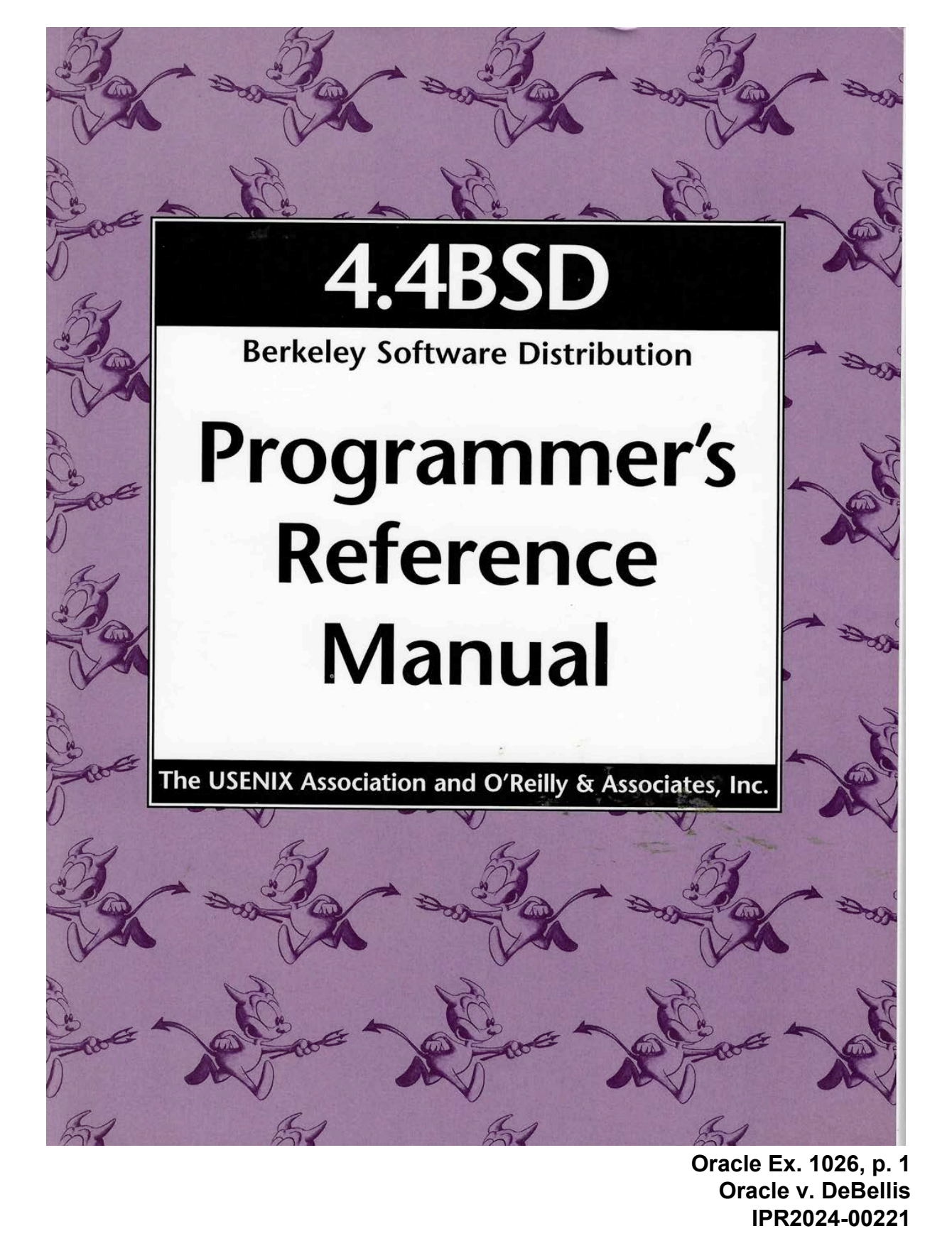




4.4BSD

Berkeley Software Distribution

Programmer's Reference Manual

The USENIX Association and O'Reilly & Associates, Inc.

First Printing, 1994

Copyright 1979, 1980, 1983, 1986, 1993, 1994 The Regents of the University of California. All rights reserved.

Other than the specific manual pages and documents listed below as copyrighted by AT&T, redistribution and use of this manual in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- 1) Redistributions of this manual must retain the copyright notices on this page, this list of conditions and the following disclaimer.
- 2) Software or documentation that incorporates part of this manual must reproduce the copyright notices on this page, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- 3) All advertising materials mentioning features or use of this software must display the following acknowledgement: "This product includes software developed by the University of California, Berkeley and its contributors."
- 4) Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The Institute of Electrical and Electronics Engineers and the American National Standards Committee X3, on Information Processing Systems have given us permission to reprint portions of their documentation.

In the following statement, the phrase "this text" refers to portions of the system documentation.

"Portions of this text are reprinted and reproduced in electronic form in 4.4BSD from IEEE Std 1003.1-1988, IEEE Standard Portable Operating System Interface for Computer Environments (POSIX), copyright 1988 by the Institute of Electrical and Electronics Engineers, Inc. In the event of any discrepancy between these versions and the original IEEE Standard, the original IEEE Standard is the referee document."

In the following statement, the phrase "This material" refers to portions of the system documentation.

"This material is reproduced with permission from American National Standards Committee X3, on Information Processing Systems. Computer and Business Equipment Manufacturers Association (CBEMA), 311 First St., NW, Suite 500, Washington, DC 20001-2178. The developmental work of Programming Language C was completed by the X3J11 Technical Committee."

Manual pages ptrace.2, ecvt.3, mp.3, plot.3, L-devices.5, L-dialcodes.5, L.aliases.5, L.cmds.5, L.sys.5, USERFILE.5, dump.5, and plot.5 are copyright 1979, AT&T Bell Laboratories, Incorporated. Holders of UNIXTM/32V, System III, or System V software licenses are permitted to copy these documents, or any portion of them, as necessary for licensed use of the software, provided this copyright notice and statement of permission are included.

The views and conclusions contained in this manual are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Regents of the University of California.

This book was printed and bound in the United States of America.
Distributed by O'Reilly & Associates, Inc.

[recycle logo] This book is printed on acid-free paper with 50% recycled content, 10-13% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-078-3

NAME

ioctl – control device

SYNOPSIS

```
#include <sys/ioctl.h>

int
ioctl(int d, unsigned long request, char *argp);
```

DESCRIPTION

The **ioctl()** function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl()** requests. The argument *d* must be an open file descriptor.

An *ioctl request* has encoded in it whether the argument is an “in” parameter or “out” parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an *ioctl request* are located in the file `<sys/ioctl.h>`.

RETURN VALUES

If an error has occurred, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

Ioctl() will fail if:

- [EBADF] *d* is not a valid descriptor.
- [ENOTTY] *d* is not associated with a character special device.
- [ENOTTY] The specified request does not apply to the kind of object that the descriptor *d* references.
- [EINVAL] *Request* or *argp* is not valid.

SEE ALSO

mt(1), **execve(2)**, **fcntl(2)**, **tty(4)**, **intro(4)**

HISTORY

An **ioctl** function call appeared in Version 7 AT&T UNIX.

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int
printf(const char *format, ...);

int
fprintf(FILE *stream, const char *format, ...);

int
sprintf(char *str, const char *format, ...);

int
snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>

int
vprintf(const char *format, va_list ap);

int
vfprintf(FILE *stream, const char *format, va_list ap);

int
vsprintf(char *str, const char *format, va_list ap);

int
vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

DESCRIPTION

The **printf()** family of functions produces output according to a *format* as described below. **Printf()** and **vprintf()** write output to *stdout*, the standard output stream; **fprintf()** and **vfprintf()** write output to the given output *stream*; **sprintf()**, **snprintf()**, **vsprintf()**, and **vsnprintf()** write to the character string *str*. These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg(3)**) are converted for output. These functions return the number of characters printed (not including the trailing '\0' used to end output to strings). **Snprintf()** and **vsnprintf()** will write at most *size-1* of the characters printed into the output string (the *size*'th character then gets the terminating '\0'); if the return value is greater than or equal to the *size* argument, the string was too short and some of the printed characters were discarded. **Sprintf()** and **vsprintf()** effectively assume an infinite *size*.

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. The arguments must correspond properly (after type promotion) with the conversion specifier. After the %, the following appear in sequence:

- Zero or more of the following flags:
 - A # character specifying that the value should be converted to an “alternate form”. For c, d, i, n, p, s, and u, conversions, this option has no effect. For o conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For x and X conversions, a non-zero result has the string '0x' (or

'0x' for X conversions) prepended to it. For e, E, f, g, and G, conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For g and G conversions, trailing zeros are not removed from the result as they would otherwise be.

- A zero '0' character specifying zero padding. For all conversions except n, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (Mc d, i, o, u, i, x, and X), the '0' flag is ignored.
- A negative field width flag '-' indicates the converted value is to be left adjusted on the field boundary. Except for n conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A '-' overrides a '0' if both are given.
- A space, specifying that a blank should be left before a positive number produced by a signed conversion (d, e, E, f, g, G, or i).
- A '+' character specifying that a sign always be placed before a number produced by a signed conversion. A '+' overrides a space if both are used.
- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.
- An optional precision, in the form of a period '.' followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point for e, E, and f conversions, the maximum number of significant digits for g and G conversions, or the maximum number of characters to be printed from a string for s conversions.
- The optional character h, specifying that a following d, i, o, u, x, or X conversion corresponds to a *short int* or *unsigned short int* argument, or that a following n conversion corresponds to a pointer to a *short int* argument.
- The optional character l (ell) specifying that a following d, i, o, u, x, or X conversion applies to a pointer to a *long int* or *unsigned long int* argument, or that a following n conversion corresponds to a pointer to a *long int* argument.
- The optional character q, specifying that a following d, i, o, u, x, or X conversion corresponds to a *quad int* or *unsigned quad int* argument, or that a following n conversion corresponds to a pointer to a *quad int* argument.
- The character L specifying that a following e, E, f, g, or G conversion corresponds to a *long double* argument (but note that long double values are not currently supported by the VAX and Tahoe compilers).
- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '*' instead of a digit string. In this case, an *int* argument supplies the field width or precision. A negative field width is treated as a left adjustment flag followed by a positive field width; a negative precision is treated as though it were missing.

The conversion specifiers and their meanings are:

- dioxX** The *int* (or appropriate variant) argument is converted to signed decimal (**d** and **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters **abcdef** are used for **x** conversions; the letters **ABCDEF** are used for conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.
- DOU** The *long int* argument is converted to signed decimal, unsigned octal, or unsigned decimal, as if the format had been **ld**, **lo**, or **lu** respectively. These conversion characters are deprecated, and will eventually disappear.
- eE** The *double* argument is rounded and converted in the style **[-]d.ddde±dd** where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An **E** conversion uses the letter **E** (rather than **e**) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is **00**.
- f** The *double* argument is rounded and converted to decimal notation in the style **[-]ddd.ddd**, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.
- g** The *double* argument is converted in style **f** or **e** (or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.
- c** The *int* argument is converted to an *unsigned char*, and the resulting character is written.
- s** The *"char *"* argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.
- p** The *"void *"* pointer argument is printed in hexadecimal (as if by **%#x** or **%#lx**).
- n** The number of characters written so far is stored into the integer indicated by the *"int *"* (or variant) pointer argument. No argument is converted.
- %** A **'%'** is written. No argument is converted. The complete conversion specification is **'%%'**.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

EXAMPLES

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

To print π to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

To allocate a 128 byte string and print into it:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *newfmt(const char *fmt, ...)
{
    char *p;
    va_list ap;
    if ((p = malloc(128)) == NULL)
        return (NULL);
    va_start(ap, fmt);
    (void) vsnprintf(p, 128, fmt, ap);
    va_end(ap);
    return (p);
}
```

SEE ALSO

printf(1), scanf(3)

STANDARDS

The `fprintf()`, `printf()`, `sprintf()`, `vprintf()`, `vfprintf()`, and `vsprintf()` functions conform to ANSI C X3.159-1989 ("ANSI C").

HISTORY

The functions `snprintf()` and `vsnprintf()` are new to this release.

BUGS

The conversion formats `%D`, `%O`, and `%U` are not standard and are provided only for backward compatibility. The effect of padding the `%p` format with zeros (either by the `'0'` flag or by specifying a precision), and the benign effect (i.e., none) of the `'#'` flag on `%n` and `%p` conversions, as well as other nonsensical combinations such as `%Ld`, are not standard; such combinations should be avoided.

Because `sprintf()` and `vsprintf()` assume an infinitely long string, callers must be careful not to overflow the actual space; this is often impossible to assure. For safety, programmers should use the `snprintf()` interface instead. Unfortunately, this interface is not portable.

NAME

stty, gtty – set and get terminal state (defunct)

SYNOPSIS

```
#include <sgtty.h>

stty(int fd, struct sgttyb *buf);
gtty(int fd, struct sgttyb *buf);
```

DESCRIPTION

These interfaces are obsoleted by `ioctl(2)`. They are available from the compatibility library, `libcompat`.

The `stty()` function sets the state of the terminal associated with `fd`. The `gtty()` function retrieves the state of the terminal associated with `fd`. To set the state of a terminal the call must have write permission.

The `stty()` call is actually `ioctl(fd, TIOCSETP, buf)`, while the `gtty()` call is `ioctl(fd, TIOCGETP, buf)`. See `ioctl(2)` and `tty(4)` for an explanation.

DIAGNOSTICS

If the call is successful 0 is returned, otherwise -1 is returned and the global variable `errno` contains the reason for the failure.

SEE ALSO

`ioctl(2)`, `tty(4)`

HISTORY

The `stty()` and `gtty()` functions appeared in 4.2BSD.

NAME

tty – general terminal interface

SYNOPSIS

```
#include <sys/ioctl.h>
```

DESCRIPTION

This section describes the interface to the terminal drivers in the system.

Terminal Special Files

Each hardware terminal port on the system usually has a terminal special device file associated with it in the directory “/dev/” (for example, “/dev/tty03”). When a user logs into the system on one of these hardware terminal ports, the system has already opened the associated device and prepared the line for normal interactive use (see `getty(8)`.) There is also a special case of a terminal file that connects not to a hardware terminal port, but to another program on the other side. These special terminal devices are called *ptys* and provide the mechanism necessary to give users the same interface to the system when logging in over a network (using `rlogin(1)`, or `telnet(1)` for example.) Even in these cases the details of how the terminal file was opened and set up is already handled by special software in the system. Thus, users do not normally need to worry about the details of how these lines are opened or used. Also, these lines are often used for dialing out of a system (through an out-calling modem), but again the system provides programs that hide the details of accessing these terminal special files (see `tip(2)`.)

When an interactive user logs in, the system prepares the line to behave in a certain way (called a *line discipline*), the particular details of which is described in `stty(1)` at the command level, and in `termios(4)` at the programming level. A user may be concerned with changing settings associated with his particular login terminal and should refer to the preceding man pages for the common cases. The remainder of this man page is concerned with describing details of using and controlling terminal devices at a low level, such as that possibly required by a program wishing to provide features similar to those provided by the system.

Line disciplines

A terminal file is used like any other file in the system in that it can be opened, read, and written to using standard system calls. For each existing terminal file, there is a software processing module called a *line discipline* is associated with it. The *line discipline* essentially glues the low level device driver code with the high level generic interface routines (such as `read(2)` and `write(2)`), and is responsible for implementing the semantics associated with the device. When a terminal file is first opened by a program, the default *line discipline* called the *termios* line discipline is associated with the file. This is the primary line discipline that is used in most cases and provides the semantics that users normally associate with a terminal. When the *termios* line discipline is in effect, the terminal file behaves and is operated according to the rules described in `termios(4)`. Please refer to that man page for a full description of the terminal semantics. The operations described here generally represent features common across all *line disciplines*, however some of these calls may not make sense in conjunction with a line discipline other than *termios*, and some may not be supported by the underlying hardware (or lack thereof, as in the case of *ptys*).

Terminal File Operations

All of the following operations are invoked using the `ioctl(2)` system call. Refer to that man page for a description of the *request* and *argp* parameters. In addition to the *ioctl requests* defined here, the specific line discipline in effect will define other *requests* specific to it (actually `termios(4)` defines them as function calls, not *ioctl requests*.) The following section lists the available *ioctl requests*. The name of the request, a description of its purpose, and the typed *argp* parameter (if any) are listed. For example, the first entry says

*TIOCSETD int *ldisc*

and would be called on the terminal associated with file descriptor zero by the following code fragment:

```
int ldisc;

ldisc = TTYDISC;
ioctl(0, TIOCSETD, &ldisc);
```

Terminal File Request Descriptions

*TIOCSETD int *ldisc*

Change to the new line discipline pointed to by *ldisc*. The available line disciplines are listed in `<sys/termios.h>` and currently are:

TTYDISC	Termios interactive line discipline.
TABLDISC	Tablet line discipline.
SLIPDISC	Serial IP line discipline.

*TIOCGETD int *ldisc*

Return the current line discipline in the integer pointed to by *ldisc*.

TIOCSBRK void

Set the terminal hardware into BREAK condition.

TIOCCBRK void

Clear the terminal hardware BREAK condition.

TIOCSDTR void

Assert data terminal ready (DTR).

TIOCCDTR void

Clear data terminal ready (DTR).

*TIOCGPGRP int *tpgrp*

Return the current process group the terminal is associated with in the integer pointed to by *tpgrp*. This is the underlying call that implements the `termios(4)` `tcgetattr()` call.

*TIOCSPGRP int *tpgrp*

Associate the terminal with the process group (as an integer) pointed to by *tpgrp*. This is the underlying call that implements the `termios(4)` `tcsetattr()` call.

*TIOCGETA struct termios *term*

Place the current value of the termios state associated with the device in the termios structure pointed to by *term*. This is the underlying call that implements the `termios(4)` `tcgetattr()` call.

*TIOCSETA struct termios *term*

Set the termios state associated with the device immediately. This is the underlying call that implements the `termios(4)` `tcsetattr()` call with the `TCSANOW` option.

*TIOCSETAW struct termios *term*

First wait for any output to complete, then set the termios state associated with the device. This is the underlying call that implements the `termios(4)` `tcsetattr()` call with the `TCSADRAIN` option.

TIOCSETAF *struct termios *term*

First wait for any output to complete, clear any pending input, then set the termios state associated with the device. This is the underlying call that implements the `termios(4)` `tcsetattr()` call with the `TCSAFLUSH` option.

TIOCOUTQ *int *num*

Place the current number of characters in the output queue in the integer pointed to by *num*.

TIOCSTI *char *cp*

Simulate typed input. Pretend as if the terminal received the character pointed to by *cp*.

TIOCNOTTY *void*

This call is obsolete but left for compatibility. In the past, when a process that didn't have a controlling terminal (see *The Controlling Terminal* in `termios(4)`) first opened a terminal device, it acquired that terminal as its controlling terminal. For some programs this was a hazard as they didn't want a controlling terminal in the first place, and this provided a mechanism to disassociate the controlling terminal from the calling process. It *must* be called by opening the file `/dev/tty` and calling `TIOCNOTTY` on that file descriptor.

The current system does not allocate a controlling terminal to a process on an `open()` call: there is a specific ioctl called `TIOCSCTTY` to make a terminal the controlling terminal. In addition, a program can `fork()` and call the `setsid()` system call which will place the process into its own session - which has the effect of disassociating it from the controlling terminal. This is the new and preferred method for programs to lose their controlling terminal.

TIOCSTOP *void*

Stop output on the terminal (like typing `^S` at the keyboard).

TIOCSTART *void*

Start output on the terminal (like typing `^Q` at the keyboard).

TIOCSCTTY *void*

Make the terminal the controlling terminal for the process (the process must not currently have a controlling terminal).

TIOC Drain *void*

Wait until all output is drained.

TIOCEXCL *void*

Set exclusive use on the terminal. No further opens are permitted except by root. Of course, this means that programs that are run by root (or `setuid`) will not obey the exclusive setting - which limits the usefulness of this feature.

TIOCNXCL *void*

Clear exclusive use of the terminal. Further opens are permitted.

TIOCFLUSH *int *what*

If the value of the *int* pointed to by *what* contains the FREAD bit as defined in `<sys/file.h>`, then all characters in the input queue are cleared. If it contains the FWRITE bit, then all characters in the output queue are cleared. If the value of the integer is zero, then it behaves as if both the FREAD and FWRITE bits were set (i.e. clears both queues).

TIOCGWINSZ *struct winsize *ws*

Put the window size information associated with the terminal in the *winsize* structure pointed to by *ws*. The window size structure contains the number of rows and columns (and pixels if appropriate) 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 defined in `<sys/ioctl.h>`.

TIOCSWINSZ *struct winsize *ws*

Set the window size associated with the terminal to be the value in the *winsize* structure pointed to by *ws* (see above).

TIOCCONS *int *on*

If *on* points to a non-zero integer, redirect kernel console output (kernel `printf`'s) to this terminal. If *on* points to a zero integer, redirect kernel console output back to the normal console. This is usually used on workstations to redirect kernel messages to a particular window.

TIOCMSET *int *state*

The integer pointed to by *state* contains bits that correspond to modem state. Following is a list of defined variables and the modem state they represent:

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.

This call sets the terminal modem state to that represented by *state*. Not all terminals may support this.

TIOCMGET *int *state*

Return the current state of the terminal modem lines as represented above in the integer pointed to by *state*.

TIOCMBS *int *state*

The bits in the integer pointed to by *state* represent modem state as described above, however the state is OR-ed in with the current state.

TIOCMBC *int *state*

The bits in the integer pointed to by *state* represent modem state as described above, however each bit which is on in *state* is cleared in the terminal.

SEE ALSO

getty(8), ioctl(2), pty(4), stty(1), termios(4)

NAME

termcap – terminal capability data base

SYNOPSIS

termcap

DESCRIPTION

The **Termcap** file is a data base describing terminals, used, for example, by **vi(1)** and **curses(3)**. Terminals are described in **termcap** by giving a set of capabilities that they have and by describing how operations are performed. Padding requirements and initialization sequences are included in **termcap**.

Entries in **termcap** consist of a number of ':'-separated fields. The first entry for each terminal gives the names that are known for the terminal, separated by 'I' characters. The first name given is the most common abbreviation for the terminal. The last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should be in lower case and contain no blanks; the last name may well contain upper case characters and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens. Modes that the hardware can be in or user preferences should be indicated by appending a hyphen and an indicator of the mode. Therefore, a "vt100" in 132-column mode would be "vt100-w". The following suffixes should be used where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	vt100-w
-am	With automatic margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-n	Number of lines on screen	aaa-60
-na	No arrow keys (leave them in local)	concept100-na
-np	Number of pages of memory	concept100-4p
-rv	Reverse video	concept100-rv

CAPABILITIES

The characters in the *Notes* function field in the table have the following meanings (more than one may apply to a capability):

N	indicates numeric parameter(s)
P	indicates that padding may be specified
*	indicates that padding may be based on the number of lines affected
o	indicates capability is obsolete

"Obsolete" capabilities have no *terminfo* equivalents, since they were considered useless, or are subsumed by other capabilities. New software should not rely on them at all.

Name	Type	Notes	Description
functions			
ae	str	(P)	End alternate character set.
AL	str	(NP*)	Add <i>n</i> new blank lines
al	str	(P*)	Add new blank line.
am	bool		Terminal has automatic margins.
as	str	(P)	Start alternate character set.

bc	str	(o)	Backspace if not. ^H .
bl	str	(P)	Audible signal (bell).
bs	bool	(o)	Terminal can backspace with ^H .
bt	str	(P)	Back tab.
bw	bool		le (backspace) wraps from column 0 to last column.
CC	str		Terminal settable command character in prototype.
cd	str	(P*)	Clear to end of display.
ce	str	(P)	Clear to end of line.
ch	str	(NP)	Set cursor column (horizontal position).
cl	str	(P*)	Clear screen and home cursor.
CM	str	(NP)	Memory-relative cursor addressing.
cm	str	(NP)	Screen-relative cursor motion.
co	num		Number of columns in a line (See BUGS section below).
cr	str	(P)	Carriage return.
cs	str	(NP)	Change scrolling region (VT100).
ct	str	(P)	Clear all tab stops.
cv	str	(NP)	Set cursor row (vertical position).
da	bool		Display may be retained above the screen.
dB	num	(o)	Milliseconds of bs delay needed (default 0).
db	bool		Display may be retained below the screen.
DC	str	(NP*)	Delete <i>n</i> characters.
dC	num	(o)	Milliseconds of cr delay needed (default 0).
dc	str	(P*)	Delete character.
dF	num	(o)	Milliseconds of ff delay needed (default 0).
DL	str	(NP*)	Delete <i>n</i> lines.
dl	str	(P*)	Delete line.
dm	str		Enter delete mode.
dN	num	(o)	Milliseconds of nl delay needed (default 0).
DO	str	(NP*)	Move cursor down: <i>n</i> lines.
do	str		Down one line.
ds	str		Disable status line.
dT	num	(o)	Milliseconds of horizontal tab delay needed (default 0).
dV	num	(o)	Milliseconds of vertical tab delay needed (default 0).
ec	str	(NP)	Erase <i>n</i> characters.
ed	str		End delete mode.
ei	str		End insert mode.
eo	bool		Can erase overstrikes with a blank.
EP	bool	(o)	Even parity.
es	bool		Escape can be used on the status line.
ff	str	(P*)	Hardcopy terminal page eject.
fs	str		Return from status line.
gn	bool		Generic line type, for example dialup, switch).
hc	bool		Hardcopy terminal.
HD	bool	(o)	Half-duplex.
hd	str		Half-line down (forward 1/2 linefeed).
ho	str	(P)	Home cursor.
hs	bool		Has extra "status line".
hu	str		Half-line up (reverse 1/2 linefeed).
hz	bool		Cannot print "~~" (Hazeltine).

i1-i3	str		Terminal initialization strings (terminfo only)
IC	str	(NP*)	Insert <i>n</i> blank characters.
ic	str	(P*)	Insert character.
if	str		Name of file containing initialization string.
im	str		Enter insert mode.
in	bool		Insert mode distinguishes nulls.
iP	str		Pathname of program for initialization (terminfo only).
ip	str	(P*)	Insert pad after character inserted.
is	str		Terminal initialization string (termcap only).
it	num		Tabs initially every <i>n</i> positions.
K1	str		Sent by keypad upper left.
K2	str		Sent by keypad upper right.
K3	str		Sent by keypad center.
K4	str		Sent by keypad lower left.
K5	str		Sent by keypad lower right.
k0-k9	str		Sent by function keys 0-9.
kA	str		Sent by insert-line key.
ka	str		Sent by clear-all-tabs key.
kb	str		Sent by backspace key.
kC	str		Sent by clear-screen or erase key.
kD	str		Sent by delete-character key.
kd	str		Sent by down-arrow key.
kE	str		Sent by clear-to-end-of-line key.
ke	str		Out of "keypad transmit" mode.
kF	str		Sent by scroll-forward/down key.
kH	str		Sent by home-down key.
kh	str		Sent by home key.
kI	str		Sent by insert-character or enter-insert-mode key.
kL	str		Sent by delete-line key.
kl	str		Sent by left-arrow key.
kM	str		Sent by insert key while in insert mode.
km	bool		Has a "meta" key (shift, sets parity bit).
kN	str		Sent by next-page key.
kn	num	(o)	Number of function (k0-k9) keys (default 0).
ko	str	(o)	Termcap entries for other non-function keys.
kP	str		Sent by previous-page key.
kR	str		Sent by scroll-backward/up key.
kr	str		Sent by right-arrow key.
kS	str		Sent by clear-to-end-of-screen key.
ks	str		Put terminal in "keypad transmit" mode.
kT	str		Sent by set-tab key.
kt	str		Sent by clear-tab key.
ku	str		Sent by up-arrow key.
l0-l9	str		Labels on function keys if not "fn".
LC	bool	(o)	Lower-case only.
LE	str	(NP)	Move cursor left <i>n</i> positions.
le	str	(P)	Move cursor left one position.
li	num		Number of lines on screen or page (See BUGS section below)
ll	str		Last line, first column

li	num		Lines of memory if > li (0 means varies).
ma	str	(o)	Arrow key map (used by v.i. version 2 only).
mb	str		Turn on blinking attribute.
md	str		Turn on bold (extra bright) attribute.
me	str		Turn off all attributes.
mh	str		Turn on half-bright attribute.
mi	bool		Safe to move while in insert mode.
mk	str		Turn on blank attribute (characters invisible).
ml	str	(o)	Memory lock on above cursor.
mm	str		Turn on "meta mode" (8th bit).
mo	str		Turn off "meta mode".
mp	str		Turn on protected attribute.
mr	str		Turn on reverse-video attribute.
ms	bool		Safe to move in standout modes.
mu	str	(o)	Memory unlock (turn off memory lock).
nc	bool	(o)	No correctly-working cr (Datamedia 2500, Hazeltine 2000).
nd	str		Non-destructive space (cursor right).
NL	bool	(o)	\n is newline, not line feed.
nl	str	(o)	Newline character if not \n .
ns	bool	(o)	Terminal is a CRT but doesn't scroll.
nw	str	(P)	Newline (behaves like cr followed by do).
OP	bool	(o)	Odd parity.
os	bool		Terminal overstrikes.
pb	num		Lowest baud where delays are required.
pc	str		Pad character (default NUL).
pf	str		Turn off the printer.
pk	str		Program function key <i>n</i> to type string <i>s</i> (terminfo only).
pl	str		Program function key <i>n</i> to execute string <i>s</i> (terminfo only).
pO	str	(N)	Turn on the printer for <i>n</i> bytes.
po	str		Turn on the printer.
ps	str		Print contents of the screen.
pt	bool	(o)	Has hardware tabs (may need to be set with is).
px	str		Program function key <i>n</i> to transmit string <i>s</i> (terminfo only).
r1-r3	str		Reset terminal completely to sane modes (terminfo only).
rc	str	(P)	Restore cursor to position of last sc .
rf	str		Name of file containing reset codes.
RI	str	(NP)	Move cursor right <i>n</i> positions.
rp	str	(NP*)	Repeat character <i>c</i> <i>n</i> times.
rs	str		Reset terminal completely to sane modes (termcap only).
sa	str	(NP)	Define the video attributes.
sc	str	(P)	Save cursor position.
se	str		End standout mode.
SF	str	(NP*)	Scroll forward <i>n</i> lines.
sf	str	(P)	Scroll text up.
sg	num		Number of garbage chars left by so or se (default 0).
so	str		Begin standout mode.
SR	str	(NP*)	Scroll backward <i>n</i> lines.
sr	str	(P)	Scroll text down.
st	str		Set a tab in all rows, current column.

ta	str	(P)	Tab to next 8-position hardware tab stop.
tc	str		Entry of similar terminal – must be last.
te	str		String to end programs that use termcap .
ti	str		String to begin programs that use termcap .
ts	str	(N)	Go to status line, column <i>n</i> .
UC	bool	(o)	Upper-case only.
uc	str		Underscore one character and move past it.
ue	str		End underscore mode.
ug	num		Number of garbage chars left by us or ue (default 0).
ul	bool		Underline character overstrikes.
UP	str	(NP*)	Move cursor up <i>n</i> lines.
up	str		Upline (cursor up).
us	str		Start underscore mode.
vb	str		Visible bell (must not move cursor).
ve	str		Make cursor appear normal (undo vs / vi).
vi	str		Make cursor invisible.
vs	str		Make cursor very visible.
vt	num		Virtual terminal number (not supported on all systems).
wi	str	(N)	Set current window.
ws	num		Number of columns in status line.
xb	bool		Beehive (f1= ESC, f2=^C).
xn	bool		Newline ignored after 80 cols (Concept).
xo	bool		Terminal uses xoff/xon (DC3/DC1) handshaking.
xr	bool	(o)	Return acts like ce cr nl (Delta Data).
xs	bool		Standout not erased by overwriting (Hewlett-Packard).
xt	bool		Tabs ruin, magic char (Teleray 1061).
xx	bool	(o)	Tektronix 4025 insert-line.

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the `termcap` file as of this writing.

```
ca | concept100 | c100 | concept | c104 | concept100-4p | HDS Concept-100:\  
 :al=3*\E^R:am:bl=~G:cd=16*\E^C:ce=16\E~U:c1=2*\L:cm=\Ea%+ %+:\  
 :co#80:.cr=9^M:db:dc=16\EA^A:d1=3*\EB^B:do=~J:e1=e\X200:eo:im=E^P:i:n:\  
 :ip=1*:is=EU/EV/E7/E5/E8/EI|\ENH|EK/E\200/Eo/\200/Eo/\47/E:k1=l:E5:\  
 :k2=\E6:k3=\E7:kb=~h:kd=E<:ke=\Ex:kh=E?:kl=E>:kr=E=:ks=\EX:\\  
 :ku=E;:le=~H:l.i#24:m:b=\EC:me=\EN\200:m:h==EE::mi:mK=\EH:mp=EI\\  
 :mr=\ED:nd=E=:pb#9600:rp=0.2*\Er%.%+:se=\Ed\Ee:sf=~J:so=\EE\xD:\\  
 :ta=8\t:t:te=Ev \_200\200\200\200\200\200\Ep\r\n:\\  
 :ti=\EU\Ev _8p\Ep\r:ue=\EG:ul:up=E;:us=\EG:\\  
 :vb=\Ek\200\200\200\200\200\200\200\200\200\200\200\200\200\200\200\EK:\\  
 :ve=Ew:vs=EW:vt#8:xn:\\  
 :bs:cr=~M:DC#9:dT#8:nl=~J:ta=~I:pt:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability (here between the last field on a line and the first field on the next). Comments may be included on lines beginning with "#".

Types of Capabilities

Capabilities in `termcap` are of three types: Boolean capabilities, which indicate particular features that the

terminal has; numeric capabilities, giving the size of the display or the size of other attributes; and string capabilities, which give character sequences that can be used to perform particular terminal operations. All capabilities have two-letter codes. For instance, the fact that the Concept has *automatic margins* (an automatic return and linefeed when the end of a line is reached) is indicated by the Boolean capability **am**. Hence the description of the Concept includes **am**.

Numeric capabilities are followed by the character '#' then the value. In the example above **co**, which indicates the number of columns the display has, gives the value '80' for the Concept.

Finally, string-valued capabilities, such as **ce** (clear-to-end-of-line sequence) are given by the two-letter code, an '=', then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, which causes padding characters to be supplied by **tputs** after the remainder of the string is sent to provide this delay. The delay can be either a number, such as '20', or a number followed by an '*', such as '3*'. An '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-line padding required. (In the case of insert-character, the factor is still the number of *lines* affected; this is always 1 unless the terminal has **in** and the software uses it.) When an '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per line to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string-valued capabilities for easy encoding of control characters there. **\E** maps to an ESC character, **\X** maps to a control-X for any appropriate X, and the sequences **\n** **\r** **\b** **\f** map to linefeed, return, tab, backspace, and formfeed, respectively. Finally, characters may be given as three octal digits after a ****, and the characters **^** and **** may be given as **\^** and ****. If it is necessary to place a **:** in a capability it must be escaped in octal as **\072**. If it is necessary to place a NUL character in a string capability it must be encoded as **\200**. (The routines that deal with **termcap** use C strings and strip the high bits of the output very late, so that a **\200** comes out as a **\000** would.)

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the first **cr** and **ta** in the example above.

Preparing Descriptions

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in **termcap** and to build up a description gradually, using partial descriptions with **vi** to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the **termcap** file to describe it or bugs in **vi**. To easily test a new terminal description you are working on you can put it in your home directory in a file called **.termcap** and programs will look there before looking in **/usr/share/misc/termcap**. You can also set the environment variable **TERMPATH** to a list of absolute file pathnames (separated by spaces or colons), one of which contains the description you are working on, and programs will search them in the order listed, and nowhere else. See **termcap(3)**. The **TERMCAP** environment variable is usually set to the **termcap** entry itself to avoid reading files when starting up a program.

To get the padding for insert-line right (if the terminal manufacturer did not document it), a severe test is to use **vi** to edit **/etc/passwd** at 9600 baud, delete roughly 16 lines from the middle of the screen, then hit the 'u' key several times quickly. If the display messes up, more padding is usually needed. A similar test can be used for insert-character.

Basic Capabilities

The number of columns on each line of the display is given by the **co** numeric capability. If the display is a CRT, then the number of lines on the screen is given by the **li** capability. If the display wraps around to the beginning of the next line when the cursor reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, the code to do this is given by the **cl** string capability. If the terminal overstrikes (rather than clearing the position when a character is overwritten), it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope

terminals, such as the Tektronix 4010 series, as well as to hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage-return, **^M**.) If there is a code to produce an audible signal (bell, beep, etc.), give this as **bl**.

If there is a code (such as backspace) to move the cursor one position to the left, that capability should be given as **le**. Similarly, codes to move to the right, up, and down should be given as **nd**, **up**, and **do**, respectively. These *local cursor motions* should not alter the text they pass over; for example, you would not normally use **nd=** unless the terminal has the **os** capability, because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in **termcap** have undefined behavior at the left and top edges of a CRT display. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up off the top using local cursor motions.

In order to scroll text up, a program goes to the bottom left corner of the screen and sends the **sf** (index) string. To scroll text down, a program goes to the top left corner of the screen and sends the **sr** (reverse index) string. The strings **sf** and **sr** have undefined behavior when not on their respective corners of the screen. Parameterized versions of the scrolling sequences are **SF** and **SR**, which have the same semantics as **sf** and **sr** except that they take one parameter and scroll that many lines. They also have undefined behavior except at the appropriate corner of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output there, but this does not necessarily apply to **nd** from the last column. Leftward local motion is defined from the left edge only when **bw** is given; then an **le** from the left edge will move to the right edge of the previous row. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch-selectable automatic margins, the **termcap** description usually assumes that this feature is on, *i.e.*, **am**. If the terminal has a command that moves to the first column of the next line, that command can be given as **nw** (newline). It is permissible for this to clear the remainder of the current line, so if the terminal has no correctly-working CR and LF it may still be possible to craft a working **nw** out of one or both of them.

These capabilities suffice to describe hardcopy and "glass-ty" terminals. Thus the Teletype model 33 is described as

```
T3 | tty33 | 33 | tty | Teletype model 33:\
      :bl=^G:co#72:cr=^M:do=^J:hc:os:
```

and the Lear Siegler ADM-3 is described as

```
l3 | adm3 | 3 | LSI ADM-3:\
      :am:bl=^G:cl=^Z:co#80:cr=^M:do=^J:le=^H:li#24:sf=^J:
```

Parameterized Strings

Cursor addressing and other strings requiring parameters are described by a parameterized string capability, with **printf(3)**-like escapes **%x** in it, while other characters are passed through unchanged. For example, to address the cursor the **cm** capability is given, using two parameters: the row and column to move to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory. If the terminal has memory-relative cursor addressing, that can be indicated by an analogous **CM** capability.)

The **%** encodings have the following meanings:

```
%%      output '%'
%d      output value as in printf %d
%2      output value as in printf %2d
```

%3 output value as in printf %3d
 % output value as in printf %c
 %+x add x to value, then do %.
 %>xy if value > x then add y, no output
 %r reverse order of two parameters, no output
 %i increment by one, no output
 %n exclusive-or all parameters with 0140 (Datamedia 2500)
 %B BCD (16*(value/10)) + (value%10), no output
 %D Reverse coding (value - 2*(value%16)), no output (Delta Data).

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent “\E&a12c03Y” padded for 6 milliseconds. Note that the order of the row and column coordinates is reversed here and that the row and column are sent as two-digit integers. Thus its **cm** capability is “cm=6\E&r%2c%2Y”.

The Datamedia 2500 needs the current row and column sent encoded in binary using “%.”. Terminals that use “%.” need to be able to backspace the cursor (**le**) and to move the cursor up one line on the screen (**up**). This is necessary because it is not always safe to transmit **\n**, **\D**, and **\r**, as the system may change or discard them. (Programs using **termcap** must set terminal modes so that tabs are not expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the Lear Siegler ADM-3a, which offsets row and column by a blank character, thus “cm=\E=%+ %+”.

Row or column absolute cursor addressing can be given as single parameter capabilities **ch** (horizontal position absolute) and **cv** (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645) and can be used in preference to **cm**. If there are parameterized local motions (e.g., move *n* positions to the right) these can be given as **DO**, **LE**, **RI**, and **UP** with a single parameter indicating how many positions to move. These are primarily useful if the terminal does not have **cm**, such as the Tektronix 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to the very upper left corner of the screen), this can be given as **ho**. Similarly, a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **up** from the home position, but a program should never do this itself (unless **ll** does), because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as cursor address (0,0): to the top left corner of the screen, not of memory. (Therefore, the “\EH” sequence on Hewlett-Packard terminals cannot be used for **ho**.)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, this should be given as **cd**. **cd** must only be invoked from the first column of a line. (Therefore, it can be simulated by a request to delete a large number of lines, if a true **cd** is not available.)

Insert/Delete Line

If the terminal can open a new blank line before the line containing the cursor, this should be given as **al**; this must be invoked only from the first position of a line. The cursor must then appear at the left of the newly blank line. If the terminal can delete the line that the cursor is on, this should be given as **dl**; this must only be used from the first position on the line to be deleted. Versions of **al** and **dl** which take a single parameter and insert or delete that many lines can be given as **AL** and **DL**. If the terminal has a settable scrolling region (like the VT100), the command to set this can be described with the **cs** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command — the **sc** and **rc**

(save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **sr** or **sf** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory which all commands affect, it should be given as the parameterized string **wi**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order. (This **terminfo** capability is described for completeness. It is unlikely that any **termcap**-using program will support it.)

If the terminal can retain display memory above the screen, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **sr** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character that can be described using **termcap**. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept-100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, then you have the second type of terminal and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line vs. multi-line insert mode, and special treatment of untyped spaces), we have seen no terminals whose insert mode cannot be described with the single attribute.

Termcap can describe both terminals that have an insert mode and terminals that send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode. Give as **ei** the sequence to leave insert mode. Now give as **ic** any sequence that needs to be sent just before each character to be inserted. Most terminals with a true insert mode will not give **ic**; terminals that use a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ic**. Do not give both unless the terminal actually requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence that may need to be sent after insertion of a single character can also be given in **ip**. If your terminal needs to be placed into an "insert mode" and needs a special code preceding each inserted character, then both **im/ ei** and **ic** can be given, and both will be used. The **IC** capability, with one parameter *n*, will repeat the effects of **ic** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode, you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify **dc** to delete a single character, **DC** with one parameter *n* to delete *n* characters, and delete mode by giving **dm** and **ed** to enter and exit delete mode (which is any mode the terminal needs to be placed in for **dc** to work).

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good high-contrast, easy-on-the-eyes format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are

given as **so** and **se**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces or garbage characters on the screen, as the TVI 912 and Teleray 1061 do, then **sg** should be given to tell how many characters are left.

Codes to begin underlining and end underlining can be given as **us** and **ue**, respectively. Underline mode change garbage is specified by **ug**, similar to **sg**. If the terminal has a code to underline the current character and move the cursor one position to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **mb** (blinking), **md** (bold or extra bright), **mh** (dim or half-bright), **mk** (blanking or invisible text), **mp** (protected), **mr** (reverse video), **me** (turn off *all* attribute modes), **as** (enter alternate character set mode), and **ae** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of mode, this should be given as **sa** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attributes is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, and alternate character set. Not all modes need be supported by **sa**, only those for which corresponding attribute commands exist. (It is unlikely that a **termcap**-using program will support this capability, which is defined for compatibility with **terminfo**.)

Terminals with the "magic cookie" glitches (**sg** and **ug**), rather than maintaining extra attribute bits for each character cell, instead deposit special "cookies", or "garbage characters", when they receive mode-setting sequences, which affect the display algorithm.

Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or when the cursor is addressed. Programs using standout mode should exit standout mode on such terminals before moving the cursor or sending a newline. On terminals where this is not a problem, the **ms** capability should be present to say that this overhead is unnecessary.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), this can be given as **vb**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to change, for example, a non-blinking underline into an easier-to-find block or blinking underline), give this sequence as **vs**. If there is a way to make the cursor completely invisible, give that as **vi**. The capability **ve**, which undoes the effects of both of these modes, should also be given.

If your terminal correctly displays underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local mode (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left-arrow, right-arrow, up-arrow, down-arrow, and home keys can be given as **kl**, **kr**, **ku**, **kd**, and **kh**, respectively. If there are function keys such as **f0**, **f1**, ..., **f9**, the codes they send can be given as **k0**, ..., **k9**. If these keys have labels other than the default **f0** through **f9**, the labels can be given as **l0**, ..., **l9**. The codes transmitted by certain other special keys can be given: **kh** (home down), **kb** (backspace), **ka** (clear all tabs), **kt** (clear the tab stop in this column), **kC** (clear screen or erase), **kD** (delete character), **kL** (delete line), **kM** (exit insert mode), **kE** (clear to end of line), **kS** (clear to end of screen), **kI** (insert character or enter insert mode), **kA** (insert line), **kN** (next page), **kP** (previous page), **kF** (scroll forward/down), **kR** (scroll backward/up), and **kT** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, then the other five keys can be given as **, , , , ,** and these keys are useful when the effects of a 3 by 3 directional pad are needed. The obsolete **o** capability formerly used to describe "other"

function keys has been completely supplanted by the above capabilities.

The **ma** entry is also used to indicate arrow keys on terminals that have single-character arrow keys. It is obsolete but still in use in version 2 of **vi** which must be run on some minicomputers due to memory limitations. This field is redundant with **kl**, **kr**, **ku**, **kd**, and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, and the second character is the corresponding **vi** command. These commands are *h* for **kl**, *j* for **kd**, *k* for **ku**, *l* for **kr**, and *H* for **kh**. For example, the Mime would have "**ma**=**^Hh^Kj^Zk^Xl**" indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the Mime.)

Tabs and Initialization

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory-relative cursor addressing and not screen-relative cursor addressing, a screen-sized window must be fixed into the display for cursor addressing to work properly. This is also used for the Tektronix 4025, where **ti** sets the command character to be the one used by **termcap**.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the **termcap** description. They are normally sent to the terminal by the **tset** program each time the user logs in. They will be printed in the following order: **is**; setting tabs using **ct** and **st**; and finally **if**. (**Terminfo** uses **i1-i2** instead of **is** and runs the program **iP** and prints **i3** after the other initializations.) A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs** and **rf**. These strings are output by the **reset** program, which is used when the terminal gets into a wedged state. (**Terminfo** uses **r1-r3** instead of **rs**.) Commands are normally placed in **rs** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the VT100 into 80-column mode would normally be part of **is**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80-column mode.

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ta** (usually **^I**). A "backtab" command which moves leftward to the previous tab stop can be given as **bt**. By convention, if the terminal driver modes indicate that tab stops are being expanded by the computer rather than being sent to the terminal, programs should not use **ta** or **bt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs that are initially set every *n* positions when the terminal is powered up, then the numeric parameter **it** is given, showing the number of positions between tab stops. This is normally used by the **tset** command to determine whether to set the driver mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the **termcap** description can assume that they are properly set.

If there are commands to set and clear tab stops, they can be given as **ct** (clear all tab stops) and **st** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is** or **if**.

Delays

Certain capabilities control padding in the terminal driver. These are primarily needed by hardcopy terminals and are used by the **tset** program to set terminal driver modes appropriately. Delays embedded in the capabilities **cr**, **sf**, **le**, **ff**, and **ta** will cause the appropriate delay bits to be set in the terminal driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**. For 4.2BSD **tset**, the delays are given as numeric capabilities **dC**, **dN**, **dB**, **dF**, and **dT** instead.

Miscellaneous

If the terminal requires other than a NUL (zero) character as a pad, this can be given as **pc**. Only the first character of the **pc** string is used.

If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, then the capability **hs** should be given. Special strings to go to a position in the status line and to return from the status line can be given as **ts** and **fs**. (**fs** must leave the cursor position in the same place that it was before **ts**. If necessary, the **sc** and **rc** strings can be included in **ts** and **fs** to get this effect.) The capability **ts** takes one parameter, which is the column number of the status line to which the cursor is to be moved. If escape sequences and other special commands such as tab work while in the status line, the flag **es** can be given. A string that turns off the status line (or otherwise erases its contents) should be given as **ds**. The status line is normally assumed to be the same width as the rest of the screen, *i.e.*, **co**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded), then its width in columns can be indicated with the numeric parameter **ws**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually "L").

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters), this can be indicated with the parameterized string **rp**. The first parameter is the character to be repeated and the second is the number of times to repeat it. (This is a **terminfo** feature that is unlikely to be supported by a program that uses **termcap**.)

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **CC**. A prototype command character is chosen which is used in all capabilities. This character is given in the **CC** capability to identify it. The following convention is supported on some UNIX systems: The environment is to be searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced by the character in the environment variable. This use of the **CC** environment variable is a very bad idea, as it conflicts with **make(1)**.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses **xoff/xon** (DC3/DC1) handshaking for flow control, give **xo**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, then this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **mm** and **mo**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. An explicit value of 0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **ps**: print the contents of the screen; **pf**: turn off the printer; and **po**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **pO** takes one parameter and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **pf**, is transparently passed to the printer while **pO** is in effect.

Strings to program function keys can be given as **pk**, **pl**, and **px**. Each of these strings takes two parameters: the function key number to program (from 0 to 9) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal-dependent manner. The differences among the capabilities are that **pk** causes pressing the given key to be the same as the user typing the given string; **pl** causes the string to be executed by the terminal in local mode; and **px** causes the string to be transmitted to the computer. Unfortunately, due to lack of a definition for string parameters in **termcap**, only **terminfo** supports these capabilities.

Glitches and Braindamage

Hazeltine terminals, which do not allow **^~** characters to be displayed, should indicate **hz**.

The **nc** capability, now obsolete, formerly indicated Datamedia terminals, which echo **^r ^n** for carriage return then ignore a following linefeed.

Terminals that ignore a linefeed immediately after an **am** wrap, such as the Concept, should indicate **xn**.

If **ce** is required to get rid of standout (instead of merely writing normal text on top of it), **xs** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a magic cookie, and that to erase standout mode it is necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the **ESC** or **^C** characters, has **xb**, indicating that the **"f1"** key is used for **ESC** and **"f2"** for **^C**. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **x x**.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last*, and the combined length of the entries must not exceed 1024. The capabilities given before **tc** override those in the terminal type invoked by **tc**. A capability can be canceled by placing **xx@** to the left of the **tc** invocation, where **xx** is the capability. For example, the entry

```
hn | 2621-nl:ks@:ke@:tc=2621:
```

defines a **"2621-nl"** that does not have the **ks** or **ke** capabilities, hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

<code>/usr/share/misc/termcap</code>	File containing terminal descriptions.
<code>/usr/share/misc/termcap.db</code>	Hash database file containing terminal descriptions (see <code>cap_mkdb(1)</code>).

SEE ALSO

`ex(1)`, `cap_mkdb(1)`, `more(1)`, `tset(1)`, `ul(1)`, `vi(1)`, `curses(3)`, `printf(3)`, `termcap(3)`, `term(7)`

CAVEATS AND BUGS

The *Note*: **termcap** functions were replaced by **terminfo** in AT&T System V UNIX Release 2.0. The transition will be relatively painless if capabilities flagged as **"obsolete"** are avoided.

Lines and columns are now stored by the kernel as well as in the **termcap** entry. Most programs now use the kernel information primarily; the information in this file is used only if the kernel does not have any information.

vi allows only 256 characters for string capabilities, and the routines in `termLib(3)` do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

Not all programs support all entries.

HISTORY

The `termcap` file format appeared in 3BSD.

4.4BSD

Berkeley Software Distribution Programmer's Reference Manual

4.4BSD is the final release of what may be one of the most significant research projects in the history of computing. When Bell Labs originally released UNIX source code to the R&D community, brilliant researchers wrote their own software and added it to UNIX in a spree of creative anarchy that hasn't been equalled since. The Berkeley Software Distribution became the repository of much of that work.

One outcome of UNIX's distributed authorship was an approach (the UNIX "man pages") that allows individual software contributors to use a standard format for documenting the software they've written. Though sometimes criticized as too terse, now that extended third party tutorials are widely available, this approach starts to look like a brilliant strategy. Nothing beats the ease of lookup of an alphabetical listing. For all their faults, the UNIX "man pages" are among the most useful and widely used documentation for any system!

This volume collects the man pages for the various UNIX programming libraries, plus related information for programmers. It consists of sections 2 through 5 of the online man page collection:

- (2) UNIX System Calls
- (3) C Library Subroutines
- (4) Special Files (device interfaces)
- (5) File Formats

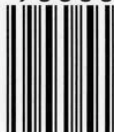
The information in this volume is useful not just for 4.4BSD systems, but also for many related UNIX implementations, including SunOS, BSDI, and Linux.

ISBN 1-56592-078-3



9 781565 920781

90000



Printed on Recycled Paper

5 Volume Set: ISBN 1-56592-082-1
ISBN 1-56592-078-3