Overview:

Why should we need to know about terminal I/O with all these nifty graphical user interfaces out there? Terminals are thing of the past, right? Sorry, in Unix, the terminal device driver controls a lot more than just terminals. modems, printers, direct connections to other computers, and other special devices that rely on streams of characters.

Terminal devices can be put into different states. The default state of the terminal when running in a shell is canonical mode, also known as cooked mode. In this mode, the terminal driver returns one line of data at a time from the terminal device. Any special characters are processed as they come into the device (ˆC, ˆZ, etc.).

The second state the terminal can be in is noncanonical mode, or raw mode. In this state, the terminal device driver returns one character at a time without assembling lines of data. Also, special characters are not processed in this mode. Programs such as vi, pine, and elm use this mode for data input and output. This allows complete control of input and output characters.

A third state, one which Posix.1 defines, is the cbreak mode. This mode is similar to raw mode, except that the processing of special characters still takes place and the corresponding signals are raised for the special characters.

P.326 Figure Here

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. Most Unix systems will echo the bell character (ˆG) when you try to type beyond this limit and the characters you type will not be echo'd or stored in the queue.
- The limit MAX_CANON is the maximum number of bytes that can be stored in an input line. This is the same as MAX_INPUT on many systems, including Linux and HPUX.
- 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 line discipline*. This takes place between the system functions and the device driver.

P. 327 Figure Here

Getting and Setting the Terminal Attributes:

All of the attributes that can be controlled in the terminal device are contained in the termios structure. This structure is defined as:

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

The c_iglag attribute is what controls any input characteristics of the terminal (map CR to NL, ring bell on input queue full, etc.). The c_oflag attribute is what you set to control any output processing of the terminal (expand tabs to spaces, map lowercase to uppercase on output, etc.). Most of the c_oflag settings are not Posix compliant. The c_cflag attribute is for setting the serial line attributes (enable parity, set flow control, etc.). The c_lflag attribute is for the settigns of the interface between the user and the device driver (local echo, enable signals generated byt the terminal, etc.).

This structure is used with two different functions, tcgetattr() and tcsetattr(). The prototypes are as follows:

```
    #include <termios.h>

    int tcgetattr(int filedes, struct termios *termptr);
```

int tcsetattr(int filedes, int opt, const struct termios *termptr);
Both return: 0 if OK, -1 on error

As the names suggest, tcgetattr() gets the current state of the terminal that the open file descriptor *filedes* points to, and tcsetattr() sets attributes for the terminal that *filedes* is associated with. These functions will return an error if the *filedes* argument is not associated with a terminal device.

The argument *opt* in tcsetattr() is for specifying when the changes are to take place. This is defined by the following macros:

: ———

TCSANOW Make the changes now. TCSADRAIN Make the changes after all output has been transmitted from the buffer. This should be used when setting output attributes. TCSAFLUSH Make the changes after all output has been transmitted, and flush the input queue of any unprocessed data. ———

How fast are we talking?

Sometimes you may find that you need to change the speed of the terminal dsession to match that of the device it is connected to. This is done with four functions in combination with the tcgetattr() and tcsetattr() functions.

```
    #include <termios.h>

    speed_t cfgetispeed(const struct termios *termptr);

    speed_t cfgetospeed(const struct termios *termptr);
```
Both return: baud rate value
```
    speed_t cfsetispeed(struct termios *termptr, speed_t speed);

    speed_t cfsetospeed(struct termios *termptr, speed_t speed);
```
Both return: 0 if OK, -1 on error

The first thing that must be done here in order to change the baud rate of the terminal is use tcgetattr() so that you can pass the termios struct to the cfset functions. You then pass the struct to the cfset functions to set the correct baud rate in the termios struct. This does not actually set the terminal speed, however. You still need to make a call to tcsetattr() with termios struct that has the changed baud rate.

1. tcgetattr() -- Get the current settings
2. cfsetispeed() -- Set the input speed in the termios struct
3. cfsetospeed() -- Set the output speed in the termios struct
4. cfsetattr() -- Make the changes to the terminal to reflect the changed struct

Terminal line control

The line control for the terminal is important if you want to prevent overflowing the buffer for the device when there is no hardware flow control implemented. Also, you can flush the input and/or output of a device discarding any data that has not already been sent or read from the buffer.

```
    #include <termios.h>

    int tcdrain(int filedes);

    int tcflow(int filedes, int action);

    int tcflush(int filedes, int queue);

    int tcsendbreak(int filedes, int duration);
```
All four return: 0 if OK, -1 on error

The tcdrain() function suspends the process until all of the data in the ouput buffer has been transmitted. The tcflow() function gives control over input and output flow control. The *action* argument to tcflow() can be any of the following macros:

: ——— ——————- TCOOFF Suspend Output TCOON Restart output TCIOFF Suspend input TCION Restart input ——— ——————-

The tcflush() function lets us discard input or output buffer data. Data in the input buffer is data that has been received but not read yet. Data in the output buffer is data that has been written but not transmitted yet. The *queue* argument can have the follow macro values:

: ———— ———————————————— TCIFLUSH Flush the input buffer TCOFLUSH Flush the output buffer TCIOFLUSH Flush both the input and output buffers ———— ————————————————

The tcsendbreak() function transmits a continous stream of zero bits. If the *duration* attribute is set to 0, then the duration of the transmition is between 0.25 and 0.5 seconds. If the *duration* is nonzero, it is implementation specific. Under Linux, if the *duration* is nonzero, the length of transmission is ***duration**N** seconds where N is between 0.25 and 0.5.

What Terminal Is This?

At some point in your career, you may want to find out what terminal device your process is attached to. In the old days, you could just open "/dev/tty" and that was the correct terminal for your process all of the time. Now, their is a POSIX.1 call that you can make to guarantee that you have the right name of your terminal device.

```
    #include <stdio.h>

    char *ctermid(char *ptr);
```
returns some stuff

If *ptr* is not null, it must be an array of char's that is as large as or larger than the macro L_ctermid and the name of the controlling terminal is stored in this array. If *ptr* is null, then the name of the controlling terminal is stored in a static array. In both cases, a pointer to the first element of the array storing the name of the controlling terminal is returned.

Two really cool and usful functions are isatty() and ttyname().

```
    #include <unistd.h>

    int isatty(int filedes);
```
Returns: 1 if terminal device, 0 otherwise
```
    char *ttyname(int filedes);
```
Returns: pointer to pathname of terminal, NULL on error

These functions are useful in finding out if *filedes* is associated with a terminal device or not. Under the hood, the function ttyname(), searches through the terminal device files in /dev/ looking for a matching special file with the same device number and i-node number as *filedes*. If this statement made no since to you, just ignore it for now. :)

Captain Cooked Mode! Arrrr! (Wait, was he a pirate?)

So, what is this cooked terminal mode anyway? When you read from the terminal, if the terminal returns a line at a time instead of each character as it is received, then you are in cooked (canonical) mode. There are a number of reasons that can make the read return:

- The requested number of bytes has been read (you hit the end of your buffer man!). The next time you do a read, you pick up where you left off if you hadn't finished reading the complete line.
- It returns when you reach the end of a line which can be any of the following characters: NL, EOL, EOL2, EOF, and CR if the ICRNL flag is set for the terminal and the IGNCR flag is not set. If EOF is the line delimiter, then it is thrown out. The others are returned to the caller.
- If a signal is caught and SA_RESTART was not specified as a flag to sigaction() the read will return.

Cooked mode is the default state of your terminal for almost all shells. At least when you execute another program with the shell, the terminal is put into cooked mode before it makes a call to an exec function.

Raw!

Raw, or noncanonical for those that don't like raw, is a bit harder to explain. It doesn't necessarily return one byte at a time. You can also set a time limit for your read to return if the number of characters you want to get have not been received from the device. The first step in going into raw mode, no matter what form you want, is to turn off the flag ICANON for your terminal device. This makes it so the input is not put into lines before it is returned. It also makes so some of the special characters are not processed: ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS, and WERASE.

So, how do we specify how long to wait for input and how many bytes to read before we return? There are two variables in the c_cc array in the termios structure that must be set: MIN and TIME. These elements are indexed by the macro defines VMIN and VTIME. MIN is the minimum number of bytes that are read in before returning (read blocks until MIN number of bytes have been read). TIME is the amount of time in tenths of a second to wait for data to arrive. So, here is the breakdown of the different cases:

1. **MIN $>$ 0 and TIME $>$ 0** In this case, read will return if MIN number of bytes have been read from the device. It will also return if the number of tenths-of-a-second specified in the TIME variable have elapsed after the first byte has been read. This means, if nothing is inputed, the read blocks indefinately. The timer only starts if a byte gets read and does not restart after more bytes are read.
2. **MIN $>$ 0 and TIME $==$ 0** In this case, there is no time limit imposed on the read. It will read until at least MIN bytes have been received. This can cause read to block forever if MIN bytes are never received.
3. **MIN $==$ 0 and TIME $>$ 0** The timer is started as soon as read is called. read returns only after TIME tenths-of-a-second have elapsed or a single byte has been received.
4. **MIN $==$ 0 and TIME $==$ 0** In this case, if data is available, the number of bytes requested (or all the data if it is less than the number of bytes requested) are returned. If no data is available, read returns 0 immediately.

This can be a little confusing at first, just read it a couple of times if you don't understand. Then let it mull over in your brain. Try writing a toy program for each case discussed above. Here is an example of the use of the c_cc variable:

```
struct termios trm;
```

```
tcgetattr(STDIN_FILENO, &trm;); /* get the current settings */
trm.c_cc[VMIN] = 1;     /* return after one byte read */
trm.c_cc[VTIME] = 0;    /* block forever until 1 byte is read */
        .
        .               /* set some other stuff */
        .
tcsetattr(STDIN_FILENO, TCSANOW, &trm;); /* set the terminal with the new
                                            settings */
```

Some source code for raw mode taken from the Stevens book:

```
#include

static struct termios   save_termios;
static int              term_saved;

int tty_raw(int fd) {       /* RAW! mode */
    struct termios  buf;

    if (tcgetattr(fd, &save;_termios) < 0) /* get the original state */
        return -1;

    buf = save_termios;

    buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
                    /* echo off, canonical mode off, extended input
                       processing off, signal chars off */

    buf.c_iflag &= ~(BRKINT | ICRNL | ISTRIP | IXON);
                    /* no SIGINT on BREAK, CR-toNL off, input parity
                       check off, don't strip the 8th bit on input,
                       ouput flow control off */

    buf.c_cflag &= ~(CSIZE | PARENB);
                    /* clear size bits, parity checking off */

    buf.c_cflag |= CS8;
                    /* set 8 bits/char */

    buf.c_oflag &= ~(OPOST);
                    /* output processing off */

    buf.c_cc[VMIN] = 1;  /* 1 byte at a time */
    buf.c_cc[VTIME] = 0; /* no timer on input */

    if (tcsetattr(fd, TCSAFLUSH, &buf;) < 0)
```

```
        return -1;

    term_saved = 1;

    return 0;
}


int tty_reset(int fd) { /* set it to normal! */
    if (term_saved)
        if (tcsetattr(fd, TCSAFLUSH, &save;_termios) < 0)
            return -1;

    return 0;
}
```

Window Size

Ever wonder how some terminal applications redraw the screen when you changed the size of your xterm? There is a structure that the kernel maintains for every terminal and pseudo terminal. This is the winsize struct:

```
    struct winsize {
        unsigned short  ws_row;     /* rows in characters */
        unsigned short  ws_col;     /* columns in characters */
        unsigned short  ws_xpixel;  /* horizontal size in pixels (not used) */
        unsigned short  ws_ypixel;  /* vertical size in pixels (not used) */
    }
```

The signal SIGWINCH is sent to the forground process any time there is a change made to this strucure in the kernel. We can get the current value of this structure by making a call to ioctl with TIOCGWINSZ request:

```
    struct winsize  size;

    ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size;);
```

To change the structure in kernel memory, a call to ioctl is made with the TIOCSWINSZ request:

```
    struct winsize  size;

    ioctl(STDIN_FILENO, TIOCSWINSZ, (char *) &size;);
```

When you set the size of the structure in the kernel, if the size is different than it was previously, SIGWINCH is sent to the foreground process.