Objectives

- Ideas and Skills
  - Difference 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

Devices are treated as files

In Unix, all peripheral devices are treated as files. Every device has a filename, an inode number, an owner, a set of permissions, and last-modified time.

```
lrwxrwxrwx 1 root root          3 Apr  9 09:20 cdrw-hdc -> hdc
lrwxrwxrwx 1 root root          3 Apr  9 09:20 cdwriter -> hdc
lrwxrwxrwx 1 root root          3 Apr  9 09:20 cdwriter-hdc -> hdc
crw------- 1 wch  root     5,   1 Apr  9 09:20 console
lrwxrwxrwx 1 root root         11 Apr  9 09:19 core -> /proc/kcore
drwxr-xr-x 6 root root        120 Apr  9 09:20 disk
lrwxrwxrwx 1 root root          3 Apr  9 09:20 dvd -> hdd
lrwxrwxrwx 1 root root          3 Apr  9 09:20 dvd-hdd -> hdd
lrwxrwxrwx 1 root root         13 Apr  9 09:19 fd -> /proc/self/fd
brw-rw---- 1 wch  floppy   2,   0 Apr  9 09:20 fd0
brw-rw---- 1 wch  floppy   2,  84 Apr  9 09:20 fd0u1040
lrwxrwxrwx 1 root root          3 Apr  9 09:20 floppy-fd0 -> fd0
crw-rw-rw- 1 root root     1,   7 Apr  9 09:20 full
srwx------ 1 wch  root          0 Apr  9 09:20 gpmctl
brw-r----- 1 root disk     3,   0 Apr  9 05:19 hda
brw-r----- 1 root disk     3,   1 Apr  9 05:19 hda1
brw-r----- 1 root disk     3,   2 Apr  9 05:19 hda2
brw-r----- 1 root disk     3,   3 Apr  9 09:20 hda3
```

Devices and System calls

Since devices are treated as files, they also accept file-related system calls. For example

```
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, bufflen);        /*read data from tape    */
close(fd);                         /*disconnect             */
```

However, some devices does not support all file operations

- Mouse does not support write
- Terminal does not support lseek

Examples of mouse and terminal accessing

- To print the file name of the terminal connected to standard input, use tty command.

```
$ tty
/dev/pts/3
$cp /etc/fstab  /dev/pts/3
LABEL=/                 /                       ext3    defaults        1 1
LABEL=/boot             /boot                   ext3    defaults        1 2
/dev/devpts             /dev/pts                devpts  gid=5,mode=620  0 0
/dev/shm                /dev/shm                tmpfs   defaults        0 0
LABEL=/home             /home                   ext3    defaults        1 2
/dev/proc               /proc                   proc    defaults        0 0
/dev/sys                /sys                    sysfs   defaults        0 0
/dev/hda7               swap                    swap    defaults        0 0
/dev/hda8               /mnt/windows            vfat    users,rw,noauto 0 0
```

- `$ cat /dev/input/mice`

Properties of device files

```
$ls -li /dev/pts/3
5 crw--w---- 1 wch tty 136, 3 Apr  9 12:26 /dev/pts/3
```

- Here 136 is the major number and 3 is the minor number. Major number specifies which subroutine handles the actual device and the minor number is passed to that subroutine.
- The permission bits allows the owner to read and write to a terminal. The other user can only write to the terminal.

Sample development–write

```
#man write
NAME
       write - send a message to another user

SYNOPSIS
       write user [ttyname]

DESCRIPTION
       Write allows you to communicate with other users, by copying lines from
       your terminal to theirs.

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

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

       Any further lines you enter will be copied to the specified users ter-
       minal.  If the other user wants to reply, they must run write as  well.

       When  you  are  done,  type an end-of-file or interrupt character.  The
       other user will see the message EOF indicating that the conversation is
       over.
```

Sample development–write

```
/* 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>

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 );
}
```

Device files and Inodes

- The difference between a device file and an disk file resides in the Inode.
  - Inode for a device file contains a pointer to a device driver in the kernel
  - Inode for a disk file contains a list of pointers to blocks in the data region
- The type of an Inode is recorded in the type portion of the `st_mode` member of `struct stat`. When listed by `ls -l`, device files begin with either **b** or **c**. **b** stands for block device, such as disks, **c** stands for character devices, such as printers and speakers etc.
- The major number of a device file tells where to get the subroutine for the device. The minor number is the parameter for the subroutine.

- A connection to disk file usually involves kernel buffers. A Connection to devices such as a terminal or modem has attributes. For example, a serial connection has a baud rate, party, and number of stop bits.

Examples of Device files

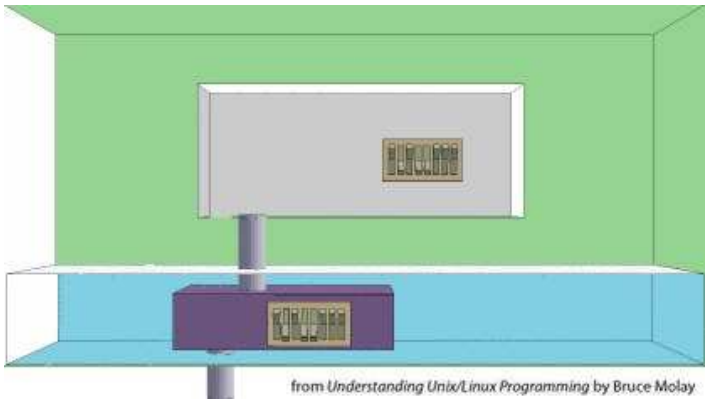
A device file represents an entity, for example

- An actual device, such as a line printer, a speaker and a modem.
- A logical subdevice, such as a large section of the disk drive
- A pseudo device, such as the physical memory of the computer (/dev/mem) or the null file (/dev/null).

Attention

Data corruption, loss of data, or loss of system integrity (a system crash) will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices on the operating system and are solely for system use in managing file systems, paging devices, and logical volumes. These files should not be used for other purposes.

Attributes of disk connections



To change driver settings

- Get settings
- modify them
- send them back

fcntl system call

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 (F_GETFL or F_SETFL)
	arg	arguments to the operation
	lock	lock information
Returns	-1	if error
	other	depends on operation

File access mode (O_RDONLY, O_WRONLY, O_RDWR) and file creation flags (e.g.,O_CREAT, O_EXCL, O_TRUNC) in arg are ignored. On Linux, this command can only change the O_APPEND, O_ASYNC, O_DIRECT, O_NOATIME, and O_NONBLOCK flags.

To get the detailed description about each arguments, read the manpage of open

```
man open
```

Example of changing attributes of disk connections by fcntl

```
/*set the O_SYNC flag*/
#include <fcntl.h>
int s;
s = fcntl(fd, F_GETFL);
s |= O_SYNC;
result = fcntl(fd, F_SETFL, s);
if (result == -1)
   perror("setting SYNC");
```

```
/*set the O_APPEND flag*/
#include <fcntl.h>

int s;
s = fcntl(fd, F_GETFL);
s |= O_APPEND;
result = fcntl(fd, F_SETFL, s);
if (result == -1)
   perror("setting APPEND");
else
   write(fd, &rec, 1);
```

O_APPEND avoids the race condition when a file is opened for writing by several processes. When the O_APPEND bit is set for a file descriptor, each call to write automatically includes an lseek to the end of the file.

Controlling file descriptors with open, create

```
$man 2 open

NAME
       open, creat - open and possibly create a file or device

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

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

The parameter flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. In addition, zero or more of the following may be bitwise-ord in flags:O_APPEND, O_ASYNC, O_CREAT, O_DIRECT,O_DIRECTORY, O_EXCL, O_LARGEFILE, O_NOATIME, O_NOCTTY, O_NOFOLLOW, O_NONBLOCK or O_NDELAY, O_SYNC, O_TRUNC.

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

Attributes of terminal connection

Let's look at a simple program

```
/*  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: usesful 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 );
}
```

Attributes of terminal connection

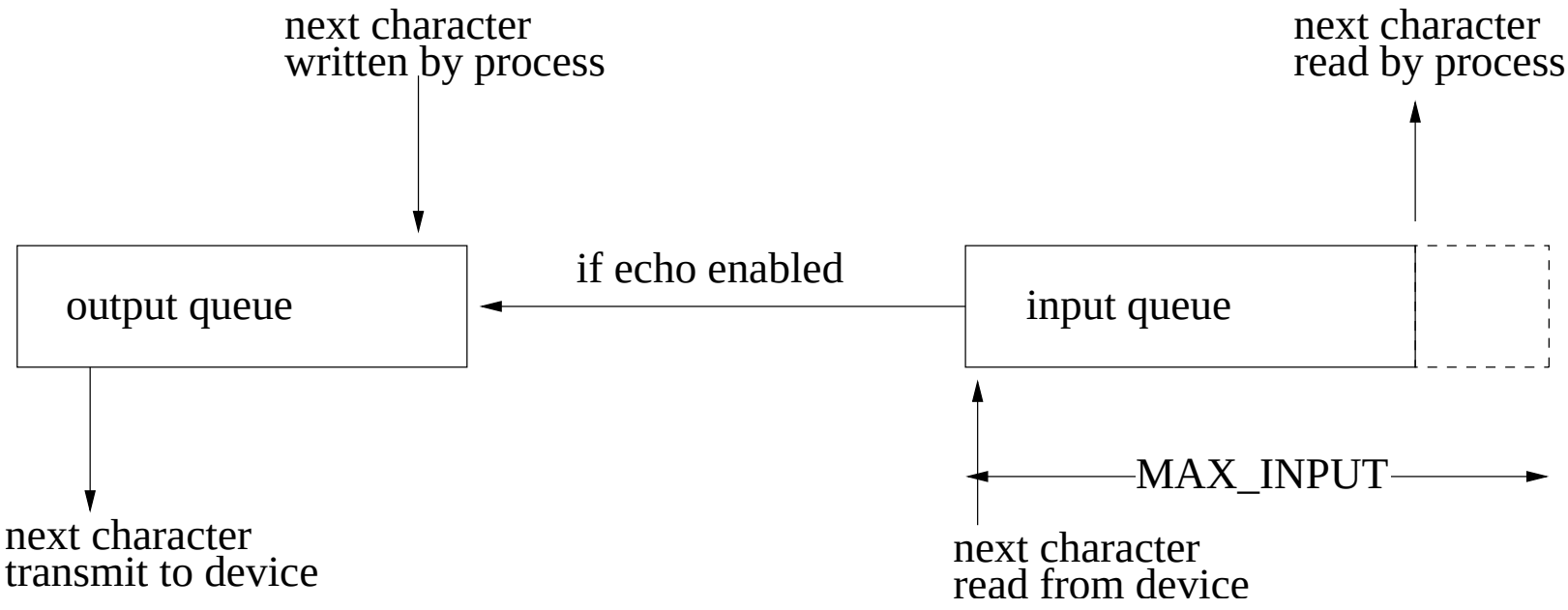
The output of this program looks like the following:

```
[wch@localhost bookcode]$ ./listchars
kdiekigmg
char   0 is k code 107
char   1 is d code 100
char   2 is i code 105
char   3 is e code 101
char   4 is k code 107
char   5 is i code 105
char   6 is g code 103
char   7 is m code 109
char   8 is g code 103
char   9 is
 code 10
```

From the above example, we can know

- The process receives no data until user presses Return.
- User press Return (ASCII 13), process sees newline (ASCII 10)
- Process sends the new line, terminal receives Return-NewLine pair (\r\n).

Logical picture of input and output queues for a terminal device

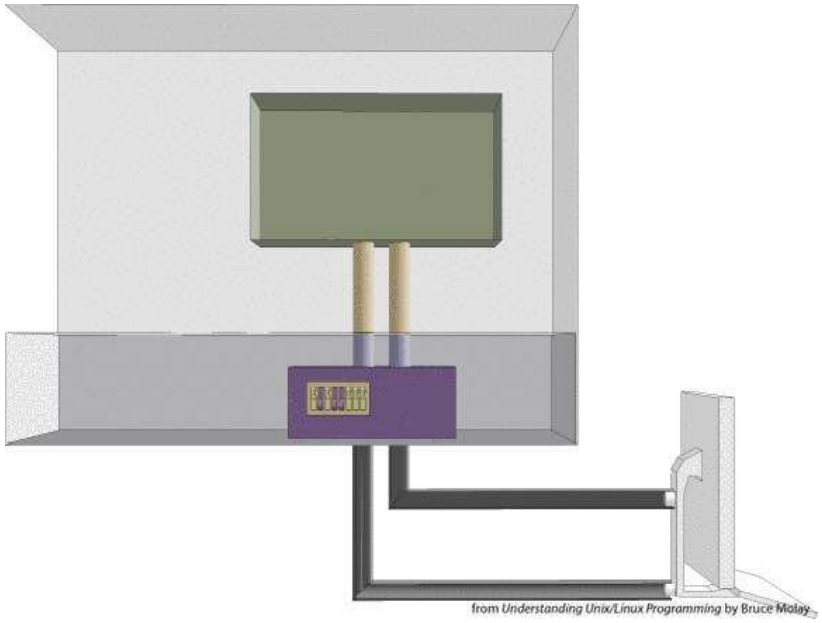


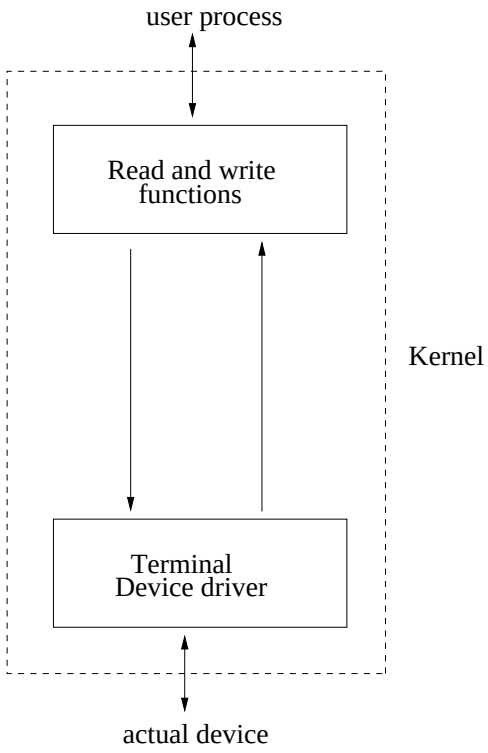
Things to remember about the Input and Output Queues:

- There is a link between the input and output queue if echoing is enabled. This means you don't need to send your keystrokes to stdout if echoing is enabled, it is done for you.

- There is a limit to the size of the input queue, this limit is defined by the macro MAX_INPUT. This means that if you try to type a line that is larger than MAX_INPUT, then the terminal device will not read anything beyond the MAX_INPUT number of characters.
- There is an output queue, but you never really need to worry about this. If a process tries to write info to the output queue and the queue is filled up, the kernel will put that process to sleep (it will block), until there is more room on the queue. This limit is not defined in any standard header file.
- Most of the processing of the input queue on Unix systems takes place in a module called the terminal driver.

Terminal driver





A terminal driver processes the data flow between a process and the external device. The driver contains many settings that control its operation. A process may read, modify and reset those driver control flags via stty command.

The stty command

```
[wch@localhost figs]$ stty -a
speed 38400 baud; rows 32; columns 76; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?;
eol2 = M-^?; swtch = M-^?; 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 -iutf8
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
```

- Using stty to change driver settings

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

For a complete list of the meaning of each setting, check the man page of stty.

Programming the terminal driver: the settings

The operation of the terminal driver can be grouped into four categories:

- **input**: what the driver does with characters coming from the terminal, including converting lowercase to uppercase, stripping off the high bit, and converting carriage returns to newlines
- **output**: what the driver does with characters going to the terminal, including replacing tab characters by sequences of spaces, converting newlines to carriage returns, and converting lowcase to uppercase.
- **control**: how characters are represented—number of bits, parity, stop bits, etc.
- **local**: what the driver does while characters are inside the driver, including echoing keystrokes back to the user and buffering input until the user presses Return.

Programming the terminal driver: the functions

Three steps to change the settings in terminal driver

- Get the attributes from the driver
- Modify any attributes you need to change
- Send those revised attributes back to the driver

For example, to turn on keystroke echoing for a connection

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

tcgetattr

Purpose	Read attributes from tty driver	
Include	#include ⟨termios.h⟩	
	#include 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

tcsetattr

Purpose	Set attributes from tty driver	
Include	#include ⟨termios.h⟩ #include 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 sucess

The values allowed for when are as follows:

- TCSANOW: Update driver settings immediately
- TCSADRAIN: Wait until all output already queued in the driver has been transmitted to the terminal. The update the driver
- TCSAFLUSH: the change occurs after all output written to the terminal has been transmitted, and all input that has been received but not read will be discarded before the change is made.

termios struct

The struct termios, defined in /usr/include/termios.h, contains several flagsets and an array of control characters. The following members are defined on all versions of Unix.

```
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;     /*ouput speed*/
};
```

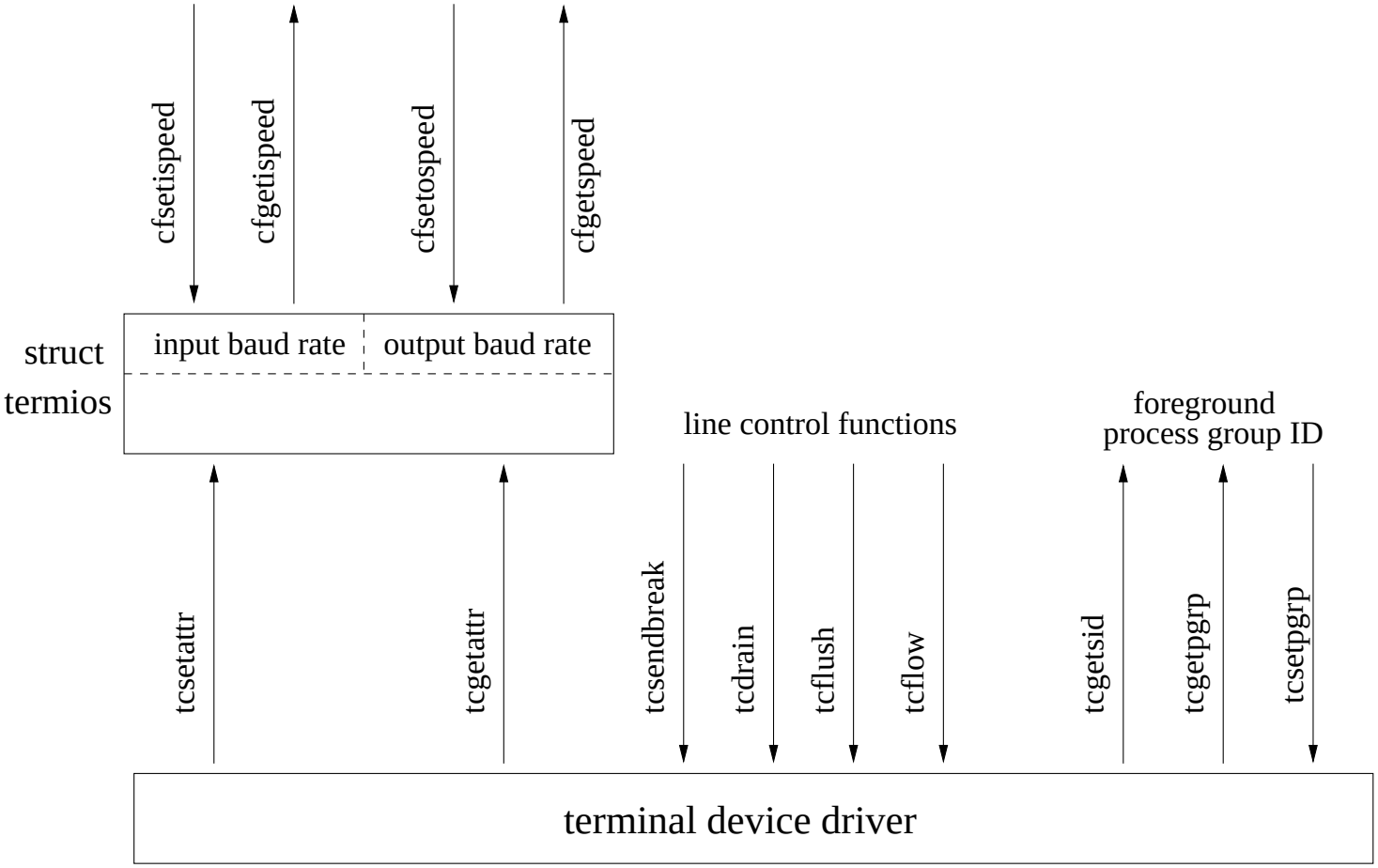
Type man termios for detailed description about termois.

```
man termios
```

Summary of terminal I/O functions

Function	Description
tcgetattr	fetch attributes (termios structure)
tcsetattr	set attributes (termios structure)
cfgetispeed	get input speed
cfgetospeed	get output speed
cfsetispeed	set input speed
cfsetospeed	set output speed
tcdrain	wait for all ouput to be transmitted
tcflow	suspend transmit or receive
tcflush	flush pending input and/or output
tcsendbreak	send BREAK character
tcgetpgrp	get foreground process group ID
tcsetpgrp	set foreground process group ID
tcgetsid	get process group ID of session leader for controlling TTY

Summary of terminal I/O functions



Terminal identification

/*return pointer to the name of controlling terminal on success
* pointer to empty string on error
*/

```
#include <stdio.h>
char *ctermid(char *ptr);
```

/*return true if terminal device, false otherwise*/

```
#include <stdio.h>
int isatty(int fd);
```

/*return a pointer to pathname of terminal, NULL on error*/

```
#include <stdio.h>
char *ttyname(int fd);
```

The following code is an example for obtaining the name of the control terminal of the current process. Upon success, ptr shall point to a string that holds the name of the terminal.

```
#include <stdio.h>
...
char term[L_ctermid];
char *ptr;


ptr = ctermid(term);
```

Bit operation

The individual bits in each flagset of termios structure are listed on page 158 of the textbook.

Action	**Code**
test a bit	if (flagset & MASK)
set a bit	flagset |= MASK
clear a bit	flagset &= ∼MASK

Sample programs

Example: echostate.c–show the state of echo bit

```
/* 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>
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");
}
```

Sample programs

```
[wch@localhost bookcode]$ ./echostate
 echo is on , since its bit is 1
[wch@localhost bookcode]$ stty -echo
[wch@localhost bookcode]$  echo if OFF, since its bit is 0
```

Sample programs

Example: setecho.c–change the state of echo bit

```
/* setecho.c
 *   usage:  setecho [y|n]
 *   shows:  how to read, change, reset tty attributes
 */
#include        <stdio.h>
#include        <termios.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);
}
```

An example - getpass function

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

#define MAX_PASS_LEN   8        /* max #chars for user to enter */

char * getpass(const char *prompt)
{
        static char buf[MAX_PASS_LEN + 1];       /* null byte at end */
        char *ptr;
        sigset_t  sig, osig;
        struct termios ts, ots;
        FILE  *fp;
        int c;

        if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
                return(NULL);
        setbuf(fp, NULL);
        //set the signal mask to block some signals
        sigemptyset(&sig);
        sigaddset(&sig, SIGINT);                /* block SIGINT */
        sigaddset(&sig, SIGTSTP);               /* block SIGTSTP */
        sigprocmask(SIG_BLOCK, &sig, &osig);    /* and save mask */
```

```
        tcgetattr(fileno(fp), &ts);             /* get tty state */
        ots = ts;                               /* structure copy */
        ts.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
        tcsetattr(fileno(fp), TCSAFLUSH, &ts);  /* set tty state */
        fputs(prompt, fp);

        ptr = buf;
        while ((c = getc(fp)) != EOF && c != '\n')
                if (ptr < &buf[MAX_PASS_LEN])
                        *ptr++ = c;
        *ptr = 0;                       /* null terminate */
        putc('\n', fp);                 /* we echo a newline */

        tcsetattr(fileno(fp), TCSAFLUSH, &ots); /* restore TTY state */
        sigprocmask(SIG_SETMASK, &osig, NULL);  /* restore mask */
        fclose(fp);       /* done with /dev/tty */
        return(buf);
}
```

Sample programs

Example: showtty.c–display many driver attributes

```
/*  showtty.c
*       displays some current tty settings
*/

#include        <stdio.h>
#include        <termios.h>

main()
{
        struct  termios ttyinfo; /* this struct holds tty info */

        if ( tcgetattr( 0 , &ttyinfo ) == -1 ){    /* get info */
                perror( "cannot get params about stdin");
                exit(1);
        }
                                                   /* show info */
        showbaud ( cfgetospeed( &ttyinfo ) );      /* get + show baud rate */
        printf("The erase character is ascii %d, Ctrl-%c\n",
                        ttyinfo.c_cc[VERASE], ttyinfo.c_cc[VERASE]-1+'A');
        printf("The line kill character is ascii %d, Ctrl-%c\n",
                        ttyinfo.c_cc[VKILL], ttyinfo.c_cc[VKILL]-1+'A');

        show_some_flags( &ttyinfo );               /* show misc. flags      */
}
```

```
showbaud( int thespeed )
/*
 *        prints the speed in english
 */
{
        printf("the baud rate is ");
        switch ( thespeed ){
                case B300:      printf("300\n");        break;
                case B600:      printf("600\n");        break;
                case B1200:     printf("1200\n");       break;
                case B1800:     printf("1800\n");       break;
                case B2400:     printf("2400\n");       break;
                case B4800:     printf("4800\n");       break;
                case B9600:     printf("9600\n");       break;
                default:        printf("Fast\n");       break;
        }
}

struct flaginfo { int    fl_value; char  *fl_name; };

struct flaginfo input_flags[] = {

        IGNBRK  ,       "Ignore break condition",
        BRKINT  ,       "Signal interrupt on break",
        IGNPAR  ,       "Ignore chars with parity errors",
        PARMRK  ,       "Mark parity errors",
        INPCK   ,       "Enable input parity check",
```

```
        ISTRIP   ,      "Strip character",
        INLCR    ,      "Map NL to CR on input",
        IGNCR    ,      "Ignore CR",
        ICRNL    ,      "Map CR to NL on input",
        IXON     ,      "Enable start/stop output control",
        /* IXANY   ,    "enable any char to restart output",    */
        IXOFF    ,      "Enable start/stop input control",
        0        ,      NULL };

struct flaginfo local_flags[] = {
        ISIG     ,      "Enable signals",
        ICANON   ,      "Canonical input (erase and kill)",
        /* XCASE        ,      "Canonical upper/lower appearance", */
        ECHO     ,      "Enable echo",
        ECHOE    ,      "Echo ERASE as BS-SPACE-BS",
        ECHOK    ,      "Echo KILL by starting new line",
        0        ,      NULL };

show_some_flags( struct termios *ttyp )
/*
 *      show the values of two of the flag sets: c_iflag and c_lflag
 *      adding c_oflag and c_cflag is pretty routine - just add new
 *      tables above and a bit more code below.
 */
{
        show_flagset( ttyp->c_iflag, input_flags );
        show_flagset( ttyp->c_lflag, local_flags );
}
```

```
show_flagset( int thevalue, struct flaginfo thebitnames[] )
/*
 * check each bit pattern and display descriptive title
 */
{
        int       i;

        for ( i=0; thebitnames[i].fl_value ; i++ ) {
                printf( "  %s is ", thebitnames[i].fl_name);
                if ( thevalue & thebitnames[i].fl_value )
                        printf("ON\n");
                else
                        printf("OFF\n");
        }
}
```

Programming with other devices: 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 the device
	operation	operation to perform
	arg...	any args required for the operation
Returns	-1	if error
	other	depends on device

The ioctl system call provides access to the attributes and operations of the device driver connected to fd. **Each type of devices has its own set of properties and ioctl operations**.

Difference between fcntl and ioctl

- fcntl is intened for performing some general operations given a file descriptor.
- ioctl is for device-specific operations. ioctl is customizable. A driver writer can define some device specific tasks & hook her own method to execute them via an ioctl