Connection Control
Studying stty

Prof. Seokin Hong

Kyungpook National University

Fall 2018

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

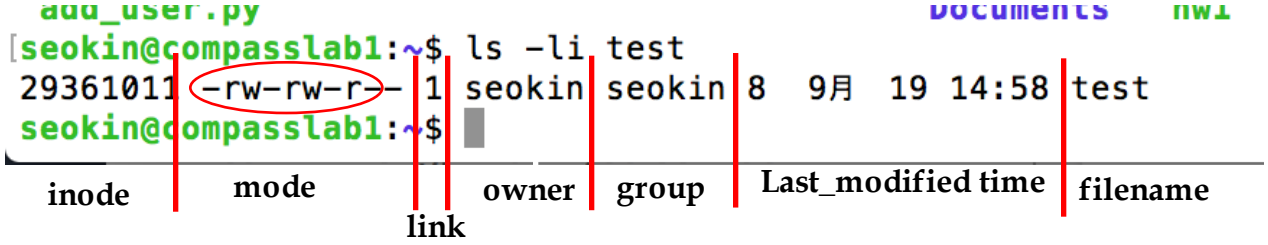
Contents

Devices Are Just Like Files

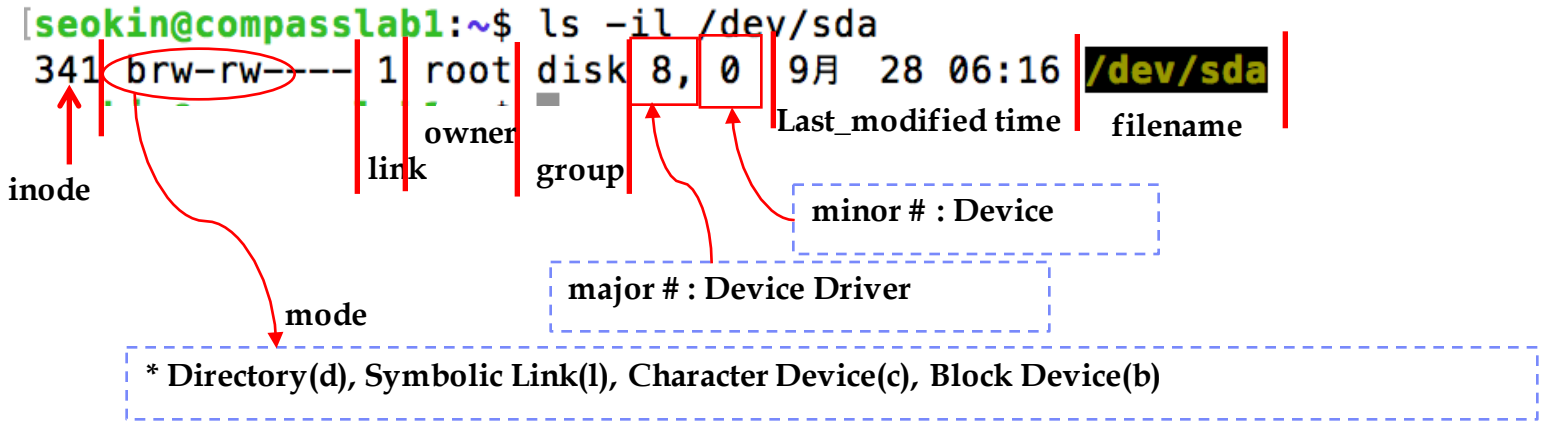
- To Unix, sound cards, terminals, mice, and disk files are the same sort of device object.
- In a Unix system, **every device is treated as a file.**
  - Every device has ..
    - a filename
    - an inode #
    - an owner
    - a set of permission bits
    - and a last-modified time
  - Everything working with files automatically applies to terminals and all other devices.
    - We can control devices with system call
      - open, read, write, lseek, close, stat

Devices Have Filenames

- Files that represent devices are in the **/dev** directory
  - Disk file (regular file)



  - Device file



Devices and System Calls

- Devices support all file-related system calls:
  open, read, write, lseek, close, stat

- Ex) code to read from a disk

```
int fd = open("/dev/sda", O_RDONLY);
lseek(fd, 0x0111, SEEK_SET);
read(fd, buffer, 20);
```

  → read data stored in the hard disk this way while bypassing the filesystem.

- Some devices do not support all file operations
  - /dev/input/mice file does not support the write()
  - Terminals support read() and write(), but they do not support lseek().

Ex: Terminals Are Just Like Files

- A terminal is anything that behaves like the classic keyboard and display unit
  - A telnet or ssh window logged in over the Internet behaves like a terminal.

- Command tty prints the file name of your terminal

- We can use any file commands and operations with that file: cp, >, mv, ln, rm, cat, ls

```
$ tty
/dev/pts/2
$ cp /etc/motd /dev/pts/2
Today is Monday, we are running low on disk space.  Please delete files.
  - your sysadmin

$ 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

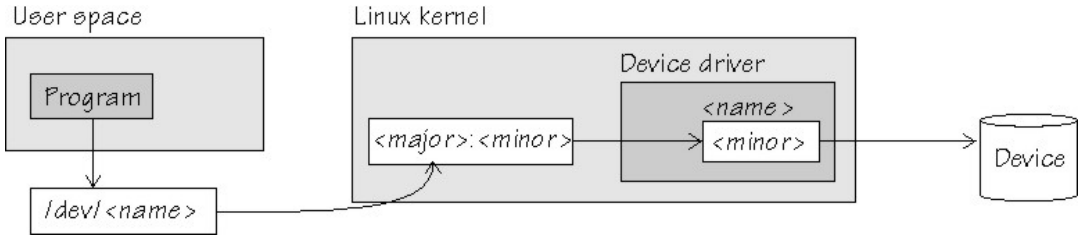
- Device files have most of the properties disk files (regular file) have.

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

file type

major number, minor number

- **Major number and minor number**
  - **The inode of a device file** stores, not a file size and storage list, but a **pointer to a device driver** in the kernel.
    - **Device driver :** a **subroutine in the kernel that gets data into and out of a device**
    - **Major number** specifies which device driver handles the device
    - **Minor number** specifies a particular instance of the device




```
seokin@compasslab1:~$ ls -li /dev/sd*
341 brw-rw---- 1 root disk 8,  0  9月  28 06:16 /dev/sda
345 brw-rw---- 1 root disk 8,  1  9月  28 06:16 /dev/sda1
344 brw-rw---- 1 root disk 8, 16  9月  28 06:16 /dev/sdb
346 brw-rw---- 1 root disk 8, 17  9月  28 06:16 /dev/sdb1
347 brw-rw---- 1 root disk 8, 18  9月  28 06:16 /dev/sdb2
348 brw-rw---- 1 root disk 8, 19  9月  28 06:16 /dev/sdb3
349 brw-rw---- 1 root disk 8, 20  9月  28 06:16 /dev/sdb4
350 brw-rw---- 1 root disk 8, 21  9月  28 06:16 /dev/sdb5
```

Properties of Device Files

- Permission Bits
  - What do permission bits mean when the file is really a device?
  - In the following example,
    - Writing permission means that the owner of the device file and members of group tty are allowed to write to the terminal
    - Reading permission means only owner of the device file can read any terminal messages

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

Writing `write`

```
$ write test /dev/pts/2
```

- `$ 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 );
}
```

```
$ cc write0.c -o write0
$ ./write0 /dev/pts/2
Is it working?...
Bye
^c
```

fgets - get a string from a stream

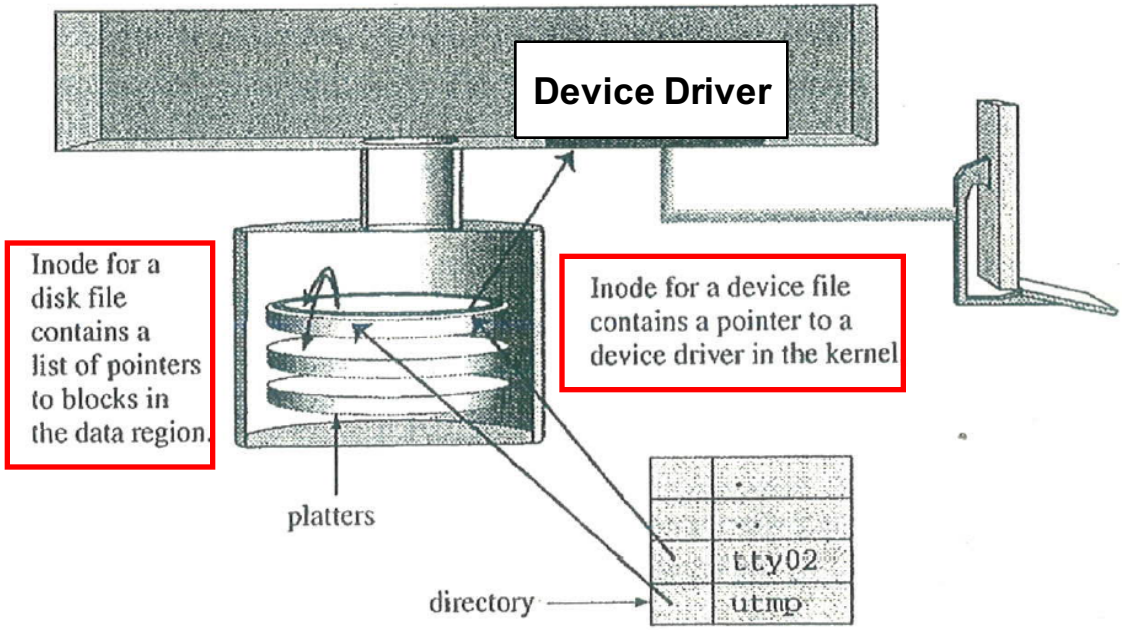
SYNOPSIS
#include <stdio.h>

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

DESCRIPTION
The *fgets*() function shall read bytes from *stream* into the array pointed to by *s*, until *n*-1 bytes are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null byte.

Device Files and Inodes

- The distinction between types of files appears at the inode level, not in a directory
- **inode** can be a disk file inode or a device-file inode
  - A disk-file inode contains a pointer to the data blocks
  - A device-file inode contains a pointer into a table of kernel subroutines called device drivers



Device Files and Inodes

- How **read()** works?
  - Kernel finds the inode for the file descriptor.
    - inode tells the kernel the type of the file
    - How?
      - type of inode is recorded int the type portion of the st_mode member of struc stat
  - If the file is a disk file,
    - the kernel gets data by consulting the block allocation list
  - If the file is a device file,
    - the kernel reads data by calling the read part of the device driver code for that device.

Contents

Devices Are Not Like Files

- Connection to a disk file is different from a connection to a terminal
  - Connection to a disk file usually involves kernel buffers. Buffering is an attribute of the connection
  - Connection to terminal send data to terminal pretty quickly
- We can control attributes in a different way.

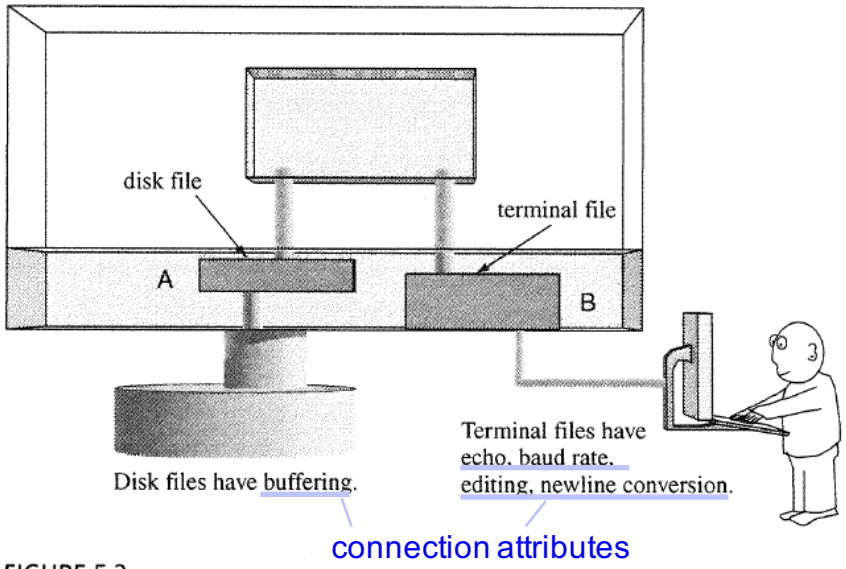
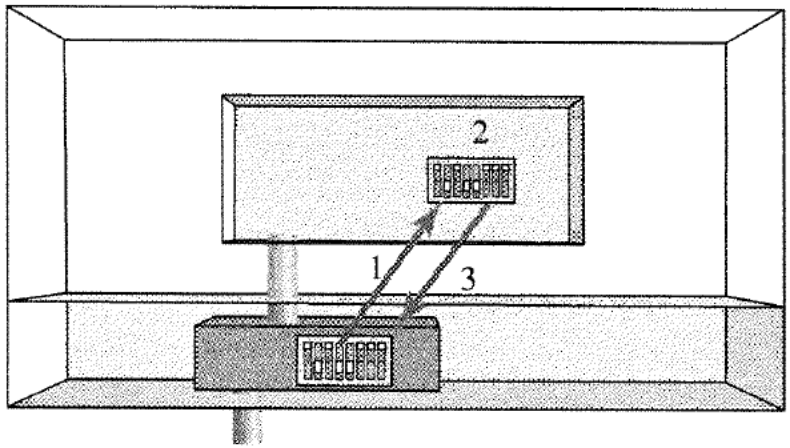


FIGURE 5.2

A process with two file descriptors.

Controlling the connection attributes

- You modify the device attributes by changing control variables of **the file descriptor** associated to the device.
- Three steps
  - Get setting
  - Modify them
  - Send them back



To change driver settings:
1. Get settings,
2. modify them
3. send them back.

FIGURE 5.4

Modifying the operation of a file descriptor.

Controlling the connection attributes

- System call: fcntl

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

Controlling the attributes of disk connection

- Ex1: 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
```

  - O_SYNC : tells the kernel that calls to write() should be return only when the bytes are written to the actual hardware

Controlling the attributes of disk connection

- Ex2: Turning on auto-append mode
  - With auto-append mode, each call to write() automatically includes an lseek to the end of file

```
#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
```

Controlling the attributes with open

- open() lets you specify fd attribute bits as part of its second argument

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

Summary of disk connections

- The kernel transfers data between disks and processes
- The code in the kernel that performs these transfers has many options
- A program can use the open() and fcntl() system calls to control the attributes of these data transfers

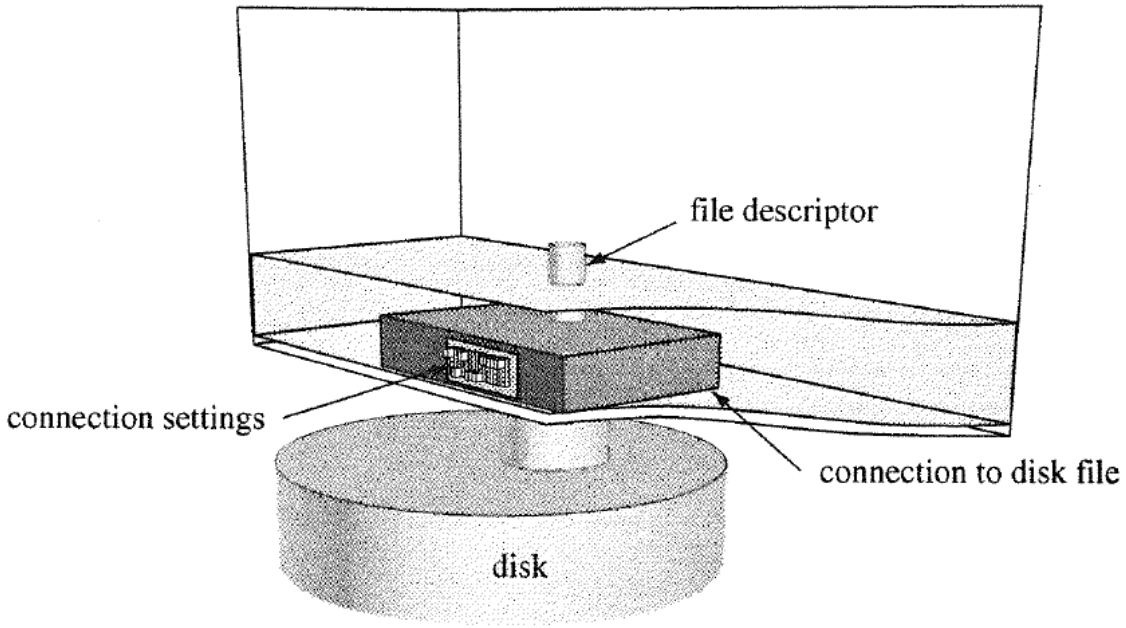


FIGURE 5.7

Connections to files have settings.

Terminal connection

- **Terminal Driver**
  - The collection of kernel subroutines that process data flowing between a process and a terminal (external device)
  - The driver contains many settings that control its operation
  - A process may read, modify, and reset those driver control flags.

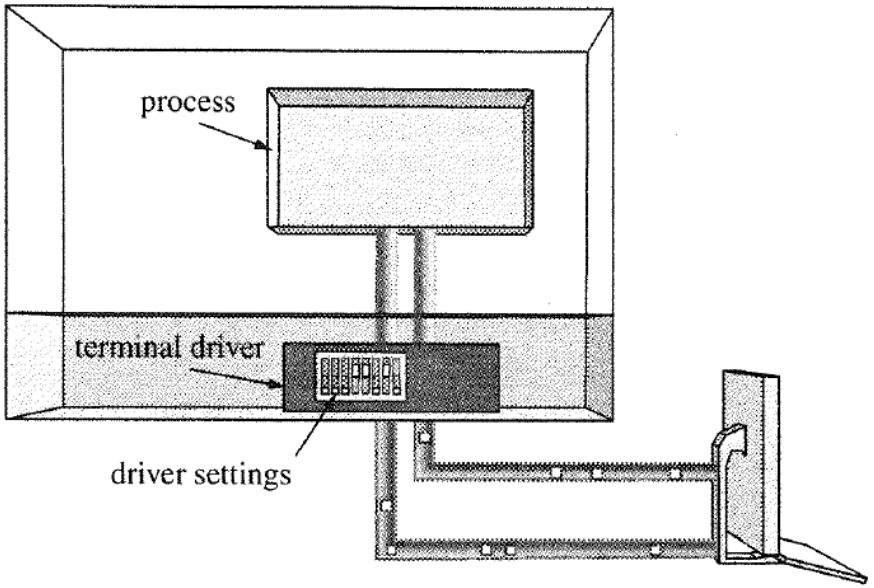


FIGURE 5.10

The terminal driver is part of the kernel.

Controlling the attributes of terminal connection

- Using stty to Display Driver Settings

```
seokin2@compasslab1:~$ stty
speed 9600 baud; line = 0;
eol = M-^?; eol2 = M-^?;
-brkint ixany iutf8
-echok
```

- Using stty to Change Driver Settings

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

※ echo on

Controlling the attributes of terminal connection

- Changing setting in the terminal driver

(a) Get the attributes from the driver.
(b) Modify any attributes you need to change.
(c) Send those revised attributes back to the driver.

- Ex1: 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 */
```

Ex1: Turning on keystroke echoing

- tcgetattr() library function

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 file descriptor connected to a terminal info pointer to a struct termios
RETURNS	-1 if error 0 if success

Ex1: Turning on keystroke echoing

- tcgetattr() library function

tcsetattr		
PURPOSE	Set attributes in tty driver	
INCLUDE	#include <termios.h> #include <unistd.h>	
USAGE	int result = tcsetattr(int fd, int when,struct termios *info);	
ARGS	fd	file descriptor connected to a terminal
	when	when to change the settings
	info	pointer to a struct termios
RETURNS	-1	if error
	0	if success

- **TCSANOW** : update driver setting immediately
- **TCSADRAIN** : wait until all output already queued in the driver has been transmitted to the terminal
- **TCSAFLUSH** : wait until all output already queued in the driver has been transmitted. Next, flush all queued input data

Ex1: Turning on keystroke echoing

- The **struct termios** data type contains several flagsets and an array of control characters

```
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 */
  };
```

Ex1: Turning on keystroke echoing

- Bits and chars in **struct termios** members

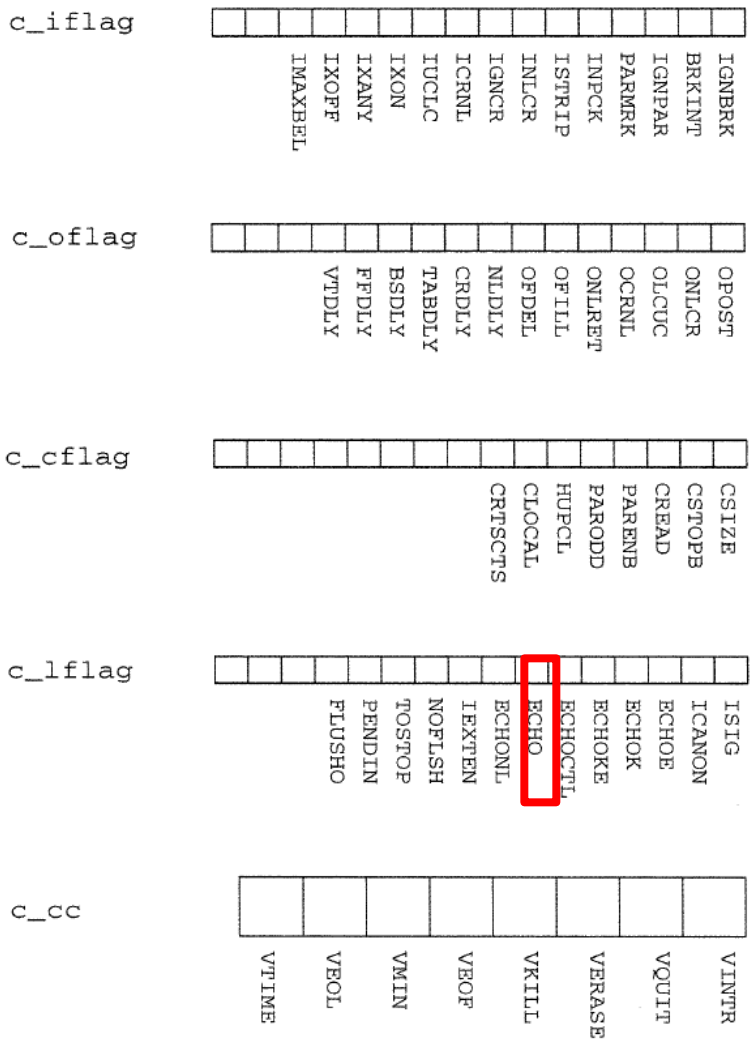


FIGURE 5.12

Bits and chars in termios members.

Ex2: Reporting the current state of the terminal

```
/* 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;
        //'fd = 0' indicates standard input which will be keyboard in the terminal
        rv = tcgetattr( 0, &info );      /* read values from driver      */

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

Ex2: Reporting the current state of the terminal

- Compile and test echostate.c

```
$ 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
```

Ex2: Turning on/off echo mode

```
/* 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);
        //'fd = 0' indicates standard input which will be keyboard in the terminal
        if ( tcgetattr(0,&info) == -1 )          /* get attribs   */
                oops("tcgettattr", 1);

        if ( av[1][0] == 'y' )
                info.c_lflag |= ECHO ;          /* turn on bit    */
        else                                            //Set a bit
                info.c_lflag &= ~ECHO ;         /* turn off bit   */
                                                        //Clear a bit
        if ( tcsetattr(0,TCSANOW,&info) == -1 ) /* set attribs    */
                oops("tcsetattr",2);
}
```

Ex2: Turning on/off echo mode

- Compile and test setecho.c

```
$ 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
```

Contents

Programming Other Devices: ioctl

- What about connections to other types of devices?
  - Ex) CD recorders, Scanners,..

- Every device file supports the ioctl()system call
  - The ioctl() provides access to the attributes and operations of the device driver to fd.
  - Each type of device has its own set of properties and ioctl operations.
    - Ex) A video terminal screen has a size measured in rows and columns or in pixels

```
#include <sys/ioctl.h>

void print_screen_dimensions()
{
    struct winsize wbuf;

    if ( ioctl(0, TIOCGWINSZ, &wbuf) != -1 ){
        printf("%d rows x %d cols\n", wbuf.ws_row, wbuf.ws_col);
        printf("%d wide x %d tall\n", wbuf.ws_xpixel, wbuf.ws_ypixel);
    }
}
```

TIOCGWINGZ is the function code

Programming Other Devices: ioctl

ioctl		
PURPOSE	Control a Device	
INCLUDE	# include <sys/ioctl.h>	
USAGE	int result = ioctl (int fd, int operation [, arg..])	
ARGS	fd	file descriptor connected to device
	operation	operation to perform
	arg...	any args required for the operation
RETURNS	-1	if error
	other	depends on device

VISUAL SUMMARY

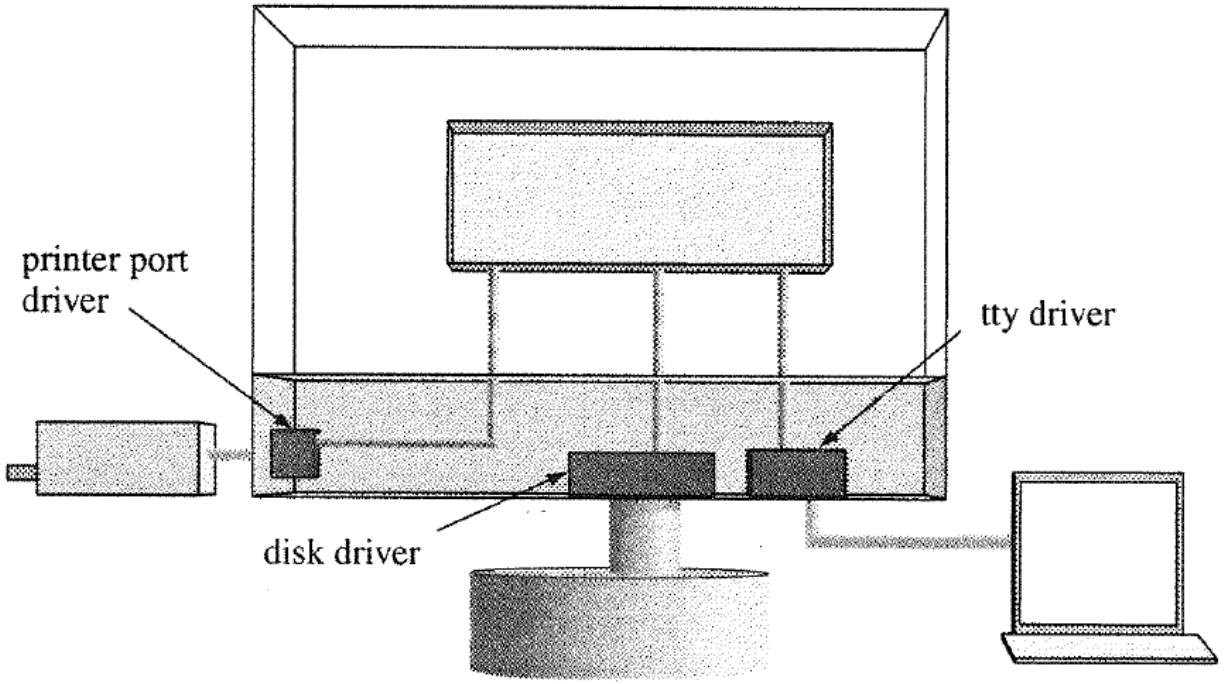


FIGURE 5.13

File descriptors, connections, and drivers.