines** that handle data flow between:
  process ↔ terminal
- It manages:
  - **Input editing**
  - **Echo**
  - **Control characters**
  - **Line buffering,** etc.
- Includes **many settings** (control flags) that define its behavior
- A process can:
  - **Read** current settings
  - **Modify** settings
  - **Reset** to defaults (e.g., with stty sane)

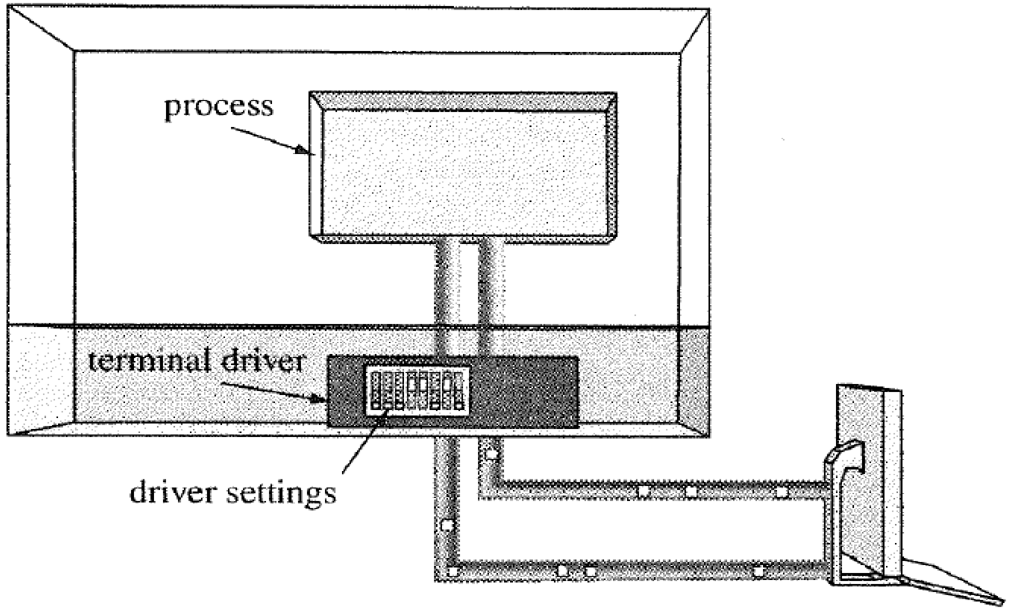


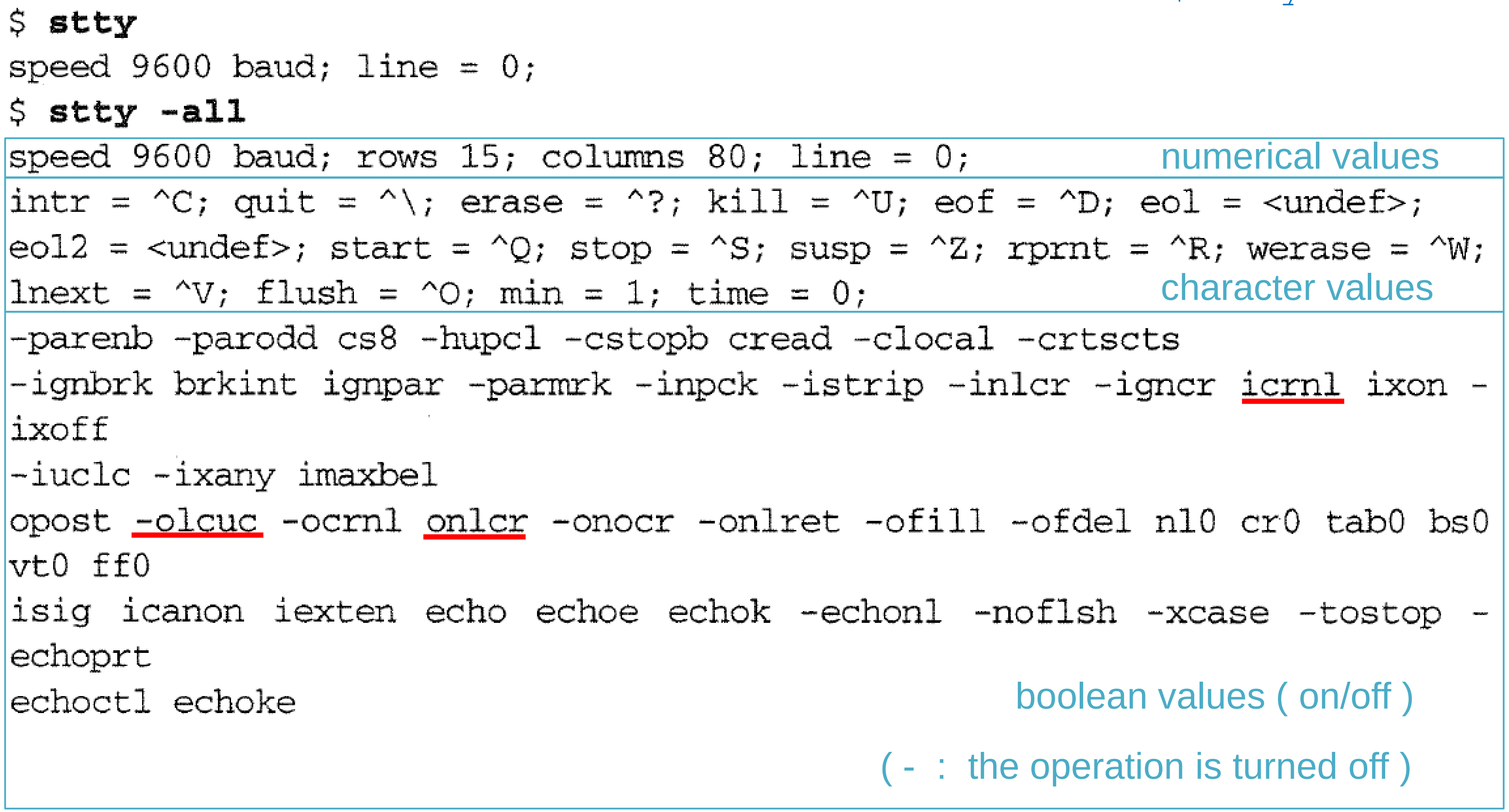
FIGURE 5.10

The terminal driver is part of the kernel.

Command : stty

- Using **stty** to **Display Driver Settings**

$ stty -a



```
$ stty
speed 9600 baud; line = 0;
$ stty -all
speed 9600 baud; rows 15; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -
ixoff
-iuclc -ixany imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0
vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -
echoprt
echoctl echoke
```

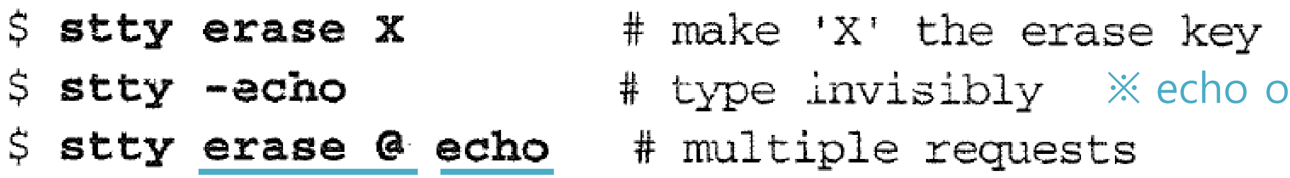
numerical values

character values

boolean values (on/off)

(- : the operation is turned off)

- Using **`stty`** to **Change Driver Settings**



```
$ stty erase X          # make 'X' the erase key
$ stty -echo            # type invisibly
$ stty erase @ echo     # multiple requests
```

※ echo off

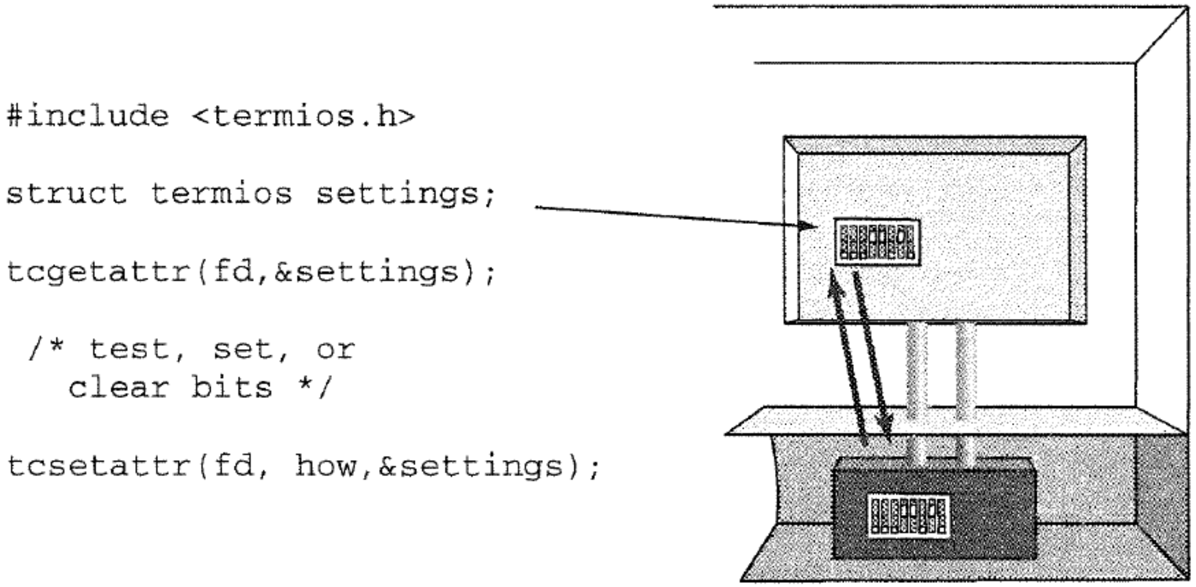
※ echo on

Programming the Terminal Driver: Settings

- **Terminal Driver's Operations:** 4 Categories
  - **Input**
    → What the driver does with characters **coming from the terminal**
  - **Output**
    → What the driver does with characters **going to the terminal**
  - **Control**
    → How characters are **represented** (e.g., bits, parity, stop bits)
  - **Local**
    → What the driver does **internally** while handling characters

Programming the Terminal Driver: Functions

- **Changing Terminal Driver Settings**
  - (a) Get the current attributes from the driver
  - (b) Modify the attributes you want to change
  - (c) Send the updated attributes back to the driver

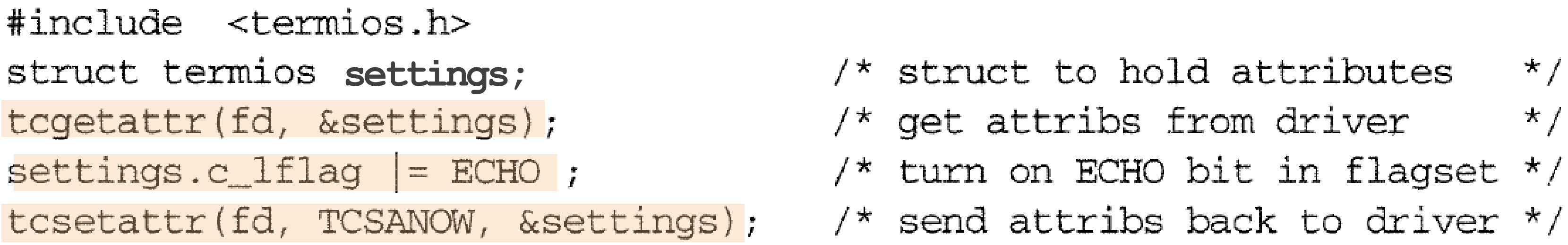


```
#include <termios.h>

struct termios settings;

tcgetattr(fd,&settings);

 /* test, set, or
   clear bits */

tcsetattr(fd, how,&settings);
```

FIGURE 5.11

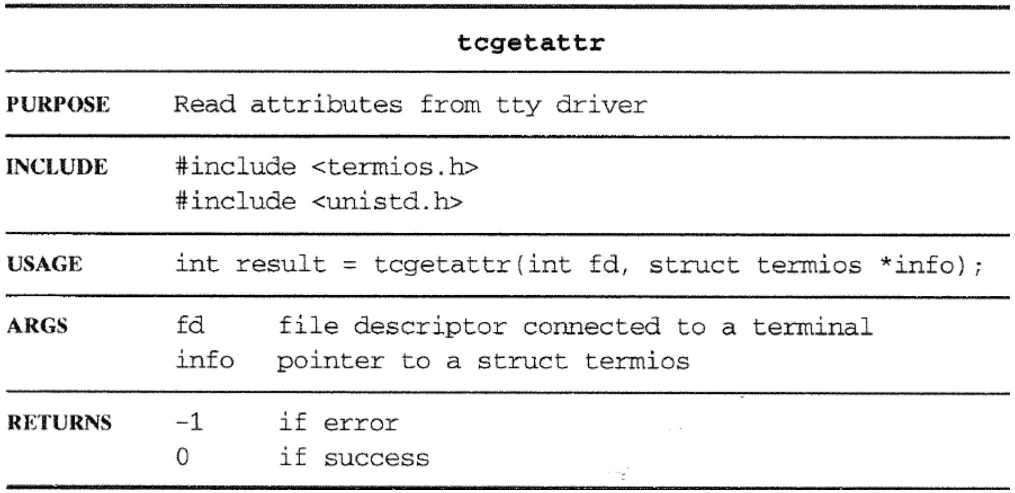
Controlling the terminal driver with tcgetattr and tcsetattr.

Ex: Turning on keystroke echoing



```
#include  <termios.h>
struct termios  settings;                /* struct to hold attributes    */
tcgetattr(fd, &settings);                /* get attribs from driver      */
settings.c_lflag |= ECHO ;               /* turn on ECHO bit in flagset */
tcsetattr(fd, TCSANOW, &settings);       /* send attribs back to driver */
```

TCSANOW
→ **즉시** 속성 변경
TCSADRAIN
→ **출력이 끝난 후** 속성 변경
TCSAFLUSH
→ **출력 완료 후** 속성 변경 + **입력 버퍼 비움**



tcgetattr		
PURPOSE	Read attributes from tty driver	

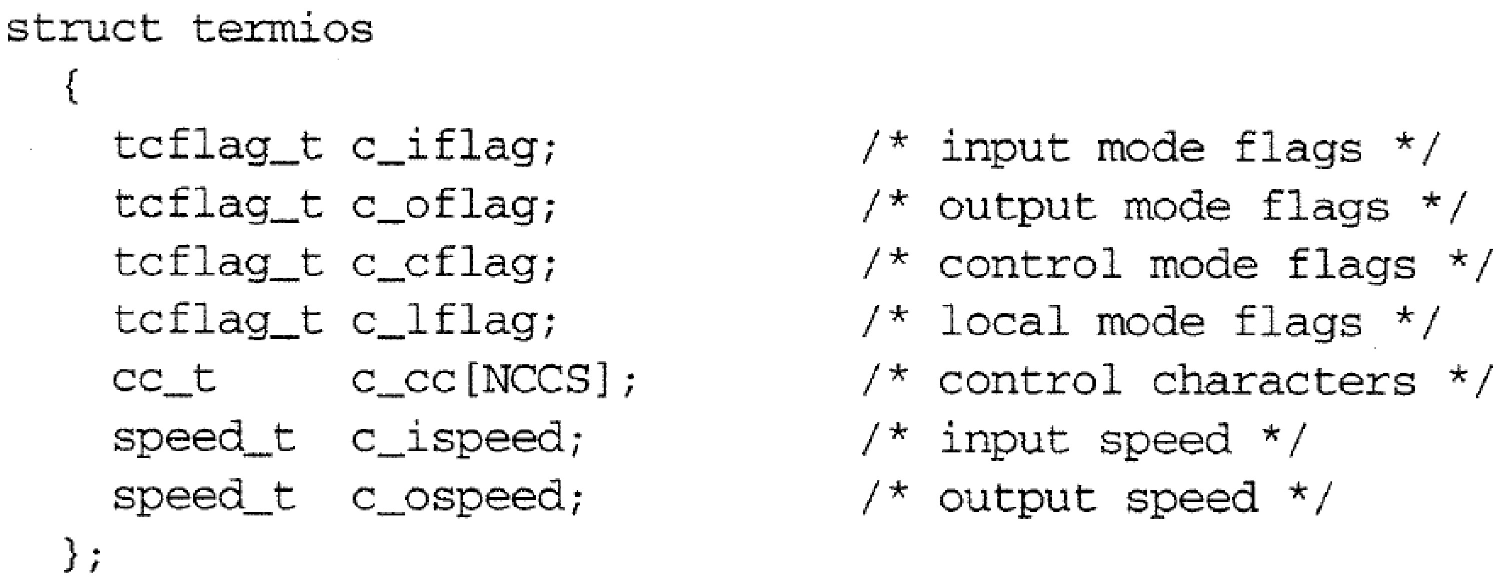
INCLUDE	#include <termios.h>
	#include <unistd.h>	

USAGE	int result = tcgetattr(int fd, struct termios *info);	
ARGS	fd info	file descriptor connected to a terminal pointer to a struct termios

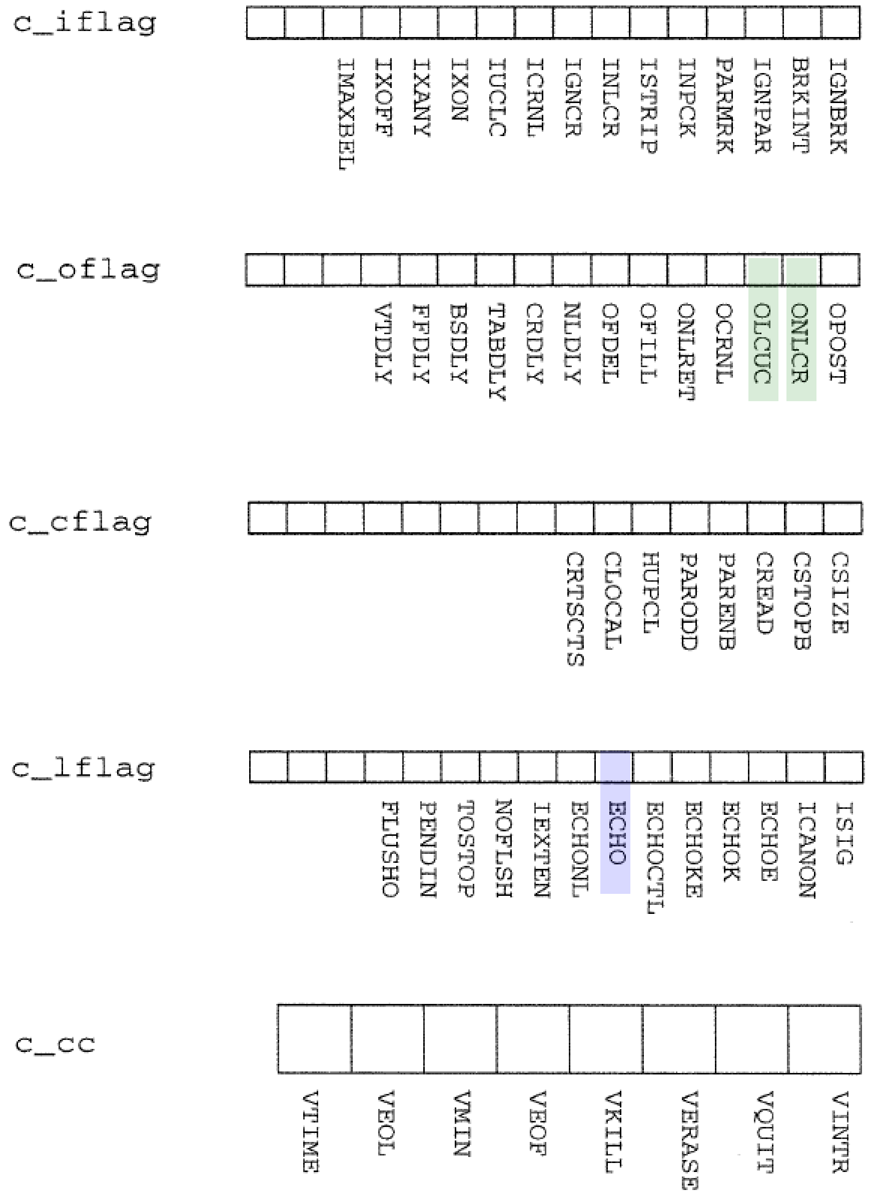
RETURNS	-1
	0
		if error
		if success

Programming the Terminal Driver: Bits

- Data type **struct termios** :
  - Defined in <termios.h>



```
struct termios
  {
    tcflag_t c_iflag;               /* input mode flags */
    tcflag_t c_oflag;               /* output mode flags */
    tcflag_t c_cflag;               /* control mode flags */
    tcflag_t c_lflag;               /* local mode flags */
    cc_t     c_cc[NCCS];            /* control characters */
    speed_t  c_ispeed;              /* input speed */
    speed_t  c_ospeed;              /* output speed */
  };
```

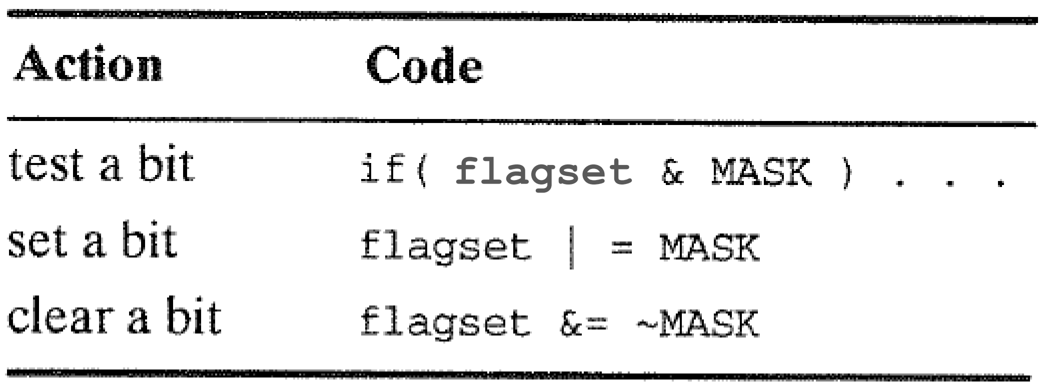


```
#include  <termios.h>
struct termios attribs;
tcgetattr(fd, &settings);
settings.c_lflag  |= ECHO ;
tcsetattr(fd, TCSANOW, &settings);
```

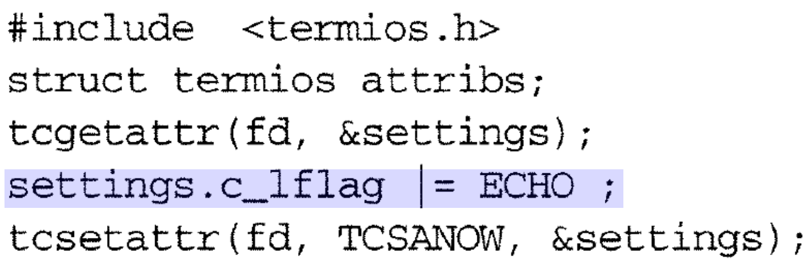
FIGURE 5.12

Bits and chars in termios members.

- **Programming the Terminal Driver: Bit Operations**
  - for modifying driver attributes



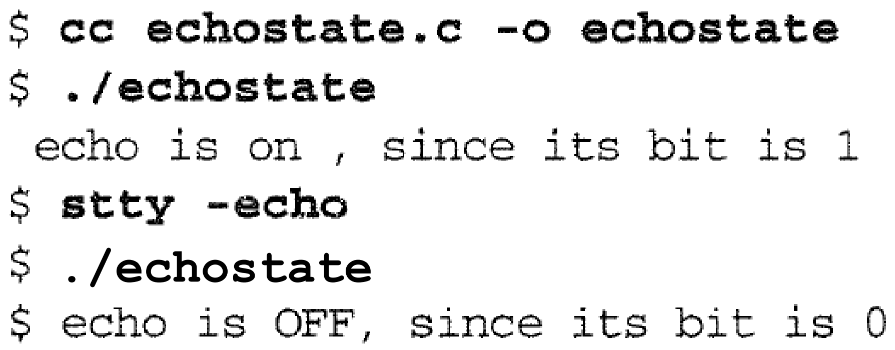
Action	Code
test a bit	`if( flagset & MASK ) . . .`
set a bit	`flagset | = MASK`
clear a bit	`flagset &= ~MASK`



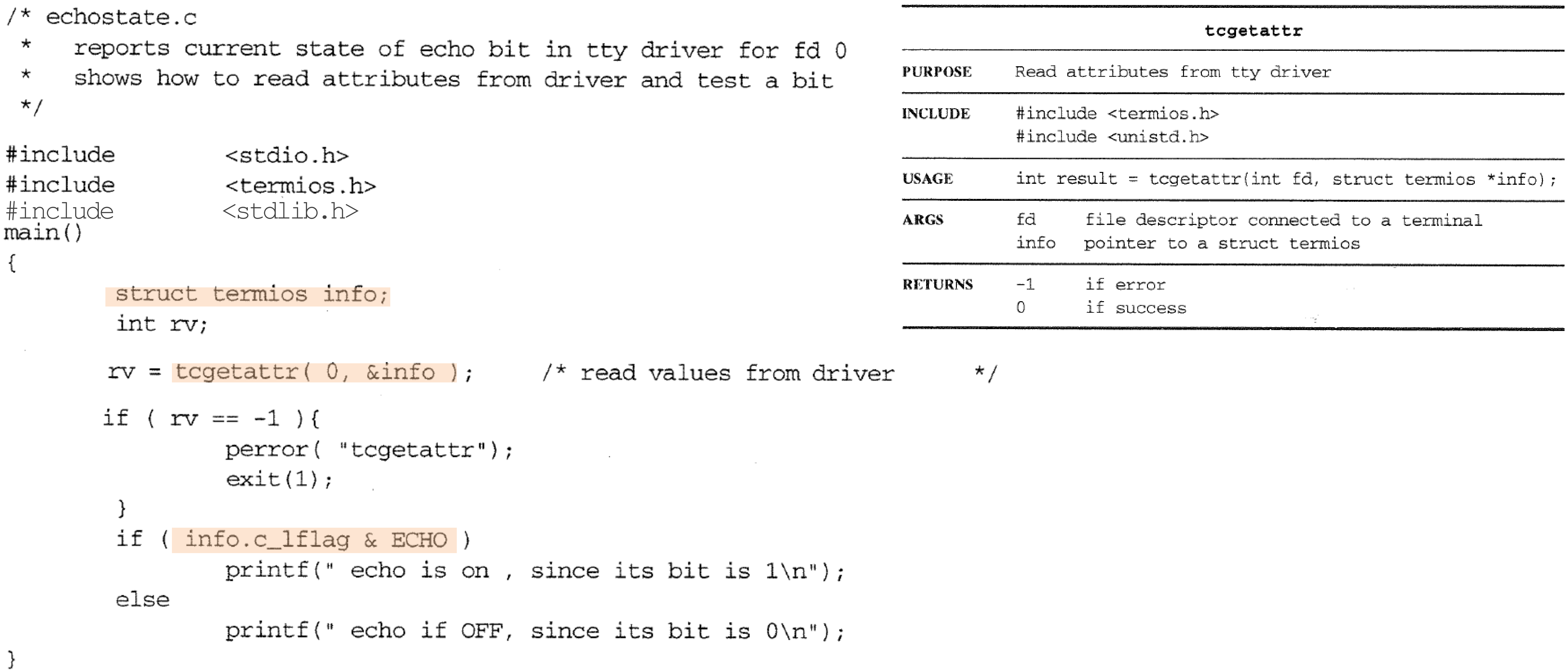
```
#include  <termios.h>
struct termios attribs;
tcgetattr(fd, &settings);
settings.c_lflag |= ECHO ;
tcsetattr(fd, TCSANOW, &settings);
```

Programming the Terminal Driver: Sample Programs

- **Ex: echostate.c -- show state of echo bit**



```
$ cc echostate.c -o echostate
$ ./echostate
 echo is on , since its bit is 1
$ stty -echo
$ ./echostate
$ echo is OFF, since its bit is 0
```



```
/* echostate.c
 *   reports current state of echo bit in tty driver for fd 0
 *   shows how to read attributes from driver and test a bit
 */

#include        <stdio.h>
#include        <termios.h>
#include        <stdlib.h>
main()
{
        struct termios info;
        int rv;

        rv = tcgetattr( 0, &info );     /* read values from driver      */

        if ( rv == -1 ){
                perror( "tcgetattr");
                exit(1);
        }
        if ( info.c_lflag & ECHO )
                printf(" echo is on , since its bit is 1\n");
        else
                printf(" echo if OFF, since its bit is 0\n");
}
```

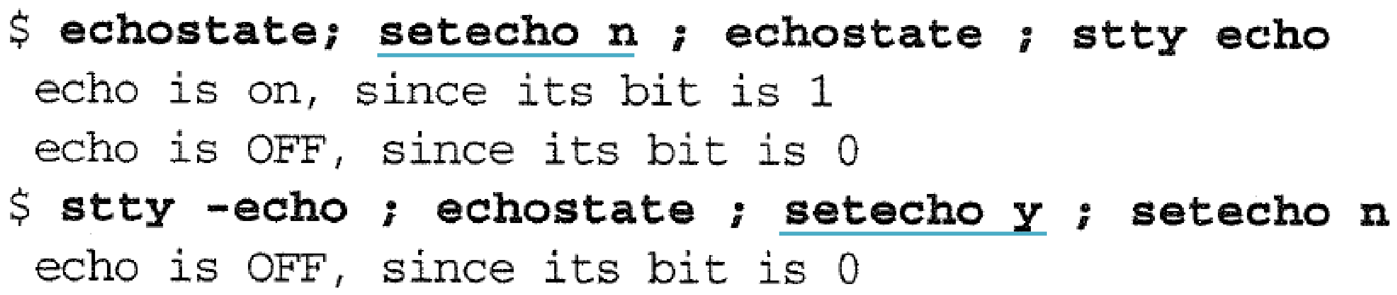
	tcgetattr	
PURPOSE	Read attributes from tty driver	

INCLUDE	#include <termios.h>
	#include <unistd.h>	

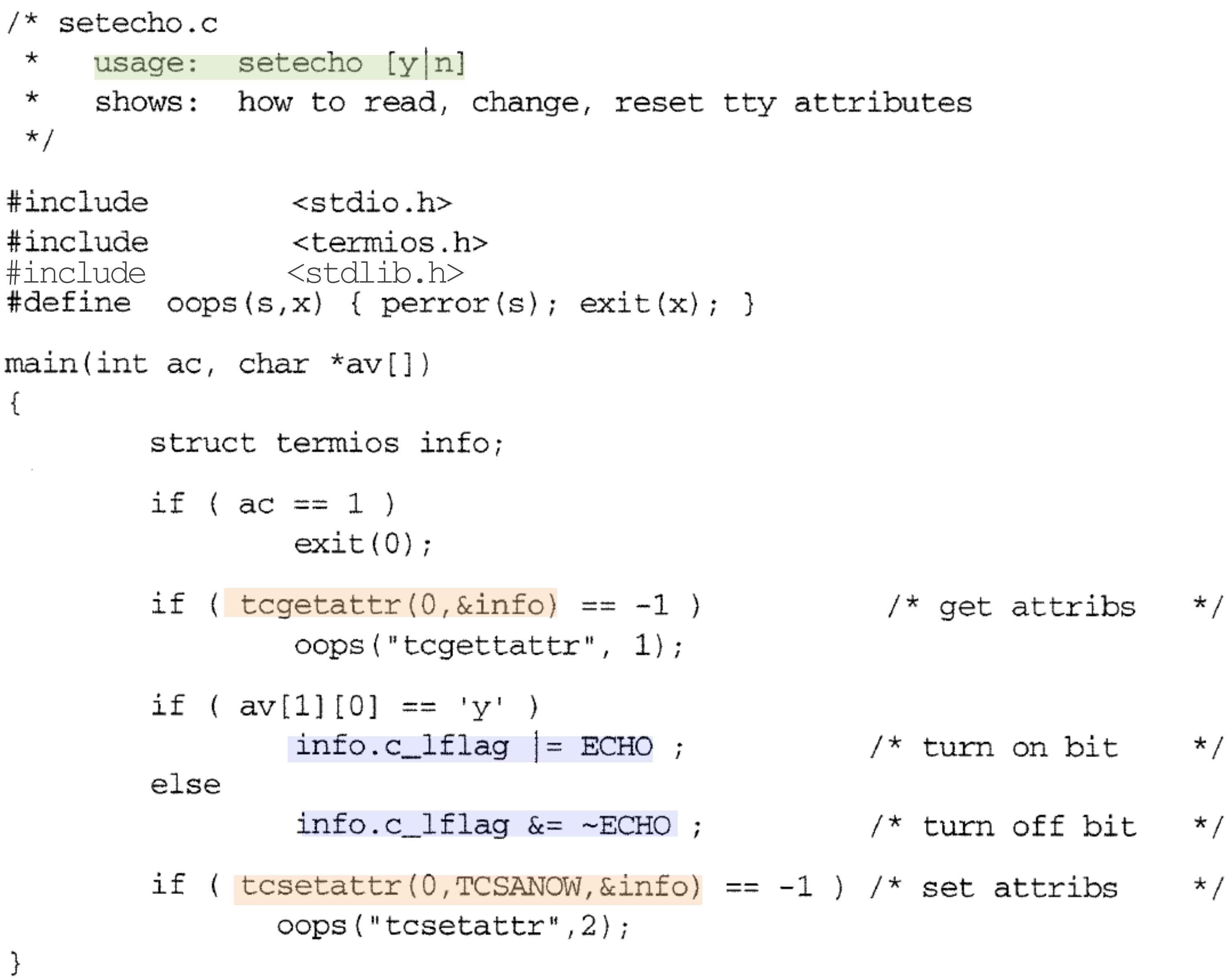
USAGE	int result = tcgetattr(int fd, struct termios *info);	
ARGS	fd info	file descriptor connected to a terminal pointer to a struct termios

RETURNS	-1
	0
		if error
		if success

- **Ex: setecho.c -- change state of echo bit**



```
$ echostate; setecho n ; echostate ; stty echo
 echo is on, since its bit is 1
 echo is OFF, since its bit is 0
$ stty -echo ; echostate ; setecho y ; setecho n
 echo is OFF, since its bit is 0
```



```
/* setecho.c
 *   usage:  setecho [y|n]
 *   shows:  how to read, change, reset tty attributes
 */

#include        <stdio.h>
#include        <termios.h>
#include        <stdlib.h>
#define  oops(s,x) { perror(s); exit(x); }

main(int ac, char *av[])
{
        struct termios info;

        if ( ac == 1 )
                exit(0);

        if ( tcgetattr(0,&info) == -1 )          /* get attribs   */
                oops("tcgettattr", 1);

        if ( av[1][0] == 'y' )
                info.c_lflag |= ECHO ;           /* turn on bit    */
        else
                info.c_lflag &= ~ECHO ;          /* turn off bit   */

        if ( tcsetattr(0,TCSANOW,&info) == -1 ) /* set attribs    */
               oops("tcsetattr",2);
}
```

VISUAL SUMMARY

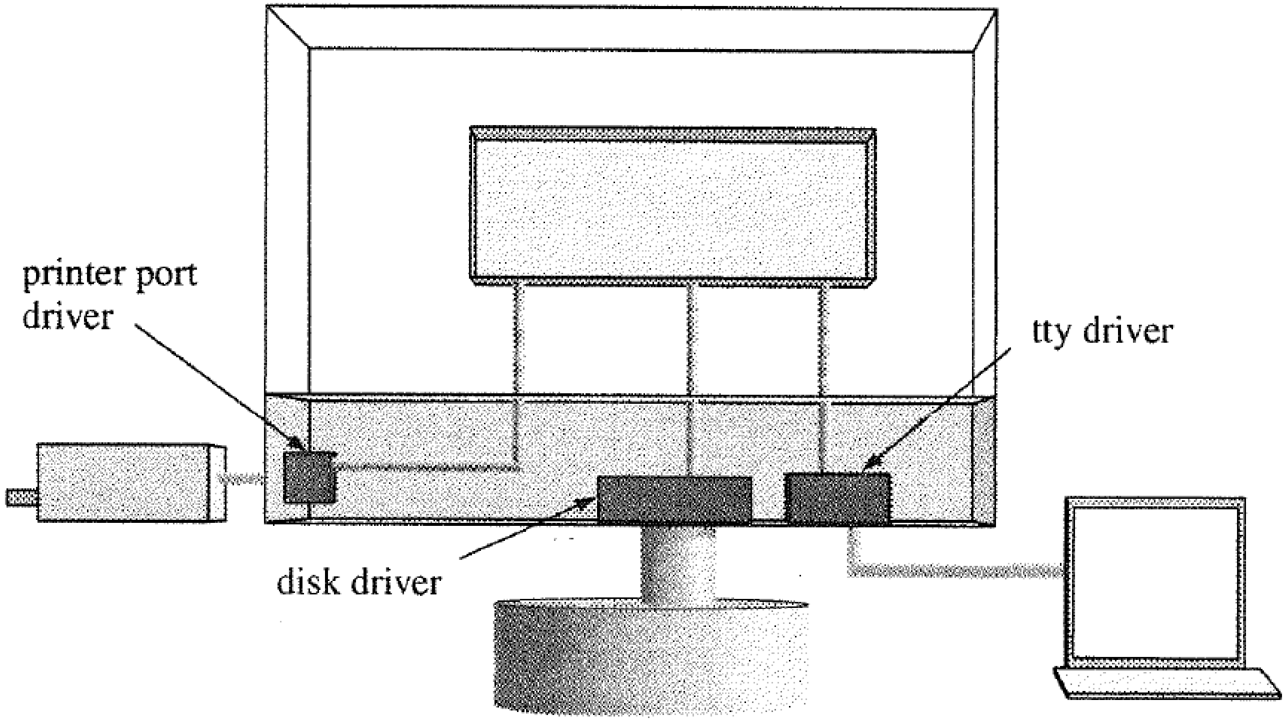


FIGURE 5.13

File descriptors, connections, and drivers.

Connection Control

- **Ideas and Skills**
  - Similarities between files and devices
  - Differences between files and devices
  - Attributes of connections
  - Race conditions and atomic operations
  - Controlling device drivers
  - Streams
- **System Calls and Functions**
  - `fcntl, ioctl`
  - `tcsetattr, tcgetattr`
- **Commands**
  - `stty`
  - `write`