Terminal Drivers

Terminal I/O is a real can of worms. In the Seventh Edition, it wasn't exactly simple. To quote the terminal driver man page,

> *The terminal handler has clearly entered the race for ever-greater complexity and generality. It's still not complex and general enough for TENEX fans.*

Since then, things have gone steadily downhill.

The most important terminal driver versions are:

- The "old" terminal driver, derived from the Seventh Edition terminal driver. This driver is still in use in XENIX and older BSD versions.
- The System III/System V terminal driver, also called *termio*.
- The POSIX.1 *termios* routines, derived from termio.

Most modern systems support more than one kind of serial line driver. This is known as the *line discipline*. Apart from terminal drivers, the most important line disciplines for asynchronous lines are *SLIP* (Serial Line Internet Protocol) and *PPP* (Point to Point Protocol). These are very implementation dependent, and we won't discuss them further. The line discipline is set with the `TIOCSETD ioctl`, described on page 259.

It's beyond the scope of this book to explain all the intricacies and kludges that have been added to terminal handlers over the decades. *Advanced Programming in the UNIX environment*, by Richard Stevens, gives you a good overview of current practice, and you shouldn't really want to know about older versions unless you have trouble with them. In the following discussion, we'll concentrate on the four areas that cause the most headaches when porting programs:

- The externally visible data structures used for passing information to and from the driver.
- A brief overview of the different operational modes (raw, cooked, cbreak, canonical and non-canonical).

- The `ioctl` request interface to the terminal driver, one of the favourite problem areas in porting terminal-related software.
- The POSIX.1 *termios* request interface.

The documentation of every driver describes at least two different modes of treating terminal input. The Seventh Edition and BSD drivers define three:

- In *raw* mode, the `read` system call passes input characters to the caller exactly as they are entered. No processing takes place in the driver. This mode is useful for programs which want to interpret characters themselves, such as full-screen editors.
- *cooked* mode interprets a number of special characters, including the new line character `\n`. A `read` call will terminate on a `\n`. This is the normal mode used by programs that don't want to be bothered by the intricacies of terminal programming.
- *cbreak* mode performs partial interpretation of the special characters, this time not including `\n`. *cbreak* mode is easier to use than raw mode, and is adequate for many purposes. It's a matter of taste whether you prefer this to raw mode or not.

By contrast, termio and termios specify two different processing modes for terminal input:

- *canonical*[*] mode performs significant processing on input before passing it to the calling function. Up to 21 input special characters may be used to tell the driver to do things as varied as start and stop output, to clear the input buffer, to send signals to the process and to terminate a line in a number of different ways.
- *Non-canonical* input mode, in which the driver does not interpret input characters specially (this corresponds roughly to BSD *cbreak* mode).

In fact, subdividing the terminal operation into modes is an oversimplification: a large number of flags modify the operational modes. Later in the chapter we'll look at how to set these modes with termios.

Typical terminal code

This is all rather abstract: let's look at a simple example: a program wants to read a single character from the terminal. To do this, it needs to set raw or non-canonical mode, read the character, and then reinstate the previous mode. For the old terminal driver, the code looks like Example 15-1:

Example 15−1:

```
struct sgttyb initial_status;               /* initial termios flags */
struct sgttyb raw_status;                   /* and the same with icanon reset */

ioctl (stdin, TIOCGETA, &initial_status);   /* get attributes */
raw_status = initial_status;                /* make a copy */
raw_status.sg_flags |= RAW;                 /* and set raw mode */
```

* The word *canon* refers to (religious) law: the intent is that this should be the correct or standard way to handle input characters. See the *New Hacker's Dictionary* for a long discussion of the term.

Example 15-1: (continued)

```
ioctl (stdin, TIOCSETN, &raw_status);        /* set the new terminal flags */
puts ("? ");
if ((reply = getchar ()) != '\n')      /* get a reply */
  puts ("\n");                                 /* and finish the line */
ioctl (stdin, TIOCSETN, &initial_status);    /* set the old terminal flags */
```

With the System V *termio* driver, it would look like Example 15-2:

Example 15-2:

```
struct termio initial_status;                /* initial termio flags */
struct termio noicanon_status;               /* and the same with icanon reset */

ioctl (stdin, TCGETA, &initial_status);      /* get attributes */
noicanon_status = initial_status;            /* make a copy */
noicanon_status.c_lflag &= ~ICANON;          /* and turn icanon off */
ioctl (stdin, TCSETA, &noicanon_status);     /* set non-canonical mode */
puts ("? ");
if ((reply = getchar ()) != '\n')      /* get a reply */
  puts ("\n");                                 /* and finish the line */
ioctl (stdin, TCSETA, &initial_status))      /* reset old terminal mode */
```

Don't rely on code like this to be *termio* code: *termios* code can look almost identical. Correct *termios* code uses the *termios* functions which we will look at on page 265, and looks like Example 15-3:

Example 15-3:

```
struct termios initial_status;               /* initial termios flags */
struct termios noicanon_status;              /* and the same with icanon reset */

tcgetattr (stdin, &initial_status)l          /* get current attributes */
noicanon_status = initial_status;            /* make a copy */
noicanon_status.c_lflag &= ~ICANON;          /* and turn icanon off */

tcsetattr (stdin, TCSANOW, &noicanon_status); /* set non-canonical mode */
puts ("? ");
if ((reply = getchar ()) != '\n')      /* get a reply */
  puts ("\n");                                 /* and finish the line */
tcsetattr (stdin, TCSANOW, &initial_status); /* reset old terminal mode */
```

Terminology

Before we start, it's a good idea to be clear about a few terms that are frequently confused:

- All terminal drivers buffer I/O in two *queues*, an input queue and an output queue. The input queue contains characters that the user has entered and the process has not yet read. The output queue contains characters that the process has written but that have not yet been output to the terminal. These queues are maintained inside the terminal driver. Don't confuse them with buffers maintained in the process data space by the *stdio* routines.

- The term *flush* can mean to discard the contents of a queue, or to wait until they have all been output to the terminal. Most of the time it means to discard the contents, and that's how we'll use it in this chapter.
- The term *drain* means to wait until the contents of the output queue have been written to the terminal. This is also one of the meanings of *flush*.
- *Special characters*, frequently called *control characters*, are input characters that cause the terminal driver to do something out of the ordinary. For example, CTRL-D usually causes the terminal driver to return an end-of-file indication. The term *special characters* is the better term, since you can set them to characters that are not ASCII control characters. For example, even today, the default erase character in System V is #: it's a special character, but not an ASCII control character.
- The *baud rate* of a modem is the number of units of information it can transmit per second. Modems are analogue devices that can represent multiple bits in a single unit of information—modern modems encode up to 6 bits per unit. For example, a modern V.32bis modem will transfer 14400 bits per second, but runs at only 2400 baud. Baud rates are of interest only to modem designers.
- As the name indicates, the *bit rate* of a serial line indicates how many bits it can transfer per second. Bit rates are often erroneously called *baud rates*, even in official documentation. The number of bytes transferred per second depends on the configuration: normally, an asynchronous serial line will transmit one start bit and one stop bit in addition to the data, so it transmits 10 bits per byte.
- *break* is an obsolescent method to signal an unusual condition over an asynchronous line. Normally, a continuous voltage or current is present on a line except when data is being transferred. Break effectively breaks (disconnects) the line for a period between .25 and .5 second. The serial hardware detects this and reports it separately. One of the problems with break is that it is intimately related to the serial line hardware.
- *DCE* and *DTE* mean *data communication equipment* and *data terminal equipment* respectively. In a modem connection, the modem is the DCE and both terminal and computer are DTEs. In a direct connect, the terminal is the DTE and the computer is the DCE. Different cabling is required for these two situations.
- *RS-232*, also known as *EIA-232*, is a standard for terminal wiring. In Europe, it is sometimes referred to as *CCITT V.24*, though V.24 does not in fact correspond exactly to RS-232. It defines a number of signals, listed in Table 15-1.

Table 15−1: RS-232 signals

Table 15−1: RS-232 signals (continued)

RS-232 name	pin	purpose
PG	1	Protective ground. Used for electrical grounding only.
TxD	2	Transmitted data.
RxD	3	Received data.
RTS	4	Request to send. Indicates that the device has data to output.
CTS	5	Clear to send. Indicates that the device can receive input. Can be used with RTS to implement flow control.
DSR	6	Data set ready. Indicates that the modem (*data set* in older parlance) is powered on.
SG	7	Signal ground. Return for the other signals.
DCD	8	Carrier detect. Indicates that the modem has connection with another modem.
DTR	20	Data terminal ready. Indicates that the terminal or computer is ready to talk to the modem.
RI	22	Ring indicator. Raised by a modem to indicate that an incoming call is ringing.

For more details about RS-232, see *RS-232 made easy*, second edition by Martin Seyer.

Terminal data structures

In this section, we'll take a detailed look at the data structures you're likely to encounter when porting software from a different platform. I have included typical literal values for the macros. *Don't ever use these values!* They're not guaranteed to be correct for every implementation, and they're included only to help you if you find that the program includes literals rather than macro names. When writing code, always use the names.

Old terminal driver definitions

In the Seventh Edition, most `ioctl` calls that took a parameter referred to a `struct sgttyb`, which was defined in */usr/include/sgtty.h*:

```
struct sgttyb
  {
  char sg_ispeed;                       /* input bit rate code */
  char sg_ospeed;                       /* output bit rate code */
  char sg_erase;                        /* erase character */
  char sg_kill;                         /* kill character */
  int  sg_flags;                        /* Terminal flags (see Table 15-3) */
  char sg_nldly;                        /* delay after \n character */
  char sg_crdly;                        /* delay after \r character */
  char sg_htdly;                        /* delay after tab character */
  char sg_vtdly;                        /* delay after vt character */
  char sg_width;                        /* terminal line width */
```

```
    char sg_length;                         /* terminal page length */
    };
```

The bit rates in sg_ispeed and sg_ospeed are encoded, and allow only a certain number of speeds:

Table 15−2: Seventh Edition bit rate codes

Parameter	value	meaning
B0	0	hang up phone
B50	1	50 bits/second
B75	2	75 bits/second
B110	3	110 bits/second
B134	4	134.5 bits/second
B150	5	150 bits/second
B200	6	200 bits/second
B300	7	300 bits/second
B600	8	600 bits/second
B1200	9	1200 bits/second
B1800	10	1800 bits/second
B2400	11	2400 bits/second
B4800	12	4800 bits/second
B9600	13	9600 bits/second
EXTA	14	External A
EXTB	15	External B

The field sg_flags contains a bit map specifying the following actions:

Table 15−3: Seventh Edition tty flags

Parameter	value (octal)	value (hex)	meaning
XTABS	02000	0x400	Replace output tabs by spaces.
INDCTL	01000	0x200	Echo control characters as ^a, ^b etc.
SCOPE	0400	0x100	Enable neat erasing functions on display terminals ("scopes").
EVENP	0200	0x80	Even parity allowed on input (most terminals).
ODDP	0100	0x40	Odd parity allowed on input.
RAW	040	0x20	Raw mode: wake up on all characters, 8-bit interface.
CRMOD	020	0x10	Map CR into LF; echo LF or CR as CR-LF.
ECHO	010	0x8	Echo (full duplex).
LCASE	04	0x4	Map upper case to lower on input.
CBREAK	02	0x2	Return each character as soon as typed.

Table 15−3: Seventh Edition tty flags (continued)

Parameter	value (octal)	value (hex)	meaning
TANDEM	01	0x1	Automatic flow control.

A second structure defines additional special characters that the driver interprets in cooked mode. They are stored in a `struct tchars`, which is also defined in */usr/include/sgtty.h*:

```
struct tchars
  {
  char t_intrc;                         /* interrupt (default DEL) */
  char t_quitc;                         /* quit (default ^\) */
  char t_startc;                        /* start output (default ^Q)*/
  char t_stopc;                         /* stop output (default ^S) */
  char t_eofc;                          /* end-of-file (default ^D) */
  char t_brkc;                          /* input delimiter (like nl, default -1) */
  };
```

Each of these characters can be disabled by setting it to -1 (octal 0377), as is done with the default `t_brkc`. This means that no key can invoke its effect.

termio and termios structures

The System V terminal driver defines a `struct termio` to represent the data that the Seventh Edition driver stored in `sgttyb` and `tchars`. In POSIX.1 termios, it is called `struct termios`. Both are very similar: compared to the Seventh Edition, they appear to have been shorter by moving the special characters, which in `sgttyb` were stored as individual elements, into the array `c_cc`:

```
struct termio
  {
  unsigned short c_iflag;               /* input modes */
  unsigned short c_oflag;               /* output modes */
  unsigned short c_cflag;               /* control modes */
  unsigned short c_lflag;               /* local modes */
  char c_line;                          /* line discipline */
  unsigned char c_cc [NCC];             /* special chars */
  long c_ispeed;                        /* input speed, some termios */
  long c_ospeed;                        /* output speed, some termios */
  };
```

The variable `c_line` specifies the line discipline. It is defined in termio, and not in the POSIX.1 termios standard, but some System V versions of termios have it anyway. `NCC` is the number of special characters. We'll look at them after the flags.

Not all versions of System V define the members `c_ispeed` and `c_ospeed`. Instead, they encode the line speed in `c_cflag`. The correct way to access them is via the termios utility functions `cfgetispeed`, `cfsetispeed`, `cfgetospeed`, `cfsetospeed` and `cfsetspeed`, which we will discuss on page 265. To make matters worse, some older System V termios implementations supplied `c_ispeed` and `c_ospeed`, but the implementation didn't use them. In addition, many systems cannot handle different input and output speeds, so setting one

speed automatically sets the other as well.

`c_iflag`, `c_oflag`, `c_cflag` and `c_lflag` (a total of 128 possible bits) take the place of the Seventh Edition `sg_flags`.

c_iflag

`c_iflag` specifies how the driver treats terminal input:

Table 15−4: termios c_iflag bits

Parameter	value (SysV)	value (BSD)	meaning
`IGNBRK`	0x1	0x1	Ignore break condition.
`BRKINT`	0x2	0x2	Generate a `SIGINT` signal on break.
`IGNPAR`	0x4	0x4	Ignore characters with parity errors.
`PARMRK`	0x8	0x8	If a parity or framing error occurs on input, accept it and insert into the input stream the three-character sequence `0xff`, `0`, and the character received.
`INPCK`	0x10	0x10	Enable input parity check.
`ISTRIP`	0x20	0x20	Strip bit 7 from character.
`INLCR`	0x40	0x40	Map NL to CR on input.
`IGNCR`	0x80	0x80	Ignore CR.
`ICRNL`	0x100	0x100	Map CR to NL on input.
`IUCLC`[1]	0x200		Map uppercase to lowercase on input.
`IXON`	0x400	0x200	Enable output flow control with XON/XOFF (`CTRL-S`/`CTRL-Q`).
`IXANY`	0x800	0x800	Allow any character to restart output after being stopped by `CTRL-S`.
`IXOFF`	0x1000	0x400	Enable input flow control with XON/XOFF.
`CTSFLOW`[1]	0x2000		Enable CTS protocol for a modem line.
`RTSFLOW`[1]	0x4000		Enable RTS signaling for a modem line.
`IMAXBEL`[2]	0x2000	0x2000	Ring the terminal bell when the input queue is full.

[1] not in POSIX.1 or BSD.
[2] not in POSIX.1 and some versions of System V.

A couple of these flags are not portable:

- `IUCLC` maps lower case to upper case: if you enter a lower case character, it is converted to an upper case character and echos that way. Many people consider this a bug, not a feature. There's no good way to implement this on a non-System V system. If you *really* want to have this behaviour, you'll have to turn off echo and provide an echo from the program.
- `CTSFLOW` and `RTSFLOW` specify flow control via the RS-232 signals `CTS` and `RTS`. These are control flags, of course, not input flags, but some versions of System V put them here

for backward compatibility with XENIX. Some other versions of System V don't define them at all, and BSD systems and yet other System V systems supply them in `c_cflags`, where they belong.

`c_oflag` specifies the behaviour on output.

Table 15−5: termios c_oflag bits

Parameter	value (SysV)	value (BSD)	meaning
OPOST	0x1	0x1	Postprocess output.
OLCUC[1]	0x2		Map lower case to upper on output.
ONLCR	0x4	0x2	Map NL to CR-NL on output.
OCRNL	0x8	0x8	Map CR to NL on output.
ONOCR	0x10	0x10	Suppress CR output at column 0.
ONLRET	0x20	0x20	NL performs CR function.
OFILL	0x40	0x40	Use fill characters for delay.
OFDEL	0x80	0x80	Fill is DEL if set, otherwise NUL.*
NLDLY[1]	0x100		Mask bit for new-line delays:
NL0	0x0		No delay after NL.
NL1	0x100		One character delay after NL.
CRDLY[1]	0x600		Mask bits for carriage-return delays:
CR0	0x0		No delay after CR.
CR1	0x200		One character delay after CR.
CR2	0x400		Two characters delay after CR.
CR3	0x600		Three characters delay after CR.
TABDLY[1]	0x18000		Mask bits for horizontal-tab delays:
TAB0	0x0		No delay after HT.
TAB1	0x800		One character delay after HT.
TAB2	0x1000		Two characters delay after HT.
TAB3	0x1800		Expand tabs to spaces.
BSDLY[1]	0x2000		Mask bit for backspace delays:
BS0	0x0		No delay after BS.
BS1	0x2000		One character delay after BS.
VTDLY[1]	0x4000		Mask bit for vertical-tab delays:
VT0	0x0		No delay after VT.
VT1	0x4000		One character delay after VT.
FFDLY[1]	0x8000		Mask bit for form-feed delays:
FF0	0x0		No delay after FF.
FF1	0x8000		One character delay after FF.

* The ASCII character represented by binary 0 (the C character constant `\0`). Not to be confused with the null pointer, which in C is usually called `NULL`.

Table 15−5: termios c_oflag bits (continued)

[1] not in POSIX.1 or BSD.

A number of these flags are not portable:

- System V supplies a large number of flags designed to compensate for mechanical delays in old hardcopy terminal equipment. It's doubtful that any of this is needed nowadays. If you do have an unbuffered hardcopy terminal connected to your BSD machine, and it loses characters at the beginning of a line or a page, you should check whether CTS/RTS flow control might not help. Or you could buy a more modern terminal.
- `OLCUC` is obsolete, of course, but if that old hardcopy terminal also doesn't support lower-case, and it doesn't upshift lower-case characters automatically, you'll have to do it programatically.

`c_cflag` specifies hardware control aspects of the terminal interface:

Table 15−6: termios c_cflag bits

Parameter	value (SysV)	value (BSD)	meaning
`CBAUD`[1]	0xf		Bit rate
`B0`	0		Hang up
`B50`	0x1		50 bps
`B75`	0x2		75 bps
`B110`	0x3		110 bps
`B134`	0x4		134 bps
`B150`	0x5		150 bps
`B200`	0x6		200 bps
`B300`	0x7		300 bps
`B600`	0x8		600 bps
`B1200`	0x9		1200 bps
`B1800`	0xa		1800 bps
`B2400`	0xb		2400 bps
`B4800`	0xc		4800 bps
`B9600`	0xd		9600 bps
`B19200`	0xe		19200 bps
`EXTA`	0xe		External A
`B38400`	0xf		38400 bps
`EXTB`	0xf		External B
`CSIZE`	0x30	0x300	Mask bits for character size:
`CS5`	0x0	0x0	5 bits
`CS6`	0x10	0x100	6 bits
`CS7`	0x20	0x200	7 bits
`CS8`	0x30	0x300	8 bits

Table 15-6: termios c_cflag bits (continued)

Parameter	value (SysV)	value (BSD)	meaning
CSTOPB	0x40	0x400	Send two stop bits (if not set, send 1 stop bit).
CREAD	0x80	0x800	Enable receiver.
PARENB	0x100	0x1000	Enable parity.
PARODD	0x200	0x2000	Set odd parity if set, otherwise even.
HUPCL	0x400	0x4000	Hang up on last close.
CLOCAL	0x800	0x8000	Disable modem control lines.
RCV1EN[3]	0x1000		*see below*
XMT1EN[3]	0x2000		*see below*
LOBLK[3]	0x4000		Block layer output.
CTSFLOW[1]	0x10000		CTS flow control of output.
CCTS_OFLOW[2]		0x10000	CTS flow control of output.
CRTSCTS[2]		0x10000	CTS flow control of output (alternative symbol).
RTSFLOW[1]		0x20000	RTS flow control of input.
CRTS_IFLOW[2]		0x20000	RTS flow control of input.
MDMBUF[2]	0x100000		Flow control output via Carrier.

[1] speeds are encoded differently in BSD—see below.
[2] not in POSIX.1 or System V.
[3] not in POSIX.1 or BSD.

Again, some of these flags are only available on specific platforms:

- RCV1EN and XMT1EN are defined in some System V header files, but they are not documented.
- BSD systems supply CRTS_IFLOW and CCTS_OFLOW for RS-232 flow control. Some System V systems supply RTSFLOW and CTSFLOW to mean the same thing, but other System V systems don't support it, and other systems again put these flags in c_iflag.

c_lflag specifies the behaviour specific to the line discipline. This flag varies so much between System V and BSD that it's easier to put them in separate tables. Table 15-7 describes the standard System V line discipline, and Table 15-8 describes the standard BSD line discipline,

Table 15-7: System V termios c_lflag bits

Parameter	value	meaning
ISIG	0x1	Allow the characters INTR, QUIT, SUSP and DSUSP to generate signals.
ICANON	0x2	Enable canonical input (erase and kill processing).

Table 15−7: System V termios c_lflag bits (continued)

Parameter	value	meaning
XCASE	0x4	In conjunction with ICANON, map upper/lower case to an upper-case only terminal. Lower case letters are displayed in upper case, and upper case letters are displayed with a preceding backslash (\).
ECHO	0x8	Enable echo.
ECHOE	0x10	Erase character removes character from screen.
ECHOK	0x20	Echo NL after line kill character.
ECHONL	0x40	Echo NL even if echo is off.
NOFLSH	0x80	Disable flush after interrupt or quit.

Here's the BSD version:

Table 15−8: BSD termios c_lflag bits

Parameter	value	meaning
ECHOKE[1]	0x1	Line kill erases line from screen.
ECHOE	0x2	Erase character removes character from screen.
ECHOK	0x4	Echo NL after line kill character.
ECHO	0x8	Enable echo.
ECHONL	0x10	Echo NL even if echo is off.
ECHOPRT[1]	0x20	Visual erase mode for hardcopy.
ECHOCTL[1]	0x40	Echo control chars as ^(Char).
ISIG	0x80	Enable signals INTR, QUIT, SUSP and DSUSP.
ICANON	0x100	Enable canonical input (erase and kill processing).
ALTWERASE[1]	0x200	Use alternate WERASE algorithm. Instead of erasing back to the first blank space, erase back to the first non-alphanumeric character.
IEXTEN	0x400	Enable DISCARD and LNEXT.
EXTPROC[1]	0x800	This flag carries the comment "External processing". Apart from that, it appears to be undocumented.
TOSTOP	0x400000	If a background process attempts output, send a SIGTTOU to it. By default this stops the process.
FLUSHO[1]	0x800000	Status return only: output being flushed.
NOKERNINFO[1]	0x2000000	Prevent the STATUS character from displaying information on the foreground process group.
PENDIN[1]	0x20000000	Pending input is currently being redisplayed.
NOFLSH	0x80000000	Don't flush input and output queues after receiving SIGINT or SIGQUIT.

[1] not in POSIX.1.

Converting the c_lflag bits is even more of a problem:

- XCASE is part of the System V upper case syndrome that we saw with c_iflag and c_oflag.
- BSD offers a number of echo flags that are not available in System V. In practice, this is a cosmetic difference in the way input works. Consider a BSD program with a line like:

  ```
  term.c_lflag = ECHOKE | ECHOE | ECHOK | ECHOCTL;
  ```

  This will fail to compile under System V because ECHOKE and ECHOCTL are undefined. You can probably ignore these flags, so the way to fix it would be something like:

  ```
  term.c_lflag = ECHOE | ECHOK
  #ifdef ECHOKE
                | ECHOKE
  #endif
  #ifdef ECHOCTL
                | ECHOCTL
  #endif
               ;
  ```

  Note the lonesome semicolon on the last line.
- The flags FLUSHO and PENDIN are status flags that cannot be set. There's no way to get this information in System V.
- NOKERNINFO refers to the STATUS character, which we will see below. This is not supported in System V.

special characters

The number of special characters has increased from 6 in the Seventh Edition (struct tchars) to 8 in termio and a total of 20 in termios (though 4 of the termios special characters are "reserved"—in other words, not defined). Despite this number, there is no provision for redefining CR and NL.

Table 15-9: termio and termios special characters

Name	Index in c_cc (SysV)	Default (SysV)	Index in c_cc (BSD)	Default (BSD)	Function
CR	(none)	\r	(none)	\r	Go to beginning of line. In canonical and cooked modes, complete a read request.
NL	(none)	\n	(none)	\n	End line. In canonical and cooked modes, complete a read request.
VINTR	0	DEL	8	CTRL-C	Generate an SIGINT signal.

Table 15−9: termio and termios special characters (continued)

Name	Index in c_cc (SysV)	Default (SysV)	Index in c_cc (BSD)	Default (BSD)	Function
VQUIT	1	CTRL-|	9	CTRL-|	Generate a SIGQUIT signal.
VERASE	2	#[4]	3	DEL	Erase last character.
VKILL	3	@[4]	5	CTRL-U	Erase current input line.
VEOF	4	CTRL-D	0	CTRL-D	Return end-of-file indication.
VEOL	5	NUL	1	\377	Alternate end-of-line character.
VEOL2[1]	6	NUL	2	\377	Alternate end-of-line character.
VSWTCH[1, 2]	7	NUL			*shl* layers: switch shell.
VSTART	8	CTRL-Q	12	CTRL-Q	Resume output after stop.
VSTOP	9	CTRL-S	13	CTRL-S	Stop output.
VSUSP	10	CTRL-Z	10	CTRL-Z	Generate a SIGTSTP signal when typed.
VDSUSP[1]	11	CTRL-Y	11	CTRL-Y	Generate a SIGTSTP signal when the character is read.
VREPRINT[1]	12	CTRL-R	6	CTRL-R	Redisplay all characters in the input queue (in other words, characters that have been input but not yet read by any process). The term "print" recalls the days of harcopy terminals.
VDISCARD[1]	13	CTRL-O	15	CTRL-O	Discard all terminal output until another DISCARD character arrives, more input is typed or the program clears the condition.
VWERASE[1]	14	CTRL-W	4	CTRL-W	Erase the preceding word.
VLNEXT[1]	15	CTRL-V	14	CTRL-V	Interpret next character literally.
VSTATUS[1,3]			18	\377	Send a SIGINFO signal to the foreground process group. If NOKERNINFO is not set, the kernel also prints a status message on the terminal.

[1] not in POSIX.1.

[2] *shl* layers are a System V method of multiplexing several shells on one terminal. They are not supported on BSD systems.

[3] not supported on System V.

[4] These archaic, teletype-related values are still the default for System V. The file */usr/include/sys/termio.h* contains alternative definitions (VERASE set to CTRL-H and VKILL set to CTRL-X), but these need to be specifically enabled by defining the preprocessor variable _NEW_TTY_CTRL.

You will frequently see these names without the leading V. For example, the *stty* program refers to VQUIT as QUIT.

Terminal driver modes

Depending on the driver, it looks as if you have a choice of two or three operational modes on input:

- With the termio and termios drivers, you have the choice of canonical and non-canonical mode.
- With the old terminal driver, you have the choice of *raw*, *cooked* and *cbreak* modes.

This distinction is not as clear-cut as it appears: in fact, you can set up both drivers to do most things you want.

Canonical mode

To quote Richard Stevens' *Advanced Programming in the UNIX environment*: "Canonical mode is simple"—it takes only about 30 pages for a brief description. For an even simpler description: everything in the rest of this chapter applies to canonical mode unless otherwise stated.

Non-canonical mode

Non-canonical mode ignores all special characters except *INTR*, *QUIT*, *SUSP*, *STRT*, *STOP*, *DISCARD* and *LNEXT*. If you don't want these to be interpreted, you can disable them by setting the corresponding entry in tchars to _POSIX_VDISABLE.

The terminal mode has a strong influence on how a read from a terminal completes. In canonical mode, a read request will complete when the number of characters requested has been input, or when the user enters one of the characters CR, NL, *VEOL* or (where supported) *VEOL2*. In non-canonical mode, no special character causes a normal read completion. The way a read request completes depends on two variables, *MIN* and *TIME*. *MIN* represents a minimum number of characters to be read, and *TIME* represents a time in units of 0.1 second. There are four possible cases:

1. Both *MIN* and *TIME* are non-zero. In this case, a read will complete when either *MIN* characters have been entered or *TIME*/10 seconds have passed since a character was entered. The timer starts when a character is entered, so at least one character must be entered for the read to complete.
2. *MIN* is non-zero, *TIME* is zero. In this case, the read will not complete until *MIN* characters have been entered.
3. *MIN* is zero and *TIME* is non-zero. The read will complete after entering one character or after *TIME/10* seconds. In the latter case, 0 characters are returned. This is not the same as setting *MIN* to 1 and leaving *TIME* as it is: in this case, the read would not

complete until at least one character is entered.

4. Both *MIN* and *TIME* are set to 0. In this case, `read` returns immediately with any characters that may be waiting.

If *MIN* is non-zero, it overrides the read count specified to `read`, even if `read` requests less than *MIN* characters: the remaining characters are kept in the input queue for the next `read` request. This can have the unpleasant and confusing effect that at first nothing happens when you type something in, and then suddenly multiple reads complete.

Non-canonical mode does not interpret all the special characters, but it needs space to store *MIN* and *TIME*. In 4.4BSD, two of the reserved characters are used for this purpose. Most other implementations, including XENIX, System V and some older BSDs do it differently, and this can cause problems:

- The value of *VEOF* is used for *VMIN*. This value is normally `CTRL-D`, which is decimal 4: if you switch from canonical to non-canonical mode and do not change *MIN*, you may find that a read of a single character will not complete until you enter a total of four characters.
- The value of *VEOL* is used for *TIME*. This is normally 0.

Raw mode

Raw mode does almost no interpretation of the input stream. In particular, no special characters are recognized, and there is no timeout. The non-canonical mode variables *MIN* and *TIME* do not exist. The result is the same as setting *MIN* to 1 and *TIME* to 0 in non-canonical mode.

Cooked mode

The cooked mode of the old terminal driver is essentially the same as canonical mode, within the limitations of the driver data structures—termios offers some features that are not available with the old terminal driver, such as alternate end-of-line characters.

Cbreak mode

To quote the Seventh Edition manual:

> *CBREAK is a sort of half-cooked (rare?) mode.*

In terms of termios, it is quite close to non-canonical mode: the only difference is that cbreak mode turns off echo. Non-canonical mode does not specify whether echo is on or off.

Emulating old terminal driver modes

Table 15-10 illustrates how you can define old driver terminal modes with termios. You'll see that a large number of entries are not defined: raw and cbreak modes do not specify how these

parameters are set. You can set them to whatever you feel appropriate.

Table 15−10: Defining terminal modes with termios

Flag	raw mode	cbreak mode
BRKINT	off	on
INPCK	off	on
ISTRIP	off	not defined
ICRNL	off	not defined
IXON	off	not defined
OPOST	off	not defined
CSIZE	CS8	not defined
PARENB	off	not defined
ECHO	off	off
ISIG	off	not defined
ICANON	off	off
IEXTEN	off	not defined
VMIN	1	1
VTIME	0	0

gtty and stty

You may still occasionally run into the system calls `stty` and `gtty`, which are leftovers from the Seventh Edition. You can replace `stty` with the `ioctl` function `TIOCSETP`, and you can replace `gtty` with the `ioctl` request `TIOCGETP`. Read more on both these requests on page 257.

The Linux terminal driver

Linux has the great advantage of being a recent development, so it doesn't have a number of the warts of older terminal drivers. It goes to some trouble to be compatible, however:

- In addition to POSIX.1 *termios*, the kernel also directly supports System V *termio*.
- The library *libbsd.a* includes `ioctl` calls for the old terminal driver, which Linux users call the BSD driver.
- The only line discipline you can expect to work under Linux is the standard tty line discipline `N_TTY`.

ioctl

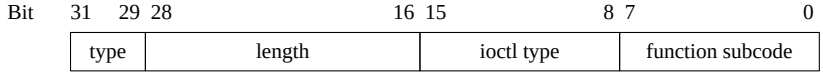
`ioctl` is the file system catchall: if there isn't any other function to do the job, then somebody will bend `ioctl` to do it. Nowhere is this more evident than in terminal I/O handling. As a result of this catchall nature, it's not easy to represent `ioctl` parameters in C.

We'll look at the semantics first. The `ioctl` function call takes three parameters:

1. A file number.
2. A *request*, which we'll look at in more detail in the next section.
3. When present, the meaining is defined by the request. It could be an integer, another request code or a pointer to some structure defined by the request.

ioctl request codes

The key to understanding `ioctl` is the request code. Request codes are usually subdivided into a number of fields. For example, 4.4BSD defines four fields:



- The first three bits specify the type of parameter. `IOC_VOID` (`0x20` in the first byte) specifies that the request takes no parameters, `IOC_OUT` (`0x40` in the first byte) specifies that the parameters are to be copied out of the kernel (in other words, that the parameters are to be returned to the user), and `IOC_IN` (`0x80` in the first byte) specifies that the parameters are to be copied in to the kernel (they are to be passed to `ioctl`).
- The next 13 bits specify the length of the parameter in bytes.
- The next byte specifies the type of request. This is frequently a mnemonic letter. In 4.4BSD, this field is set to the lower-case letter `t` for terminal ioctls.
- Finally, the last byte is a number used to identify the request uniquely.

This encoding depends heavily on the operating system. Other systems (especially, of course, 16 bit systems) encode things differently, but the general principle remains the same.

Both the request code and the third parameter, where present, do not map easily to C language data structures. As a result, the definition of the function varies significantly. For example, XENIX and BSD declare it as:

```
#include <sys/ioctl.h>
int ioctl (int fd, unsigned long request, char *argp)
```

and System V.4 has

```
#include <unistd.h>
int ioctl (int fs, int request,             /* arg */ ...);
```

Strictly speaking, since the request code is not a number, both `int` and `unsigned long` are incorrect, but they both do the job.

When debugging a program, it's not always easy to determine which request has been passed to `ioctl`. If you have the source code, you will see something like

```
ioctl (stdin, TIOCGETA, &termstat);
```

Unfortunately, a number of `ioctl` calls are embedded in libraries to which you probably don't have source, but you can figure out what's going on by setting a breakpoint on `ioctl`. In this example, when you hit the breakpoint, you will see something like:

```
(gdb) bt
#0  ioctl (file=0, request=1076655123, parameter=0xefbfd58c "") at ioctl.c:6
#1  0x10af in main () at foo.c:12
```

The value of `request` looks completely random. In hexadecimal it starts to make a little more sense:

```
(gdb) p/x request
$1 = 0x402c7413
```

If we compare this with the request code layout in the example above, we can recognize a fair amount of information:

- The first byte starts with `0x40`, `IOC_OUT`: the parameter exists and defines a return value.
- The next 13 bits are `0x2c`, the length to be returned (this is the length of `struct termios`).
- The next byte is `0x74`, the ASCII character `t`, indicating that this is a terminal `ioctl` request.
- The last byte is `0x13` (decimal 19).

It's easy enough to understand this when it's deciphered like this, but doing it yourself is a lot different. The first problem is that there is no agreed place where the `ioctl` requests are defined. The best place to start is in the header file *sys/ioctl.h*, which in the case of 4.4BSD will lead you to the file *sys/ioccom.h* (*sys/sys/ioccom.h* in the 4.4BSD distribution). Here you will find code like:

```
#define IOCPARM_MASK    0x1fff              /* parameter length, at most 13 bits */
#define IOCPARM_LEN(x)  (((x) >> 16) & IOCPARM_MASK)
#define IOCBASECMD(x)   ((x) & ~(IOCPARM_MASK << 16))
#define IOCGROUP(x)     (((x) >> 8) & 0xff)
#define IOC_VOID        0x20000000          /* no parameters */
#define IOC_OUT         0x40000000          /* copy out parameters */
#define IOC_IN          0x80000000          /* copy in parameters */
```

These define the basic parts of the request. Next come the individual types of request:

```
#define _IOC(inout,group,num,len) \  pass a structure of length len as parameter
  (inout | ((len & IOCPARM_MASK) << 16) | ((group) << 8) | (num))
#define _IO(g,n)     _IOC(IOC_VOID,     (g), (n), 0)    No parameter
#define _IOR(g,n,t)  _IOC(IOC_OUT,      (g), (n), sizeof(t))  Return parameter from kernel
#define _IOW(g,n,t)  _IOC(IOC_IN,       (g), (n), sizeof(t))  Pass parameter to kernel
/* this should be _IORW, but stdio got there first */
#define _IOWR(g,n,t) _IOC(IOC_INOUT,    (g), (n), sizeof(t))  Pass and return parameter
```

With these building blocks, we can now understand the real definitions:

```
#define TIOCSBRK  _IO('t', 123)             /* set break bit */
#define TIOCCBRK  _IO('t', 122)             /* clear break bit */
#define TIOCSDTR  _IO('t', 121)             /* set data terminal ready */
#define TIOCCDTR  _IO('t', 120)             /* clear data terminal ready */
#define TIOCGPGRP _IOR('t', 119, int)       /* get pgrp of tty */
#define TIOCSPGRP _IOW('t', 118, int)       /* set pgrp of tty */
```

These define four requests without parameters (`_IO`), a request that returns an `int` parameter from the kernel (`_IOR`), and a request that passes an `int` parameter to the kernel (`_IOW`).

Terminal ioctls

For a number of reasons, it's difficult to categorize terminal driver `ioctl` calls:

- As the terminal driver has changed over the course of time, some implementors have chosen to keep the old `ioctl` codes and give them new parameters. For example, the Seventh Edition call `TIOCGETA` returned the terminal parameters to a `struct sgttyb`. The same call in System V returns the values to a `struct termio`, and in 4.4BSD it returns the values to a `struct termios`.
- The documentation for many `ioctl` calls is extremely hazy: although System V supports the old terminal driver discipline, the documentation is very scant. Just because an `ioctl` function is not documented in the man pages doesn't mean that it isn't supported: it's better to check in the header files (usually something like *sys/termio.h* or *sys/termios.h*).
- Many `ioctl` calls seem to duplicate functionality. There are minor differences, but even they are treacherous. For example, in the Seventh Edition the `TIOCSETA` function drains the output queue and discards the content of the input queue before setting the terminal state. The same function in 4.4BSD performs the function immediately. To get the Seventh Edition behaviour, you need to use `TIOCSETAF`. The behaviour in System V is not documented, which means that you may be at the mercy of the implementor of the device driver: on one system, it may behave like the Seventh Edition, on another like 4.4BSD.

In the following sections, we'll attempt to categorize the most frequent `ioctl` functions in the

kind of framework that POSIX.1 uses for termios. Here's an index to the mess:

Table 15−11: ioctl parameters

Name	Function	Parameter 3	Page
TCFLSH	Flush I/O	int *	263
TCGETA	Get terminal state	struct termio *	258
TCGETS	Get terminal state	struct termios *	258
TCSBRK	Drain output, send break	int *	261
TCSETA	Set terminal state	struct termio *	259
TCSETAF	Drain I/O and set state	struct termio *	259
TCSETAW	Drain output and set state	struct termio *	259
TCSETS	Set terminal state	struct termios *	258
TCSETSF	Drain I/O and set state	struct termios *	258
TCSETSW	Drain output and set state	struct termios *	258
TCXONC	Set fbw control	int *	262
TIOCCBRK	Clear break	*(none)*	260
TIOCCDTR	Clear *DTR*	*(none)*	260
TIOCCONS	Set console	int *	264
TIOCDRAIN	Drain output queue	*(none)*	262
TIOCFLUSH	Flush I/O	int *	263
TIOCGETA	Get current state	struct termio *	256
TIOCGETC	Get special chars	struct tchars *	258
TIOCGETD	Set line discipline	int *ldisc	259
TIOCGETP	Get current state	struct sgttyb *	257
TIOCGPGRP	Get process group ID	pid_t *	263
TIOCGSID	Get session ID	pid_t *	264
TIOCGSOFTCAR	Get DCD indication	int *	265
TIOCGWINSZ	Get window size	struct winsize *	259
TIOCHPCL	Hang up on clear	*(none)*	258
TIOCMBIC	Clear modem state bits	int *	261
TIOCMBIS	Set modem state bits	int *	261
TIOCMGET	Get modem state	int *	261
TIOCMSET	Set modem state	int *	261
TIOCNXCL	Clear exclusive use	*(none)*	264
TIOCNOTTY	Drop controlling terminal	*(none)*	264
TIOCOUTQ	Get output queue length	int *	262
TIOCSBRK	Send break	*(none)*	260
TIOCSCTTY	Set controlling tty	*(none)*	263
TIOCSDTR	Set *DTR*	*(none)*	260
TIOCSETA	Set terminal state	struct sgttyb *	257
TIOCSETAF	Drain I/O and set state	struct termios *	257
TIOCSETAW	Drain output and set state	struct termios *	257

Table 15−11: ioctl parameters (continued)

Name	Function	Parameter 3	Page
TIOCSETC	Set special chars	struct tchars *	258
TIOCSETD	Set line discipline	int *ldisc	259
TIOCSETN	Set state immediately	struct sgttyb *	257
TIOCSETP	Get current state	struct sgttyb *	257
TIOCSPGRP	Set process group ID	pid_t *	263
TIOCSSOFTCAR	Set DCD indication	int *	265
TIOCSTART	Start output	*(none)*	262
TIOCSTI	Simulate input	char *	262
TIOCSTOP	Stop output	*(none)*	262
TIOCSWINSZ	Set window size	struct winsize *	259

Terminal attributes

One of the most fundamental groups of `ioctl` requests get and set the terminal state. This area is the biggest mess of all. Each terminal driver has its own group of requests, the request names are similar enough to be confusing, different systems use the same request names to mean different things, and even in *termios*, there is no agreement between BSD and System V about the names of the requests.

Table 15-12 gives an overview.

Table 15−12: Comparison of sgttyb, termio and termios ioctls

Function	`sgtty` request	`termio` request	`termios` request (BSD)	`termios` request (System V)
Get current state	TIOCGETA	TCGETA	TIOCGETA	TCGETS
Get special chars	TIOCGETC	TCGETA	TIOCGETA	TCGETS
Set terminal state immediately	TIOCSETN	TCSETA	TIOCSETA	TCSETS
Drain output and set state		TCSETAW	TIOCSETAW	TCSETSW
Drain I/O and set state	TIOCSETA	TCSETAF	TIOCSETAF	TCSETSF
Set special chars	TIOCSETC	TCSETAF	TIOCSETAF	TCSETSF

TIOCGETA

The call `ioctl (fd, TIOCGETA, term)` places the current terminal parameters in the structure `term`. The usage differs depending on the system:

- In the Seventh Edition, `term` was of type `struct sgttyb *`.
- In System V, `term` is of type `struct termio *`.

- In 4.4BSD, `term` is of type `struct termios *`.
- The Seventh Edition request `TIOCSETN` only sets the terminal state described in the first 6 bytes of `struct sgettyb`.

TIOCSETA

The call `ioctl (fd, TIOCSETA, term)` sets the current terminal state from `term`. The usage differs depending on the system:

- In the Seventh Edition, `term` was of type `struct sgttyb *`. The system drained the output queue and flushed the input queue before setting the parameters.
- In System V.3, `term` is of type `struct termio *`. The drain and flush behaviour is not documented.
- In 4.4BSD, `term` is of type `struct termios *`. The action is performed immediately with no drain or flush. This is used to implement the `tcsetattr` function with the `TCSANOW` option.

TIOCGETP and TIOCSETP

`TIOCGETP` and `TIOCSETP` are obsolete versions of `TIOCGETA` and `TIOCSETA` respectively. They affect only the first 6 bytes of the `sgttyb` structure (`sg_ispeed` to `sg_flags`). These requests correspond in function to the obsolete Seventh Edition system calls `stty` and `gtty`.

TIOCSETAW

The call `ioctl (fd, TIOCSETAW, void *term)` waits for any output to complete, then sets the terminal state associated with the device. 4.4BSD uses this call to implement the `tcsetattr` function with the `TCSADRAIN` option. In XENIX, the parameter `term` is of type `struct termio`; in other systems is it of type `struct termios`.

TIOCSETAF

The call `ioctl (fd, TIOCSETAF, void *term)` waits for any output to complete, flushes any pending input and then sets the terminal state. 4.4BSD uses this call to implement the `tcsetattr` function with the `TCSAFLUSH` option. In XENIX, the parameter `term` is of type `struct termio`, in other systems is it of type `struct termios`.

TIOCSETN

The call `ioctl (fd, TIOCSETN, struct sgttyb *term)` sets the parameters but does not delay or flush input. This call is supported by System V.3. and the Seventh Edition. In the Seventh Edition, this function works only on the first 6 bytes of the `sgttyb` structure.

TIOCHPCL

The call `ioctl (fd, TIOCHPCL, NULL)` specifies that the terminal line is to be disconnected (hung up) when the file is closed for the last time.

TIOCGETC

The call `ioctl (fd, TIOCGETC, struct tchars *chars)` returns the terminal special characters to `chars`.

TIOCSETC

The call `ioctl (fd, TIOCSETC, struct tchars *chars)` sets the terminal special characters from `chars`.

TCGETS

The call `ioctl (fd, TCGETS, struct termios *term)` returns the current terminal parameters to `term`. This function is supported by System V.4.

TCSETS

The call `ioctl (fd, TCSETS, struct termios *term)` immediately sets the current terminal parameters from `term`. This function is supported by System V.4 and corresponds to the 4.4BSD call `TIOCSETA`.

TCSETSW

The call `ioctl (fd, TCSETSW, struct termios *term)` sets the current terminal parameters from `term` after all output characters have been output. This function is supported by System V.4 and corresponds to the 4.4BSD call `TIOCSETAW`.

TCSETSF

The call `ioctl (fd, TCSETSF, struct termios *term)` flushes the input queue and sets the current terminal parameters from `term` after all output characters have been output. This function is supported by System V.4 and corresponds to the 4.4BSD call `TIOCSETAF`.

TCGETA

The call `ioctl (fd, TCGETA, struct termio *term)` stores current terminal parameters in `term`. Not all `termios` parameters can be stored in a `struct termio`; you may find it advantageous to use `TCGETS` instead (see above).

TCSETA

The call `ioctl (fd, TCSETA, struct termio *term)` sets the current terminal status from `term`. Parameters that cannot be stored in `struct termio` are not affected. This corresponds to `TCSETA`, except that it uses a `struct termio *` instead of a `struct termios *`.

TCSETAW

The call `ioctl (fd, TCSETAW, struct termio *term)` sets the current terminal parameters from `term` after draining the output queue. This corresponds to `TCSETW`, except that it uses a `struct termio *` instead of a `struct termios *`.

TCSETAF

The call `ioctl (fd, TCSETAF, struct termio *term)` input queue" flushes the input queue and sets the current terminal parameters from `term` after all output characters have been output. This corresponds to `TCSETF`, except that it uses a `struct termio *` instead of a `struct termios *`.

TIOCGWINSZ

The call `ioctl (fd, TIOCGWINSZ, struct winsize *ws)` puts the window size information associated with the terminal in `ws`. The window size structure contains the number of rows and columns (and pixels if appropiate) of the devices attached to the terminal. It is set by user software and is the means by which most full screen oriented programs determine the screen size. The winsize structure is defi ned as:

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

Many implementations ignore the members `ws_xpixel` and `ws_ypixel` and set them to 0.

TIOCSWINSZ

The call `ioctl (fd, TIOCSWINSZ, struct winsize *ws)` sets the window size associated with the terminal to the value at `ws`. If the new size is different from the old size, a `SIGWINCH` (window changed) signal is sent to the process group of the terminal. See `TIOCGWINSZ` for more details.

TIOCSETD

The call `ioctl (fd, TIOCSETD, int *ldisc);` changes the line discipline to `ldisc`. Not all systems support multiple line disciplines, and both the available line disciplines and their names depend on the system. Here are some typical ones:

- `OTTYDISC`: In System V, the “old” (Seventh Edition) tty discipline.
- `NETLDISC`: The Berknet line discipline.
- `NTTYDISC`: In System V, the “new” (termio) tty discipline.
- `TABLDISC`: The Hitachi tablet discipline.
- `NTABLDISC`: The GTCO tablet discipline.
- `MOUSELDISC`: The mouse discipline.
- `KBDLDISC`: The keyboard line discipline.
- `TTYDISC`: The termios interactive line discipline.
- `TABLDISC`: The tablet line discipline.
- `SLIPDISC`: The Serial IP (SLIP) line discipline.

TIOCGETD

The call `ioctl (fd, TIOCGETD, int *ldisc)` returns the current line discipline at `ldisc`. See the discussion in the section on `TIOCSETD` above.

Hardware control

TIOCSBRK

The call `ioctl (fd, TIOCSBRK, NULL)` sets the terminal hardware into break condition. This function is supported by 4.4BSD.

TIOCCBRK

The call `ioctl (fd, TIOCCBRK, NULL)` clears a terminal hardware BREAK condition. This function is supported by 4.4BSD.

TIOCSDTR

The call `ioctl (fd, TIOCSDTR, NULL)` asserts Data Terminal Ready (DTR). This function is supported by 4.4BSD. See page 239 for details of the DTR signal.

TIOCCDTR

The call `ioctl (fd, TIOCCDTR, NULL)` resets Data Terminal Ready (DTR). This function is supported by 4.4BSD. See page 239 for details of the DTR signal.

TIOCMSET

The call `ioctl (fd, TIOCMSET, int *state)` sets modem state. It is supported by 4.4BSD, SunOS and System V.4, but not all terminals support this call. `*state` is a bit map representing the parameters listed in table Table 15-13:

Table 15−13: TIOCMSET and TIOCMGET state bits

Parameter	meaning
TIOCM_LE	Line Enable
TIOCM_DTR	Data Terminal Ready
TIOCM_RTS	Request To Send
TIOCM_ST	Secondary Transmit
TIOCM_SR	Secondary Receive
TIOCM_CTS	Clear To Send
TIOCM_CAR	Carrier Detect
TIOCM_CD	Carrier Detect (synonym)
TIOCM_RNG	Ring Indication
TIOCM_RI	Ring Indication (synonym)
TIOCM_DSR	Data Set Ready

TIOCMGET

The call `ioctl (fd, TIOCMGET, int *state)` returns the current state of the terminal modem lines. See the description of `TIOCMSET` for the use of the bit mapped variable `state`.

TIOCMBIS

The call `ioctl (fd, TIOCMBIS, int *state)` sets the modem state in the same manner as `TIOMSET`, but instead of setting the `state` bits unconditionally, each bit is logically *or*ed with the current state.

TIOCMBIC

The call `ioctl (fd, TIOCMBIC, int *state)` clears the modem state: each bit set in the bitmap `state` is reset in the modem state. The other state bits are not affected.

TCSBRK

The call `ioctl (fd, TCSBRK, int nobreak)` drains the output queue and then sends a break if `nobreak` is not set. This function is supported in System V and SunOS. In contrast to the 4.4BSD function `TIOCSBRK`, `TCSBRK` resets the break condition automatically.

TCXONC

The call `ioctl (fd, TCXONC, int type)` specifies flow control. It is supported in System V and SunOS. Table 15-14 shows the possible values of `type`.

Table 15−14: TCXONC and tcflow type bits

Parameter	value	meaning
TCOOFF	0	suspend output
TCOON	1	restart suspended output
TCIOFF	2	suspend input
TCION	3	restart suspended input

Not all drivers support input flow control via `TCXONC`.

Queue control

TIOCOUTQ

The call `ioctl (fd, TIOCOUTQ, int *num)` sets the current number of characters in the output queue to `*num`. This function is supported by BSD and SunOS.

TIOCSTI

The call `ioctl (fd, TIOCSTI, char *cp)` simulates typed input. It inserts the character at `*cp` into the input queue. This function is supported by BSD and SunOS.

TIOCSTOP

The call `ioctl (fd, TIOCSTOP, NULL)` stops output on the terminal. It's like typing `CTRL-S` at the keyboard. This function is supported by 4.4BSD.

TIOCSTART

The call `ioctl (fd, TIOCSTART, NULL)` restarts output on the terminal, like typing `CTRL-Q` at the keyboard. This function is supported by 4.4BSD.

TIOCDRAIN

The call `ioctl (fd, TIOCDRAIN, NULL)` suspends process execution until all output is drained. This function is supported by 4.4BSD.

TIOCFLUSH

The call `ioctl (fd, TIOCFLUSH, int *what)` flushes the input and output queues. This function is supported by 4.4BSD, System V.3 and the Seventh Edition. The System V.3 and Seventh Edition implementations ignore the parameter `what` and flush both queues. 4.4BSD flushes the queues if the corresponding bits `FREAD` and `FWRITE` are set in `*what`. If no bits are set, it clears both queues.

TCFLSH

The call `ioctl (fd, TCFLSH, int type)` flushes the input or output queues, depending on the flags defined in Table 15-15.

Table 15−15: TCFLSH type bits

Parameter	value	meaning
TCIFLUSH	0	flush the input queue
TCOFLUSH	1	flush the output queue
TCIOFLUSH	2	flush both queues

This function is supported by System V. It does the same thing as `TIOCFLUSH`, but the semantics are different.

Session control

TIOCGPGRP

The call `ioctl (fd, TIOCGPGRP, pid_t *tpgrp)` sets `*tpgrp` to the ID of the current process group with which the terminal is associated. 4.4BSD uses this call to implement the function `tcgetpgrp`.

TIOCSPGRP

The call `ioctl (fd, TIOCSPGRP, pid_t *tpgrp)` associates the terminal with the process group `tpgrp`. 4.4BSD uses this call to implement the function `tcsetpgrp`.

TIOCSCTTY

`TIOCSCTTY` makes the terminal the controlling terminal for the process. This function is supported by BSD and SunOS systems. On BSD systems, the call is `ioctl (fd, TIOCSCTTY, NULL)` and on SunOS systems it is `ioctl (fd, TIOCSCTTY, int type)`. Normally the controlling terminal will be set only if no other process already owns it. In those implementations that support `type` the superuser can set `type` to 1 in order to force the takeover of the terminal, even if another process owns it. In 4.4BSD, you would first use the *revoke* system call (see Chapter 14, *File systems*, page 213) to force a close of all file descriptors associated with the file.

System V and older versions of BSD have no equivalent of this function. In these systems, when a process group leader without a controlling terminal opens a terminal, it automatically becomes the controlling terminal. There are methods to ovverride this behaviour: in System V, you set the flag `O_NOCTTY` when you open ther terminal. In old BSD versions, you subsequently release the control of the terminal with the `TIOCNOTTY` request, which we'll look at in the next section.

TIOCNOTTY

Traditionally, the first time a process without a controlling terminal opened a terminal, it acquired that terminal as its controlling terminal. We saw in the section on `TIOCSCTTY` above that this is no longer the default behaviour in BSD, and that you can override it in System V. Older BSD versions, including SunOS, did not offer either of these choices. Instead, you had to accept that you acquired a controlling terminal, and then release the controlling terminal again with ioctl `TIOCNOTTY`. If you find this code in a package, and your system doesn't support it, you can eliminate it. If your system is based on System V, you should check the call to `open` for the terminal and ensure that the flag `O_NOCTTY` is set.

A second use for `TIOCNOTTY` was after a `fork`, when the child might want to relinquish the controlling terminal. This can also be done with `setsid` (see Chapter 12, *Kernel dependencies*, page 171).

TIOCGSID

The call `ioctl (fd, TIOCGSID, pid_t *pid)` stores the terminal's session ID at `pid`. This function is supported by System V.4.

Miscellaneous functions

TIOCEXCL

The call `ioctl (fd, TIOCEXCL, NULL)` sets exclusive use on the terminal. No further opens are permitted except by root.

TIOCNXCL

The call `ioctl (fd, TIOCNXCL, NULL)` clears exclusive use of the terminal (see `TIOCEXCL`). Further opens are permitted.

TIOCCONS

The call `ioctl (fd, TIOCCONS, int *on)` sets the console file. If `on` points to a non-zero integer, kernel console output is redirected to the terminal specified in the call. If `on` points to zero, kernel console output is redirected to the standard console. This is usually used on work stations to redirect kernel messages to a particular window.

TIOCGSOFTCAR

The call `ioctl (fd, TIOCGSOFTCAR, int *set)` sets `*set` to 1 if the terminal "Data carrier detect" (DCD) signal or the software carrier flag is asserted, and to 0 otherwise. This function is supported only in SunOS 4.X, and is no longer present in Solaris 2. See page 239 for a description of the DSR line.

TIOCSSOFTCAR

The call `ioctl (fd, TIOCSSOFTCAR, int *set)` is a method to fake a modem carrier detect signal. It resets software carrier mode if `*set` is zero and sets it otherwise. In software carrier mode, the `TIOCGSOFTCAR` call always returns 1; otherwise it returns the real value of the DCD interface signal. This function is supported only in SunOS 4.X, and is no longer present in Solaris 2.

termios functions

It should come as no surprise that people have long wanted a less bewildering interface to terminals than the `ioctl` calls that we looked at in the previous section. In POSIX.1, a number of new functions were introduced with the intent of bringing some sort of order into the chaos. A total of 8 new functions were introduced, split into three groups. In addition, a further 6 auxiliary functions were added:

- `tcgetattr` and `tcsetattr` get and set terminal attributes using `struct termios`.
- `tcgetpgrp` and `tcsetpgrp` get and set the program group ID.
- `tcdrain`, `tcflow`, `tcflush` and `tcsendbreak` manipulate the terminal hardware.
- `cfgetispeed`, `cfsetispeed`, `cfgetospeed`, `cfsetospeed`, `cfsetspeed` and `cfmakeraw` are auxiliary functions to manipulate `termios` entries.

These functions do not add new functionality, but attempt to provide a more uniform interface. In some systems, they are system calls, whereas in others they are library functions that build on the `ioctl` interface. If you are porting a package that uses `termios`, and your system doesn't supply it, you have the choice of rewriting the code to use `ioctl` calls, or you can use the 4.4BSD library calls supplied in the 4.4BSD Lite distribution (usr/src/lib/libc/gen/termios.c). In the following sections we'll look briefly at each function.

Direct termios functions

tcgetattr

`tcgetattr` corresponds to `TIOCGETA` described on page 256. It returns the current `termios` state to `term`.

```
#include <termios.h>
int tcgetattr (int fd, struct termios *term)
```

tcsetattr

`tcgetattr` sets the current `termios` state from `term`.

```
#include <termios.h>
int tcsetattr (int fd, int action, struct termios *t)
```

`action` can have one of the values listed in Table 15-16.

Table 15−16: tcsetattr action flags

Parameter	meaning
TCSANOW	Change terminal parameters immediately. Corresponds to the `ioctl` request TIOCSETA.
TCSADRAIN	First drain output, then change the parameters. Used when changing parameters that affect output. Corresponds to the `ioctl` call TIOCSETAW.
TCSAFLUSH	Discard any pending input, drain output, then change the parameters. Corresponds to `ioctl` call TIOCSETAF.

See page 257 for details of the corresponding `ioctl` interfaces.

In addition, some implementations define the parameter TCSASOFT: if this is specified in addition to one of the above flags, the values of the fields `c_cflag`, `c_ispeed` and `c_ospeed` are ignored. This is typically used when the device in question is not a serial line terminal.

tcgetpgrp

`tcgetpgrp` returns the ID of the current process group with which the terminal is associated. It corresponds to the `ioctl` call TIOCGPGRP described on page 263.

```
#include <sys/types.h>
#include <unistd.h>
pid_t tcgetpgrp (int fd);
```

tcsetpgrp

`tcsetpgrp` associates the terminal with the process group `tpgrp`. It corresponds to the `ioctl` call TIOCSPGRP described on page 263.

```
#include <sys/types.h>
#include <unistd.h>
int tcsetpgrp (int fd, pid_t pgrp_id);
```

tcdrain

`tcdrain` suspends the process until all output is drained. It corresponds to the `ioctl` call TIOCDRAIN described on page 262.

```
#include <termios.h>
int tcdrain (int fd);
```

tcflow

`tcflow` specifies flow control. It corresponds to the `ioctl` call `TCXONC`. See the description of `TCXONC` on page 262 for the meaning of the parameter `action`.

```
#include <termios.h>
int tcflow (int fd, int action);
```

tcflush

`tcflush` flushes input or output queues for `fd`.

```
#include <termios.h>
int tcflush (int fd, int action);
```

`action` may take the values shown in Table 15-17.

Table 15−17: `tcflush` action bits

Parameter	meaning
TCIFLUSH	Flush data received but not read
TCOFLUSH	Flush data written but not transmitted
TCIOFLUSH	Flush both data received but not read and data written but not transmitted

This function corresponds to the `ioctl` request `TCFLSH` described on page 263.

tcsendbreak

`tcsendbreak` sends a break indication on the line. This is equivalent to the `ioctl` request `TCSBRK` described on page 261.

```
#include <termios.h>
int tcsendbreak (int fd, int len);
```

termios auxiliary functions

In addition to the `termios` functions in the previous section, a number of library functions manipulate `termios` struct entries. With one exception, they handle line speeds. They don't have any direct effect on the line—you need a `tcsetattr` for that—but they provide a link between the viewpoint of the application and the underlying implementation.

There is still no agreement on how to represent line speeds. BSD systems use the bit rate as an integer and store it in the fields `c_ispeed` and `c_ospeed`. They leave it to the driver to explain it to the hardware, so you can effectively specify any speed the hardware is capable of handling. By contrast, System V still uses the small numeric indices that were used in the

Seventh Edition* (see page 240), which allows the field to be stored in 4 bits. They are located in the field `c_cflag`. This is *not* a good idea, because these speeds are the only ones System V knows about. If you have a V.32bis, V.42bis modem that claims to be able to transfer data at up to 57,600 bps, you will not be able to take full advantage of its capabilities with System V. In addition, there is only one speed constant, which sets both the input and output speeds. The functions for setting input and output speed are effectively the same thing.

In addition to these problems, SCO UNIX System V.3 further complicates the issue by providing the fields `s_ospeed` and `s_ispeed` in the `struct termios`. The functions `cfsetispeed` and `cfsetospeed` set these fields in addition to the four bits in `c_cflag`, but the functions `cfgetispeed` and `cfgetospeed` retrieve the values from `c_cflags`, so it's not clear what use the fields `c_ispeed` and `c_ospeed` are intended to be.

Setting the bit rates is thus not quite as simple as it might appear: the preprocessor variables `B9600` and friends might not equate to the kind of constant that the termios implementation needs, and there is no designated place in the `termios` structure to store the bit rates.

This problem is solved by the following functions, which are normally macros:

- `speed_t cfgetispeed (struct termios *t)` returns `t`'s input speed in `speed_t` format. It is undefined if the speed is not representable as `speed_t`.
- `int cfsetispeed (struct termios *t, speed_t speed)` sets `t`'s input speed to the internal representation of `speed`.
- `speed_t cfgetospeed (struct termios *t)` returns `t`'s output speed in `speed_t` format. The result is undefined if the speed is not representable as `speed_t`.
- `int cfsetospeed (struct termios *t, speed_t speed)` sets `t`'s output speed to the internal representation of `speed`.
- `void cfsetspeed (struct termios *t, speed_t speed)` sets both input and output speed to the internal representation of `speed`.
- `void cfmakeraw (struct termios *t)` sets the whole structure `t` to default values.

* These constants were originally the values that were written to the interface hardware to set the speed.