62

TERMINALS

Historically, users accessed a UNIX system using a terminal connected via a serial line (an RS-232 connection). Terminals were cathode ray tubes (CRTs) capable of displaying characters and, in some cases, primitive graphics. Typically, CRTs provided a monochrome display of 24 lines by 80 columns. By today's standards, these CRTs were small and expensive. In even earlier times, terminals were sometimes hard-copy teletype devices. Serial lines were also used to connect other devices, such as printers and modems, to a computer or to connect one computer to another.

> On early UNIX systems, the terminal lines connected to the system were represented by character devices with names of the form `/dev/tty`*n*. (On Linux, the `/dev/tty`*n* devices are the virtual consoles on the system.) It is common to see the abbreviation *tty* (derived from teletype) as a shorthand for *terminal*.

Especially during the early years of UNIX, terminal devices were not standardized, which meant that different character sequences were required to perform operations such as moving the cursor to the beginning of the line or the top of the screen. (Eventually, some vendor implementations of such *escape sequences*—for example, Digital's VT-100—became de facto and, ultimately, ANSI standards, but a wide variety of terminal types continued to exist.) This lack of standardization meant that it was difficult to write portable programs that used terminal features. The *vi* editor was an early example of a program with such requirements. The *termcap* and *terminfo* databases (described in [Strang et al., 1988]), which tabulate

how to perform various screen-control operations for a wide variety of terminal types, and the *curses* library ([Strang, 1986]) were developed in response to this lack of standardization.

It is no longer common to see a conventional terminal. The usual interface to modern UNIX systems is an X Window System window manager on a high-performance bit-mapped graphical monitor. (An old-style terminal provided functionality roughly equivalent to a single terminal window—an *xterm* or similar—in an X Window System. The fact that the user of such a terminal had only a single "window" to the system was the driving force behind the development of the job-control facilities described in Section 34.7.) Similarly, many devices (e.g., printers) that were once connected directly to a computer are now often intelligent devices connected via a network.

All of the above is a preamble to saying that the need to program terminal devices is less frequent than it used to be. Therefore, this chapter focuses on the aspects of terminal programming that are particularly relevant to software terminal emulators (i.e., *xterm* and similar). It gives only brief coverage to serial lines; references for further information about serial programming are provided at the end of this chapter.

62.1 Overview

Both a conventional terminal and a terminal emulator have an associated terminal driver that handles input and output on the device. (In the case of a terminal emulator, the device is a pseudoterminal. We describe pseudoterminals in Chapter 64.) Various aspects of the operation of the terminal driver can be controlled using the functions described in this chapter.

When performing input, the driver is capable of operating in either of the following modes:

- *Canonical mode*: In this mode, terminal input is processed line by line, and line editing is enabled. Lines are terminated by a newline character, generated when the user presses the *Enter* key. A *read()* from the terminal returns only when a complete line of input is available, and returns at most one line. (If the *read()* requests fewer bytes than are available in the current line, then the remaining bytes are available for the next *read()*.) This is the default input mode.
- *Noncanonical mode*: Terminal input is not gathered into lines. Programs such as *vi*, *more*, and *less* place the terminal in noncanonical mode so that they can read single characters without the user needing to press the *Enter* key.

The terminal driver also interprets a range of special characters, such as the *interrupt* character (normally *Control-C*) and the *end-of-file* character (normally *Control-D*). Such interpretation may result in a signal being generated for the foreground process group or some type of input condition occurring for a program reading from the terminal. Programs that place the terminal in noncanonical mode typically also disable processing of some or all of these special characters.

A terminal driver operates two queues (Figure 62-1): one for input characters transmitted from the terminal device to the reading process(es) and the other for output characters transmitted from processes to the terminal. If terminal echoing

is enabled, then the terminal driver automatically appends a copy of any input character to the end of the output queue, so that input characters are also output on the terminal.

SUSv3 specifies the limit `MAX_INPUT`, which an implementation can use to indicate the maximum length of the terminal's input queue. A related limit, `MAX_CANON`, defines the maximum number of bytes in a line of input in canonical mode. On Linux, *sysconf(_SC_MAX_INPUT)* and *sysconf(_SC_MAX_CANON)* both return the value 255. However, neither of these limits is actually employed by the kernel, which simply imposes a limit of 4096 bytes on the input queue. A corresponding limit on the size of the output queue also exists. However, applications don't need to be concerned with this, since, if a process produces output faster than the terminal driver can handle it, the kernel suspends execution of the writing process until space is once more available in the output queue.

On Linux, we can call *ioctl(fd, FIONREAD, &cnt)* to obtain the number of unread bytes in the input queue of the terminal referred to by the file descriptor *fd*. This feature is not specified in SUSv3.

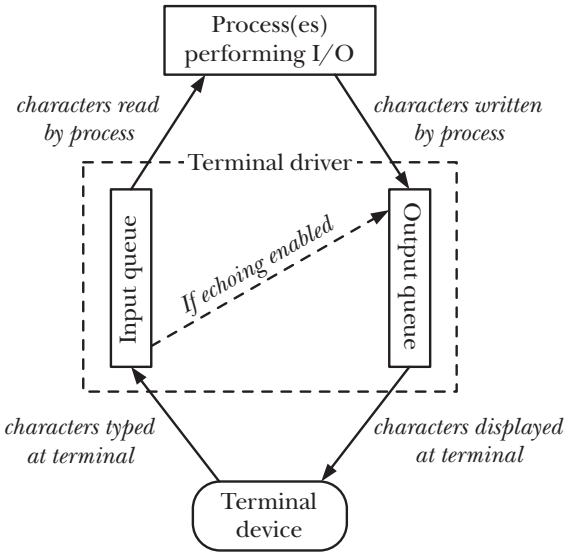

Figure 62-1: Input and output queues for a terminal device

62.2 Retrieving and Modifying Terminal Attributes

The *tcgetattr()* and *tcsetattr()* functions retrieve and modify the attributes of a terminal.

```
#include <termios.h>

int tcgetattr(int fd, struct termios *termios_p);
int tcsetattr(int fd, int optional_actions, const struct termios *termios_p);
```

Both return 0 on success, or −1 on error

The *fd* argument is a file descriptor that refers to a terminal. (If *fd* doesn't refer to a terminal, these functions fail with the error ENOTTY.)

The *termios_p* argument is a pointer to a *termios* structure, which records terminal attributes:

```
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 modes */
    cc_t     c_line;            /* Line discipline (nonstandard)*/
    cc_t     c_cc[NCCS];        /* Terminal special characters */
    speed_t  c_ispeed;          /* Input speed (nonstandard; unused) */
    speed_t  c_ospeed;          /* Output speed (nonstandard; unused) */
};
```

The first four fields of the *termios* structure are bit masks (the *tcflag_t* data type is an integer type of suitable size) containing flags that control various aspects of the terminal driver's operation:

- *c_iflag* contains flags controlling terminal input;
- *c_oflag* contains flags controlling terminal output;
- *c_cflag* contains flags relating to hardware control of the terminal line; and
- *c_lflag* contains flags controlling the user interface for terminal input.

All of the flags used in the above fields are listed in Table 62-2 (on page 1302).

The *c_line* field specifies the line discipline for this terminal. For the purposes of programming terminal emulators, the line discipline will always be set to N_TTY, the so-called *new* discipline, a component of the kernel terminal-handling code that implements canonical mode I/O processing. Setting the line discipline can be relevant when programming serial lines.

The *c_cc* array contains the terminal special characters (*interrupt*, *suspend*, and so on) as well as fields controlling the operation of noncanonical mode input. The *cc_t* data type is an unsigned integer type suitable for holding these values, and the NCCS constant specifies the number of elements in this array. We describe the terminal special characters in Section 62.4.

The *c_ispeed* and *c_ospeed* fields are unused on Linux (and are not specified in SUSv3). We explain how Linux stores terminal line speeds in Section 62.7.

> The Seventh Edition and early BSD terminal driver (known as the *tty* driver) had grown over time so that it used no less than four different data structures to represent the information equivalent to the *termios* structure. System V replaced this baroque arrangement with a single structure called *termio*. The initial POSIX committee selected the System V API almost as is, in the process renaming it to *termios*.

When changing terminal attributes with *tcsetattr()*, the *optional_actions* argument determines when the change takes effect. This argument is specified as one of the following values:

`TCSANOW`
: The change is carried out immediately.

`TCSADRAIN`
: The change takes effect after all currently queued output has been transmitted to the terminal. Normally, this flag should be specified when making changes that affect terminal output, so that we don't affect output that has already been queued but not yet displayed.

`TCSAFLUSH`
: The change takes effect as for `TCSADRAIN`, but, in addition, any input that is still pending at the time the change takes effect is discarded. This is useful, for example, when reading a password, where we wish to disable terminal echoing and prevent user type-ahead.

The usual (and recommended) way to change terminal attributes is to use *tcgetattr()* to retrieve a *termios* structure containing a copy of the current settings, change the desired attributes, and then use *tcsetattr()* to push the updated structure back to the driver. (This approach ensures that we pass a fully initialized structure to *tcsetattr()*.) For example, we can use the following sequence to turn terminal echoing off:

```
struct termios tp;

if (tcgetattr(STDIN_FILENO, &tp) == -1)
    errExit("tcgetattr");
tp.c_lflag &= ~ECHO;
if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &tp) == -1)
    errExit("tcsetattr");
```

The *tcsetattr()* function returns successfully if *any* of the requested changes to terminal attributes could be performed; it fails only if *none* of the requested changes could be made. This means that, when making multiple attribute changes, it may sometimes be necessary to make a further call to *tcgetattr()* to retrieve the new terminal attributes and compare them against the requested changes.

In Section 34.7.2, we noted that if *tcsetattr()* is called from a process in a background process group, then the terminal driver suspends the process group by delivering a `SIGTTOU` signal, and that, if called from an orphaned process group, *tcsetattr()* fails with the error `EIO`. The same statements apply for various other functions described in this chapter, including *tcflush()*, *tcflow()*, *tcsendbreak()*, and *tcdrain()*.

In earlier UNIX implementations, terminal attributes were accessed using *ioctl()* calls. Like several other functions described in this chapter, the *tcgetattr()* and *tcsetattr()* functions are POSIX inventions designed to address the problem that type checking of the third argument of *ioctl()* isn't possible. On Linux, as in many other UNIX implementations, these are library functions layered on top of *ioctl()*.

62.3 The *stty* Command

The *stty* command is the command-line analog of the *tcgetattr()* and *tcsetattr()* functions, allowing us to view and change terminal attributes from the shell. This is useful when trying to monitor, debug, or undo the effects of programs that modify terminal attributes.

We can view the current settings of all terminal attributes using the following command (here carried out on a virtual console):

```
$ stty -a
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

The first line of the above output shows the terminal line speed (bits per second), the terminal window size, and the line discipline in numeric form (0 corresponds to `N_TTY`, the new line discipline).

The next three lines show the settings of the various terminal special characters. The notation `^C` denotes *Control-C*, and so on. The string `<undef>` means that the corresponding terminal special character is not currently defined. The *min* and *time* values relate to noncanonical mode input, and are described in Section 62.6.2.

The remaining lines show the settings of the various flags from (in order) the *c_cflag*, *c_iflag*, *c_oflag*, and *c_lflag* fields of the *termios* structure. Where a flag name is preceded by a hyphen (-), the flag is currently disabled; otherwise, it is currently enabled.

When entered without command-line arguments, *stty* shows just the terminal line speed, the line discipline, and any of the other settings that deviate from sane values.

We can change the settings of terminal special characters using commands such as the following:

```
$ stty intr ^L                  Make the interrupt character Control-L
```

When specifying a control character as the final argument, we can do so in a variety of ways:

- as a 2-character sequence consisting of a caret (^) followed by the corresponding character (as shown above);
- as an octal or hexadecimal number (thus: `014` or `0xC`); or
- by typing the actual character itself.

If we employ the final option, and the character in question is one interpreted specially by the shell or the terminal driver, then we must precede it with the *literal next* character (usually *Control-V*):

```
$ stty intr Control-V Control-L
```

(Although, for readability, a space is shown between the *Control-V* and the *Control-L* in the above example, no white-space characters should be typed between the *Control-V* and the desired character.)

It is possible, though unusual, to define terminal special characters to be something other than control characters:

```
$ stty intr q                   Make the interrupt character q
```

Of course, when we do this, we can no longer use the *q* key for its usual purpose (i.e., generating the letter *q*).

To change terminal flags, such as the TOSTOP flag, we can use commands such as the following:

```
$ stty tostop                   Enable the TOSTOP flag
$ stty -tostop                  Disable the TOSTOP flag
```

Sometimes, when developing programs that modify terminal attributes, a program may crash, leaving the terminal in a state that renders it all but unusable. On a terminal emulator, we have the luxury of simply closing the terminal window and starting another. Alternatively, we can type in the following character sequence to restore the terminal flags and special characters to a reasonable state:

```
Control-J stty sane Control-J
```

The *Control-J* character is the real newline character (ASCII 10 decimal). We use this character because, in some modes, the terminal driver may no longer map the *Enter* key (ASCII 13 decimal) into a newline character. We type an initial *Control-J* to ensure that we have a fresh command line with no preceding characters. This may not be easy to verify if, for example, terminal echoing has been disabled.

The *stty* command operates on the terminal referred to by standard input. Using the *−F* option, we can (subject to permission checks) monitor and set the attributes of a terminal other than the one on which the *stty* command is run:

```
$ su                            Need privilege to access another user's terminal
Password:
# stty -a -F /dev/tty3          Fetch attributes for terminal /dev/tty3
Output omitted for brevity
```

The *−F* option is a Linux-specific extension to the *stty* command. On many other UNIX implementations, *stty* always acts on the terminal referred to by standard input, and we must use the following alternative form (which can also be used on Linux):

```
# stty -a < /dev/tty3
```

62.4 Terminal Special Characters

Table 62-1 lists the special characters recognized by the Linux terminal driver. The first two columns show the name of the character and the corresponding constant that can be used as a subscript in the *c_cc* array. (As can be seen, these constants simply prefix the letter *V* in front of the character name.) The CR and NL characters don't have corresponding *c_cc* subscripts, because the values of these characters can't be changed.

The *Default setting* column of the table shows the usual default value for the special character. As well as being able to set a terminal special character to a specific value, it is also possible to disable the character by setting it to the value returned by the call *fpathconf(fd, _PC_VDISABLE)*, where *fd* is a file descriptor referring to a terminal. (On most UNIX implementations, this call returns the value 0.)

The operation of each of the special characters is subject to the setting of various flags in the *termios* bit-mask fields (described in Section 62.5), as shown in the penultimate column of the table.

The final column indicates which of these characters are specified by SUSv3. Regardless of the SUSv3 specification, most of these characters are supported on all UNIX implementations.

Table 62-1: Terminal special characters

Character	*c_cc* subscript	Description	Default setting	Relevant bit-mask flags	SUSv3
CR	(none)	Carriage return	^M	ICANON, IGNCR, ICRNL, OPOST, OCRNL, ONOCR	•
DISCARD	VDISCARD	Discard output	^O	(not implemented)	
EOF	VEOF	End-of-file	^D	ICANON	•
EOL	VEOL	End-of-line		ICANON	•
EOL2	VEOL2	Alternate end-of-line		ICANON, IEXTEN	
ERASE	VERASE	Erase character	^?	ICANON	•
INTR	VINTR	Interrupt (SIGINT)	^C	ISIG	•
KILL	VKILL	Erase line	^U	ICANON	•
LNEXT	VLNEXT	Literal next	^V	ICANON, IEXTEN	
NL	(none)	Newline	^J	ICANON, INLCR, ECHONL, OPOST, ONLCR, ONLRET	•
QUIT	VQUIT	Quit (SIGQUIT)	^\	ISIG	•
REPRINT	VREPRINT	Reprint input line	^R	ICANON, IEXTEN, ECHO	
START	VSTART	Start output	^Q	IXON, IXOFF	•
STOP	VSTOP	Stop output	^S	IXON, IXOFF	•
SUSP	VSUSP	Suspend (SIGTSTP)	^Z	ISIG	•
WERASE	VWERASE	Erase word	^W	ICANON, IEXTEN	

The following paragraphs provide more detailed explanations of the terminal special characters. Note that if the terminal driver performs its special input interpretation on one of these characters, then—with the exception of CR, EOL, EOL2, and NL—the character is discarded (i.e., it is not passed to any reading process).

CR

CR is the *carriage return* character. This character is passed to the reading process. In canonical mode (`ICANON` flag set) with the `ICRNL` (*map CR to NL on input*) flag set (the default), this character is first converted to a newline (ASCII 10 decimal, `^J`) before being passed to the reading process. If the `IGNCR` (*ignore CR*) flag is set, then this character is ignored on input (in which case the true newline character must be used to terminate a line of input). An output CR character causes the terminal to move the cursor to the beginning of the line.

DISCARD

DISCARD is the *discard output* character. Although this character is defined within the *c_cc* array, it has no effect on Linux. On some other UNIX implementations, typing this character once causes program output to be discarded. This character is a toggle—typing it once more reenables the display of output. This is useful when a program is generating a large amount of output and we want to skip some of it. (This was much more useful on traditional terminals, where line speeds were slower and other "terminal windows" were unavailable.) This character is not passed to the reading process.

EOF

EOF is the canonical mode *end-of-file* character (usually *Control-D*). Entering this character at the beginning of a line causes an end-of-file condition to be detected by a process reading from the terminal (i.e., *read()* returns 0). If typed anywhere other than the initial character of a line, then this character simply causes *read()* to complete immediately, returning the characters so far input in the line. In both cases, the EOF character itself is not passed to the reading process.

EOL and EOL2

EOL and EOL2 are *additional line-delimiter* characters that operate like the standard newline (NL) character for canonical mode input, terminating a line of input and making it available to the reading process. By default, these characters are undefined. If they are defined, they are passed to the reading process. The EOL2 character is operational only if the `IEXTEN` (*extended input processing*) flag is set (the default).

These characters are rarely used. One application that does use them is *telnet*. By setting EOL or EOL2 to be the *telnet* escape character (usually *Control-]*, or, alternatively, the tilde character, ~, if operating in *rlogin* mode), *telnet* is able to immediately catch that character, even while reading input in canonical mode.

ERASE

In canonical mode, typing the ERASE character erases the previously input character on the current line. The erased character and the ERASE character itself are not passed to the reading process.

INTR

INTR is the *interrupt* character. If the `ISIG` (*enable signals*) flag is set (the default), typing this character causes an *interrupt* signal (`SIGINT`) to be sent to the terminal's

foreground process group (Section 34.2). The INTR character itself is not passed to the reading process.

KILL

KILL is the *erase line* (also known as *kill line*) character. In canonical mode, typing this character causes the current line of input to be discarded (i.e., the characters typed so far, as well as the KILL character itself, are not passed to the reading process).

LNEXT

LNEXT is the *literal next* character. In some circumstances, we may wish to treat one of the terminal special characters as though it were a normal character for input to a reading process. Typing the literal next character (usually *Control-V*) causes the next character to be treated literally, voiding any special interpretation of the character that the terminal driver would normally perform. Thus, we could enter the 2-character sequence *Control-V Control-C* to supply a real *Control-C* character (ASCII 3) as input to the reading process. The LNEXT character itself is not passed to the reading process. This character is interpreted only in canonical mode with the `IEXTEN` (*extended input processing*) flag set (which is the default).

NL

NL is the *newline* character. In canonical mode, this character terminates an input line. The NL character itself is included in the line returned to the reading process. (The CR character is normally converted to NL in canonical mode.) An output NL character causes the terminal to move the cursor down one line. If the `OPOST` and `ONLCR` (*map NL to CR-NL*) flags are set (the default), then, on output, a newline character is mapped to the 2-character sequence CR plus NL. (The combination of the `ICRNL` and `ONLCR` flags means that an input CR is converted to a NL, and then echoed as CR plus NL.)

QUIT

If the `ISIG` flag is set (the default), typing the QUIT character causes a *quit* signal (`SIGQUIT`) to be sent to the terminal's foreground process group (Section 34.2). The QUIT character itself is not passed to the reading process.

REPRINT

REPRINT is the *reprint input* character. In canonical mode with the `IEXTEN` flag set (the default), typing this character causes the current (as yet incomplete) line of input to be redisplayed on the terminal. This is useful if some other program (e.g., *wall(1)* or *write(1)*) has written output that has messed up the terminal display. The REPRINT character itself is not passed to the reading process.

START and STOP

START and STOP are the *start output* and *stop output* characters, which operate if the `IXON` (*enable start/stop output control*) flag is enabled (the default). (The START and STOP characters are not honored by some terminal emulators.)

Typing the STOP character suspends terminal output. The STOP character itself is not passed to the reading process. If the `IXOFF` flag is set and the terminal

input queue is full, then the terminal driver automatically sends a STOP character to throttle the input.

Typing the START character causes terminal output to resume after previously being stopped by the STOP character. The START character itself is not passed to the reading process. If the IXOFF (*enable start/stop input control*) flag is set (this flag is disabled by default) and the terminal driver had previously sent a STOP character because the input queue was full, the terminal driver automatically generates a START character when space once more becomes available in the input queue.

If the IXANY flag is set, then any character, not just START, may be typed in order to restart output (and that character is similarly not passed to the reading process).

The START and STOP characters are used for software flow control in either direction between the computer and the terminal device. One function of these characters is to allow users to stop and start terminal output. This is output flow control, as enabled by IXON. However, flow control in the other direction (i.e., control of input flow from the device to the computer, as enabled by IXOFF) is also important when, for example, the device in question is a modem or another computer. Input flow control makes sure that no data is lost if the application is slow to handle input and the kernel buffers fill up.

With the higher line speeds that are nowadays typical, software flow control has been superseded by hardware (RTS/CTS) flow control, whereby data flow is enabled and disabled using signals sent via separate wires on the serial port. (RTS stands for *Request To Send*, and CTS stands for *Clear To Send*.)

SUSP

SUSP is the *suspend* character. If the ISIG flag is set (the default), typing this character causes a *terminal suspend* signal (SIGTSTP) to be sent to the terminal's foreground process group (Section 34.2). The SUSP character itself is not passed to the reading process.

WERASE

WERASE is the *word erase* character. In canonical mode, with the IEXTEN flag set (the default), typing this character erases all characters back to the beginning of the previous word. A word is considered to be a sequence of letters, digits, and the underscore character. (On some UNIX implementations, a word is considered to be delimited by white space.)

Other terminal special characters

Other UNIX implementations provide terminal special characters in addition to those listed in Table 62-1.

BSD provides the DSUSP and STATUS characters. The DSUSP character (typically *Control-Y*) operates in a fashion similar to the SUSP character, but suspends the foreground process group only when it attempts to read the character (i.e., after all preceding input has been read). Several non-BSD-derived implementations also provide the DSUSP character.

The STATUS character (typically *Control-T*) causes the kernel to display status information on the terminal (including the state of the foreground process and how much CPU time it has consumed), and sends a SIGINFO signal to the foreground process group. If desired, processes can catch this signal and display further status

information. (Linux provides a vaguely similar feature in the form of the *magic SysRq key*; see the kernel source file `Documentation/sysrq.txt` for details.)

System V derivatives provide the SWTCH character. This character is used to switch shells under *shell layers*, a System V predecessor to job control.

Example program

Listing 62-1 shows the use of *tcgetattr()* and *tcsetattr()* to change the terminal *interrupt* character. This program sets the *interrupt* character to be the character whose numeric value is supplied as the program's command-line argument, or disables the *interrupt* character if no command-line argument is supplied.

The following shell session demonstrates the use of this program. We begin by setting the *interrupt* character to *Control-L* (ASCII 12), and then verify the change with *stty*:

```
$ ./new_intr 12
$ stty
speed 38400 baud; line = 0;
intr = ^L;
```

We then start a process running *sleep(1)*. We find that typing *Control-C* no longer has its usual effect of terminating a process, but typing *Control-L* does terminate the process.

```
$ sleep 10
^C                              Control-C has no effect; it is just echoed
Type Control-L to terminate sleep
```

We now display the value of the shell *$?* variable, which shows the termination status of the last command:

```
$ echo $?
130
```

We see that the termination status of the process was 130. This indicates that the process was killed by signal 130 − 128 = 2; signal number 2 is `SIGINT`.

Next, we use our program to disable the *interrupt* character.

```
$ ./new_intr
$ stty                          Verify the change
speed 38400 baud; line = 0;
intr = <undef>;
```

Now we find that neither *Control-C* nor *Control-L* generates a `SIGINT` signal, and we must instead use *Control-* to terminate a program:

```
$ sleep 10
^C^L                            Control-C and Control-L are simply echoed
Type Control-\ to generate SIGQUIT
Quit
$ stty sane                     Return terminal to a sane state
```

Listing 62-1: Changing the terminal *interrupt* character

tty/new_intr.c

```
#include <termios.h>
#include <ctype.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct termios tp;
    int intrChar;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [intr-char]\n", argv[0]);

    /* Determine new INTR setting from command line */

    if (argc == 1) {                                    /* Disable */
        intrChar = fpathconf(STDIN_FILENO, _PC_VDISABLE);
        if (intrChar == -1)
            errExit("Couldn't determine VDISABLE");
    } else if (isdigit((unsigned char) argv[1][0])) {
        intrChar = strtoul(argv[1], NULL, 0);           /* Allows hex, octal */
    } else {                                            /* Literal character */
        intrChar = argv[1][0];
    }

    /* Fetch current terminal settings, modify INTR character, and
       push changes back to the terminal driver */

    if (tcgetattr(STDIN_FILENO, &tp) == -1)
        errExit("tcgetattr");
    tp.c_cc[VINTR] = intrChar;
    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &tp) == -1)
        errExit("tcsetattr");

    exit(EXIT_SUCCESS);
}
```

tty/new_intr.c

62.5 Terminal Flags

Table 62-2 lists the settings controlled by each of the four flag fields of the *termios* structure. The constants listed in this table correspond to single bits, except those specifying the term *mask*, which are values spanning several bits; these may contain one of a range of values, shown in parentheses. The column labeled *SUSv3* indicates whether the flag is specified in SUSv3. The *Default* column shows the default settings for a virtual console login.

Many shells that provide command-line editing facilities perform their own manipulations of the flags listed in Table 62-2. This means that if we try using *stty(1)* to experiment with these settings, then the changes may not be effective when entering shell commands. To circumvent this behavior, we must disable command-line editing in the shell. For example, command-line editing can be disabled by specifying the command-line option *--noediting* when invoking *bash*.

Table 62-2: Terminal flags

Field/Flag	Description	Default	SUSv3
c_iflag			
BRKINT	Signal interrupt (SIGINT) on BREAK condition	on	•
ICRNL	Map CR to NL on input	on	•
IGNBRK	Ignore BREAK condition	off	•
IGNCR	Ignore CR on input	off	•
IGNPAR	Ignore characters with parity errors	off	•
IMAXBEL	Ring bell when terminal input queue is full (unused)	(on)	
INLCR	Map NL to CR on input	off	•
INPCK	Enable input parity checking	off	•
ISTRIP	Strip high bit (bit 8) from input characters	off	•
IUTF8	Input is UTF-8 (since Linux 2.6.4)	off	
IUCLC	Map uppercase to lowercase on input (if IEXTEN also set)	off	
IXANY	Allow any character to restart stopped output	off	•
IXOFF	Enable start/stop input flow control	off	•
IXON	Enable start/stop output flow control	on	•
PARMRK	Mark parity errors (with 2 prefix bytes: 0377 + 0)	off	•
c_oflag			
BSDLY	Backspace delay mask (BS0, BS1)	BS0	•
CRDLY	CR delay mask (CR0, CR1, CR2, CR3)	CR0	•
FFDLY	Form-feed delay mask (FF0, FF1)	FF0	•
NLDLY	Newline delay mask (NL0, NL1)	NL0	•
OCRNL	Map CR to NL on output (see also ONOCR)	off	•
OFDEL	Use DEL (0177) as fill character; otherwise NUL (0)	off	•
OFILL	Use fill characters for delay (rather than timed delay)	off	•
OLCUC	Map lowercase to uppercase on output	off	
ONLCR	Map NL to CR-NL on output	on	•
ONLRET	Assume NL performs CR function (move to start of line)	off	•
ONOCR	Don't output CR if already at column 0 (start of line)	off	•
OPOST	Perform output postprocessing	on	•
TABDLY	Horizontal-tab delay mask (TAB0, TAB1, TAB2, TAB3)	TAB0	•
VTDLY	Vertical-tab delay mask (VT0, VT1)	VT0	•
c_cflag			
CBAUD	Baud (bit rate) mask (B0, B2400, B9600, and so on)	B38400	
CBAUDEX	Extended baud (bit rate) mask (for rates > 38,400)	off	
CIBAUD	Input baud (bit rate), if different from output (unused)	(off)	
CLOCAL	Ignore modem status lines (don't check carrier signal)	off	•
CMSPAR	Use "stick" (mark/space) parity	off	

Table 62-2: Terminal flags (continued)

Field/Flag	Description	Default	SUSv3
CREAD	Allow input to be received	on	•
CRTSCTS	Enable RTS/CTS (hardware) flow control	off	
CSIZE	Character-size mask (5 to 8 bits: CS5, CS6, CS7, CS8)	CS8	•
CSTOPB	Use 2 stop bits per character; otherwise 1	off	•
HUPCL	Hang up (drop modem connection) on last close	on	•
PARENB	Parity enable	off	•
PARODD	Use odd parity; otherwise even	off	•
c_lflag			
ECHO	Echo input characters	on	•
ECHOCTL	Echo control characters visually (e.g., ^L)	on	
ECHOE	Perform ERASE visually	on	•
ECHOK	Echo KILL visually	on	•
ECHOKE	Don't output a newline after echoed KILL	on	
ECHONL	Echo NL (in canonical mode) even if echoing is disabled	off	•
ECHOPRT	Echo deleted characters backward (between \ and /)	off	
FLUSHO	Output is being flushed (unused)	-	
ICANON	Canonical mode (line-by-line) input	on	•
IEXTEN	Enable extended processing of input characters	on	•
ISIG	Enable signal-generating characters (INTR, QUIT, SUSP)	on	•
NOFLSH	Disable flushing on INTR, QUIT, and SUSP	off	•
PENDIN	Redisplay pending input at next read (not implemented)	(off)	
TOSTOP	Generate SIGTTOU for background output (Section 34.7.1)	off	•
XCASE	Canonical upper/lowercase presentation (unimplemented)	(off)	

Several of the flags listed in Table 62-2 were provided for historical terminals with limited capabilities, and these flags have little use on modern systems. For example, the IUCLC, OLCUC, and XCASE flags were used with terminals that were capable of displaying only uppercase letters. On many older UNIX systems, if a user tried logging in with an uppercase username, the *login* program assumed that the user was sitting at such a terminal and would set these flags, and then the following password prompt would appear:

```
\PASSWORD:
```

From this point on, all lowercase characters were output in uppercase, and real uppercase characters were preceded by a backslash (\). Similarly, for input, a real uppercase character could be specified by a preceding backslash. The ECHOPRT flag was also designed for limited-capability terminals.

The various delay masks are also historical, allowing for terminals and printers that took longer to echo characters such as carriage return and form feed. The related OFILL and OFDEL flags specified how such a delay was to be performed. Most of these flags are unused on Linux. One exception is the TAB3 setting for the TABDLY mask, which causes tab characters to be output as (up to eight) spaces.

The following paragraphs provide more details about some of the *termios* flags.

BRKINT

If the BRKINT flag is set and the IGNBRK flag is not set, then a SIGINT signal is sent to the foreground process group when a BREAK condition occurs.

> Most conventional dumb terminals provided a *BREAK* key. Pressing this key doesn't actually generate a character, but instead causes a *BREAK condition*, whereby a series of 0 bits is sent to the terminal driver for a specified length of time, typically 0.25 or 0.5 seconds (i.e., longer than the time required to transmit a single byte). (Unless the IGNBRK flag has been set, the terminal driver consequently delivers a single 0 byte to the reading process.) On many UNIX systems, the BREAK condition acted as a signal to a remote host to change its line speed (baud) to something suitable for the terminal. Thus, the user would press the *BREAK* key until a valid login prompt appeared, indicating that the line speed was now suitable for this terminal.
>
> On a virtual console, we can generate a BREAK condition by pressing *Control-Break*.

ECHO

Setting the ECHO flag enables echoing of input characters. Disabling echoing is useful when reading passwords. Echoing is also disabled within the command mode of *vi*, where keyboard characters are interpreted as editing commands rather than text input. The ECHO flag is effective in both canonical and noncanonical modes.

ECHOCTL

If ECHO is set, then enabling the ECHOCTL flag causes control characters other than tab, newline, START, and STOP to be echoed in the form ^A (for *Control-A*), and so on. If ECHOCTL is disabled, control characters are not echoed.

> The control characters are those with ASCII codes less than 32, plus the DEL character (127 decimal). A control character, *x*, is echoed using a caret (^) followed by the character resulting from the expression *(x ^ 64)*. For all characters except DEL, the effect of the XOR (^) operator in this expression is to add 64 to the value of the character. Thus, *Control-A* (ASCII 1) is echoed as caret plus A (ASCII 65). For DEL, the expression has the effect of subtracting 64 from 127, yielding the value 63, the ASCII code for ?, so that DEL is echoed as ^?.

ECHOE

In canonical mode, setting the ECHOE flag causes ERASE to be performed visually, by outputting the sequence backspace-space-backspace to the terminal. If ECHOE is disabled, then the ERASE character is instead echoed (e.g., as ^?), but still performs its function of deleting a character.

ECHOK and ECHOKE

The ECHOK and ECHOKE flags control the visual display when using the KILL (erase line) character in canonical mode. In the default case (both flags enabled), a line is erased visually (see ECHOE). If either of these flags is disabled, then a visual erase is not performed (but the input line is still discarded), and the KILL character is echoed (e.g., as ^U). If ECHOK is set, but not ECHOKE, then a newline character is also output.

ICANON

Setting the ICANON flag enables canonical mode input. Input is gathered into lines, and special interpretation of the EOF, EOL, EOL2, ERASE, LNEXT, KILL, REPRINT, and WERASE characters is enabled (but note the effect of the IEXTEN flag described below).

IEXTEN

Setting the IEXTEN flag enables extended processing of input characters. This flag (as well as ICANON) must be set in order for the following characters to be interpreted: EOL2, LNEXT, REPRINT, and WERASE. The IEXTEN flag must also be set for the IUCLC flag to be effective. SUSv3 merely says that the IEXTEN flag enables extended (implementation-defined) functions; details may vary on other UNIX implementations.

IMAXBEL

On Linux, the setting of the IMAXBEL flag is ignored. On a console login, the bell is always rung when the input queue is full.

IUTF8

Setting the IUTF8 flag enables cooked mode (Section 62.6.3) to correctly handle UTF-8 input when performing line editing.

NOFLSH

By default, when a signal is generated by typing the INTR, QUIT, or SUSP character, any outstanding data in the terminal input and output queues is flushed (discarded). Setting the NOFLSH flag disables such flushing.

OPOST

Setting the OPOST flag enables output postprocessing. This flag must be set in order for any of the flags in the *c_oflag* field of the *termios* structure to be effective. (Put conversely, disabling the OPOST flag prevents all output postprocessing.)

PARENB, IGNPAR, INPCK, PARMRK, and PARODD

The PARENB, IGNPAR, INPCK, PARMRK, and PARODD flags are concerned with parity generation and checking.

The PARENB flag enables generation of parity check bits for output characters and parity checking for input characters. If we want to perform only output parity generation, then we can disable input parity checking by turning INPCK off. If the PARODD flag is set, then odd parity is used in both cases; otherwise, even parity is used.

The remaining flags specify how an input character with parity errors should be handled. If the IGNPAR flag is set, the character is discarded (not passed to the reading process). Otherwise, if the PARMRK flag is set, then the character is passed to the reading process, but is preceded by the 2-byte sequence 0377 + 0. (If the PARMRK flag is set and ISTRIP is clear, then a real 0377 character is doubled to become 0377 + 0377.) If PARMRK is not set, but INPCK is set, then the character is discarded, and a 0 byte is

passed to the reading process instead. If none of IGNPAR, PARMRK, or INPCK is set, then the character is passed as is to the reading process.

Example program

Listing 62-2 demonstrates the use of *tcgetattr()* and *tcsetattr()* to turn off the ECHO flag, so that input characters are not echoed. Here is an example of what we see when running this program:

```
$ ./no_echo
Enter text:                             We type some text, which is not echoed,
Read: Knock, knock, Neo.                but was nevertheless read
```

Listing 62-2: Disabling terminal echoing

tty/no_echo.c

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

#define BUF_SIZE 100

int
main(int argc, char *argv[])
{
    struct termios tp, save;
    char buf[BUF_SIZE];

    /* Retrieve current terminal settings, turn echoing off */

    if (tcgetattr(STDIN_FILENO, &tp) == -1)
        errExit("tcgetattr");
    save = tp;                          /* So we can restore settings later */
    tp.c_lflag &= ~ECHO;                /* ECHO off, other bits unchanged */
    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &tp) == -1)
        errExit("tcsetattr");

    /* Read some input and then display it back to the user */

    printf("Enter text: ");
    fflush(stdout);
    if (fgets(buf, BUF_SIZE, stdin) == NULL)
        printf("Got end-of-file/error on fgets()\n");
    else
        printf("\nRead: %s", buf);

    /* Restore original terminal settings */

    if (tcsetattr(STDIN_FILENO, TCSANOW, &save) == -1)
        errExit("tcsetattr");

    exit(EXIT_SUCCESS);
}
```

tty/no_echo.c

62.6 Terminal I/O Modes

We have already noted that the terminal driver is capable of handling input in either canonical or noncanonical mode, depending on the setting of the ICANON flag. We now describe these two modes in detail. We then describe three useful terminal modes—cooked, cbreak, and raw—that were available in Seventh Edition UNIX, and show how these modes are emulated on modern UNIX systems by setting appropriate values in the *termios* structure.

62.6.1 Canonical Mode

Canonical mode input is enabled by setting the ICANON flag. Terminal input in canonical mode is distinguished by the following features:

- Input is gathered into lines, terminated by one of the line-delimiter characters: NL, EOL, EOL2 (if the IEXTEN flag is set), EOF (at anything other than the initial position in the line), or CR (if the ICRNL flag is enabled). Except in the case of EOF, the line delimiter is passed back to the reading process (as the last character in the line).
- Line editing is enabled, so that the current line of input can be modified. Thus, the following characters are enabled: ERASE, KILL, and, if the IEXTEN flag is set, WERASE.
- If the IEXTEN flag is set, the REPRINT and LNEXT characters are also enabled.

In canonical mode, a terminal *read()* returns when a complete line of input is available. (The *read()* itself may fetch only part of that line if it requested fewer bytes; remaining bytes will be fetched by subsequent calls to *read()*.) A *read()* may also terminate if interrupted by a signal handler and restarting of system calls is not enabled for this signal (Section 21.5).

> While describing the NOFLSH flag in Section 62.5, we noted that the characters that generate signals also cause the terminal driver to flush the terminal input queue. This flushing occurs regardless of whether the signal is caught or ignored by an application. We can prevent such flushing by enabling the NOFLSH flag.

62.6.2 Noncanonical Mode

Some applications (e.g., *vi* and *less*) need to read characters from the terminal without the user supplying a line delimiter. Noncanonical mode is provided for this purpose. In noncanonical mode (ICANON unset), no special input processing is performed. In particular, input is no longer gathered into lines, but is instead available immediately.

In what circumstances does a noncanonical *read()* complete? We can specify that a noncanonical *read()* terminates after a certain time, after a certain number of bytes have been read, or both in combination. Two elements of the *termios c_cc* array determine the behavior: TIME and MIN. The TIME element (indexed using the constant VTIME) specifies a timeout value in tenths of a second. The MIN element (indexed using VMIN) specifies the minimum number of bytes to be read. (The MIN and TIME settings have no effect on canonical-mode terminal I/O.)

The precise operation and interaction of the MIN and TIME parameters depends on whether they each have nonzero values. The four possible cases are described below. Note that in all cases, if, at the time of a *read()*, sufficient bytes are already available to satisfy the requirements specified by MIN, *read()* returns immediately with the lesser of the number of bytes available and the number of bytes requested.

MIN == 0, TIME == 0 (polling read)

If data is available at the time of the call, then *read()* returns immediately with the lesser of the number of bytes available or the number of bytes requested. If no bytes are available, *read()* completes immediately, returning 0.

This case serves the usual requirements of polling, allowing the application to check if input is available without blocking if it is not. This mode is somewhat similar to setting the `O_NONBLOCK` flag for the terminal (Section 5.9). However, with `O_NONBLOCK`, if no bytes are available for reading, then *read()* returns −1 with the error `EAGAIN`.

MIN > 0, TIME == 0 (blocking read)

The *read()* blocks (possibly indefinitely) until the lesser of the number of bytes requested or MIN bytes are available, and returns the lesser of the two values.

Programs such as *less* typically set MIN to 1 and TIME to 0. This allows the program to wait for single key presses without needing to waste CPU time by polling in a busy loop.

If a terminal is placed in noncanonical mode with MIN set to 1 and TIME set to 0, then the techniques described in Chapter 63 can be used to check whether a single character (rather than a complete line) has been typed at the terminal.

MIN == 0, TIME > 0 (read with timeout)

A timer is started when *read()* is called. The call returns as soon as at least 1 byte is available, or when TIME tenths of a second have elapsed. In the latter case, *read()* returns 0.

This case is useful for programs talking to a serial device (e.g., a modem). The program can send data to the device and then wait for a response, using a timeout to avoid hanging forever in case the device doesn't respond.

MIN > 0, TIME > 0 (read with interbyte timeout)

After the initial byte of input becomes available, a timer is restarted as each further byte is received. The *read()* returns when either the lesser of MIN bytes or the number of bytes requested have been read, or when the time between receiving successive bytes exceeds TIME tenths of a second. Since the timer is started only after the initial byte becomes available, at least 1 byte is returned. (A *read()* can block indefinitely for this case.)

This case is useful for handling terminal keys that generate escape sequences. For example, on many terminals, the left-arrow key generates the 3-character sequence consisting of *Escape* followed by `OD`. These characters are transmitted in quick succession. Applications handling such sequences need to distinguish the pressing of such a key from the situation where the user slowly types each of the

characters individually. This can be done by performing a *read()* with a small inter-byte timeout, say 0.2 seconds. Such a technique is used in the command mode of some versions of *vi*. (Depending on the length of the timeout, in such applications, we may be able to simulate a left-arrow key press by quickly typing the aforementioned 3-character sequence.)

Portably modifying and restoring MIN and TIME

For historical compatibility with some UNIX implementations, SUSv3 allows the values of the VMIN and VTIME constants to be the same as VEOF and VEOL, respectively, which means that these elements of the *termios c_cc* array may coincide. (On Linux, the values of these constants are distinct.) This is possible because VEOF and VEOL are unused in noncanonical mode. The fact that VMIN and VEOF may have the same value means that caution is needed in a program that enters noncanonical mode, sets MIN (typically to 1), and then later returns to canonical mode. On return to canonical mode, EOF will no longer have its usual value of ASCII 4 (*Control-D*). The portable way to deal with this problem is to save a copy of the *termios* settings prior to changing to noncanonical mode, and then use this saved structure to return to canonical mode.

62.6.3 Cooked, Cbreak, and Raw Modes

The terminal driver in Seventh Edition UNIX (as well as in early versions of BSD) was capable of handling input in three modes: *cooked*, *cbreak*, and *raw*. The differences between the three modes are summarized in Table 62-3.

Table 62-3: Differences between cooked, cbreak, and raw terminal modes

Feature	Mode		
	Cooked	**Cbreak**	**Raw**
Input available	line by line	char. by char.	char. by char.
Line-editing?	yes	no	no
Signal-generating characters interpreted?	yes	yes	no
START/STOP interpreted?	yes	yes	no
Other special characters interpreted?	yes	no	no
Other input processing performed?	yes	yes	no
Other output processing performed?	yes	yes	no
Input echoed?	yes	maybe	no

Cooked mode was essentially canonical mode with all of the default special character processing enabled (i.e., interpretation of CR, NL, and EOF; enabling of line editing; handling of signal-generating characters; ICRNL; OCRNL; and so on).

Raw mode was the converse: noncanonical mode, with all input and output processing, as well as echoing, switched off. (An application that needed to ensure that the terminal driver makes absolutely no changes to the data transferred across a serial line would use this mode.)

Cbreak mode was intermediate between cooked and raw modes. Input was noncanonical, but signal-generating characters were interpreted, and the various input and output transformations could still occur (depending on individual flag

settings). Cbreak mode did not disable echoing, but applications employing this mode would usually disable echoing as well. Cbreak mode was useful in screen-handling applications (such as *less*) that permitted character-by-character input, but still needed to allow interpretation of characters such as INTR, QUIT, and SUSP.

Example: setting raw and cbreak mode

In the Seventh Edition and the original BSD terminal drivers, it was possible to switch to raw or cbreak mode by tweaking single bits (called `RAW` and `CBREAK`) in the terminal driver data structures. With the transition to the POSIX *termios* interface (now supported on all UNIX implementations), single bits for selecting raw and cbreak mode are no longer available, and applications emulating these modes must explicitly change the required fields of the *termios* structure. Listing 62-3 provides two functions, *ttySetCbreak()* and *ttySetRaw()*, that implement the equivalents of these terminal modes.

> Applications that use the *ncurses* library can call the functions *cbreak()* and *raw()*, which perform similar tasks to our functions in Listing 62-3.

Listing 62-3: Switching a terminal to cbreak and raw modes

tty/tty_functions.c

```
#include <termios.h>
#include <unistd.h>
#include "tty_functions.h"              /* Declares functions defined here */

/* Place terminal referred to by 'fd' in cbreak mode (noncanonical mode
   with echoing turned off). This function assumes that the terminal is
   currently in cooked mode (i.e., we shouldn't call it if the terminal
   is currently in raw mode, since it does not undo all of the changes
   made by the ttySetRaw() function below). Return 0 on success, or -1
   on error. If 'prevTermios' is non-NULL, then use the buffer to which
   it points to return the previous terminal settings. */

int
ttySetCbreak(int fd, struct termios *prevTermios)
{
    struct termios t;

    if (tcgetattr(fd, &t) == -1)
        return -1;

    if (prevTermios != NULL)
        *prevTermios = t;

    t.c_lflag &= ~(ICANON | ECHO);
    t.c_lflag |= ISIG;

    t.c_iflag &= ~ICRNL;
```

```
    t.c_cc[VMIN] = 1;                   /* Character-at-a-time input */
    t.c_cc[VTIME] = 0;                  /* with blocking */

    if (tcsetattr(fd, TCSAFLUSH, &t) == -1)
        return -1;

    return 0;
}

/* Place terminal referred to by 'fd' in raw mode (noncanonical mode
   with all input and output processing disabled). Return 0 on success,
   or -1 on error. If 'prevTermios' is non-NULL, then use the buffer to
   which it points to return the previous terminal settings. */

int
ttySetRaw(int fd, struct termios *prevTermios)
{
    struct termios t;

    if (tcgetattr(fd, &t) == -1)
        return -1;

    if (prevTermios != NULL)
        *prevTermios = t;

    t.c_lflag &= ~(ICANON | ISIG | IEXTEN | ECHO);
                        /* Noncanonical mode, disable signals, extended
                           input processing, and echoing */

    t.c_iflag &= ~(BRKINT | ICRNL | IGNBRK | IGNCR | INLCR |
                       INPCK | ISTRIP | IXON | PARMRK);
                        /* Disable special handling of CR, NL, and BREAK.
                           No 8th-bit stripping or parity error handling.
                           Disable START/STOP output flow control. */

    t.c_oflag &= ~OPOST;                /* Disable all output processing */

    t.c_cc[VMIN] = 1;                   /* Character-at-a-time input */
    t.c_cc[VTIME] = 0;                  /* with blocking */

    if (tcsetattr(fd, TCSAFLUSH, &t) == -1)
        return -1;

    return 0;
}
```

tty/tty_functions.c

A program that places the terminal in raw or cbreak mode must be careful to return the terminal to a usable mode when it terminates. Among other tasks, this entails handling all of the signals that are likely to be sent to the program, so that

the program is not prematurely terminated. (Job-control signals can still be generated from the keyboard in cbreak mode.)

An example of how to do this is provided in Listing 62-4. This program performs the following steps:

- Set the terminal to either cbreak mode ⑨ or raw mode ⑫, depending on whether a command-line argument (any string) is supplied ⑧. The previous terminal settings are saved in the global variable *userTermios* ①.
- If the terminal was placed in cbreak mode, then signals can be generated from the terminal. These signals need to be handled so that terminating or suspending the program leaves the terminal in a state that the user expects. The program installs the same handler for `SIGQUIT` and `SIGINT` ⑩. The `SIGTSTP` signal requires special treatment, so a different handler is installed for that signal ⑪.
- Install a handler for the `SIGTERM` signal, in order to catch the default signal sent by the *kill* command ⑬.
- Execute a loop that reads characters one at a time from *stdin* and echoes them on standard output ⑭. The program treats various input characters specially before outputting them ⑮:
  - All letters are converted to lowercase before being output.
  - The newline (`\n`) and carriage return (`\r`) characters are echoed without change.
  - Control characters other than the newline and carriage return are echoed as a 2-character sequence: caret (`^`) plus the corresponding uppercase letter (e.g., *Control-A* echoes as `^A`).
  - All other characters are echoed as asterisks (`*`).
  - The letter *q* causes the loop to terminate ⑯.
- On exit from the loop, restore the terminal to its state as last set by the user, and then terminate ⑰.

The program installs the same handler for `SIGQUIT`, `SIGINT`, and `SIGTERM`. This handler restores the terminal to its state as last set by the user and terminates the program ②.

The handler for the `SIGTSTP` signal ③ deals with the signal in the manner described in Section 34.7.3. Note the following additional details of the operation of this signal handler:

- Upon entry, the handler saves the current terminal settings (in *ourTermios*) ④, and then resets the terminal to the settings that were in effect (saved in *userTermios*) when the program was started ⑤, before once more raising `SIGTSTP` to actually stop the process.
- Upon resumption of execution after receipt of `SIGCONT`, the handler once more saves the current terminal settings in *userTermios* ⑥, since the user may have changed the settings while the program was stopped (using the *stty* command, for example). The handler then returns the terminal to the state (*ourTermios*) required by the program ⑦.

Listing 62-4: Demonstrating cbreak and raw modes

tty/test_tty_functions.c

```
#include <termios.h>
#include <signal.h>
#include <ctype.h>
#include "tty_functions.h"              /* Declarations of ttySetCbreak()
                                           and ttySetRaw() */
#include "tlpi_hdr.h"

① static struct termios userTermios;
                        /* Terminal settings as defined by user */

   static void             /* General handler: restore tty settings and exit */
   handler(int sig)
   {
②      if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &userTermios) == -1)
           errExit("tcsetattr");
       _exit(EXIT_SUCCESS);
   }

   static void             /* Handler for SIGTSTP */
③ tstpHandler(int sig)
   {
       struct termios ourTermios;          /* To save our tty settings */
       sigset_t tstpMask, prevMask;
       struct sigaction sa;
       int savedErrno;

       savedErrno = errno;                 /* We might change 'errno' here */

       /* Save current terminal settings, restore terminal to
          state at time of program startup */

④      if (tcgetattr(STDIN_FILENO, &ourTermios) == -1)
           errExit("tcgetattr");
⑤      if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &userTermios) == -1)
           errExit("tcsetattr");

       /* Set the disposition of SIGTSTP to the default, raise the signal
          once more, and then unblock it so that we actually stop */

       if (signal(SIGTSTP, SIG_DFL) == SIG_ERR)
           errExit("signal");
       raise(SIGTSTP);

       sigemptyset(&tstpMask);
       sigaddset(&tstpMask, SIGTSTP);
       if (sigprocmask(SIG_UNBLOCK, &tstpMask, &prevMask) == -1)
           errExit("sigprocmask");

       /* Execution resumes here after SIGCONT */

       if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
           errExit("sigprocmask");         /* Reblock SIGTSTP */
```

```
        sigemptyset(&sa.sa_mask);           /* Reestablish handler */
        sa.sa_flags = SA_RESTART;
        sa.sa_handler = tstpHandler;
        if (sigaction(SIGTSTP, &sa, NULL) == -1)
            errExit("sigaction");

        /* The user may have changed the terminal settings while we were
           stopped; save the settings so we can restore them later */

⑥       if (tcgetattr(STDIN_FILENO, &userTermios) == -1)
            errExit("tcgetattr");

        /* Restore our terminal settings */

⑦       if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &ourTermios) == -1)
            errExit("tcsetattr");

        errno = savedErrno;
    }

    int
    main(int argc, char *argv[])
    {
        char ch;
        struct sigaction sa, prev;
        ssize_t n;

        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_RESTART;

⑧       if (argc > 1) {                     /* Use cbreak mode */
⑨           if (ttySetCbreak(STDIN_FILENO, &userTermios) == -1)
                errExit("ttySetCbreak");

            /* Terminal special characters can generate signals in cbreak
               mode. Catch them so that we can adjust the terminal mode.
               We establish handlers only if the signals are not being ignored. */

⑩           sa.sa_handler = handler;

            if (sigaction(SIGQUIT, NULL, &prev) == -1)
                errExit("sigaction");
            if (prev.sa_handler != SIG_IGN)
                if (sigaction(SIGQUIT, &sa, NULL) == -1)
                    errExit("sigaction");

            if (sigaction(SIGINT, NULL, &prev) == -1)
                errExit("sigaction");
            if (prev.sa_handler != SIG_IGN)
                if (sigaction(SIGINT, &sa, NULL) == -1)
                    errExit("sigaction");

⑪           sa.sa_handler = tstpHandler;
```

```
            if (sigaction(SIGTSTP, NULL, &prev) == -1)
                errExit("sigaction");
            if (prev.sa_handler != SIG_IGN)
                if (sigaction(SIGTSTP, &sa, NULL) == -1)
                    errExit("sigaction");
        } else {                            /* Use raw mode */
⑫          if (ttySetRaw(STDIN_FILENO, &userTermios) == -1)
                errExit("ttySetRaw");
        }

⑬      sa.sa_handler = handler;
        if (sigaction(SIGTERM, &sa, NULL) == -1)
            errExit("sigaction");

        setbuf(stdout, NULL);               /* Disable stdout buffering */

⑭      for (;;) {                          /* Read and echo stdin */
            n = read(STDIN_FILENO, &ch, 1);
            if (n == -1) {
                errMsg("read");
                break;
            }

            if (n == 0)                     /* Can occur after terminal disconnect */
                break;

⑮          if (isalpha((unsigned char) ch))          /* Letters --> lowercase */
                putchar(tolower((unsigned char) ch));
            else if (ch == '\n' || ch == '\r')
                putchar(ch);
            else if (iscntrl((unsigned char) ch))
                printf("^%c", ch ^ 64);     /* Echo Control-A as ^A, etc. */
            else
                putchar('*');               /* All other chars as '*' */

⑯          if (ch == 'q')                  /* Quit loop */
                break;
        }

⑰      if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &userTermios) == -1)
            errExit("tcsetattr");
        exit(EXIT_SUCCESS);
    }
```

tty/test_tty_functions.c

Here is an example of what we see when we ask the program in Listing 62-4 to use raw mode:

```
$ stty                              Initial terminal mode is sane (cooked)
speed 38400 baud; line = 0;
$ ./test_tty_functions
abc                                 Type abc, and Control-J
   def                              Type DEF, Control-J, and Enter
^C^Z                                Type Control-C, Control-Z, and Control-J
    q$                              Type q to exit
```

In the last line of the preceding shell session, we see that the shell printed its prompt on the same line as the *q* character that caused the program to terminate.

The following shows an example run using cbreak mode:

```
$ ./test_tty_functions x
XYZ                                  Type XYZ and Control-Z
[1]+  Stopped       ./test_tty_functions x
$ stty                               Verify that terminal mode was restored
speed 38400 baud; line = 0;
$ fg                                 Resume in foreground
./test_tty_functions x
***                                  Type 123 and Control-J
   $                                 Type Control-C to terminate program
Press Enter to get next shell prompt
$ stty                               Verify that terminal mode was restored
speed 38400 baud; line = 0;
```

62.7 Terminal Line Speed (Bit Rate)

Different terminals (and serial lines) are capable of transmitting and receiving at different speeds (bits per second). The *cfgetispeed()* and *cfsetispeed()* functions retrieve and modify the input line speed. The *cfgetospeed()* and *cfsetospeed()* functions retrieve and modify the output line speed.

> The term *baud* is commonly used as a synonym for the terminal line speed (in bits per second), although this usage is not technically correct. Precisely, baud is the per-second rate at which signal changes can occur on the line, which is not necessarily the same as the number of bits transmitted per second, since the latter depends on how bits are encoded into signals. Nevertheless, the term *baud* continues to be used synonymously with *bit rate* (bits per second). (The term *baud rate* is often also used synonymously with *baud*, but this is redundant; the baud is by definition a rate.) To avoid confusion, we'll generally use terms like *line speed* or *bit rate*.

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);
speed_t cfgetospeed(const struct termios *termios_p);
```

Both return a line speed from given *termios* structure

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

Both return 0 on success, or −1 on error

Each of these functions works on a *termios* structure that must be previously initialized by a call to *tcgetattr()*.

For example, to find out the current terminal output line speed, we would do the following:

```
struct termios tp;
speed_t rate;

if (tcgetattr(fd, &tp) == -1)
    errExit("tcgetattr");
rate = cfgetospeed(&tp);
if (rate == -1)
    errExit("cfgetospeed");
```

If we then wanted to change this line speed, we would continue as follows:

```
if (cfsetospeed(&tp, B38400) == -1)
    errExit("cfsetospeed");
if (tcsetattr(fd, TCSAFLUSH, &tp) == -1)
    errExit("tcsetattr");
```

The *speed_t* data type is used to store a line speed. Rather than directly assigning numeric values for line speeds, a set of symbolic constants (defined in `<termios.h>`) is used. These constants define a series of discrete values. Some examples of such constants are `B300`, `B2400`, `B9600`, and `B38400`, corresponding, respectively, to the line speeds 300, 2400, 9600, and 38,400 bits per second. The use of a set of discrete values reflects the fact that terminals are normally designed to work with a limited set of different (standardized) line speeds, derived from the division of some base rate (e.g., 115,200 is typical on PCs) by integral values (e.g., 115,200 / 12 = 9600).

SUSv3 specifies that the terminal line speeds are stored in the *termios* structure, but (deliberately) does not specify where. Many implementations, including Linux, maintain these values in the *c_cflag* field, using the `CBAUD` mask and the `CBAUDEX` flag. (In Section 62.2, we noted that the nonstandard *c_ispeed* and *c_ospeed* fields of the Linux *termios* structure are unused.)

Although the *cfsetispeed()* and *cfsetospeed()* functions allow separate input and output line speeds to be specified, on many terminals, these two speeds must be the same. Furthermore, Linux uses only a single field to store the line speed (i.e., the two rates are assumed to be always the same), which means that all of the input and output line-speed functions access the same *termios* field.

> Specifying *speed* as 0 in a call to *cfsetispeed()* means "set the input speed to whatever the output speed is when *tcsetattr()* is later called." This is useful on systems where the two line speeds are maintained as separate values.

62.8 Terminal Line Control

The *tcsendbreak()*, *tcdrain()*, *tcflush()*, and *tcflow()* functions perform tasks that are usually collectively grouped under the term *line control*. (These functions are POSIX inventions designed to replace various *ioctl()* operations.)

```
#include <termios.h>

int tcsendbreak(int fd, int duration);
int tcdrain(int fd);
int tcflush(int fd, int queue_selector);
int tcflow(int fd, int action);
```

All return 0 on success, or −1 on error

In each function, *fd* is a file descriptor that refers to a terminal or other remote device on a serial line.

The *tcsendbreak()* function generates a BREAK condition, by transmitting a continuous stream of 0 bits. The *duration* argument specifies the length of the transmission. If *duration* is 0, 0 bits are transmitted for 0.25 seconds. (SUSv3 specifies at least 0.25 and not more than 0.5 seconds.) If *duration* is greater than 0, 0 bits are transmitted for *duration* milliseconds. SUSv3 leaves this case unspecified; the handling of a nonzero *duration* varies widely on other UNIX implementations (the details described here are for *glibc*).

The *tcdrain()* function blocks until all output has been transmitted (i.e., until the terminal output queue has been emptied).

The *tcflush()* function flushes (discards) the data in the terminal input queue, the terminal output queue, or both queues (see Figure 62-1). Flushing the input queue discards data that has been received by the terminal driver but not yet read by any process. For example, an application can use *tcflush()* to discard terminal type-ahead before prompting for a password. Flushing the output queue discards data that has been written (passed to the terminal driver) but not yet transmitted to the device. The *queue_selector* argument specifies one of the values shown in Table 62-4.

> Note that the term *flush* is used in a different sense with *tcflush()* than when talking about file I/O. For file I/O, *flushing* means forcing the output to be transferred either user-space memory to the buffer cache in the case of the *stdio fflush()*, or from the buffer cache to the disk, in the case of *fsync()*, *fdatasync()*, and *sync()*.

Table 62-4: Values for the *tcflush() queue_selector* argument

Value	Description
TCIFLUSH	Flush the input queue
TCOFLUSH	Flush the output queue
TCIOFLUSH	Flush both the input and the output queues

The *tcflow()* function controls the flow of data in either direction between the computer and the terminal (or other remote device). The *action* argument is one of the values shown in Table 62-5. The TCIOFF and TCION values are effective only if the terminal is capable of interpreting STOP and START characters, in which case these operations respectively cause the terminal to suspend and resume sending data to the computer, respectively.

Table 62-5: Values for the *tcflush() action* argument

Value	Description
TCOOFF	Suspend output to the terminal
TCOON	Resume output to the terminal
TCIOFF	Transmit a STOP character to the terminal
TCION	Transmit a START character to the terminal

62.9 Terminal Window Size

In a windowing environment, a screen-handling application needs to be able to monitor the size of a terminal window, so that the screen can be redrawn appropriately if the user modifies the window size. The kernel provides two pieces of support to allow this to happen:

- A SIGWINCH signal is sent to the foreground process group after a change in the terminal window size. By default, this signal is ignored.
- At any time—usually following the receipt of a SIGWINCH signal—a process can use the *ioctl()* TIOCGWINSZ operation to find out the current size of the terminal window.

The *ioctl()* TIOCGWINSZ operation is used as follows:

```
if (ioctl(fd, TIOCGWINSZ, &ws) == -1)
    errExit("ioctl");
```

The *fd* argument is a file descriptor referring to a terminal window. The final argument to *ioctl()* is a pointer to a *winsize* structure (defined in <sys/ioctl.h>), used to return the terminal window size:

```
struct winsize {
    unsigned short ws_row;          /* Number of rows (characters) */
    unsigned short ws_col;          /* Number of columns (characters) */
    unsigned short ws_xpixel;       /* Horizontal size (pixels) */
    unsigned short ws_ypixel;       /* Vertical size (pixels) */
};
```

Like many other implementations, Linux doesn't use the pixel-size fields of the *winsize* structure.

Listing 62-5 demonstrates the use of the SIGWINCH signal and the *ioctl()* TIOCGWINSZ operation. The following shows an example of the output produced when this program is run under a window manager and the terminal window size is changed three times:

```
$ ./demo_SIGWINCH
Caught SIGWINCH, new window size: 35 rows * 80 columns
Caught SIGWINCH, new window size: 35 rows * 73 columns
Caught SIGWINCH, new window size: 22 rows * 73 columns
Type Control-C to terminate program
```

Listing 62-5: Monitoring changes in the terminal window size

tty/demo_SIGWINCH.c

```
#include <signal.h>
#include <termios.h>
#include <sys/ioctl.h>
#include "tlpi_hdr.h"

static void
sigwinchHandler(int sig)
{
}

int
main(int argc, char *argv[])
{
    struct winsize ws;
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigwinchHandler;
    if (sigaction(SIGWINCH, &sa, NULL) == -1)
        errExit("sigaction");

    for (;;) {
        pause();                        /* Wait for SIGWINCH signal */

        if (ioctl(STDIN_FILENO, TIOCGWINSZ, &ws) == -1)
            errExit("ioctl");
        printf("Caught SIGWINCH, new window size: "
                "%d rows * %d columns\n", ws.ws_row, ws.ws_col);
    }
}
```

tty/demo_SIGWINCH.c

It is also possible to change the terminal driver's notion of the window size by passing an initialized *winsize* structure in an *ioctl()* `TIOCSWINSZ` operation:

```
ws.ws_row = 40;
ws.ws_col = 100;
if (ioctl(fd, TIOCSWINSZ, &ws) == -1)
    errExit("ioctl");
```

If the new values in the *winsize* structure differ from the terminal driver's current notion of the terminal window size, two things happen:

- The terminal driver data structures are updated using the values supplied in the *ws* argument.
- A `SIGWINCH` signal is sent to the foreground process group of the terminal.

Note, however, that these events on their own are insufficient to change the actual dimensions of the displayed window, which are controlled by software outside the kernel (such as a window manager or a terminal emulator program).

Although not standardized in SUSv3, most UNIX implementations provide access to the terminal window size using the *ioctl()* operations described in this section.

62.10 Terminal Identification

In Section 34.4, we described the *ctermid()* function, which returns the name of the controlling terminal for a process (usually `/dev/tty` on UNIX systems). The functions described in this section are also useful for identifying a terminal.

The *isatty()* function enables us to determine whether a file descriptor, *fd*, is associated with a terminal (as opposed to some other file type).

```
#include <unistd.h>

int isatty(int fd);
```

Returns true (1) if *fd* is associated with a terminal, otherwise false (0)

The *isatty()* function is useful in editors and other screen-handling programs that need to determine whether their standard input and output are directed to a terminal.

Given a file descriptor, the *ttyname()* function returns the name of the associated terminal device.

```
#include <unistd.h>

char *ttyname(int fd);
```

Returns pointer to (statically allocated) string containing terminal name, or `NULL` on error

To find the name of the terminal, *ttyname()* uses the *opendir()* and *readdir()* functions described in Section 18.8 to walk through the directories holding terminal device names, looking at each directory entry until it finds one whose device ID (the *st_rdev* field of the *stat* structure) matches that of the device referred to by the file descriptor *fd*. Terminal device entries normally reside in two directories: `/dev` and `/dev/pts`. The `/dev` directory contains entries for virtual consoles (e.g., `/dev/tty1`) and BSD pseudoterminals. The `/dev/pts` directory contains entries for (System V-style) pseudoterminal slave devices. (We describe pseudoterminals in Chapter 64.)

A reentrant version of *ttyname()* exists in the form of *ttyname_r()*.

The *tty(1)* command, which displays the name of the terminal referred to by its standard input, is the command-line analog of *ttyname()*.

62.11 Summary

On early UNIX systems, terminals were real hardware devices connected to a computer via serial lines. Early terminals were not standardized, meaning that different escape sequences were required to program the terminals produced by different vendors. On modern workstations, such terminals have been superseded by bit-mapped monitors running the X Window System. However, the ability to program terminals is still required when dealing with virtual devices, such as virtual consoles and terminal emulators (which employ pseudoterminals), and real devices connected via serial lines.

Terminal settings (with the exception of the terminal window size) are maintained in a structure of type *termios*, which contains four bit-mask fields that control various terminal settings and an array that defines the various special characters interpreted by the terminal driver. The *tcgetattr()* and *tcsetattr()* functions allow a program to retrieve and modify the terminal settings.

When performing input, the terminal driver can operate in two different modes. In canonical mode, input is gathered into lines (terminated by one of the line-delimiter characters) and line editing is enabled. By contrast, noncanonical mode allows an application to read terminal input a character at a time, without needing to wait for the user to type a line-delimiter character. Line editing is disabled in noncanonical mode. Completion of a noncanonical mode read is controlled by the MIN and TIME fields of the *termios* structure, which determine the minimum number of characters to be read and a timeout to be applied to the read operation. We described four distinct cases for the operation of noncanonical reads.

Historically, the Seventh Edition and BSD terminal drivers provided three input modes—cooked, cbreak, and raw—which performed varying degrees of processing of terminal input and output. Cbreak and raw modes can be emulated by changing various fields within the *termios* structure.

A range of functions perform various other terminal operations. These include changing the terminal line speed and performing line-control operations (generating a break condition, pausing until output has been transmitted, flushing terminal input and output queues, and suspending or resuming transmission of data in either direction between the terminal and the computer). Other functions allow us to check if a given file descriptor refers to a terminal and to obtain the name of that terminal. The *ioctl()* system call can be used to retrieve and modify the terminal window size recorded by the kernel, and to perform a range of other terminal-related operations.

Further information

[Stevens, 1992] also describes terminal programming and goes into much more detail on programming serial ports. Several good online resources discuss terminal programming. Hosted at the LDP web site (*http://www.tldp.org*) are the *Serial HOWTO* and the *Text-terminal HOWTO*, both by David S. Lawyer. Another useful source of information is the *Serial Programming Guide for POSIX Operating Systems* by Michael R. Sweet, available online at *http://www.easysw.com/~mike/serial/*.

62.12 Exercises

62-1. Implement *isatty()*. (You may find it useful to read the description of *tcgetattr()* in Section 62.2.)

62-2. Implement *ttyname()*.

62-3. Implement the *getpass()* function described in Section 8.5. (The *getpass()* function can obtain a file descriptor for the controlling terminal by opening `/dev/tty`.)

62-4. Write a program that displays information indicating whether the terminal referred to by standard input is in canonical or noncanonical mode. If in noncanonical mode, then display the values of TIME and MIN.