18

Terminal I/O

530128

18.1 Introduction

The handling of terminal I/O is a messy area, regardless of the operating system. The UNIX System is no exception. The manual page for terminal I/O is usually one of the longest in most editions of the programmer's manuals.

With the UNIX System, a schism formed in the late 1970s when System III developed a different set of terminal routines from those of Version 7. The System III style of terminal I/O continued through System V, and the Version 7 style became the standard for the BSD-derived systems. As with signals, this difference between the two worlds has been conquered by POSIX.1. In this chapter, we look at all the POSIX.1 terminal functions and some of the platform-specific additions.

Part of the complexity of the terminal I/O system occurs because people use terminal I/O for so many different things: terminals, hard-wired lines between computers, modems, printers, and so on.

18.2 Overview

Terminal I/O has two modes:

1. Canonical mode input processing. In this mode, terminal input is processed as lines. The terminal driver returns at most one line per read request.
2. Noncanonical mode input processing. The input characters are not assembled into lines.

If we don't do anything special, canonical mode is the default. For example, if the shell redirects standard input to the terminal and we use `read` and `write` to copy standard input to standard output, the terminal is in canonical mode, and each `read` returns at most one line. Programs that manipulate the entire screen, such as the `vi` editor, use noncanonical mode, since the commands may be single characters and are not terminated by newlines. Also, this editor doesn't want processing by the system of the special characters, since they may overlap with the editor commands. For example, the Control-D character is often the end-of-file character for the terminal, but it's also a `vi` command to scroll down one-half screen.

> The Version 7 and older BSD-style terminal drivers supported three modes for terminal input: (a) cooked mode (the input is collected into lines, and the special characters are processed), (b) raw mode (the input is not assembled into lines, and there is no processing of special characters), and (c) cbreak mode (the input is not assembled into lines, but some of the special characters are processed). Figure 18.20 shows a POSIX.1 function that places a terminal in cbreak or raw mode.

POSIX.1 defines 11 special input characters, 9 of which we can change. We've been using some of these throughout the text—the end-of-file character (usually Control-D) and the suspend character (usually Control-Z), for example. Section 18.3 describes each of these characters.

We can think of a terminal device as being controlled by a terminal driver, usually within the kernel. Each terminal device has an input queue and an output queue, shown in Figure 18.1.

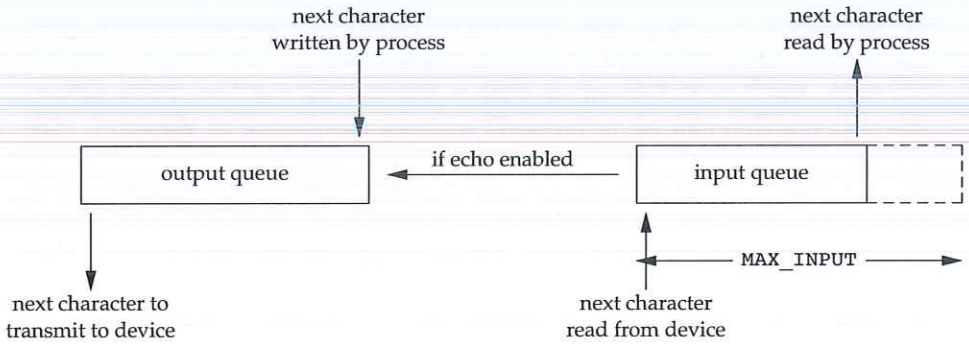

Figure 18.1 Logical picture of input and output queues for a terminal device

There are several points to consider from this picture.

- If echoing is enabled, there is an implied link between the input queue and the output queue.
- The size of the input queue, `MAX_INPUT` (see Figure 2.12), is finite. When the input queue for a particular device fills, the system behavior is implementation dependent. Most UNIX systems echo the bell character when this happens.

- There is another input limit, `MAX_CANON`, that we don't show here. This limit is the maximum number of bytes in a canonical input line.
- Although the size of the output queue is finite, no constants defining that size are accessible to the program, because when the output queue starts to fill up, the kernel simply puts the writing process to sleep until room is available.
- We'll see how the `tcflush` flush function allows us to flush either the input queue or the output queue. Similarly, when we describe the `tcsetattr` function, we'll see how we can tell the system to change the attributes of a terminal device only after the output queue is empty. (We want to do this, for example, if we're changing the output attributes.) We can also tell the system to discard everything in the input queue when changing the terminal attributes. (We want to do this if we're changing the input attributes or changing between canonical and noncanonical modes, so that previously entered characters aren't interpreted in the wrong mode.)

Most UNIX systems implement all the canonical processing in a module called the *terminal line discipline*. We can think of this module as a box that sits between the kernel's generic read and write functions and the actual device driver (see Figure 18.2).

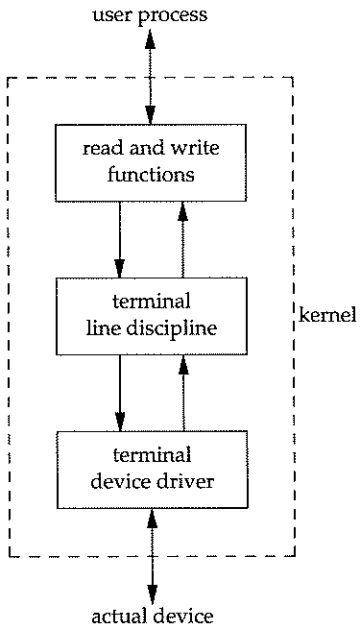

Figure 18.2 Terminal line discipline

By isolating the canonical processing in a separate module, all terminal drivers can support canonical processing consistently. We'll return to this picture when we discuss pseudo terminals in Chapter 19.

All the terminal device characteristics that we can examine and change are contained in a `termios` structure. This structure is defined in the header `<termios.h>`, which we use throughout this chapter:

```
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 */
};
```

Roughly speaking, the input flags control the input of characters by the terminal device driver (e.g., strip eighth bit on input, enable input parity checking), the output flags control the driver output (e.g., perform output processing, map newline to CR/LF), the control flags affect the RS-232 serial lines (e.g., ignore modem status lines, one or two stop bits per character), and the local flags affect the interface between the driver and the user (e.g., echo on or off, visually erase characters, enable terminal-generated signals, job control stop signal for background output).

The type `tcflag_t` is big enough to hold each of the flag values and is often defined as an `unsigned int` or an `unsigned long`. The `c_cc` array contains all the special characters that we can change. `NCCS` is the number of elements in this array and is typically between 15 and 20 (since most implementations of the UNIX System support more than the 11 POSIX-defined special characters). The `cc_t` type is large enough to hold each special character and is typically an `unsigned char`.

> Versions of System V that predated the POSIX standard had a header named `<termio.h>` and a structure named `termio`. POSIX.1 added an `s` to the names, to differentiate them from their predecessors.

Figures 18.3 through 18.6 list all the terminal flags that we can change to affect the characteristics of a terminal device. Note that even though the Single UNIX Specification defines a common subset that all platforms start from, all the implementations have their own additions. Most of these additions come from the historical differences between the systems. We'll discuss each of these flag values in detail in Section 18.5.

Given all the options available, how do we examine and change these characteristics of a terminal device? Figure 18.7 summarizes the various functions defined by the Single UNIX Specification that operate on terminal devices. (All the functions listed are part of the base POSIX specification. We described `tcgetpgrp`, `tcgetsid`, and `tcsetpgrp` in Section 9.7.)

Note that the Single UNIX Specification doesn't use the classic `ioctl` on terminal devices. Instead, it uses the 13 functions shown in Figure 18.7. The reason is that the `ioctl` function for terminal devices uses a different data type for its final argument, which depends on the action being performed. This makes type checking of the arguments impossible.

Although only 13 functions operate on terminal devices, the first two functions in Figure 18.7 (`tcgetattr` and `tcsetattr`) manipulate almost 70 different flags (see

Figures 18.3 through 18.6). The handling of terminal devices is complicated by the large number of options available for terminal devices and the challenge of trying to determine which options are required for a particular device (be it a terminal, modem, printer, or whatever).

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
CBAUDEXT	extended baud rate					•
CCAR_OFLOW	DCD flow control of output		•		•	
CCTS_OFLOW	CTS flow control of output		•		•	•
CDSR_OFLOW	DSR flow control of output		•		•	
CDTR_IFLOW	DTR flow control of input		•		•	
CIBAUDEXT	extended input baud rate					•
CIGNORE	ignore control flags		•		•	
CLOCAL	ignore modem status lines	•	•	•	•	•
CMSPAR	mark or space parity			•		
CREAD	enable receiver	•	•	•	•	•
CRTSCTS	enable hardware flow control		•	•	•	•
CRTS_IFLOW	RTS flow control of input		•		•	•
CRTSXOFF	enable input hardware flow control					•
CSIZE	character size mask	•	•	•	•	•
CSTOPB	send two stop bits, else one	•	•	•	•	•
HUPCL	hang up on last close	•	•	•	•	•
MDMBUF	same as CCAR_OFLOW		•		•	
PARENB	parity enable	•	•	•	•	•
PAREXT	mark or space parity					•
PARODD	odd parity, else even	•	•	•	•	•

Figure 18.3 `c_cflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
BRKINT	generate SIGINT on BREAK	•	•	•	•	•
ICRNL	map CR to NL on input	•	•	•	•	•
IGNBRK	ignore BREAK condition	•	•	•	•	•
IGNCR	ignore CR	•	•	•	•	•
IGNPAR	ignore characters with parity errors	•	•	•	•	•
IMAXBEL	ring bell on input queue full		•	•	•	•
INLCR	map NL to CR on input	•	•	•	•	•
INPCK	enable input parity checking	•	•	•	•	•
ISTRIP	strip eighth bit off input characters	•	•	•	•	•
IUCLC	map uppercase to lowercase on input			•		•
IUTF8	input is UTF-8			•	•	
IXANY	enable any characters to restart output	•	•	•	•	•
IXOFF	enable start/stop input flow control	•	•	•	•	•
IXON	enable start/stop output flow control	•	•	•	•	•
PARMRK	mark parity errors	•	•	•	•	•

Figure 18.4 `c_iflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
ALTWERASE	use alternate WERASE algorithm		•		•	
ECHO	enable echo	•	•	•	•	•
ECHOCTL	echo control chars as ^(Char)		•	•	•	•
ECHOE	visually erase chars	•	•	•	•	•
ECHOK	echo kill	•	•	•	•	•
ECHOKE	visual erase for kill		•	•	•	•
ECHONL	echo NL	•	•	•	•	•
ECHOPRT	visual erase mode for hard copy		•	•	•	•
EXTPROC	external character processing		•	•	•	
FLUSHO	output being flushed		•	•	•	•
ICANON	canonical input	•	•	•	•	•
IEXTEN	enable extended input char processing	•	•	•	•	•
ISIG	enable terminal-generated signals	•	•	•	•	•
NOFLSH	disable flush after interrupt or quit	•	•	•	•	•
NOKERNINFO	no kernel output from STATUS		•		•	
PENDIN	retype pending input		•	•	•	•
TOSTOP	send SIGTTOU for background output	•	•	•	•	•
XCASE	canonical upper/lower presentation			•		•

Figure 18.5 `c_lflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
BSDLY	backspace delay mask	XSI		•		•
CRDLY	CR delay mask	XSI		•		•
FFDLY	form feed delay mask	XSI		•		•
NLDLY	NL delay mask	XSI		•		•
OCRNL	map CR to NL on output	XSI	•	•		•
OFDEL	fill is DEL, else NUL	XSI		•		•
OFILL	use fill character for delay	XSI		•		•
OLCUC	map lowercase to uppercase on output			•		•
ONLCR	map NL to CR-NL	XSI	•	•	•	•
ONLRET	NL performs CR function	XSI	•	•		•
ONOCR	no CR output at column 0	XSI	•	•		•
ONOEOT	discard EOTs (^D) on output		•		•	
OPOST	perform output processing	•	•	•	•	•
OXTABS	expand tabs to spaces		•		•	
TABDLY	horizontal tab delay mask	XSI	•	•		•
VTDLY	vertical tab delay mask	XSI		•		•

Figure 18.6 `c_oflag` terminal flags

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 output 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

Figure 18.7 Summary of terminal I/O functions

The relationships among the 13 functions shown in Figure 18.7 are illustrated in Figure 18.8.

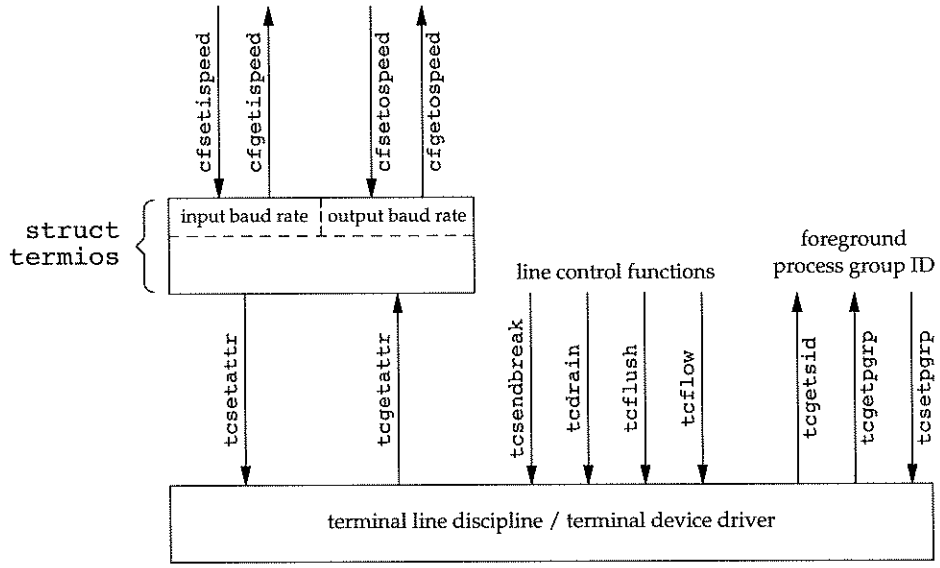

Figure 18.8 Relationships among the terminal-related functions

POSIX.1 doesn't specify where in the termios structure the baud rate information is stored; that is an implementation detail. Some systems, such as Solaris, store this information in the c_cflag field. Linux and BSD-derived systems, such as FreeBSD and Mac OS X, have two separate fields in the structure: one for the input speed and one for the output speed.

18.3 Special Input Characters

POSIX.1 defines 11 characters that are handled specially on input. Implementations define additional special characters. Figure 18.9 summarizes these special characters.

Character	Description	c_cc subscript	Enabled by field	Enabled by flag	Typical value	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
CR	carriage return	(can't change)	c_lflag	ICANON	\r	•	•	•	•	•
DISCARD	discard output	VDISCARD	c_lflag	IEXTEN	^O		•	•	•	•
DSUSP	delayed suspend (SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		•		•	•
EOF	end of file	VEOF	c_lflag	ICANON	^D	•	•	•	•	•
EOL	end of line	VEOL	c_lflag	ICANON		•	•	•	•	•
EOL2	alternate end of line	VEOL2	c_lflag	ICANON			•	•	•	•
ERASE	backspace one character	VERASE	c_lflag	ICANON	^H, ^?	•	•	•	•	•
ERASE2	alternate backspace character	VERASE2	c_lflag	ICANON	^H, ^?		•			
INTR	interrupt signal (SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•	•	•	•	•
KILL	erase line	VKILL	c_lflag	ICANON	^U	•	•	•	•	•
LNEXT	literal next	VLNEXT	c_lflag	IEXTEN	^V		•	•	•	•
NL	line feed (newline)	(can't change)	c_lflag	ICANON	\n	•	•	•	•	•
QUIT	quit signal (SIGQUIT)	VQUIT	c_lflag	ISIG	^\	•	•	•	•	•
REPRINT	reprint all input	VREPRINT	c_lflag	ICANON	^R		•	•	•	•
START	resume output	VSTART	c_iflag	IXON/IXOFF	^Q	•	•	•	•	•
STATUS	status request	VSTATUS	c_lflag	ICANON	^T		•		•	
STOP	stop output	VSTOP	c_iflag	IXON/IXOFF	^S	•	•	•	•	•
SUSP	suspend signal (SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•	•	•	•	•
WERASE	backspace one word	VWERASE	c_lflag	ICANON	^W		•	•	•	•

Figure 18.9 Summary of special terminal input characters

Of the 11 POSIX.1 special characters, we can change 9 of them to almost any value that we like. The exceptions are the newline and carriage return characters (`\n` and `\r`, respectively) and perhaps the STOP and START characters (depends on the implementation). To do this, we modify the appropriate entry in the `c_cc` array of the `termios` structure. The elements in this array are referred to by name, with each name beginning with a `V` (the third column in Figure 18.9).

POSIX.1 allows us to disable these characters. If we set the value of an entry in the `c_cc` array to the value of `_POSIX_VDISABLE`, then we disable the corresponding special character.

> In early versions of the Single UNIX Specification, support for `_POSIX_VDISABLE` was optional. It is now required.
>
> All four platforms discussed in this text support this feature. Linux 3.2.0 and Solaris 10 define `_POSIX_VDISABLE` as 0; FreeBSD 8.0 and Mac OS X 10.6.8 define it as `0xff`.
>
> Some earlier UNIX systems disabled a feature if the corresponding special input character was 0.

Example

Before describing all the special characters in detail, let's look at a small program that changes them. The program in Figure 18.10 disables the interrupt character and sets the end-of-file character to Control-B.

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios  term;
    long            vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("standard input is not a terminal device");

    if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
        err_sys("tcgetattr error");

    term.c_cc[VINTR] = vdisable;    /* disable INTR character */
    term.c_cc[VEOF]  = 2;           /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Figure 18.10 Disable interrupt character and change end-of-file character

Note the following points regarding this program.

- We modify the terminal characters only if standard input is a terminal device. We call `isatty` (Section 18.9) to check this.
- We fetch the `_POSIX_VDISABLE` value using `fpathconf`.
- The function `tcgetattr` (Section 18.4) fetches a `termios` structure from the kernel. After we've modified this structure, we call `tcsetattr` to set the attributes. The only attributes that change are the ones we specifically modified.
- Disabling the interrupt key is different from ignoring the interrupt signal. The program in Figure 18.10 simply disables the special character that causes the terminal driver to generate `SIGINT`. We can still use the `kill` function to send the signal to the process. □

We now describe each of the special characters in more detail. We call these the special input characters, but two of the characters, STOP and START (Control-S and Control-Q), are also handled specially when output. Note that when recognized by the terminal driver and processed specially, most of these special characters are then discarded: they are not returned to the process in a read operation. The exceptions are the newline characters (NL, EOL, EOL2) and the carriage return (CR).

CR The carriage return character. We cannot change this character. This character is recognized on input in canonical mode. When both `ICANON` (canonical mode) and `ICRNL` (map CR to NL) are set and `IGNCR` (ignore CR) is not set, the CR character is translated to NL and has the same effect as a NL character. This character is returned to the reading process (perhaps after being translated to a NL).

DISCARD The discard character. This character, recognized on input in extended mode (`IEXTEN`), causes subsequent output to be discarded until another DISCARD character is entered or the discard condition is cleared (see the `FLUSHO` option). This character is discarded when processed (i.e., it is not passed to the process).

DSUSP The delayed-suspend job-control character. This character is recognized on input in extended mode (`IEXTEN`) if job control is supported and if the `ISIG` flag is set. Like the SUSP character, this delayed-suspend character generates the `SIGTSTP` signal that is sent to all processes in the foreground process group (refer to Figure 9.7). However, the delayed-suspend character generates a signal only when a process reads from the controlling terminal, not when the character is typed. This character is discarded when processed (i.e., it is not passed to the process).

EOF The end-of-file character. This character is recognized on input in canonical mode (`ICANON`). When we type this character, all bytes waiting to be read are immediately passed to the reading process. If no bytes are waiting to be read, a count of 0 is returned. Entering an EOF character at the beginning of the line is the normal way to indicate the end of file to a program. This character is discarded when processed in canonical mode (i.e., it is not passed to the process).

EOL The additional line delimiter character, like NL. This character is recognized on input in canonical mode (`ICANON`) and is returned to the reading process; however, this character is not normally used.

EOL2 Another line delimiter character, like NL. This character is treated identically to the EOL character.

ERASE The erase character (backspace). This character is recognized on input in canonical mode (`ICANON`) and erases the previous character in the line, not erasing beyond the beginning of the line. The erase character is discarded when processed in canonical mode (i.e., it is not passed to the process).

ERASE2 The alternate erase character (backspace). This character is treated exactly like the erase character (ERASE).

INTR The interrupt character. This character is recognized on input if the `ISIG` flag is set and generates the `SIGINT` signal that is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

KILL The kill character. (The name "kill" is overused; recall the `kill` function used to send a signal to a process. This character should be called the line-erase character; it has nothing to do with signals.) It is recognized on input in canonical mode (`ICANON`). It erases the entire line and is discarded when processed (i.e., it is not passed to the process).

LNEXT The literal-next character. This character is recognized on input in extended mode (`IEXTEN`) and causes any special meaning of the next character to be ignored. This works for all special characters listed in this section. We can use this character to type any character to a program. The LNEXT character is discarded when processed, but the next character entered is passed to the process.

NL The newline character, also called the line delimiter. We cannot change this character. It is recognized on input in canonical mode (`ICANON`). This character is returned to the reading process.

QUIT The quit character. This character is recognized on input if the `ISIG` flag is set. The quit character generates the `SIGQUIT` signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

Recall from Figure 10.1 that the difference between INTR and QUIT is that the QUIT character not only terminates the process by default, but also generates a `core` file.

REPRINT The reprint character. This character is recognized on input in extended, canonical mode (both `IEXTEN` and `ICANON` flags set) and causes all unread input to be output (reechoed). This character is discarded when processed (i.e., it is not passed to the process).

START The start character. This character is recognized on input if the `IXON` flag is set and is automatically generated as output if the `IXOFF` flag is set. A received START character with `IXON` set causes stopped output (from a previously entered STOP character) to restart. In this case, the START character is discarded when processed (i.e., it is not passed to the process).

When `IXOFF` is set, the terminal driver automatically generates a START character to resume input that it had previously stopped, when the new input will not overflow the input buffer.

STATUS The BSD status-request character. This character is recognized on input in extended, canonical mode (both `IEXTEN` and `ICANON` flags set) and generates the `SIGINFO` signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). Additionally, if the `NOKERNINFO` flag is not set, status information on the foreground process group is displayed on the terminal. This character is discarded when processed (i.e., it is not passed to the process).

STOP The stop character. This character is recognized on input if the `IXON` flag is set and is automatically generated as output if the `IXOFF` flag is set. A received STOP character with `IXON` set stops the output. In this case, the STOP character is discarded when processed (i.e., it is not passed to the process). The stopped output is restarted when a START character is entered.

When `IXOFF` is set, the terminal driver automatically generates a STOP character to prevent the input buffer from overflowing.

SUSP The suspend job-control character. This character is recognized on input if job control is supported and if the `ISIG` flag is set. The suspend character generates the `SIGTSTP` signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

WERASE The word-erase character. This character is recognized on input in extended, canonical mode (both `IEXTEN` and `ICANON` flags set) and causes the previous word to be erased. First, it skips backward over any white space (spaces or tabs), then skips backward over the previous token, leaving the cursor positioned where the first character of the previous token was located. Normally, the previous token ends when a white space character is encountered. We can change this behavior, however, by setting the `ALTWERASE` flag. This flag causes the previous token to end when the first nonalphanumeric character is encountered. The word-erase character is discarded when processed (i.e., it is not passed to the process).

Another "character" that we need to define for terminal devices is the BREAK character. BREAK is not really a character, but rather a condition that occurs during asynchronous serial data transmission. A BREAK condition is signaled to the device driver in various ways, depending on the serial interface.

> Most old serial terminals have a key labeled BREAK that generates the BREAK condition, which is why most people think of BREAK as a character. Some newer terminal keyboards don't have a BREAK key. On PCs, the break key might be mapped for another purpose. For example, the Windows command interpreter can be interrupted by pressing Control-BREAK.

For asynchronous serial data transmission, a BREAK is a sequence of zero-valued bits that continues for longer than the time required to send one byte. The entire sequence of zero-valued bits is considered a single BREAK. In Section 18.8, we'll see how to send a BREAK with the `tcsendbreak` function.

18.4 Getting and Setting Terminal Attributes

To get and set a `termios` structure, we call two functions: `tcgetattr` and `tcsetattr`. This is how we examine and modify the various option flags and special characters to make the terminal operate the way we want it to.

```
#include <termios.h>

int tcgetattr(int fd, struct termios *termptr);

int tcsetattr(int fd, int opt, const struct termios *termptr);
```

Both return: 0 if OK, –1 on error

Both functions take a pointer to a `termios` structure and either return the current terminal attributes or set the terminal's attributes. Since these two functions operate only on terminal devices, `errno` is set to `ENOTTY` and –1 is returned if *fd* does not refer to a terminal device.

The argument *opt* for `tcsetattr` lets us specify when we want the new terminal attributes to take effect. This argument is specified as one of the following constants.

`TCSANOW` The change occurs immediately.

`TCSADRAIN` The change occurs after all output has been transmitted. This option should be used if we are changing the output parameters.

`TCSAFLUSH` The change occurs after all output has been transmitted. Furthermore, when the change takes place, all input data that has not been read is discarded (flushed).

The return status of `tcsetattr` can be confusing to use correctly. This function returns OK if it was able to perform *any* of the requested actions, even if it couldn't perform all the requested actions. If the function returns OK, it is our responsibility to see whether all the requested actions were performed. This means that after we call `tcsetattr` to set the desired attributes, we need to call `tcgetattr` and compare the actual terminal's attributes to the desired attributes to detect any differences.

What are the attributes of a terminal we open for the first time? The answer is "it depends." Some systems might initialize the terminal attributes to implementation-defined values. Other systems might leave the attributes with the values they had the last time that the terminal was used. If we want to be sure that the terminal behavior conforms to the standard, we can open the terminal device with the `O_TTY_INIT` flag (see Section 3.3). This will ensure that when we call `tcgetattr`, any nonstandard portions of the `termios` structure will be initialized so the terminal will behave as expected when we change the attributes and call `tcsetattr`.

18.5 Terminal Option Flags

In this section, we list all the various terminal option flags, expanding the descriptions from Figures 18.3 through 18.6. This list is alphabetical and indicates in which of the four terminal flag fields the option appears. (The field that controls a given option is usually not apparent from the option name alone.) We also note whether each option is defined by the Single UNIX Specification and list the platforms that support it.

All the flags listed specify one or more bits that we turn on or clear, unless we call the flag a *mask*. A mask defines multiple bits grouped together from which a set of values is defined. We have a defined name for the mask and a name for each value. For example, to set the character size, we first zero the bits using the character-size mask `CSIZE`, and then set one of the values `CS5`, `CS6`, `CS7`, or `CS8`.

The six delay values supported by Linux and Solaris are also masks: `BSDLY`, `CRDLY`, `FFDLY`, `NLDLY`, `TABDLY`, and `VTDLY`. Refer to the `termio`(7I) manual page on Solaris for the length of each delay value. In all cases, a delay mask of 0 means no delay. If a delay is specified, the `OFILL` and `OFDEL` flags determine whether the driver does an actual delay or whether fill characters are transmitted instead.

Example

Figure 18.11 demonstrates the use of these masks to extract a value and to set a value.

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios  term;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 bits/byte\n");
        break;
    case CS6:
        printf("6 bits/byte\n");
        break;
    case CS7:
        printf("7 bits/byte\n");
        break;
    case CS8:
        printf("8 bits/byte\n");
        break;
    default:
        printf("unknown bits/byte\n");
    }

    term.c_cflag &= ~CSIZE;      /* zero out the bits */
    term.c_cflag |= CS8;         /* set 8 bits/byte */
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Figure 18.11 Example of `tcgetattr` and `tcsetattr`

□

We now describe each of the flags.

`ALTWERASE` (`c_lflag`, FreeBSD, Mac OS X) If set, an alternate word-erase algorithm is used when the WERASE character is entered. Instead of moving backward until the previous white space character, this flag causes the WERASE character to move backward until the first nonalphanumeric character is encountered.

`BRKINT` (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If this flag is set and `IGNBRK` is not set, the input and output queues are flushed when a BREAK is received, and a `SIGINT` signal is generated. This signal is generated for the foreground process group if the terminal device is a controlling terminal.

If neither `IGNBRK` nor `BRKINT` is set, then a BREAK is read as a single character `\0`, unless `PARMRK` is set; in that case the BREAK is read as the 3-byte sequence `\377`, `\0`, `\0`.

`BSDLY` (`c_oflag`, XSI, Linux, Solaris) Backspace delay mask. The values for the mask are `BS0` or `BS1`.

`CBAUDEXT` (`c_cflag`, Solaris) Extended baud rates. Used to enable baud rates greater than B38400. (We discuss baud rates in Section 18.7.)

`CCAR_OFLOW` (`c_cflag`, FreeBSD, Mac OS X) Enable hardware flow control of the output using the RS-232 modem carrier signal Data-Carrier-Detect (DCD). This is the same as the old `MDMBUF` flag.

`CCTS_OFLOW` (`c_cflag`, FreeBSD, Mac OS X, Solaris) Enable hardware flow control of the output using the Clear-To-Send (CTS) RS-232 signal.

`CDSR_OFLOW` (`c_cflag`, FreeBSD, Mac OS X) Flow control the output according to the Data-Set-Ready (DSR) RS-232 signal.

`CDTR_IFLOW` (`c_cflag`, FreeBSD, Mac OS X) Flow control the input according to the Data-Terminal-Ready (DTR) RS-232 signal.

`CIBAUDEXT` (`c_cflag`, Solaris) Extended input baud rates. Used to enable input baud rates greater than B38400. (We discuss baud rates in Section 18.7.)

`CIGNORE` (`c_cflag`, FreeBSD, Mac OS X) Ignore control flags.

`CLOCAL` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the modem status lines are ignored. This usually means that the device is directly attached. When this flag is not set, an open of a terminal device usually blocks until the modem answers a call and establishes a connection, for example.

`CMSPAR` (`c_oflag`, Linux) Select mark or space parity. If `PARODD` is set, the parity bit is always 1 (mark parity). Otherwise, the parity bit is always 0 (space parity).

`CRDLY` (`c_oflag`, XSI, Linux, Solaris) Carriage return delay mask. Possible values for the mask are `CR0`, `CR1`, `CR2`, and `CR3`.

`CREAD` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the receiver is enabled and characters can be received.

`CRTSCTS` (`c_cflag`, FreeBSD, Linux, Mac OS X, Solaris) Behavior depends on platform. For Solaris, enables outbound hardware flow control if set. On the other three platforms, enables both inbound and outbound hardware flow control (equivalent to `CCTS_OFLOW|CRTS_IFLOW`).

`CRTS_IFLOW` (`c_cflag`, FreeBSD, Mac OS X, Solaris) Request-To-Send (RTS) flow control of input.

`CRTSXOFF` (`c_cflag`, Solaris) If set, inbound hardware flow control is enabled. The state of the Request-To-Send RS-232 signal controls the flow control.

`CSIZE` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) This field is a mask that specifies the number of bits per byte for both transmission and reception. This size does not include the parity bit, if any. The values for the field defined by this mask are `CS5`, `CS6`, `CS7`, and `CS8`, for 5, 6, 7, and 8 bits per byte, respectively.

`CSTOPB` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, two stop bits are used; otherwise, one stop bit is used.

`ECHO` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, input characters are echoed back to the terminal device. Input characters can be echoed in either canonical or noncanonical mode.

`ECHOCTL` (`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ECHO` is set, ASCII control characters (those characters in the range 0 through octal 37, inclusive) other than the ASCII TAB, the ASCII NL, and the START and STOP characters are echoed as ^*X*, where *X* is the character formed by adding octal 100 to the control character. For example, the ASCII Control-A character (octal 1) is echoed as `^A` and the ASCII DELETE character (octal 177) is echoed as `^?`. If this flag is not set, the ASCII control characters are echoed as themselves. As with the `ECHO` flag, this flag affects the echoing of control characters in both canonical and noncanonical modes.

Be aware that some systems echo the EOF character differently, since its typical value is Control-D. (Control-D is the ASCII EOT character, which can cause some terminals to hang up.) Check your manual.

`ECHOE` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ICANON` is set, the ERASE character erases the last character in the current line from the display. This is usually done in the terminal driver by writing the three-character sequence backspace, space, backspace.

If the WERASE character is supported, `ECHOE` causes the previous word to be erased using one or more of the same three-character sequence.

If the `ECHOPRT` flag is supported, the actions described here for `ECHOE` assume that the `ECHOPRT` flag is not set.

ECHOK (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if ICANON is set, the KILL character erases the current line from the display or outputs the NL character (to emphasize that the entire line was erased).

If the ECHOKE flag is supported, this description of ECHOK assumes that ECHOKE is not set.

ECHOKE (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set and if ICANON is set, the KILL character is echoed by erasing each character on the line. The way in which each character is erased is selected by the ECHOE and ECHOPRT flags.

ECHONL (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if ICANON is set, the NL character is echoed, even if ECHO is not set.

ECHOPRT (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set and if both ICANON and ECHO are set, then the ERASE character (and WERASE character, if supported) cause all the characters being erased to be printed as they are erased. This is often useful on a hard-copy terminal to see exactly which characters are being deleted.

EXTPROC (c_lflag, FreeBSD, Linux, Mac OS X) If set, canonical character processing is performed external to the operating system. This can be the case if the serial communication peripheral card can offload the host processor by doing some of the line discipline processing. This can also be the case when using pseudo terminals (Chapter 19).

FFDLY (c_oflag, XSI, Linux, Solaris) Form feed delay mask. The values for the mask are FF0 or FF1.

FLUSHO (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set, output is being flushed. This flag is set when we type the DISCARD character; the flag is cleared when we type another DISCARD character. We can also set or clear this condition by setting or clearing this terminal flag.

HUPCL (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the modem control lines are lowered (i.e., the modem connection is broken) when the last process closes the device.

ICANON (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, canonical mode is in effect (Section 18.10). This enables the following characters: EOF, EOL, EOL2, ERASE, KILL, REPRINT, STATUS, and WERASE. The input characters are assembled into lines.

If canonical mode is not enabled, read requests are satisfied directly from the input queue. A read does not return until at least MIN bytes have been received or the timeout value TIME has expired between bytes. Refer to Section 18.11 for additional details.

ICRNL (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if IGNCR is not set, a received CR character is translated into a NL character.

IEXTEN (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the extended, implementation-defined special characters are recognized and processed.

IGNBRK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, a BREAK condition on input is ignored. See BRKINT for a way to have a BREAK condition either generate a SIGINT signal or be read as data.

IGNCR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, a received CR character is ignored. If this flag is not set, it is possible to translate the received CR into a NL character if the ICRNL flag is set.

IGNPAR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, an input byte with a framing error (other than a BREAK) or an input byte with a parity error is ignored.

IMAXBEL (c_iflag, FreeBSD, Linux, Mac OS X, Solaris) Ring bell when input queue is full.

INLCR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, a received NL character is translated into a CR character.

INPCK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled.

Parity "generation and detection" and "input parity checking" are two different things. The generation and detection of parity bits is controlled by the PARENB flag. Setting this flag usually causes the device driver for the serial interface to generate parity for outgoing characters and to verify the parity of incoming characters. The flag PARODD determines whether the parity should be odd or even. If an input character arrives with the wrong parity, then the state of the INPCK flag is checked. If this flag is set, then the IGNPAR flag is checked (to see whether the input byte with the parity error should be ignored); if the byte should not be ignored, then the PARMRK flag is checked to see which characters should be passed to the reading process.

ISIG (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the input characters are compared against the special characters that cause the terminal-generated signals to be generated (INTR, QUIT, SUSP, and DSUSP); if equal, the corresponding signal is generated.

ISTRIP (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, valid input bytes are stripped to 7 bits. When this flag is not set, all 8 bits are processed.

IUCLC (c_iflag, Linux, Solaris) Map uppercase to lowercase on input.

`IUTF8` (`c_iflag`, Linux, Mac OS X) Allow character erase processing to work with UTF-8 multibyte characters.

`IXANY` (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Enable any characters to restart output.

`IXOFF` (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, start–stop input control is enabled. When it notices that the input queue is getting full, the terminal driver outputs a STOP character. This character should be recognized by the device that is sending the data and cause the device to stop. Later, when the characters on the input queue have been processed, the terminal driver will output a START character. This should cause the device to resume sending data.

`IXON` (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, start–stop output control is enabled. When the terminal driver receives a STOP character, output stops. While the output is stopped, the next START character resumes the output. If this flag is not set, the START and STOP characters are read by the process as normal characters.

`MDMBUF` (`c_cflag`, FreeBSD, Mac OS X) Flow control the output according to the modem carrier flag. This is the old name for the `CCAR_OFLOW` flag.

`NLDLY` (`c_oflag`, XSI, Linux, Solaris) Newline delay mask. The values for the mask are `NL0` or `NL1`.

`NOFLSH` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) By default, when the terminal driver generates the `SIGINT` and `SIGQUIT` signals, both the input and output queues are flushed. Also, when it generates the `SIGSUSP` signal, the input queue is flushed. If the `NOFLSH` flag is set, this normal flushing of the queues does not occur when these signals are generated.

`NOKERNINFO` (`c_lflag`, FreeBSD, Mac OS X) When set, this flag prevents the STATUS character from printing information on the foreground process group. Regardless of whether this flag is set, however, the STATUS character still causes the `SIGINFO` signal to be sent to the foreground process group.

`OCRNL` (`c_oflag`, XSI, FreeBSD, Linux, Solaris) If set, map CR to NL on output.

`OFDEL` (`c_oflag`, XSI, Linux, Solaris) If set, the output fill character is ASCII DEL; otherwise, it's ASCII NUL. See the `OFILL` flag.

`OFILL` (`c_oflag`, XSI, Linux, Solaris) If set, fill characters (either ASCII DEL or ASCII NUL; see the `OFDEL` flag) are transmitted for a delay, instead of using a timed delay. See the six delay masks: `BSDLY`, `CRDLY`, `FFDLY`, `NLDLY`, `TABDLY`, and `VTDLY`.

`OLCUC` (`c_oflag`, Linux, Solaris) If set, map lowercase characters to uppercase characters on output.

ONLCR (c_oflag, XSI, FreeBSD, Linux, Mac OS X, Solaris) If set, map NL to CR-NL on output.

ONLRET (c_oflag, XSI, FreeBSD, Linux, Solaris) If set, the NL character is assumed to perform the carriage return function on output.

ONOCR (c_oflag, XSI, FreeBSD, Linux, Solaris) If set, a CR is not output at column 0.

ONOEOT (c_oflag, FreeBSD, Mac OS X) If set, EOT (^D) characters are discarded on output. This may be necessary on some terminals that interpret Control-D as a hangup.

OPOST (c_oflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, implementation-defined output processing takes place. Refer to Figure 18.6 for the various implementation-defined flags for the c_oflag field.

OXTABS (c_oflag, FreeBSD, Mac OS X) If set, tabs are expanded to spaces on output. This produces the same effect as setting the horizontal tab delay (TABDLY) to XTABS or TAB3.

PARENB (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, parity generation is enabled for outgoing characters, and parity checking is performed on incoming characters. The parity is odd if PARODD is set; otherwise, it is even parity. See also the discussion of the INPCK, IGNPAR, and PARMRK flags.

PAREXT (c_cflag, Solaris) Select mark or space parity. If PARODD is set, the parity bit is always 1 (mark parity). Otherwise, the parity bit is always 0 (space parity).

PARMRK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set and if IGNPAR is not set, a byte with a framing error (other than a BREAK) or a byte with a parity error is read by the process as the three-character sequence \377, \0, *X*, where *X* is the byte received in error. If ISTRIP is not set, a valid \377 is passed to the process as \377, \377. If neither IGNPAR nor PARMRK is set, a byte with a framing error (other than a BREAK) or with a parity error is read as a single character \0.

PARODD (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the parity for outgoing and incoming characters is odd parity. Otherwise, the parity is even parity. Note that the PARENB flag controls the generation and detection of parity.

The PARODD flag also controls whether mark or space parity is used when either the CMSPAR or PAREXT flag is set.

PENDIN (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set, any input that has not been read is reprinted by the system when the next character is input. This action is similar to what happens when we type the REPRINT character.

TABDLY (c_oflag, XSI, Linux, Mac OS X, Solaris) Horizontal tab delay mask. The values for the mask are TAB0, TAB1, TAB2, or TAB3.

The value XTABS is equal to TAB3. This value causes the system to expand tabs into spaces. The system assumes a tab stop every eight spaces, and we can't change this assumption.

TOSTOP (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if the implementation supports job control, the SIGTTOU signal is sent to the process group of a background process that tries to write to its controlling terminal. By default, this signal stops all the processes in the process group. This signal is not generated by the terminal driver if the background process that is writing to the controlling terminal is either ignoring or blocking the signal.

VTDLY (c_oflag, XSI, Linux, Solaris) Vertical tab delay mask. The values for the mask are VT0 and VT1.

XCASE (c_lflag, Linux, Solaris) If set and if ICANON is also set, the terminal is assumed to be uppercase only, and all input is converted to lowercase. To input an uppercase character, precede it with a backslash. Similarly, the system outputs an uppercase character by preceding it with a backslash. (This option flag is obsolete today, since most, if not all, uppercase-only terminals have disappeared.)

18.6 stty Command

All the options described in the previous section can be examined and changed from within a program with the tcgetattr and tcsetattr functions (Section 18.4) or from the command line (or a shell script) with the stty(1) command. This command is simply an interface to the first six functions that we listed in Figure 18.7. If we execute this command with its -a option, it displays all the terminal options:

```
$ stty -a
speed 9600 baud; 25 rows; 80 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -ocrnl -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts
        -dsrflow -dtrflow -mdmbuf
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; erase2 = ^?; intr = ^C; kill = ^U;
        lnext = ^V; min = 1; quit = ^; reprint = ^R; start = ^Q;
        status = ^T; stop = ^S; susp = ^Z; time = 0; werase = ^W;
```

Option names preceded by a hyphen are disabled. The last four lines display the current settings for each of the terminal special characters (Section 18.3). The first line displays the number of rows and columns for the current terminal window; we discuss the terminal window size in Section 18.12.

The `stty` command uses its standard input to get and set the terminal option flags. Although some older implementations used standard output, POSIX.1 requires that the standard input be used. All four implementations discussed in this text provide versions of `stty` that operate on the standard input. This means that we can type

```
stty -a </dev/tty1a
```

if we are interested in discovering the settings on the terminal named `tty1a`.

18.7 Baud Rate Functions

The term *baud rate* is a historical term that should be referred to today as "bits per second." Although most terminal devices use the same baud rate for both input and output, the capability exists to set the two rates to different values, if the hardware allows this.

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);

speed_t cfgetospeed(const struct termios *termptr);
```

Both return: baud rate value

```
int cfsetispeed(struct termios *termptr, speed_t speed);

int cfsetospeed(struct termios *termptr, speed_t speed);
```

Both return: 0 if OK, –1 on error

The return value from the two `cfget` functions and the *speed* argument to the two `cfset` functions are one of the following constants: `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, or `B38400`. The constant `B0` means "hang up." If `B0` is specified as the output baud rate when `tcsetattr` is called, the modem control lines are no longer asserted.

Most systems define additional baud rate values, such as `B57600` and `B115200`.

To use these functions, we must realize that the input and output baud rates are stored in the device's `termios` structure, as shown in Figure 18.8. Before calling either of the `cfget` functions, we first have to obtain the device's `termios` structure using `tcgetattr`. Similarly, after calling either of the two `cfset` functions, all we've done is set the baud rate in a `termios` structure. For this change to affect the device, we have to call `tcsetattr`. If there is an error in either of the baud rates that we set, we may not find out about the error until we call `tcsetattr`.

The four baud rate functions exist to insulate applications from differences in the way that implementations represent baud rates in the `termios` structure. Linux and BSD-derived platforms tend to store baud rates as numeric values equal to the rates (i.e., 9,600 baud is stored as the value 9,600), whereas System V–derived platforms (such as Solaris) tend to encode the baud rate in a bitmask. The speed values we get from the `cfget` functions and pass to the `cfset` functions are untranslated from their representation as they are stored in the `termios` structure.

18.8 Line Control Functions

The following four functions provide line control capability for terminal devices. All four require that *fd* refer to a terminal device; otherwise, –1 is returned with `errno` set to `ENOTTY`.

```
#include <termios.h>

int tcdrain(int fd);

int tcflow(int fd, int action);

int tcflush(int fd, int queue);

int tcsendbreak(int fd, int duration);
```

All four return: 0 if OK, –1 on error

The `tcdrain` function waits for all output to be transmitted. The `tcflow` function gives us control over both input and output flow control. The *action* argument must be one of the following four values:

`TCOOFF` Output is suspended.

`TCOON` Output that was previously suspended is restarted.

`TCIOFF` The system transmits a STOP character, which should cause the terminal device to stop sending data.

`TCION` The system transmits a START character, which should cause the terminal device to resume sending data.

The `tcflush` function lets us flush (throw away) either the input buffer (data that has been received by the terminal driver, which we have not read) or the output buffer (data that we have written, which has not yet been transmitted). The *queue* argument must be one of the following three constants:

`TCIFLUSH` The input queue is flushed.

`TCOFLUSH` The output queue is flushed.

`TCIOFLUSH` Both the input and the output queues are flushed.

The `tcsendbreak` function transmits a continuous stream of zero bits for a specified duration. If the *duration* argument is 0, the transmission lasts between 0.25 second and 0.5 second. POSIX.1 specifies that if *duration* is nonzero, the transmission time is implementation dependent.

18.9 Terminal Identification

Historically, the name of the controlling terminal in most versions of the UNIX System has been `/dev/tty`. POSIX.1 provides a runtime function that we can call to determine the name of the controlling terminal.

```
#include <stdio.h>

char *ctermid(char *ptr);
```

Returns: pointer to name of controlling terminal on success, pointer to empty string on error

If *ptr* is non-null, it is assumed to point to an array of at least `L_ctermid` bytes, and the name of the controlling terminal of the process is stored in the array. The constant `L_ctermid` is defined in `<stdio.h>`. If *ptr* is a null pointer, the function allocates room for the array (usually as a static variable). Again, the name of the controlling terminal of the process is stored in the array.

In both cases, the starting address of the array is returned as the value of the function. Since most UNIX systems use `/dev/tty` as the name of the controlling terminal, this function is intended to aid portability to other operating systems.

All four platforms described in this text return the string `/dev/tty` when we call `ctermid`.

Example — `ctermid` Function

Figure 18.12 shows an implementation of the POSIX.1 `ctermid` function.

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

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty"));    /* strcpy() returns str */
}
```

Figure 18.12 Implementation of POSIX.1 `ctermid` function

Note that we can't protect against overrunning the caller's buffer, because we have no way to determine its size. □

Two functions that are more interesting for a UNIX system are `isatty`, which returns true if a file descriptor refers to a terminal device, and `ttyname`, which returns the pathname of the terminal device that is open on a file descriptor.

```
#include <unistd.h>

int isatty(int fd);
```

Returns: 1 (true) if terminal device, 0 (false) otherwise

```
char *ttyname(int fd);
```

Returns: pointer to pathname of terminal, `NULL` on error

Example — `isatty` Function

The `isatty` function is trivial to implement, as we show in Figure 18.13. We simply try one of the terminal-specific functions (that doesn't change anything if it succeeds) and look at the return value.

```
#include    <termios.h>

int
isatty(int fd)
{
    struct termios  ts;

    return(tcgetattr(fd, &ts) != -1); /* true if no error (is a tty) */
}
```

Figure 18.13 Implementation of POSIX.1 `isatty` function

We test our `isatty` function with the program in Figure 18.14.

```
#include "apue.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "not a tty");
    exit(0);
}
```

Figure 18.14 Test the `isatty` function

When we run the program from Figure 18.14, we get the following output:

```
$ ./a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ ./a.out </etc/passwd 2>/dev/null
fd 0: not a tty
fd 1: tty
fd 2: not a tty
```

□

Example — `ttyname` Function

The `ttyname` function (Figure 18.15) is longer, as we have to search all the device entries, looking for a match.

```
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    <string.h>
#include    <termios.h>
#include    <unistd.h>
#include    <stdlib.h>

struct devdir {
    struct devdir   *d_next;
    char            *d_name;
};

static struct devdir    *head;
static struct devdir    *tail;
static char             pathname[_POSIX_PATH_MAX + 1];

static void
add(char *dirname)
{
    struct devdir   *ddp;
    int             len;

    len = strlen(dirname);

    /*
     * Skip ., .., and /dev/fd.
     */
    if ((dirname[len-1] == '.') && (dirname[len-2] == '/' ||
      (dirname[len-2] == '.' && dirname[len-3] == '/')))
        return;
    if (strcmp(dirname, "/dev/fd") == 0)
        return;
    if ((ddp = malloc(sizeof(struct devdir))) == NULL)
        return;
```

```
    if ((ddp->d_name = strdup(dirname)) == NULL) {
        free(ddp);
        return;
    }

    ddp->d_next = NULL;
    if (tail == NULL) {
        head = ddp;
        tail = ddp;
    } else {
        tail->d_next = ddp;
        tail = ddp;
    }
}

static void
cleanup(void)
{
    struct devdir   *ddp, *nddp;

    ddp = head;
    while (ddp != NULL) {
        nddp = ddp->d_next;
        free(ddp->d_name);
        free(ddp);
        ddp = nddp;
    }
    head = NULL;
    tail = NULL;
}

static char *
searchdir(char *dirname, struct stat *fdstatp)
{
    struct stat     devstat;
    DIR             *dp;
    int             devlen;
    struct dirent   *dirp;

    strcpy(pathname, dirname);
    if ((dp = opendir(dirname)) == NULL)
        return(NULL);
    strcat(pathname, "/");
    devlen = strlen(pathname);
    while ((dirp = readdir(dp)) != NULL) {
        strncpy(pathname + devlen, dirp->d_name,
          _POSIX_PATH_MAX - devlen);

        /*
         * Skip aliases.
         */
        if (strcmp(pathname, "/dev/stdin") == 0 ||
```

```
            strcmp(pathname, "/dev/stdout") == 0 ||
            strcmp(pathname, "/dev/stderr") == 0)
                continue;
        if (stat(pathname, &devstat) < 0)
            continue;
        if (S_ISDIR(devstat.st_mode)) {
            add(pathname);
            continue;
        }
        if (devstat.st_ino == fdstatp->st_ino &&
          devstat.st_dev == fdstatp->st_dev) {  /* found a match */
            closedir(dp);
            return(pathname);
        }
    }

    closedir(dp);
    return(NULL);
}

char *
ttyname(int fd)
{
    struct stat     fdstat;
    struct devdir   *ddp;
    char            *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    rval = searchdir("/dev", &fdstat);
    if (rval == NULL) {
        for (ddp = head; ddp != NULL; ddp = ddp->d_next)
            if ((rval = searchdir(ddp->d_name, &fdstat)) != NULL)
                break;
    }

    cleanup();
    return(rval);
}
```

Figure 18.15 Implementation of POSIX.1 `ttyname` function

The technique is to read the `/dev` directory, looking for an entry with the same device number and i-node number. Recall from Section 4.24 that each file system has a unique device number (the `st_dev` field in the `stat` structure, from Section 4.2), and each directory entry in that file system has a unique i-node number (the `st_ino` field in the `stat` structure). We assume in this function that when we hit a matching device

number and matching i-node number, we've located the desired directory entry. We could also verify that the two entries have matching `st_rdev` fields (the major and minor device numbers for the terminal device) and that the directory entry is a character special file. However, since we've already verified that the file descriptor argument is both a terminal device and a character special file, and since a matching device number and i-node number pair is unique on a UNIX system, there is no need for the additional comparisons.

The name of our terminal might reside in a subdirectory in `/dev`. Thus, we might need to search the entire file system tree under `/dev`. We skip several directories that might produce incorrect or odd-looking results: `/dev/.`, `/dev/..`, and `/dev/fd`. We also skip the aliases `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`, since they are symbolic links to files in `/dev/fd`.

We can test this implementation with the program shown in Figure 18.16.

```
#include "apue.h"

int
main(void)
{
    char *name;

    if (isatty(0)) {
        name = ttyname(0);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 0: %s\n", name);

    if (isatty(1)) {
        name = ttyname(1);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 1: %s\n", name);

    if (isatty(2)) {
        name = ttyname(2);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 2: %s\n", name);

    exit(0);
}
```

Figure 18.16 Test the `ttyname` function

Running the program from Figure 18.16 gives us

```
$ ./a.out < /dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttys001
fd 2: not a tty
```

□

18.10 Canonical Mode

Canonical mode is simple: we issue a read, and the terminal driver returns when a line has been entered. Several conditions cause the read to return.

- The read returns when the requested number of bytes have been read. We don't have to read a complete line. If we read a partial line, no information is lost; the next read starts where the previous read stopped.
- The read returns when a line delimiter is encountered. Recall from Section 18.3 that the following characters are interpreted as end of line in canonical mode: NL, EOL, EOL2, and EOF. Also, recall from Section 18.5 that if `ICRNL` is set and if `IGNCR` is not set, then the CR character also terminates a line, since it acts just like the NL character.

  Of these five line delimiters, one (EOF) is discarded by the terminal driver when it's processed. The other four are returned to the caller as the last character of the line.
- The read also returns if a signal is caught and if the function is not automatically restarted (Section 10.5).

Example—`getpass` Function

We now examine the function `getpass`, which reads a password of some type from the user at a terminal. This function is called by the `login`(1) and `crypt`(1) programs. To read the password, the function must turn off echoing, but it can leave the terminal in canonical mode, as whatever we type as the password forms a complete line. Figure 18.17 shows a typical implementation on a UNIX system.

There are several points to consider in this example.

- Instead of hard-wiring `/dev/tty` into the program, we call the function `ctermid` to open the controlling terminal.
- We read and write only to the controlling terminal and return an error if we can't open this device for reading and writing. There are other conventions to use. The version of `getpass` in the GNU C library reads from standard input and writes to standard error if the controlling terminal can't be opened for reading and writing. The Solaris version fails if it can't open the controlling terminal.

```
#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);

    sigemptyset(&sig);
    sigaddset(&sig, SIGINT);        /* block SIGINT */
    sigaddset(&sig, SIGTSTP);       /* block SIGTSTP */
    sigprocmask(SIG_BLOCK, &sig, &osig);    /* and save mask */

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

Figure 18.17 Implementation of getpass function

- We block the two signals SIGINT and SIGTSTP. If we didn't do this, entering the INTR character would abort the program and leave the terminal with echoing disabled. Similarly, entering the SUSP character would stop the program and return to the shell with echoing disabled. We choose to block the signals while we have echoing disabled. If they are generated while we're reading the password, they are held until we return. There are other ways to

handle these signals. Some versions just ignore `SIGINT` (saving its previous action) while in `getpass`, resetting the action for this signal to its previous value before returning. This means that any occurrence of the signal while it's ignored is lost. Other versions catch `SIGINT` (saving its previous action) and if the signal is caught, send themselves the signal with the `kill` function after resetting the terminal state and signal action. None of the versions of `getpass` catch, ignore, or block `SIGQUIT`, so entering the QUIT character aborts the program and probably leaves the terminal with echoing disabled.

- Be aware that some shells, notably the Korn shell, turn echoing back on whenever they read interactive input. These shells are the ones that provide command-line editing and therefore manipulate the state of the terminal every time we enter an interactive command. So, if we invoke this program under one of these shells and abort it with the QUIT character, it may reenable echoing for us. Other shells that don't provide this form of command-line editing, such as the Bourne shell, will abort the program and leave the terminal in no-echo mode. If we do this to our terminal, the `stty` command can reenable echoing.
- We use standard I/O to read and write the controlling terminal. We specifically set the stream to be unbuffered; otherwise, there might be some interactions between the writing and reading of the stream (we would need some calls to `fflush`). We could have also used unbuffered I/O (Chapter 3), but we would have to simulate the `getc` function using `read`.
- We store only up to eight characters as the password. Any additional characters that are entered are ignored.

The program in Figure 18.18 calls `getpass` and prints what we enter to let us verify that the ERASE and KILL characters work (as they should in canonical mode).

```
#include "apue.h"

char    *getpass(const char *);

int
main(void)
{
    char    *ptr;

    if ((ptr = getpass("Enter password:")) == NULL)
        err_sys("getpass error");
    printf("password: %s\n", ptr);

    /* now use password (probably encrypt it) ... */

    while (*ptr != 0)
        *ptr++ = 0;     /* zero it out when we're done with it */
    exit(0);
}
```

Figure 18.18 Call the `getpass` function

Whenever a program that calls `getpass` is done with the cleartext password, the program should zero it out in memory, just to be safe. If the program were to generate a `core` file that others might be able to read or if some other process were somehow able to read our memory, they might be able to read the cleartext password. (By "cleartext," we mean the password that we type at the prompt that is printed by `getpass`. Most UNIX system programs then modify this cleartext password, turning it into an "encrypted" password. The `pw_passwd` field in the password file (Section 6.2), for example, contains the encrypted password, not the cleartext password.) □

18.11 Noncanonical Mode

Noncanonical mode is specified by turning off the `ICANON` flag in the `c_lflag` field of the `termios` structure. In noncanonical mode, the input data is not assembled into lines. The following special characters (Section 18.3) are not processed: ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS, and WERASE.

As we said, understanding canonical mode is easy: the system returns up to one line at a time. But with noncanonical mode, how does the system know when to return data to us? If it returned one byte at a time, overhead would be excessive. (Recall Figure 3.6, which showed the overhead in reading one byte at a time. Each time we doubled the amount of data returned, we halved the system call overhead.) The system can't always return multiple bytes at a time, since sometimes we don't know how much data to read until we start reading it.

The solution is to tell the system to return when either a specified amount of data has been read or after a given amount of time has passed. This technique uses two variables in the `c_cc` array in the `termios` structure: MIN and TIME. These two elements of the array are indexed by the names `VMIN` and `VTIME`.

MIN specifies the minimum number of bytes before a `read` returns. TIME specifies the number of tenths of a second to wait for data to arrive. There are four cases.

Case A: MIN > 0, TIME > 0

TIME specifies an interbyte timer that is started only when the first byte is received. If MIN bytes are received before the timer expires, `read` returns MIN bytes. If the timer expires before MIN bytes are received, `read` returns the bytes received. (At least one byte is returned if the timer expires, because the timer is not started until the first byte is received.) In this case, the caller blocks until the first byte is received. If data is already available when `read` is called, it is as if the data had been received immediately after the `read`.

Case B: MIN > 0, TIME == 0

The `read` does not return until MIN bytes have been received. This can cause a `read` to block indefinitely.

Case C: MIN == 0, TIME > 0

TIME specifies a read timer that is started when `read` is called. (Compare this to case A, in which a nonzero TIME represented an interbyte timer that was not started until the first byte was received.) The `read` returns when a single byte is received or when the timer expires. If the timer expires, `read` returns 0.

Case D: MIN == 0, TIME == 0

If some data is available, `read` returns up to the number of bytes requested. If no data is available, `read` returns 0 immediately.

Realize in all these cases that MIN is only a minimum. If the program requests more than MIN bytes of data, it's possible to receive up to the requested amount. This also applies to cases C and D, in which MIN is 0.

Figure 18.19 summarizes the four cases for noncanonical input. In this figure, *nbytes* is the third argument to `read` (the maximum number of bytes to return).

	MIN > 0	MIN == 0
TIME > 0	**A:** `read` returns [MIN, *nbytes*] before timer expires; `read` returns [1, MIN) if timer expires. (TIME = interbyte timer. Caller can block indefinitely.)	**C:** `read` returns [1, *nbytes*] before timer expires; `read` returns 0 if timer expires. (TIME = `read` timer.)
TIME == 0	**B:** `read` returns [MIN, *nbytes*] when available. (Caller can block indefinitely.)	**D:** `read` returns [0, *nbytes*] immediately.

Figure 18.19 Four cases for noncanonical input

Be aware that POSIX.1 allows the subscripts `VMIN` and `VTIME` to have the same values as `VEOF` and `VEOL`, respectively. Indeed, Solaris does this for backward compatibility with older versions of System V. This creates a portability problem, however. In going from noncanonical to canonical mode, we must now restore `VEOF` and `VEOL` as well. If `VMIN` equals `VEOF` and we don't restore their values, when we set `VMIN` to its typical value of 1, the end-of-file character becomes Control-A. The easiest way around this problem is to save the entire `termios` structure when going into noncanonical mode and restore it when going back to canonical mode.

Example

The program in Figure 18.20 defines the `tty_cbreak` and `tty_raw` functions that set the terminal in *cbreak mode* and *raw mode*. (The terms *cbreak* and *raw* come from the Version 7 terminal driver.) We can reset the terminal to its original state (the state before either of these functions was called) by calling the function `tty_reset`.

If we've called `tty_cbreak`, we need to call `tty_reset` before calling `tty_raw`. The same goes for calling `tty_cbreak` after calling `tty_raw`. This improves the chances that the terminal will be left in a usable state if we encounter any errors.

Two additional functions are provided: `tty_atexit` can be established as an exit handler to ensure that the terminal mode is reset by `exit`, and `tty_termios` returns a pointer to the original canonical mode `termios` structure.

```
#include "apue.h"
#include <termios.h>
#include <errno.h>

static struct termios       save_termios;
static int                  ttysavefd = -1;
static enum { RESET, RAW, CBREAK }  ttystate = RESET;

int
tty_cbreak(int fd)  /* put terminal into a cbreak mode */
{
    int             err;
    struct termios  buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* structure copy */

    /*
     * Echo off, canonical mode off.
     */
    buf.c_lflag &= ~(ECHO | ICANON);

    /*
     * Case B: 1 byte at a time, no timer.
     */
    buf.c_cc[VMIN] = 1;
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);

    /*
     * Verify that the changes stuck.  tcsetattr can return 0 on
     * partial success.
     */
    if (tcgetattr(fd, &buf) < 0) {
        err = errno;
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = err;
        return(-1);
    }
    if ((buf.c_lflag & (ECHO | ICANON)) || buf.c_cc[VMIN] != 1 ||
      buf.c_cc[VTIME] != 0) {
```

```
            /*
             * Only some of the changes were made.  Restore the
             * original settings.
             */
            tcsetattr(fd, TCSAFLUSH, &save_termios);
            errno = EINVAL;
            return(-1);
        }

        ttystate = CBREAK;
        ttysavefd = fd;
        return(0);
}

int
tty_raw(int fd)     /* put terminal into a raw mode */
{
        int             err;
        struct termios  buf;

        if (ttystate != RESET) {
            errno = EINVAL;
            return(-1);
        }
        if (tcgetattr(fd, &buf) < 0)
            return(-1);
        save_termios = buf; /* structure copy */

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

        /*
         * No SIGINT on BREAK, CR-to-NL off, input parity
         * check off, don't strip 8th bit on input, output
         * flow control off.
         */
        buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

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

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

        /*
```

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

        /*
         * Case B: 1 byte at a time, no timer.
         */
        buf.c_cc[VMIN] = 1;
        buf.c_cc[VTIME] = 0;
        if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
            return(-1);

        /*
         * Verify that the changes stuck.  tcsetattr can return 0 on
         * partial success.
         */
        if (tcgetattr(fd, &buf) < 0) {
            err = errno;
            tcsetattr(fd, TCSAFLUSH, &save_termios);
            errno = err;
            return(-1);
        }
        if ((buf.c_lflag & (ECHO | ICANON | IEXTEN | ISIG)) ||
          (buf.c_iflag & (BRKINT | ICRNL | INPCK | ISTRIP | IXON)) ||
          (buf.c_cflag & (CSIZE | PARENB | CS8)) != CS8 ||
          (buf.c_oflag & OPOST) || buf.c_cc[VMIN] != 1 ||
          buf.c_cc[VTIME] != 0) {
            /*
             * Only some of the changes were made.  Restore the
             * original settings.
             */
            tcsetattr(fd, TCSAFLUSH, &save_termios);
            errno = EINVAL;
            return(-1);
        }

        ttystate = RAW;
        ttysavefd = fd;
        return(0);
    }

    int
    tty_reset(int fd)       /* restore terminal's mode */
    {
        if (ttystate == RESET)
            return(0);
        if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
            return(-1);
        ttystate = RESET;
        return(0);
    }
```

```
void
tty_atexit(void)        /* can be set up by atexit(tty_atexit) */
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}

struct termios *
tty_termios(void)       /* let caller see original tty state */
{
    return(&save_termios);
}
```

Figure 18.20 Set terminal mode to cbreak or raw

Our definition of cbreak mode is the following:

- Noncanonical mode. As we mentioned at the beginning of this section, this mode turns off some input character processing. It does not turn off signal handling, so the user can always type one of the characters that triggers a terminal-generated signal. Be aware that the caller should catch these signals; otherwise, there's a chance that the signal will terminate the program, and the terminal will be left in cbreak mode.

  As a general rule, whenever we write a program that changes the terminal mode, we should catch most signals. This allows us to reset the terminal mode before terminating.
- Echo off.
- One byte at a time input. To do this, we set MIN to 1 and TIME to 0. This is case B from Figure 18.19. A `read` won't return until at least one byte is available.

We define raw mode as follows:

- Noncanonical mode. We also turn off processing of the signal-generating characters (`ISIG`) and the extended input character processing (`IEXTEN`). Additionally, we disable a BREAK character from generating a signal, by turning off `BRKINT`.
- Echo off.
- We disable the CR-to-NL mapping on input (`ICRNL`), input parity detection (`INPCK`), the stripping of the eighth bit on input (`ISTRIP`), and output flow control (`IXON`).
- Eight-bit characters (`CS8`), and parity checking is disabled (`PARENB`).
- All output processing is disabled (`OPOST`).
- One byte at a time input (MIN = 1, TIME = 0).

The program in Figure 18.21 tests raw and cbreak modes.

```
#include "apue.h"

static void
sig_catch(int signo)
{
    printf("signal caught\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

int
main(void)
{
    int     i;
    char    c;

    if (signal(SIGINT, sig_catch) == SIG_ERR)   /* catch signals */
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("signal(SIGTERM) error");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("Enter raw mode characters, terminate with DELETE\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177)     /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");
    if (tty_cbreak(STDIN_FILENO) < 0)
        err_sys("tty_cbreak error");
    printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        c &= 255;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");

    exit(0);
}
```

Figure 18.21 Test raw and cbreak terminal modes

Running the program in Figure 18.21, we can see what happens with these two terminal modes:

```
$ ./a.out
Enter raw mode characters, terminate with DELETE
                                                    4
                                                   33
                                                  133
                                                   61
                                                   70
                                                  176
                                    type DELETE
Enter cbreak mode characters, terminate with SIGINT
1                                   type Control-A
10                                  type backspace
signal caught                       type interrupt key
```

In raw mode, the characters entered were Control-D (04) and the special function key F7. On the terminal being used, this function key generated five characters: *ESC* (033), *[* (0133), *1* (061), *8* (070), and *~* (0176). Note that with the output processing turned off in raw mode (`~OPOST`), we do not get a carriage return output after each character. Also note that special-character processing is disabled in cbreak mode (so, for example, Control-D, the end-of-file character, and backspace aren't handled specially), whereas the terminal-generated signals are still processed. □

18.12 Terminal Window Size

Most UNIX systems provide a way to keep track of the current terminal window size and to have the kernel notify the foreground process group when the size changes. The kernel maintains a `winsize` structure for every terminal and pseudo terminal:

```
struct winsize {
  unsigned short  ws_row;     /* rows, in characters */
  unsigned short  ws_col;     /* columns, in characters */
  unsigned short  ws_xpixel;  /* horizontal size, pixels (unused) */
  unsigned short  ws_ypixel;  /* vertical size, pixels (unused) */
};
```

The rules for this structure are as follows:

- We can fetch the current value of this structure using an `ioctl` (Section 3.15) of `TIOCGWINSZ`.
- We can store a new value of this structure in the kernel using an `ioctl` of `TIOCSWINSZ`. If this new value differs from the current value stored in the kernel, a `SIGWINCH` signal is sent to the foreground process group. (Note from Figure 10.1 that the default action for this signal is to be ignored.)

- Other than storing the current value of the structure and generating a signal when the value changes, the kernel does nothing else with this structure. Interpreting the structure is entirely up to the application.

This feature is provided to notify applications (such as the `vi` editor) when the window size changes. When it receives the signal, the application can fetch the new size and redraw the screen.

Example

Figure 18.22 shows a program that prints the current window size and goes to sleep. Each time the window size changes, `SIGWINCH` is caught and the new size is printed. We have to terminate this program with a signal.

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

static void
pr_winsize(int fd)
{
    struct winsize  size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
}

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);
    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("signal error");
    pr_winsize(STDIN_FILENO);   /* print initial size */
    for ( ; ; )                 /* and sleep forever */
        pause();
}
```

Figure 18.22 Print window size

Running the program in Figure 18.22 on a windowed terminal gives us

```
$ ./a.out
35 rows, 80 columns          initial size
SIGWINCH received            change window size: signal is caught
40 rows, 123 columns
SIGWINCH received            and again
42 rows, 33 columns
^C $                         type the interrupt key to terminate
```

□

18.13 termcap, terminfo, and curses

termcap stands for "terminal capability," and it refers to the text file /etc/termcap and a set of routines used to read this file. The termcap scheme was developed at Berkeley to support the vi editor. The termcap file contains descriptions of various terminals: which features the terminal supports (e.g., how many lines and rows, whether the terminal support backspace) and how to make the terminal perform certain operations (e.g., clear the screen, move the cursor to a given location). Taking this information out of the compiled program and placing it into a text file that can easily be edited allows the vi editor to run on many different terminals.

The routines that support the termcap file were eventually extracted from the vi editor and placed into a separate curses library. Many features were added to make this library usable for any program that wanted to manipulate the screen.

The termcap scheme was not perfect. As more and more terminals were added to the data file, it took longer to scan the file, looking for a specific terminal. The data file also used two-character names to identify the various terminal attributes. These deficiencies led to development of the terminfo scheme and its associated curses library. The terminal descriptions in terminfo are basically compiled versions of a textual description and can be located faster at runtime. terminfo appeared with SVR2 and has been included in all System V releases since then.

> Historically, System V–based systems used terminfo, and BSD-derived systems used termcap, but it is now common for systems to provide both. Mac OS X, however, supports only terminfo.

A description of terminfo and the curses library is provided by Goodheart [1991], but this book is currently out of print. Strang [1986] describes the Berkeley version of the curses library. Strang, Mui, and O'Reilly [1988] provide a description of termcap and terminfo.

> The ncurses library, a free version that is compatible with the SVR4 curses interface, can be found at http://invisible-island.net/ncurses/ncurses.html. It can also be found at http://www.gnu.org/software/ncurses.

Neither termcap nor terminfo, by itself, addresses the problems we've been looking at in this chapter: changing the terminal's mode, changing one of the terminal special characters, handling the window size, and so on. What they do provide is a way

to perform typical operations (clear the screen, move the cursor) on a wide variety of terminals. On the other hand, `curses` does help with some of the details that we've addressed in this chapter. Functions are provided by `curses` to set raw mode, set cbreak mode, turn echo on and off, and the like. Note that the `curses` library is designed for character-based dumb terminals, which have mostly been replaced by pixel-based graphics terminals today.

18.14 Summary

Terminals have many features and options, most of which we're able to change to suit our needs. In this chapter, we described numerous functions that change a terminal's operation—namely, special input characters and the option flags. We also looked at all the terminal special characters and the many options that can be set or reset for a terminal device.

There are two modes of terminal input—canonical (line at a time) and noncanonical. We showed examples of both modes and provided functions that map between the POSIX.1 terminal options and the older BSD cbreak and raw modes. We also described how to fetch and change the window size of a terminal.

Exercises

18.1 Write a program that calls `tty_raw` and terminates (without resetting the terminal mode). If your system provides the `reset`(1) command (all four systems described in this text do), use it to restore the terminal mode.

18.2 The `PARODD` flag in the `c_cflag` field allows us to specify even or odd parity. The BSD `tip` program, however, also allows the parity bit to be 0 or 1. How does it do this?

18.3 If your system's `stty`(1) command outputs the MIN and TIME values, do the following exercise. Log in to the system twice and start the `vi` editor from one login. Use the `stty` command from your other login to determine which values `vi` sets MIN and TIME to (since `vi` sets the terminal to noncanonical mode). (If you are running a windowing system on your terminal, you can do this same test by logging in once and using two separate windows instead.)

This page intentionally left blank