The Termcap Manual

The Termcap Library and Data Base
Second Edition
December 1992

Richard M. Stallman

Published by the Free Software Foundation (675 Mass Ave, Cambridge MA 02139). Printed copies are available for $10 each.

Cover art by Etienne Suvasa.

Introduction

Termcap is a library and data base that enables programs to use display terminals in a terminal-independent manner. It originated in Berkeley Unix.

The termcap data base describes the capabilities of hundreds of different display terminals in great detail. Some examples of the information recorded for a terminal could include how many columns wide it is, what string to send to move the cursor to an arbitrary position (including how to encode the row and column numbers), how to scroll the screen up one or several lines, and how much padding is needed for such a scrolling operation.

The termcap library is provided for easy access this data base in programs that want to do terminal-independent character-based display output.

This manual describes the GNU version of the termcap library, which has some extensions over the Unix version. All the extensions are identified as such, so this manual also tells you how to use the Unix termcap.

The GNU version of the termcap library is available free as source code, for use in free programs, and runs on Unix and VMS systems (at least). You can find it in the GNU Emacs distribution in the files '`termcap.c`' and '`tparam.c`'.

This manual was written for the GNU project, whose goal is to develop a complete free operating system upward-compatible with Unix for user programs. The project is approximately two thirds complete. For more information on the GNU project, including the GNU Emacs editor and the mostly-portable optimizing C compiler, send one dollar to

Free Software Foundation
675 Mass Ave
Cambridge, MA 02139

1 The Termcap Library

The termcap library is the application programmer's interface to the termcap data base. It contains functions for the following purposes:

- Finding the description of the user's terminal type (`tgetent`).
- Interrogating the description for information on various topics (`tgetnum`, `tgetflag`, `tgetstr`).
- Computing and performing padding (`tputs`).
- Encoding numeric parameters such as cursor positions into the terminal-specific form required for display commands (`tparam`, `tgoto`).

1.1 Preparing to Use the Termcap Library

To use the termcap library in a program, you need two kinds of preparation:

- The compiler needs declarations of the functions and variables in the library.

  On GNU systems, it suffices to include the header file '`termcap.h`' in each source file that uses these functions and variables.

  On Unix systems, there is often no such header file. Then you must explictly declare the variables as external. You can do likewise for the functions, or let them be implicitly declared and cast their values from type `int` to the appropriate type.

  We illustrate the declarations of the individual termcap library functions with ANSI C prototypes because they show how to pass the arguments. If you are not using the GNU C compiler, you probably cannot use function prototypes, so omit the argument types and names from your declarations.
- The linker needs to search the library. Usually either '`-ltermcap`' or '`-ltermlib`' as an argument when linking will do this.

1.2 Finding a Terminal Description: `tgetent`

An application program that is going to use termcap must first look up the description of the terminal type in use. This is done by calling `tgetent`, whose declaration in ANSI Standard C looks like:

`int tgetent (char *`*buffer*`, char *`*termtype*`);`

This function finds the description and remembers it internally so that you can interrogate it about specific terminal capabilities (see Section 1.3 [Interrogate], page 5).

The argument *termtype* is a string which is the name for the type of terminal to look up. Usually you would obtain this from the environment variable `TERM` using `getenv ("TERM")`.

If you are using the GNU version of termcap, you can alternatively ask `tgetent` to allocate enough space. Pass a null pointer for *buffer*, and `tgetent` itself allocates the storage using `malloc`. There is no way to get the address that was allocated, and you shouldn't try to free the storage.

With the Unix version of termcap, you must allocate space for the description yourself and pass the address of the space as the argument *buffer*. There is no way you can tell how much space is needed, so the convention is to allocate a buffer 2048 characters long and assume that is enough. (Formerly the convention was to allocate 1024 characters and assume that was enough. But one day, for one kind of terminal, that was not enough.)

No matter how the space to store the description has been obtained, termcap records its address internally for use when you later interrogate the description with `tgetnum`, `tgetstr` or `tgetflag`. If the buffer was allocated by termcap, it will be freed by termcap too if you call `tgetent` again. If the buffer was provided by you, you must make sure that its contents remain unchanged for as long as you still plan to interrogate the description.

The return value of `tgetent` is -1 if there is some difficulty accessing the data base of terminal types, 0 if the data base is accessible but the specified type is not defined in it, and some other value otherwise.

Here is how you might use the function `tgetent`:

```
#ifdef unix
static char term_buffer[2048];
#else
#define term_buffer 0
#endif

init_terminal_data ()
{
  char *termtype = getenv ("TERM");
  int success;

  if (termtype == 0)
    fatal ("Specify a terminal type with `setenv TERM <yourtype>'.\n");

  success = tgetent (term_buffer, termtype);
  if (success < 0)
    fatal ("Could not access the termcap data base.\n");
  if (success == 0)
    fatal ("Terminal type `%s' is not defined.\n", termtype);
}
```

Here we assume the function `fatal` prints an error message and exits.

If the environment variable `TERMCAP` is defined, its value is used to override the terminal type data base. The function `tgetent` checks the value of `TERMCAP` automatically. If the value starts with '`/`' then it is taken as a file name to use as the data base file, instead of '`/etc/termcap`' which is the standard data base. If the value does not start with '`/`' then it is itself used as the terminal description, provided that the terminal type *termtype* is among the types it claims to apply to. See Chapter 2 [Data Base], page 17, for information on the format of a terminal description.

1.3 Interrogating the Terminal Description

Each piece of information recorded in a terminal description is called a *capability*. Each defined terminal capability has a two-letter code name and a specific meaning. For example, the number of columns is named '`co`'. See Chapter 3 [Capabilities], page 23, for definitions of all the standard capability names.

Once you have found the proper terminal description with `tgetent` (see Section 1.2 [Find], page 3), your application program must *interrogate* it for various terminal capabilities. You must specify the two-letter code of the capability whose value you seek.

Capability values can be numeric, boolean (capability is either present or absent) or strings. Any particular capability always has the same value type; for example, '`co`' always has a numeric value, while '`am`' (automatic wrap at margin) is always a flag, and '`cm`' (cursor motion command) always has a string value. The documentation of each capability says which type of value it has.

There are three functions to use to get the value of a capability, depending on the type of value the capability has. Here are their declarations in ANSI C:

```
int tgetnum (char *name);
int tgetflag (char *name);
char *tgetstr (char *name, char **area);
```

`tgetnum` Use `tgetnum` to get a capability value that is numeric. The argument *name* is the two-letter code name of the capability. If the capability is present, `tgetnum` returns the numeric value (which is nonnegative). If the capability is not mentioned in the terminal description, `tgetnum` returns −1.

`tgetflag` Use `tgetflag` to get a boolean value. If the capability *name* is present in the terminal description, `tgetflag` returns 1; otherwise, it returns 0.

`tgetstr` Use `tgetstr` to get a string value. It returns a pointer to a string which is the capability value, or a null pointer if the capability is not present in the terminal description.

There are two ways `tgetstr` can find space to store the string value:

- You can ask `tgetstr` to allocate the space. Pass a null pointer for the argument *area*, and `tgetstr` will use `malloc` to allocate storage big enough for the value. Termcap will never free this storage or refer to it again; you should free it when you are finished with it.

  This method is more robust, since there is no need to guess how much space is needed. But it is supported only by the GNU termcap library.

- You can provide the space. Provide for the argument *area* the address of a pointer variable of type `char *`. Before calling `tgetstr`, initialize the variable to point at available space. Then `tgetstr` will store the string value in that space and will increment the pointer variable to point after the space that has been used. You can use the same pointer variable for many calls to `tgetstr`.

  There is no way to determine how much space is needed for a single string, and no way for you to prevent or handle overflow of the area you have provided. However, you can be sure that the total size of all the string values you will obtain from the terminal description is no greater than the size of the description (unless you get the same capability twice). You can determine that size with `strlen` on the buffer you provided to `tgetent`. See below for an example.

  Providing the space yourself is the only method supported by the Unix version of termcap.

Note that you do not have to specify a terminal type or terminal description for the interrogation functions. They automatically use the description found by the most recent call to `tgetent`.

Here is an example of interrogating a terminal description for various capabilities, with conditionals to select between the Unix and GNU methods of providing buffer space.

```
char *tgetstr ();

char *cl_string, *cm_string;
int height;
int width;
int auto_wrap;
```

```
char PC;   /* For tputs.  */
char *BC;  /* For tgoto.  */
char *UP;

interrogate_terminal ()
{
#ifdef UNIX
  /* Here we assume that an explicit term_buffer
     was provided to tgetent.  */
  char *buffer
    = (char *) malloc (strlen (term_buffer));
#define BUFFADDR &buffer
#else
#define BUFFADDR 0
#endif

  char *temp;

  /* Extract information we will use.  */
  cl_string = tgetstr ("cl", BUFFADDR);
  cm_string = tgetstr ("cm", BUFFADDR);
  auto_wrap = tgetflag ("am");
  height = tgetnum ("li");
  width = tgetnum ("co");

  /* Extract information that termcap functions use.  */
  temp = tgetstr ("pc", BUFFADDR);
  PC = temp ? *temp : 0;
  BC = tgetstr ("le", BUFFADDR);
  UP = tgetstr ("up", BUFFADDR);
}
```

See Section 1.5 [Padding], page 8, for information on the variable `PC`. See Section 1.6.2 [Using Parameters], page 15, for information on `UP` and `BC`.

1.4 Initialization for Use of Termcap

Before starting to output commands to a terminal using termcap, an application program should do two things:

- Initialize various global variables which termcap library output functions refer to. These include `PC` and `ospeed` for padding (see Section 1.5.4 [Output Padding], page 10) and `UP` and `BC` for cursor motion (see Section 1.6.2.2 [tgoto], page 16).

- Tell the kernel to turn off alteration and padding of horizontal-tab characters sent to the terminal.

To turn off output processing in Berkeley Unix you would use `ioctl` with code `TIOCLSET` to set the bit named `LLITOUT`, and clear the bits `ANYDELAY` using `TIOCSETN`. In POSIX or System V, you must clear the bit named `OPOST`. Refer to the system documentation for details.

If you do not set the terminal flags properly, some older terminals will not work. This is because their commands may contain the characters that normally signify newline, carriage return and horizontal tab—characters which the kernel thinks it ought to modify before output.

When you change the kernel's terminal flags, you must arrange to restore them to their normal state when your program exits. This implies that the program must catch fatal signals such as `SIGQUIT` and `SIGINT` and restore the old terminal flags before actually terminating.

Modern terminals' commands do not use these special characters, so if you do not care about problems with old terminals, you can leave the kernel's terminal flags unaltered.

1.5 Padding

Padding means outputting null characters following a terminal display command that takes a long time to execute. The terminal description says which commands require padding and how much; the function `tputs`, described below, outputs a terminal command while extracting from it the padding information, and then outputs the padding that is necessary.

1.5.1 Why Pad, and How

Most types of terminal have commands that take longer to execute than they do to send over a high-speed line. For example, clearing the screen may take 20msec once the entire command is received. During that time, on a 9600 bps line, the terminal could receive about 20 additional output characters while still busy clearing the screen. Every terminal has a certain amount of buffering capacity to remember output characters that cannot be processed yet, but too many slow commands in a row can cause the buffer to fill up. Then any additional output that cannot be processed immediately will be lost.

To avoid this problem, we normally follow each display command with enough useless charaters (usually null characters) to fill up the time that the display command needs to execute. This does the job if the terminal throws away null characters without using up space in the buffer (which most terminals do). If enough padding is used, no output can ever be lost. The right amount of padding avoids loss of output without slowing down operation, since the time used to transmit padding is time that nothing else could be done.

The number of padding characters needed for an operation depends on the line speed. In fact, it is proportional to the line speed. A 9600 baud line transmits about one character per msec, so the clear screen command in the example above would need about 20 characters of padding. At 1200 baud, however, only about 3 characters of padding are needed to fill up 20msec.

1.5.2 When There Is Not Enough Padding

There are several common manifestations of insufficient padding.

- Emacs displays '`I-search: ^Q-`' at the bottom of the screen.

  This means that the terminal thought its buffer was getting full of display commands, so it tried to tell the computer to stop sending any.

- The screen is garbled intermittently, or the details of garbling vary when you repeat the action. (A garbled screen could be due to a command which is simply incorrect, or to user option in the terminal which doesn't match the assumptions of the terminal description, but this usually leads to reproducible failure.)

  This means that the buffer did get full, and some commands were lost. Many changeable factors can change which ones are lost.

- Screen is garbled at high output speeds but not at low speeds. Padding problems nearly always go away at low speeds, usually even at 1200 baud.

  This means that a high enough speed permits commands to arrive faster than they can be executed.

Although any obscure command on an obscure terminal might lack padding, in practice problems arise most often from the clearing commands '`cl`' and '`cd`' (see Section 3.7 [Clearing], page 33), the scrolling commands '`sf`' and '`sr`' (see Section 3.5 [Scrolling], page 30), and the line insert/delete commands '`al`' and '`dl`' (see Section 3.8 [Insdel Line], page 34).

Occasionally the terminal description fails to define '`sf`' and some programs will use '`do`' instead, so you may get a problem with '`do`'. If so, first define '`sf`' just like '`do`', then add some padding to '`sf`'.

The best strategy is to add a lot of padding at first, perhaps 200 msec. This is much more than enough; in fact, it should cause a visible slowdown. (If you don't see a slowdown, the change has not taken effect; see Section 2.5 [Changing], page 21.) If this makes the problem go away, you have found the right place to add padding; now reduce the amount until the problem comes back, then increase it again. If the problem remains, either it is in some other capability or it is not a matter of padding at all.

Keep in mind that on many terminals the correct padding for insert/delete line or for scrolling is cursor-position dependent. If you get problems from scrolling a large region of the screen but not from scrolling a small part (just a few lines moving), it may mean that fixed padding should be replaced with position-dependent padding.

1.5.3 Specifying Padding in a Terminal Description

In the terminal description, the amount of padding required by each display command is recorded as a sequence of digits at the front of the command. These digits specify the padding time in milliseconds (msec). They can be followed optionally by a decimal point and one more digit, which is a number of tenths of msec.

Sometimes the padding needed by a command depends on the cursor position. For example, the time taken by an "insert line" command is usually proportional to the number of lines that need to be moved down or cleared. An asterisk ('`*`') following the padding time says that the time should be multiplied by the number of screen lines affected by the command.

```
:al=1.3*\E[L:
```

is used to describe the "insert line" command for a certain terminal. The padding required is 1.3 msec per line affected. The command itself is '⟨ESC⟩ `[ L`'.

The padding time specified in this way tells `tputs` how many pad characters to output. See Section 1.5.4 [Output Padding], page 10.

Two special capability values affect padding for all commands. These are the '`pc`' and '`pb`'. The variable '`pc`' specifies the character to pad with, and '`pb`' the speed below which no padding is needed. The defaults for these variables, a null character and 0, are correct for most terminals. See Section 3.17 [Pad Specs], page 48.

1.5.4 Performing Padding with `tputs`

Use the termcap function `tputs` to output a string containing an optional padding spec of the form described above (see Section 1.5.3 [Describe Padding], page 10). The function `tputs` strips off and decodes the padding spec, outputs the rest of the string, and then outputs the appropriate padding. Here is its declaration in ANSI C:

```
char PC;
short ospeed;

int tputs (char *string, int nlines, int (*outfun) ());
```

Here *string* is the string (including padding spec) to be output; *nlines* is the number of lines affected by the operation, which is used to multiply the amount of padding if the padding spec ends with a '*'. Finally, *outfun* is a function (such as `fputchar`) that is called to output each character. When actually called, *outfun* should expect one argument, a character.

The operation of `tputs` is controlled by two global variables, `ospeed` and `PC`. The value of `ospeed` is supposed to be the terminal output speed, encoded as in the `ioctl` system call which gets the speed information. This is needed to compute the number of padding characters. The value of `PC` is the character used for padding.

You are responsible for storing suitable values into these variables before using `tputs`. The value stored into the `PC` variable should be taken from the '`pc`' capability in the terminal description (see Section 3.17 [Pad Specs], page 48). Store zero in `PC` if there is no '`pc`' capability.

The argument *nlines* requires some thought. Normally, it should be the number of lines whose contents will be cleared or moved by the command. For cursor motion commands, or commands that do editing within one line, use the value 1. For most commands that affect multiple lines, such as '`al`' (insert a line) and '`cd`' (clear from the cursor to the end of the screen), *nlines* should be the screen height minus the current vertical position (origin 0). For multiple insert and scroll commands such as '`AL`' (insert multiple lines), that same value for *nlines* is correct; the number of lines being inserted is *not* correct.

If a "scroll window" feature is used to reduce the number of lines affected by a command, the value of *nlines* should take this into account. This is because the delay time required depends on how much work the terminal has to do, and the scroll window feature reduces the work. See Section 3.5 [Scrolling], page 30.

Commands such as '`ic`' and '`dc`' (insert or delete characters) are problematical because the padding needed by these commands is proportional to the number of characters affected, which is the number of columns from the cursor to the end of the line. It would be nice to have a way to specify such a dependence, and there is no need for dependence on vertical position in these commands, so it is an obvious idea to say that for these commands *nlines* should really be the number of columns affected. However, the definition of termcap clearly says that *nlines* is always the number of lines affected, even in this case, where it is always 1. It is not easy to change this rule now, because too many programs and terminal descriptions have been written to follow it.

Because *nlines* is always 1 for the '`ic`' and '`dc`' strings, there is no reason for them to use '*', but some of them do. These should be corrected by deleting the '*'. If, some day, such entries have disappeared, it may be possible to change to a more useful convention for the *nlines* argument for these operations without breaking any programs.

1.6 Filling In Parameters

Some terminal control strings require numeric *parameters*. For example, when you move the cursor, you need to say what horizontal and vertical positions to move it to. The value of the terminal's '`cm`' capability, which says how to move the cursor, cannot simply be a string of characters; it must say how to express the cursor position numbers and where to put them within the command.

The specifications of termcap include conventions as to which string-valued capabilities require parameters, how many parameters, and what the parameters mean; for example, it defines the '`cm`' string to take two parameters, the vertical and horizontal positions, with 0,0 being the upper left corner. These conventions are described where the individual commands are documented.

Termcap also defines a language used within the capability definition for specifying how and where to encode the parameters for output. This language uses character sequences starting with '`%`'. (This is the same idea as `printf`, but the details are different.) The language for parameter encoding is described in this section.

A program that is doing display output calls the functions `tparam` or `tgoto` to encode parameters according to the specifications. These functions produce a string containing the actual commands to be output (as well a padding spec which must be processed with `tputs`; see Section 1.5 [Padding], page 8).

1.6.1 Describing the Encoding

A terminal command string that requires parameters contains special character sequences starting with '`%`' to say how to encode the parameters. These sequences control the actions of `tparam` and `tgoto`.

The parameters values passed to `tparam` or `tgoto` are considered to form a vector. A pointer into this vector determines the next parameter to be processed. Some of the '`%`'-sequences encode one parameter and advance the pointer to the next parameter. Other '`%`'-sequences alter the pointer or alter the parameter values without generating output.

For example, the '`cm`' string for a standard ANSI terminal is written as '`\E[%i%d;%dH`'. ('`\E`' stands for ⟨ESC⟩.) '`cm`' by convention always requires two parameters, the vertical and horizontal goal positions, so this string specifies the encoding of two parameters. Here '`%i`' increments the two values supplied, and each '`%d`' encodes one of the values in decimal. If the cursor position values 20,58 are encoded with this string, the result is '`\E[21;59H`'.

First, here are the '`%`'-sequences that generate output. Except for '`%%`', each of them encodes one parameter and advances the pointer to the following parameter.

'`%%`' Output a single '`%`'. This is the only way to represent a literal '`%`' in a terminal command with parameters. '`%%`' does not use up a parameter.

'`%d`' As in `printf`, output the next parameter in decimal.

'`%2`' Like '`%02d`' in `printf`: output the next parameter in decimal, and always use at least two digits.

'`%3`' Like '`%03d`' in `printf`: output the next parameter in decimal, and always use at least three digits. Note that '`%4`' and so on are *not* defined.

'`%.`' Output the next parameter as a single character whose ASCII code is the parameter value. Like '`%c`' in `printf`.

'`%+`*char*' Add the next parameter to the character *char*, and output the resulting character. For example, '`%+ `' represents 0 as a space, 1 as '`!`', etc.

The following '`%`'-sequences specify alteration of the parameters (their values, or their order) rather than encoding a parameter for output. They generate no output; they are used only for their side effects on the parameters. Also, they do not advance the "next parameter" pointer except as explicitly stated. Only '`%i`', '`%r`' and '`%>`' are defined in standard Unix termcap. The others are GNU extensions.

'`%i`' Increment the next two parameters. This is used for terminals that expect cursor positions in origin 1. For example, '`%i%d,%d`' would output two parameters with '`1`' for 0, '`2`' for 1, etc.

'`%r`' Interchange the next two parameters. This is used for terminals whose cursor positioning command expects the horizontal position first.

'`%s`' Skip the next parameter. Do not output anything.

'`%b`' Back up one parameter. The last parameter used will become once again the next parameter to be output, and the next output command will use it. Using '`%b`' more than once, you can back up any number of parameters, and you can refer to each parameter any number of times.

'`%>`*c1c2*' Conditionally increment the next parameter. Here *c1* and *c2* are characters which stand for their ASCII codes as numbers. If the next parameter is greater than the ASCII code of *c1*, the ASCII code of *c2* is added to it.

'`%a` *op type pos*'
Perform arithmetic on the next parameter, do not use it up, and do not output anything. Here *op* specifies the arithmetic operation, while *type* and *pos* together specify the other operand.

Spaces are used above to separate the operands for clarity; the spaces don't appear in the data base, where this sequence is exactly five characters long.

The character *op* says what kind of arithmetic operation to perform. It can be any of these characters:

'`=`' assign a value to the next parameter, ignoring its old value. The new value comes from the other operand.

'`+`' add the other operand to the next parameter.

'`-`' subtract the other operand from the next parameter.

'`*`' multiply the next parameter by the other operand.

'`/`' divide the next parameter by the other operand.

The "other operand" may be another parameter's value or a constant; the character *type* says which. It can be:

'`p`' Use another parameter. The character *pos* says which parameter to use. Subtract 64 from its ASCII code to get the position of the desired parameter relative to this one. Thus, the character '`A`' as *pos* means the parameter after the next one; the character '`?`' means the parameter before the next one.

'`c`' Use a constant value. The character *pos* specifies the value of the constant. The 0200 bit is cleared out, so that 0200 can be used to represent zero.

The following '`%`'-sequences are special purpose hacks to compensate for the weird designs of obscure terminals. They modify the next parameter or the next two parameters but do not generate output and do not use up any parameters. '`%m`' is a GNU extension; the others are defined in standard Unix termcap.

'`%n`' Exclusive-or the next parameter with 0140, and likewise the parameter after next.

'`%m`' Complement all the bits of the next parameter and the parameter after next.

'`%B`' Encode the next parameter in BCD. It alters the value of the parameter by adding six times the quotient of the parameter by ten. Here is a C statement that shows how the new value is computed:

```
parm = (parm / 10) * 16 + parm % 10;
```

'`%D`' Transform the next parameter as needed by Delta Data terminals. This involves subtracting twice the remainder of the parameter by 16.

```
parm -= 2 * (parm % 16);
```

1.6.2 Sending Display Commands with Parameters

The termcap library functions `tparam` and `tgoto` serve as the analog of `printf` for terminal string parameters. The newer function `tparam` is a GNU extension, more general but missing from Unix termcap. The original parameter-encoding function is `tgoto`, which is preferable for cursor motion.

1.6.2.1 `tparam`

The function `tparam` can encode display commands with any number of parameters and allows you to specify the buffer space. It is the preferred function for encoding parameters for all but the '`cm`' capability. Its ANSI C declaration is as follows:

```
char *tparam (char *ctlstring, char *buffer, int size, int parm1,...)
```

The arguments are a control string *ctlstring* (the value of a terminal capability, presumably), an output buffer *buffer* and *size*, and any number of integer parameters to be encoded. The effect of `tparam` is to copy the control string into the buffer, encoding parameters according to the '`%`' sequences in the control string.

You describe the output buffer by its address, *buffer*, and its size in bytes, *size*. If the buffer is not big enough for the data to be stored in it, `tparam` calls `malloc` to get a larger buffer. In either case, `tparam` returns the address of the buffer it ultimately uses. If the value equals *buffer*, your original buffer was used. Otherwise, a new buffer was allocated, and you must free it after you are done with printing the results. If you pass zero for *size* and *buffer*, `tparam` always allocates the space with `malloc`.

All capabilities that require parameters also have the ability to specify padding, so you should use `tputs` to output the string produced by `tparam`. See Section 1.5 [Padding], page 8. Here is an example.

```
{
char *buf;
char buffer[40];

buf = tparam (command, buffer, 40, parm);
tputs (buf, 1, fputchar);
if (buf != buffer)
free (buf);
}
```

If a parameter whose value is zero is encoded with '`%.`'-style encoding, the result is a null character, which will confuse `tputs`. This would be a serious problem, but luckily '`%.`' encoding is used only by a few old models of terminal, and only for the '`cm`' capability. To solve the problem, use `tgoto` rather than `tparam` to encode the '`cm`' capability.

1.6.2.2 tgoto

The special case of cursor motion is handled by `tgoto`. There are two reasons why you might choose to use `tgoto`:

- For Unix compatibility, because Unix termcap does not have `tparam`.
- For the '`cm`' capability, since `tgoto` has a special feature to avoid problems with null characters, tabs and newlines on certain old terminal types that use '`%.`' encoding for that capability.

Here is how `tgoto` might be declared in ANSI C:

`char *tgoto (char *`*cstring*`, int `*hpos*`, int `*vpos*`)`

There are three arguments, the terminal description's '`cm`' string and the two cursor position numbers; `tgoto` computes the parametrized string in an internal static buffer and returns the address of that buffer. The next time you use `tgoto` the same buffer will be reused.

Parameters encoded with '`%.`' encoding can generate null characters, tabs or newlines. These might cause trouble: the null character because `tputs` would think that was the end of the string, the tab because the kernel or other software might expand it into spaces, and the newline becaue the kernel might add a carriage-return, or padding characters normally used for a newline. To prevent such problems, `tgoto` is careful to avoid these characters. Here is how this works: if the target cursor position value is such as to cause a problem (that is to say, zero, nine or ten), `tgoto` increments it by one, then compensates by appending a string to move the cursor back or up one position.

The compensation strings to use for moving back or up are found in global variables named `BC` and `UP`. These are actual external C variables with upper case names; they are declared `char *`. It is up to you to store suitable values in them, normally obtained from the '`le`' and '`up`' terminal capabilities in the terminal description with `tgetstr`. Alternatively, if these two variables are both zero, the feature of avoiding nulls, tabs and newlines is turned off.

It is safe to use `tgoto` for commands other than '`cm`' only if you have stored zero in `BC` and `UP`.

Note that `tgoto` reverses the order of its operands: the horizontal position comes before the vertical position in the arguments to `tgoto`, even though the vertical position comes before the horizontal in the parameters of the '`cm`' string. If you use `tgoto` with a command such as '`AL`' that takes one parameter, you must pass the parameter to `tgoto` as the "vertical position".

2 The Format of the Data Base

The termcap data base of terminal descriptions is stored in the file '/etc/termcap'. It contains terminal descriptions, blank lines, and comments.

A terminal description starts with one or more names for the terminal type. The information in the description is a series of *capability names* and values. The capability names have standard meanings (see Chapter 3 [Capabilities], page 23) and their values describe the terminal.

2.1 Terminal Description Format

Aside from comments (lines starting with '#', which are ignored), each nonblank line in the termcap data base is a terminal description. A terminal description is nominally a single line, but it can be split into multiple lines by inserting the two characters '\ newline'. This sequence is ignored wherever it appears in a description.

The preferred way to split the description is between capabilities: insert the four characters ': \ newline tab' immediately before any colon. This allows each sub-line to start with some indentation. This works because, after the '\ newline' are ignored, the result is ': tab :'; the first colon ends the preceding capability and the second colon starts the next capability. If you split with '\ newline' alone, you may not add any indentation after them.

Here is a real example of a terminal description:

```
dw|vt52|DEC vt52:\
        :cr=^M:do=^J:nl=^J:bl=^G:\
        :le=^H:bs:cd=\EJ:ce=\EK:cl=\EH\EJ:\
        :cm=\EY%+ %+ :co#80:li#24:\
        :nd=\EC:ta=^I:pt:sr=\EI:up=\EA:\
        :ku=\EA:kd=\EB:kr=\EC:kl=\ED:kb=^H:
```

Each terminal description begins with several names for the terminal type. The names are separated by '|' characters, and a colon ends the last name. The first name should be two characters long; it exists only for the sake of very old Unix systems and is never used in modern systems. The last name should be a fully verbose name such as "DEC vt52" or "Ann Arbor Ambassador with 48 lines". The other names should include whatever the user ought to be able to specify to get this terminal type, such as 'vt52' or 'aaa-48'. See Section 2.3 [Naming], page 19, for information on how to choose terminal type names.

After the terminal type names come the terminal capabilities, separated by colons and with a colon after the last one. Each capability has a two-letter name, such as 'cm' for "cursor motion string" or 'li' for "number of display lines".

2.2 Writing the Capabilities

There are three kinds of capabilities: flags, numbers, and strings. Each kind has its own way of being written in the description. Each defined capability has by convention a particular kind of value; for example, 'li' always has a numeric value and 'cm' always a string value.

A flag capability is thought of as having a boolean value: the value is true if the capability is present, false if not. When the capability is present, just write its name between two colons.

A numeric capability has a value which is a nonnegative number. Write the capability name, a '#', and the number, between two colons. For example, '...:li#48:...' is how you specify the 'li' capability for 48 lines.

A string-valued capability has a value which is a sequence of characters. Usually these are the characters used to perform some display operation. Write the capability name, a '=', and the characters of the value, between two colons. For example, '...:cm=\E[%i%d;%dH:...' is how the cursor motion command for a standard ANSI terminal would be specified.

Special characters in the string value can be expressed using '\'-escape sequences as in C; in addition, '\E' stands for ⟨ESC⟩. '^' is also a kind of escape character; '^' followed by *char* stands for the control-equivalent of *char*. Thus, '^a' stands for the character control-a, just like '\001'. '\' and '^' themselves can be represented as '\\' and '\^'.

To include a colon in the string, you must write '\072'. You might ask, "Why can't '\:' be used to represent a colon?" The reason is that the interrogation functions do not count slashes while looking for a capability. Even if ':ce=ab\:cd:' were interpreted as giving the 'ce' capability the value 'ab:cd', it would also appear to define 'cd' as a flag.

The string value will often contain digits at the front to specify padding (see Section 1.5 [Padding], page 8) and/or '%'-sequences within to specify how to encode parameters (see Section 1.6 [Parameters], page 12). Although these things are not to be output literally to the terminal, they are considered part of the value of the capability. They are special only when the string value is processed by tputs, tparam or tgoto. By contrast, '\' and '^' are considered part of the syntax for specifying the characters in the string.

Let's look at the VT52 example again:

```
dw|vt52|DEC vt52:\
        :cr=^M:do=^J:nl=^J:bl=^G:\
        :le=^H:bs:cd=\EJ:ce=\EK:cl=\EH\EJ:\
        :cm=\EY%+ %+ :co#80:li#24:\
        :nd=\EC:ta=^I:pt:sr=\EI:up=\EA:\
```

```
:ku=\EA:kd=\EB:kr=\EC:kl=\ED:kb=^H:
```

Here we see the numeric-valued capabilities 'co' and 'li', the flags 'bs' and 'pt', and many string-valued capabilities. Most of the strings start with ⟨ESC⟩ represented as '\E'. The rest contain control characters represented using '^'. The meanings of the individual capabilities are defined elsewhere (see Chapter 3 [Capabilities], page 23).

2.3 Terminal Type Name Conventions

There are conventions for choosing names of terminal types. For one thing, all letters should be in lower case. The terminal type for a terminal in its most usual or most fundamental mode of operation should not have a hyphen in it.

If the same terminal has other modes of operation which require different terminal descriptions, these variant descriptions are given names made by adding suffixes with hyphens. Such alternate descriptions are used for two reasons:

- When the terminal has a switch that changes its behavior. Since the computer cannot tell how the switch is set, the user must tell the computer by choosing the appropriate terminal type name.

  For example, the VT-100 has a setup flag that controls whether the cursor wraps at the right margin. If this flag is set to "wrap", you must use the terminal type 'vt100-am'. Otherwise you must use 'vt100-nam'. Plain 'vt100' is defined as a synonym for either 'vt100-am' or 'vt100-nam' depending on the preferences of the local site.

  The standard suffix '-am' stands for "automatic margins".
- To give the user a choice in how to use the terminal. This is done when the terminal has a switch that the computer normally controls.

  For example, the Ann Arbor Ambassador can be configured with many screen sizes ranging from 20 to 60 lines. Fewer lines make bigger characters but more lines let you see more of what you are editing. As a result, users have different preferences. Therefore, termcap provides terminal types for many screen sizes. If you choose type 'aaa-30', the terminal will be configured to use 30 lines; if you choose 'aaa-48', 48 lines will be used, and so on.

Here is a list of standard suffixes and their conventional meanings:

'-w'
: Short for "wide". This is a mode that gives the terminal more columns than usual. This is normally a user option.

'-am'
: "Automatic margins". This is an alternate description for use when the terminal's margin-wrap switch is on; it contains the 'am' flag. The implication is that normally the switch is off and the usual description for the terminal says that the switch is off.

'`-nam`' "No automatic margins". The opposite of '`-am`', this names an alternative description which lacks the '`am`' flag. This implies that the terminal is normally operated with the margin-wrap switch turned on, and the normal description of the terminal says so.

'`-na`' "No arrows". This terminal description initializes the terminal to keep its arrow keys in local mode. This is a user option.

'`-rv`' "Reverse video". This terminal description causes text output for normal video to appear as reverse, and text output for reverse video to come out as normal. Often this description differs from the usual one by interchanging the two strings which turn reverse video on and off.

This is a user option; you can choose either the "reverse video" variant terminal type or the normal terminal type, and termcap will obey.

'`-s`' "Status". Says to enable use of a status line which ordinary output does not touch (see Section 3.18 [Status Line], page 49).

Some terminals have a special line that is used only as a status line. For these terminals, there is no need for an '`-s`' variant; the status line commands should be defined by default. On other terminals, enabling a status line means removing one screen line from ordinary use and reducing the effective screen height. For these terminals, the user can choose the '`-s`' variant type to request use of a status line.

'`-`*nlines*' Says to operate with *nlines* lines on the screen, for terminals such as the Ambassador which provide this as an option. Normally this is a user option; by choosing the terminal type, you control how many lines termcap will use.

'`-`*npages*`p`' Says that the terminal has *npages* pages worth of screen memory, for terminals where this is a hardware option.

'`-unk`' Says that description is not for direct use, but only for reference in '`tc`' capabilities. Such a description is a kind of subroutine, because it describes the common characteristics of several variant descriptions that would use other suffixes in place of '`-unk`'.

2.4 Inheriting from Related Descriptions

When two terminal descriptions are similar, their identical parts do not need to be given twice. Instead, one of the two can be defined in terms of the other, using the '`tc`' capability. We say that one description *refers to* the other, or *inherits from* the other.

The '`tc`' capability must be the last one in the terminal description, and its value is a string which is the name of another terminal type which is referred to. For example,

```
N9|aaa|ambassador|aaa-30|ann arbor ambassador/30 lines:\
        :ti=\E[2J\E[30;0;0;30p:\
        :te=\E[60;0;0;30p\E[30;1H\E[J:\
        :li#30:tc=aaa-unk:
```

defines the terminal type '`aaa-30`' (also known as plain '`aaa`') in terms of '`aaa-unk`', which defines everything about the Ambassador that is independent of screen height. The types '`aaa-36`', '`aaa-48`' and so on for other screen heights are likewise defined to inherit from '`aaa-unk`'.

The capabilities overridden by '`aaa-30`' include '`li`', which says how many lines there are, and '`ti`' and '`te`', which configure the terminal to use that many lines.

The effective terminal description for type '`aaa`' consists of the text shown above followed by the text of the description of '`aaa-unk`'. The '`tc`' capability is handled automatically by `tgetent`, which finds the description thus referenced and combines the two descriptions (see Section 1.2 [Find], page 3). Therefore, only the implementor of the terminal descriptions needs to think about using '`tc`'. Users and application programmers do not need to be concerned with it.

Since the reference terminal description is used last, capabilities specified in the referring description override any specifications of the same capabilities in the reference description.

The referring description can cancel out a capability without specifying any new value for it by means of a special trick. Write the capability in the referring description, with the character '`@`' after the capability name, as follows:

```
NZ|aaa-30-nam|ann arbor ambassador/30 lines/no automatic-margins:\
        :am@:tc=aaa-30:
```

2.5 When Changes in the Data Base Take Effect

Each application program must read the terminal description from the data base, so a change in the data base is effective for all jobs started after the change is made.

The change will usually have no effect on a job that have been in existence since before the change. The program probably read the terminal description once, when it was started, and is continuing to use what it read then. If the program does not have a feature for reexamining the data base, then you will need to run it again (probably killing the old job).

If the description in use is coming from the `TERMCAP` environment variable, then the data base file is effectively overridden, and changes in it will have no effect until you change the `TERMCAP` variable as well. For example, some users' '`.login`' files automatically copy the terminal description into `TERMCAP` to speed startup of applications. If you have done this, you will need to change the `TERMCAP` variable to make the changed data base take effect.

3 Definitions of the Terminal Capabilities

This section is divided into many subsections, each for one aspect of use of display terminals. For writing a display program, you usually need only check the subsections for the operations you want to use. For writing a terminal description, you must read each subsection and fill in the capabilities described there.

String capabilities that are display commands may require numeric parameters (see Section 1.6 [Parameters], page 12). Most such capabilities do not use parameters. When a capability requires parameters, this is explicitly stated at the beginning of its definition. In simple cases, the first or second sentence of the definition mentions all the parameters, in the order they should be given, using a name in italics for each one. For example, the '`rp`' capability is a command that requires two parameters; its definition begins as follows:

> String of commands to output a graphic character *c*, repeated *n* times.

In complex cases or when there are many parameters, they are described explicitly.

When a capability is described as obsolete, this means that programs should not be written to look for it, but terminal descriptions should still be written to provide it.

When a capability is described as very obsolete, this means that it should be omitted from terminal descriptions as well.

3.1 Basic Characteristics

This section documents the capabilities that describe the basic and nature of the terminal, and also those that are relevant to the output of graphic characters.

'`os`' Flag whose presence means that the terminal can overstrike. This means that outputting a graphic character does not erase whatever was present in the same character position before. The terminals that can overstrike include printing terminals, storage tubes (all obsolete nowadays), and many bit-map displays.

'`eo`' Flag whose presence means that outputting a space erases a character position even if the terminal supports overstriking. If this flag is not present and overstriking is supported, output of a space has no effect except to move the cursor.

(On terminals that do not support overstriking, you can always assume that outputting a space at a position erases whatever character was previously displayed there.)

'gn' Flag whose presence means that this terminal type is a generic type which does not really describe any particular terminal. Generic types are intended for use as the default type assigned when the user connects to the system, with the intention that the user should specify what type he really has. One example of a generic type is the type '`network`'.

Since the generic type cannot say how to do anything interesting with the terminal, termcap-using programs will always find that the terminal is too weak to be supported if the user has failed to specify a real terminal type in place of the generic one. The '`gn`' flag directs these programs to use a different error message: "You have not specified your real terminal type", rather than "Your terminal is not powerful enough to be used".

'hc' Flag whose presence means this is a hardcopy terminal.

'rp' String of commands to output a graphic character *c*, repeated *n* times. The first parameter value is the ASCII code for the desired character, and the second parameter is the number of times to repeat the character. Often this command requires padding proportional to the number of times the character is repeated. This effect can be had by using parameter arithmetic with '`%`'-sequences to compute the amount of padding, then generating the result as a number at the front of the string so that `tputs` will treat it as padding.

'hz' Flag whose presence means that the ASCII character '`~`' cannot be output on this terminal because it is used for display commands.

Programs handle this flag by checking all text to be output and replacing each '`~`' with some other character(s). If this is not done, the screen will be thoroughly garbled.

The old Hazeltine terminals that required such treatment are probably very rare today, so you might as well not bother to support this flag.

'CC' String whose presence means the terminal has a settable command character. The value of the string is the default command character (which is usually ⟨ESC⟩).

All the strings of commands in the terminal description should be written to use the default command character. If you are writing an application program that changes the command character, use the '`CC`' capability to figure out how to translate all the display commands to work with the new command character.

Most programs have no reason to look at the '`CC`' capability.

`xb` Flag whose presence identifies Superbee terminals which are unable to transmit the characters ⟨ESC⟩ and *Control-C*. Programs which support this flag are supposed to check the input for the code sequences sent by the ⟨F1⟩ and ⟨F2⟩ keys, and pretend that ⟨ESC⟩ or *Control-C* (respectively) had been read. But this flag is obsolete, and not worth supporting.

3.2 Screen Size

A terminal description has two capabilities, `co` and `li`, that describe the screen size in columns and lines. But there is more to the question of screen size than this.

On some operating systems the "screen" is really a window and the effective width can vary. On some of these systems, `tgetnum` uses the actual width of the window to decide what value to return for the `co` capability, overriding what is actually written in the terminal description. On other systems, it is up to the application program to check the actual window width using a system call. For example, on BSD 4.3 systems, the system call `ioctl` with code `TIOCGWINSZ` will tell you the current screen size.

On all window systems, termcap is powerless to advise the application program if the user resizes the window. Application programs must deal with this possibility in a system-dependent fashion. On some systems the C shell handles part of the problem by detecting changes in window size and setting the `TERMCAP` environment variable appropriately. This takes care of application programs that are started subsequently. It does not help application programs already running.

On some systems, including BSD 4.3, all programs using a terminal get a signal named `SIGWINCH` whenever the screen size changes. Programs that use termcap should handle this signal by using `ioctl TIOCGWINSZ` to learn the new screen size.

`co` Numeric value, the width of the screen in character positions. Even hardcopy terminals normally have a `co` capability.

`li` Numeric value, the height of the screen in lines.

3.3 Cursor Motion

Termcap assumes that the terminal has a *cursor*, a spot on the screen where a visible mark is displayed, and that most display commands take effect at the position of the cursor. It follows that moving the cursor to a specified location is very important.

There are many terminal capabilities for different cursor motion operations. A terminal description should define as many as possible, but most programs do not need to use most of them. One capability, '`cm`', moves the cursor to an arbitrary place on the screen; this by itself is sufficient for any application as long as there is no need to support hardcopy terminals or certain old, weak displays that have only relative motion commands. Use of other cursor motion capabilities is an optimization, enabling the program to output fewer characters in some common cases.

If you plan to use the relative cursor motion commands in an application program, you must know what the starting cursor position is. To do this, you must keep track of the cursor position and update the records each time anything is output to the terminal, including graphic characters. In addition, it is necessary to know whether the terminal wraps after writing in the rightmost column. See Section 3.4 [Wrapping], page 29.

One other motion capability needs special mention: '`nw`' moves the cursor to the beginning of the following line, perhaps clearing all the starting line after the cursor, or perhaps not clearing at all. This capability is a least common denominator that is probably supported even by terminals that cannot do most other things such as '`cm`' or '`do`'. Even hardcopy terminals can support '`nw`'.

'`cm`'
: String of commands to position the cursor at line *l*, column *c*. Both parameters are origin-zero, and are defined relative to the screen, not relative to display memory.

  All display terminals except a few very obsolete ones support '`cm`', so it is acceptable for an application program to refuse to operate on terminals lacking '`cm`'.

'`ho`'
: String of commands to move the cursor to the upper left corner of the screen (this position is called the *home position*). In terminals where the upper left corner of the screen is not the same as the beginning of display memory, this command must go to the upper left corner of the screen, not the beginning of display memory.

  Every display terminal supports this capability, and many application programs refuse to operate if the '`ho`' capability is missing.

'`ll`'
: String of commands to move the cursor to the lower left corner of the screen. On some terminals, moving up from home position does this, but programs should never assume that will work. Just output the '`ll`' string (if it is provided); if moving to home position and then moving up is the best way to get there, the '`ll`' command will do that.

'`cr`'
: String of commands to move the cursor to the beginning of the line it is on. If this capability is not specified, many programs assume they can use the ASCII carriage return character for this.

`le` String of commands to move the cursor left one column. Unless the `bw` flag capability is specified, the effect is undefined if the cursor is at the left margin; do not use this command there. If `bw` is present, this command may be used at the left margin, and it wraps the cursor to the last column of the preceding line.

`nd` String of commands to move the cursor right one column. The effect is undefined if the cursor is at the right margin; do not use this command there, not even if `am` is present.

`up` String of commands to move the cursor vertically up one line. The effect of sending this string when on the top line is undefined; programs should never use it that way.

`do` String of commands to move the cursor vertically down one line. The effect of sending this string when on the bottom line is undefined; programs should never use it that way.

Some programs do use `do` to scroll up one line if used at the bottom line, if `sf` is not defined but `sr` is. This is only to compensate for certain old, incorrect terminal descriptions. (In principle this might actually lead to incorrect behavior on other terminals, but that seems to happen rarely if ever.) But the proper solution is that the terminal description should define `sf` as well as `do` if the command is suitable for scrolling.

The original idea was that this string would not contain a newline character and therefore could be used without disabling the kernel's usual habit of converting of newline into a carriage-return newline sequence. But many terminal descriptions do use newline in the `do` string, so this is not possible; a program which sends the `do` string must disable output conversion in the kernel (see Section 1.4 [Initialize], page 7).

`bw` Flag whose presence says that `le` may be used in column zero to move to the last column of the preceding line. If this flag is not present, `le` should not be used in column zero.

`nw` String of commands to move the cursor to start of next line, possibly clearing rest of line (following the cursor) before moving.

`DO`, `UP`, `LE`, `RI`
Strings of commands to move the cursor *n* lines down vertically, up vertically, or *n* columns left or right. Do not attempt to move past any edge of the screen with these commands; the effect of trying that is undefined. Only a few terminal descriptions provide these commands, and most programs do not use them.

`CM` String of commands to position the cursor at line *l*, column *c*, relative to display memory. Both parameters are origin-zero.

This capability is present only in terminals where there is a difference between screen-relative and memory-relative addressing, and not even in all such terminals.

`ch` String of commands to position the cursor at column *c* in the same line it is on. This is a special case of `cm` in which the vertical position is not changed. The `ch` capability is provided only when it is faster to output than `cm` would be in this special case. Programs should not assume most display terminals have `ch`.

`cv` String of commands to position the cursor at line *l* in the same column. This is a special case of `cm` in which the horizontal position is not changed. The `cv` capability is provided only when it is faster to output than `cm` would be in this special case. Programs should not assume most display terminals have `cv`.

`sc` String of commands to make the terminal save the current cursor position. Only the last saved position can be used. If this capability is present, `rc` should be provided also. Most terminals have neither.

`rc` String of commands to make the terminal restore the last saved cursor position. If this capability is present, `sc` should be provided also. Most terminals have neither.

`ff` String of commands to advance to the next page, for a hardcopy terminal.

`ta` String of commands to move the cursor right to the next hardware tab stop column. Missing if the terminal does not have any kind of hardware tabs. Do not send this command if the kernel's terminal modes say that the kernel is expanding tabs into spaces.

`bt` String of commands to move the cursor left to the previous hardware tab stop column. Missing if the terminal has no such ability; many terminals do not. Do not send this command if the kernel's terminal modes say that the kernel is expanding tabs into spaces.

The following obsolete capabilities should be included in terminal descriptions when appropriate, but should not be looked at by new programs.

`nc` Flag whose presence means the terminal does not support the ASCII carriage return character as `cr`. This flag is needed because old programs assume, when the `cr` capability is missing, that ASCII carriage return can be used for the purpose. We use 'nc' to tell the old programs that carriage return may not be used.

New programs should not assume any default for 'cr', so they need not look at 'nc'. However, descriptions should contain 'nc' whenever they do not contain 'cr'.

'xt' Flag whose presence means that the ASCII tab character may not be used for cursor motion. This flag exists because old programs assume, when the 'ta' capability is missing, that ASCII tab can be used for the purpose. We use 'xt' to tell the old programs not to use tab.

New programs should not assume any default for 'ta', so they need not look at 'xt' in connection with cursor motion. Note that 'xt' also has implications for standout mode (see Section 3.10 [Standout], page 39). It is obsolete in regard to cursor motion but not in regard to standout.

In fact, 'xt' means that the terminal is a Teleray 1061.

'bc' Very obsolete alternative name for the 'le' capability.

'bs' Flag whose presence means that the ASCII character backspace may be used to move the cursor left. Obsolete; look at 'le' instead.

'nl' Obsolete capability which is a string that can either be used to move the cursor down or to scroll. The same string must scroll when used on the bottom line and move the cursor when used on any other line. New programs should use 'do' or 'sf', and ignore 'nl'.

If there is no 'nl' capability, some old programs assume they can use the newline character for this purpose. These programs follow a bad practice, but because they exist, it is still desirable to define the 'nl' capability in a terminal description if the best way to move down is *not* a newline.

3.4 Wrapping

Wrapping means moving the cursor from the right margin to the left margin of the following line. Some terminals wrap automatically when a graphic character is output in the last column, while others do not. Most application programs that use termcap need to know whether the terminal wraps. There are two special flag capabilities to describe what the terminal does when a graphic character is output in the last column.

'am' Flag whose presence means that writing a character in the last column causes the cursor to wrap to the beginning of the next line.

If '`am`' is not present, writing in the last column leaves the cursor at the place where the character was written.

Writing in the last column of the last line should be avoided on terminals with '`am`', as it may or may not cause scrolling to occur (see Section 3.5 [Scrolling], page 30). Scrolling is surely not what you would intend.

If your program needs to check the '`am`' flag, then it also needs to check the '`xn`' flag which indicates that wrapping happens in a strange way. Many common terminals have the '`xn`' flag.

'`xn`' Flag whose presence means that the cursor wraps in a strange way. At least two distinct kinds of strange behavior are known; the termcap data base does not contain anything to distinguish the two.

On Concept-100 terminals, output in the last column wraps the cursor almost like an ordinary '`am`' terminal. But if the next thing output is a newline, it is ignored.

DEC VT-100 terminals (when the wrap switch is on) do a different strange thing: the cursor wraps only if the next thing output is another graphic character. In fact, the wrap occurs when the following graphic character is received by the terminal, before the character is placed on the screen.

On both of these terminals, after writing in the last column a following graphic character will be displayed in the first column of the following line. But the effect of relative cursor motion characters such as newline or backspace at such a time depends on the terminal. The effect of erase or scrolling commands also depends on the terminal. You can't assume anything about what they will do on a terminal that has '`xn`'. So, to be safe, you should never do these things at such a time on such a terminal.

To be sure of reliable results on a terminal which has the '`xn`' flag, output a '`cm`' absolute positioning command after writing in the last column. Another safe thing to do is to output carriage-return newline, which will leave the cursor at the beginning of the following line.

'`LP`' Flag whose presence means that it is safe to write in the last column of the last line without worrying about undesired scrolling. '`LP`' indicates the DEC flavor of '`xn`' strangeness.

3.5 Scrolling

Scrolling means moving the contents of the screen up or down one or more lines. Moving the contents up is *forward scrolling*; moving them down is *reverse scrolling*.

Scrolling happens after each line of output during ordinary output on most display terminals. But in an application program that uses termcap for random-access output, scrolling happens only when explicitly requested with the commands in this section.

Some terminals have a *scroll region* feature. This lets you limit the effect of scrolling to a specified range of lines. Lines outside the range are unaffected when scrolling happens. The scroll region feature is available if either '`cs`' or '`cS`' is present.

'`sf`' String of commands to scroll the screen one line up, assuming it is output with the cursor at the beginning of the bottom line.

'`sr`' String of commands to scroll the screen one line down, assuming it is output with the cursor at the beginning of the top line.

'`do`' A few programs will try to use '`do`' to do the work of '`sf`'. This is not really correct—it is an attempt to compensate for the absence of a '`sf`' command in some old terminal descriptions.

Since these terminal descriptions do define '`sr`', perhaps at one time the definition of '`do`' was different and it could be used for scrolling as well. But it isn't desirable to combine these two functions in one capability, since scrolling often requires more padding than simply moving the cursor down. Defining '`sf`' and '`do`' separately allows you to specify the padding properly. Also, all sources agree that '`do`' should not be relied on to do scrolling.

So the best approach is to add '`sf`' capabilities to the descriptions of these terminals, copying the definition of '`do`' if that does scroll.

'`SF`' String of commands to scroll the screen n lines up, assuming it is output with the cursor at the beginning of the bottom line.

'`SR`' String of commands to scroll the screen n lines down, assuming it is output with the cursor at the beginning of the top line.

'`cs`' String of commands to set the scroll region. This command takes two parameters, *start* and *end*, which are the line numbers (origin-zero) of the first line to include in the scroll region and of the last line to include in it. When a scroll region is set, scrolling is limited to the specified range of lines; lines outside the range are not affected by scroll commands.

Do not try to move the cursor outside the scroll region. The region remains set until explicitly removed. To remove the scroll region, use another '`cs`' command specifying the full height of the screen.

The cursor position is undefined after the '`cs`' command is set, so position the cursor with '`cm`' immediately afterward.

'cS' String of commands to set the scroll region using parameters in different form. The effect is the same as if 'cs' were used. Four parameters are required:

1. Total number of lines on the screen.
2. Number of lines above desired scroll region.
3. Number of lines below (outside of) desired scroll region.
4. Total number of lines on the screen, the same as the first parameter.

This capability is a GNU extension that was invented to allow the Ann Arbor Ambassador's scroll-region command to be described; it could also be done by putting non-Unix '%'-sequences into a 'cs' string, but that would have confused Unix programs that used the 'cs' capability with the Unix termcap. Currently only GNU Emacs uses the 'cS' capability.

'ns' Flag which means that the terminal does not normally scroll for ordinary sequential output. For modern terminals, this means that outputting a newline in ordinary sequential output with the cursor on the bottom line wraps to the top line. For some obsolete terminals, other things may happen.

The terminal may be able to scroll even if it does not normally do so. If the 'sf' capability is provided, it can be used for scrolling regardless of 'ns'.

'da' Flag whose presence means that lines scrolled up off the top of the screen may come back if scrolling down is done subsequently.

The 'da' and 'db' flags do not, strictly speaking, affect how to scroll. But programs that scroll usually need to clear the lines scrolled onto the screen, if these flags are present.

'db' Flag whose presence means that lines scrolled down off the bottom of the screen may come back if scrolling up is done subsequently.

'lm' Numeric value, the number of lines of display memory that the terminal has. A value of zero means that the terminal has more display memory than can fit on the screen, but no fixed number of lines. (The number of lines may depend on the amount of text in each line.)

Any terminal description that defines 'SF' should also define 'sf'; likewise for 'SR' and 'sr'. However, many terminals can only scroll by one line at a time, so it is common to find 'sf' and not 'SF', or 'sr' without 'SR'.

Therefore, all programs that use the scrolling facilities should be prepared to work with 'sf' in the case that 'SF' is absent, and likewise with 'sr'. On the other hand, an application program that uses only '`sf`' and not '`SF`' is acceptable, though slow on some terminals.

When outputting a scroll command with `tputs`, the *nlines* argument should be the total number of lines in the portion of the screen being scrolled. Very often these commands require padding proportional to this number of lines. See Section 1.5 [Padding], page 8.

3.6 Windows

A *window*, in termcap, is a rectangular portion of the screen to which all display operations are restricted. Wrapping, clearing, scrolling, insertion and deletion all operate as if the specified window were all the screen there was.

'`wi`' String of commands to set the terminal output screen window. This string requires four parameters, all origin-zero:

1. The first line to include in the window.
2. The last line to include in the window.
3. The first column to include in the window.
4. The last column to include in the window.

Most terminals do not support windows.

3.7 Clearing Parts of the Screen

There are several terminal capabilities for clearing parts of the screen to blank. All display terminals support the '`cl`' string, and most display terminals support all of these capabilities.

'`cl`' String of commands to clear the entire screen and position the cursor at the upper left corner.

'`cd`' String of commands to clear the line the cursor is on, and all the lines below it, down to the bottom of the screen. This command string should be used only with the cursor in column zero; their effect is undefined if the cursor is elsewhere.

'`ce`' String of commands to clear from the cursor to the end of the current line.

'`ec`' String of commands to clear *n* characters, starting with the character that the cursor is on. This command string is expected to leave the cursor position unchanged. The parameter *n* should never be large enough to reach past the right margin; the effect of such a large parameter would be undefined.

Clear to end of line (`ce`) is extremely important in programs that maintain an updating display. Nearly all display terminals support this operation, so it is acceptable for a an application program to refuse to work if `ce` is not present. However, if you do not want this limitation, you can accomplish clearing to end of line by outputting spaces until you reach the right margin. In order to do this, you must know the current horizontal position. Also, this technique assumes that writing a space will erase. But this happens to be true on all the display terminals that fail to support `ce`.

3.8 Insert/Delete Line

Inserting a line means creating a blank line in the middle of the screen, and pushing the existing lines of text apart. In fact, the lines above the insertion point do not change, while the lines below move down, and one is normally lost at the bottom of the screen.

Deleting a line means causing the line to disappear from the screen, closing up the gap by moving the lines below it upward. A new line appears at the bottom of the screen. Usually this line is blank, but on terminals with the `db` flag it may be a line previously moved off the screen bottom by scrolling or line insertion.

Insertion and deletion of lines is useful in programs that maintain an updating display some parts of which may get longer or shorter. They are also useful in editors for scrolling parts of the screen, and for redisplaying after lines of text are killed or inserted.

Many terminals provide commands to insert or delete a single line at the cursor position. Some provide the ability to insert or delete several lines with one command, using the number of lines to insert or delete as a parameter. Always move the cursor to column zero before using any of these commands.

`al` String of commands to insert a blank line before the line the cursor is on. The existing line, and all lines below it, are moved down. The last line in the screen (or in the scroll region, if one is set) disappears and in most circumstances is discarded. It may not be discarded if the `db` is present (see Section 3.5 [Scrolling], page 30).

The cursor must be at the left margin before this command is used. This command does not move the cursor.

`dl` String of commands to delete the line the cursor is on. The following lines move up, and a blank line appears at the bottom of the screen (or bottom of the scroll region). If the terminal has the `db` flag, a nonblank line previously pushed off the screen bottom may reappear at the bottom.

The cursor must be at the left margin before this command is used. This command does not move the cursor.

'AL' String of commands to insert *n* blank lines before the line that the cursor is on. It is like 'al' repeated *n* times, except that it is as fast as one 'al'.

'DL' String of commands to delete *n* lines starting with the line that the cursor is on. It is like 'dl' repeated *n* times, except that it is as fast as one 'dl'.

Any terminal description that defines 'AL' should also define 'al'; likewise for 'DL' and 'dl'. However, many terminals can only insert or delete one line at a time, so it is common to find 'al' and not 'AL', or 'dl' without 'DL'.

Therefore, all programs that use the insert and delete facilities should be prepared to work with 'al' in the case that 'AL' is absent, and likewise with 'dl'. On the other hand, it is acceptable to write an application that uses only 'al' and 'dl' and does not look for 'AL' or 'DL' at all.

If a terminal does not support line insertion and deletion directly, but does support a scroll region, the effect of insertion and deletion can be obtained with scrolling. However, it is up to the individual user program to check for this possibility and use the scrolling commands to get the desired result. It is fairly important to implement this alternate strategy, since it is the only way to get the effect of line insertion and deletion on the popular VT100 terminal.

Insertion and deletion of lines is affected by the scroll region on terminals that have a settable scroll region. This is useful when it is desirable to move any few consecutive lines up or down by a few lines. See Section 3.5 [Scrolling], page 30.

The line pushed off the bottom of the screen is not lost if the terminal has the 'db' flag capability; instead, it is pushed into display memory that does not appear on the screen. This is the same thing that happens when scrolling pushes a line off the bottom of the screen. Either reverse scrolling or deletion of a line can bring the apparently lost line back onto the bottom of the screen. If the terminal has the scroll region feature as well as 'db', the pushed-out line really is lost if a scroll region is in effect.

When outputting an insert or delete command with `tputs`, the *nlines* argument should be the total number of lines from the cursor to the bottom of the screen (or scroll region). Very often these commands require padding proportional to this number of lines. See Section 1.5 [Padding], page 8.

For 'AL' and 'DL' the *nlines* argument should *not* depend on the number of lines inserted or deleted; only the total number of lines affected. This is because it is just as fast to insert two or *n* lines with 'AL' as to insert one line with 'al'.

3.9 Insert/Delete Character

Inserting a character means creating a blank space in the middle of a line, and pushing the rest of the line rightward. The character in the rightmost column is lost.

Deleting a character means causing the character to disappear from the screen, closing up the gap by moving the rest of the line leftward. A blank space appears in the rightmost column.

Insertion and deletion of characters is useful in programs that maintain an updating display some parts of which may get longer or shorter. It is also useful in editors for redisplaying the results of editing within a line.

Many terminals provide commands to insert or delete a single character at the cursor position. Some provide the ability to insert or delete several characters with one command, using the number of characters to insert or delete as a parameter.

Many terminals provide an insert mode in which outputting a graphic character has the added effect of inserting a position for that character. A special command string is used to enter insert mode and another is used to exit it. The reason for designing a terminal with an insert mode rather than an insert command is that inserting character positions is usually followed by writing characters into them. With insert mode, this is as fast as simply writing the characters, except for the fixed overhead of entering and leaving insert mode. However, when the line speed is great enough, padding may be required for the graphic characters output in insert mode.

Some terminals require you to enter insert mode and then output a special command for each position to be inserted. Or they may require special commands to be output before or after each graphic character to be inserted.

Deletion of characters is usually accomplished by a straightforward command to delete one or several positions; but on some terminals, it is necessary to enter a special delete mode before using the delete command, and leave delete mode afterward. Sometimes delete mode and insert mode are the same mode.

Some terminals make a distinction between character positions in which a space character has been output and positions which have been cleared. On these terminals, the effect of insert or delete character runs to the first cleared position rather than to the end of the line. In fact, the effect may run to more than one line if there is no cleared position to stop the shift on the first line. These terminals are identified by the '`in`' flag capability.

On terminals with the '`in`' flag, the technique of skipping over characters that you know were cleared, and then outputting text later on in the same line, causes later insert and delete character operations on that line to do nonstandard things. A program that has any chance of doing this must check for the '`in`' flag and must be careful to write explicit space characters into the intermediate columns when '`in`' is present.

A plethora of terminal capabilities are needed to describe all of this complexity. Here is a list of them all. Following the list, we present an algorithm for programs to use to take proper account of all of these capabilities.

'`im`' String of commands to enter insert mode.

If the terminal has no special insert mode, but it can insert characters with a special command, '`im`' should be defined with a null value, because the '`vi`' editor assumes that insertion of a character is impossible if '`im`' is not provided.

New programs should not act like '`vi`'. They should pay attention to '`im`' only if it is defined.

'`ei`' String of commands to leave insert mode. This capability must be present if '`im`' is.

On a few old terminals the same string is used to enter and exit insert mode. This string turns insert mode on if it was off, and off it it was on. You can tell these terminals because the '`ei`' string equals the '`im`' string. If you want to support these terminals, you must always remember accurately whether insert mode is in effect. However, these terminals are obsolete, and it is reasonable to refuse to support them. On all modern terminals, you can safely output '`ei`' at any time to ensure that insert mode is turned off.

'`ic`' String of commands to insert one character position at the cursor. The cursor does not move.

If outputting a graphic character while in insert mode is sufficient to insert the character, then the '`ic`' capability should be defined with a null value.

If your terminal offers a choice of ways to insert—either use insert mode or use a special command—then define '`im`' and do not define '`ic`', since this gives the most efficient operation when several characters are to be inserted. *Do not* define both strings, for that means that *both* must be used each time insertion is done.

'`ip`' String of commands to output following an inserted graphic character in insert mode. Often it is used just for a padding spec, when padding is needed after an inserted character (see Section 1.5 [Padding], page 8).

'`IC`' String of commands to insert *n* character positions at and after the cursor. It has the same effect as repeating the '`ic`' string and a space, *n* times.

If 'IC' is provided, application programs may use it without first entering insert mode.

'mi' Flag whose presence means it is safe to move the cursor while in insert mode and assume the terminal remains in insert mode.

'in' Flag whose presence means that the terminal distinguishes between character positions in which space characters have been output and positions which have been cleared.

An application program can assume that the terminal can do character insertion if *any one of* the capabilities 'IC', 'im', 'ic' or 'ip' is provided.

To insert *n* blank character positions, move the cursor to the place to insert them and follow this algorithm:

1. If an 'IC' string is provided, output it with parameter *n* and you are finished. Otherwise (or if you don't want to bother to look for an 'IC' string) follow the remaining steps.
2. Output the 'im' string, if there is one, unless the terminal is already in insert mode.
3. Repeat steps 4 through 6, *n* times.
4. Output the 'ic' string if any.
5. Output a space.
6. Output the 'ip' string if any.
7. Output the 'ei' string, eventually, to exit insert mode. There is no need to do this right away. If the 'mi' flag is present, you can move the cursor and the cursor will remain in insert mode; then you can do more insertion elsewhere without reentering insert mode.

To insert *n* graphic characters, position the cursor and follow this algorithm:

1. If an 'IC' string is provided, output it with parameter *n*, then output the graphic characters, and you are finished. Otherwise (or if you don't want to bother to look for an 'IC' string) follow the remaining steps.
2. Output the 'im' string, if there is one, unless the terminal is already in insert mode.
3. For each character to be output, repeat steps 4 through 6.
4. Output the 'ic' string if any.
5. Output the next graphic character.
6. Output the 'ip' string if any.
7. Output the 'ei' string, eventually, to exit insert mode. There is no need to do this right away. If the 'mi' flag is present, you can move the cursor and the cursor will remain in insert mode; then you can do more insertion elsewhere without reentering insert mode.

Note that this is not the same as the original Unix termcap specifications in one respect: it assumes that the '`IC`' string can be used without entering insert mode. This is true as far as I know, and it allows you be able to avoid entering and leaving insert mode, and also to be able to avoid the inserted-character padding after the characters that go into the inserted positions.

Deletion of characters is less complicated; deleting one column is done by outputting the '`dc`' string. However, there may be a delete mode that must be entered with '`dm`' in order to make '`dc`' work.

'`dc`' String of commands to delete one character position at the cursor. If '`dc`' is not present, the terminal cannot delete characters.

'`DC`' String of commands to delete *n* characters starting at the cursor. It has the same effect as repeating the '`dc`' string *n* times. Any terminal description that has '`DC`' also has '`dc`'.

'`dm`' String of commands to enter delete mode. If not present, there is no delete mode, and '`dc`' can be used at any time (assuming there is a '`dc`').

'`ed`' String of commands to exit delete mode. This must be present if '`dm`' is.

To delete *n* character positions, position the cursor and follow these steps:

1. If the '`DC`' string is present, output it with parameter *n* and you are finished. Otherwise, follow the remaining steps.
2. Output the '`dm`' string, unless you know the terminal is already in delete mode.
3. Output the '`dc`' string *n* times.
4. Output the '`ed`' string eventually. If the flag capability '`mi`' is present, you can move the cursor and do more deletion without leaving and reentering delete mode.

As with the '`IC`' string, we have departed from the original termcap specifications by assuming that '`DC`' works without entering delete mode even though '`dc`' would not.

If the '`dm`' and '`im`' capabilities are both present and have the same value, it means that the terminal has one mode for both insertion and deletion. It is useful for a program to know this, because then it can do insertions after deletions, or vice versa, without leaving insert/delete mode and reentering it.

3.10 Standout and Appearance Modes

Appearance modes are modifications to the ways characters are displayed. Typical appearance modes include reverse video, dim, bright, blinking, underlined, invisible, and alternate character set. Each kind of terminal supports various among these, or perhaps none.

For each type of terminal, one appearance mode or combination of them that looks good for highlighted text is chosen as the *standout mode*. The capabilities '`so`' and '`se`' say how to enter and leave standout mode. Programs that use appearance modes only to highlight some text generally use the standout mode so that they can work on as many terminals as possible. Use of specific appearance modes other than "underlined" and "alternate character set" is rare.

Terminals that implement appearance modes fall into two general classes as to how they do it.

In some terminals, the presence or absence of any appearance mode is recorded separately for each character position. In these terminals, each graphic character written is given the appearance modes current at the time it is written, and keeps those modes until it is erased or overwritten. There are special commands to turn the appearance modes on or off for characters to be written in the future.

In other terminals, the change of appearance modes is represented by a marker that belongs to a certain screen position but affects all following screen positions until the next marker. These markers are traditionally called *magic cookies*.

The same capabilities ('`so`', '`se`', '`mb`' and so on) for turning appearance modes on and off are used for both magic-cookie terminals and per-character terminals. On magic cookie terminals, these give the commands to write the magic cookies. On per-character terminals, they change the current modes that affect future output and erasure. Some simple applications can use these commands without knowing whether or not they work by means of cookies.

However, a program that maintains and updates a display needs to know whether the terminal uses magic cookies, and exactly what their effect is. This information comes from the '`sg`' capability.

The '`sg`' capability is a numeric capability whose presence indicates that the terminal uses magic cookies for appearance modes. Its value is the number of character positions that a magic cookie occupies. Usually the cookie occupies one or more character positions on the screen, and these character positions are displayed as blank, but in some terminals the cookie has zero width.

The '`sg`' capability describes both the magic cookie to turn standout on and the cookie to turn it off. This makes the assumption that both kinds of cookie have the same width on the screen. If that is not true, the narrower cookie must be "widened" with spaces until it has the same width as the other.

On some magic cookie terminals, each line always starts with normal display; in other words, the scope of a magic cookie never extends over more than one line. But on other terminals, one magic cookie affects all the lines below it unless explicitly canceled. Termcap does not define any way to distinguish these two ways magic cookies can work. To be safe, it is best to put a cookie at the beginning of each line.

On some per-character terminals, standout mode or other appearance modes may be canceled by moving the cursor. On others, moving the cursor has no effect on the state of the appearance modes. The latter class of terminals are given the flag capability '`ms`' ("can move in standout"). All programs that might have occasion to move the cursor while appearance modes are turned on must check for this flag; if it is not present, they should reset appearance modes to normal before doing cursor motion.

A program that has turned on only standout mode should use '`se`' to reset the standout mode to normal. A program that has turned on only alternate character set mode should use '`ae`' to return it to normal. If it is possible that any other appearance modes are turned on, use the '`me`' capability to return them to normal.

Note that the commands to turn on one appearance mode, including '`so`' and '`mb`' ... '`mr`', if used while some other appearance modes are turned on, may combine the two modes on some terminals but may turn off the mode previously enabled on other terminals. This is because some terminals do not have a command to set or clear one appearance mode without changing the others. Programs should not attempt to use appearance modes in combination except with '`sa`', and when switching from one single mode to another should always turn off the previously enabled mode and then turn on the new desired mode.

On some old terminals, the '`so`' and '`se`' commands may be the same command, which has the effect of turning standout on if it is off, or off it is on. It is therefore risky for a program to output extra '`se`' commands for good measure. Fortunately, all these terminals are obsolete.

Programs that update displays in which standout-text may be replaced with non-standout text must check for the '`xs`' flag. In a per-character terminal, this flag says that the only way to remove standout once written is to clear that portion of the line with the '`ce`' string or something even more powerful (see Section 3.7 [Clearing], page 33); just writing new characters at those screen positions will not change the modes in effect there. In a magic cookie terminal, '`xs`' says that the only way to remove a cookie is to clear a portion of the line that includes the cookie; writing a different cookie at the same position does not work.

Such programs must also check for the '`xt`' flag, which means that the terminal is a Teleray 1061. On this terminal it is impossible to position the cursor at the front of a magic cookie, so the only two ways to remove a cookie are (1) to delete the line it is on or (2) to position the cursor at least one character before it (possibly on a previous line) and output the '`se`' string, which on these terminals finds and removes the next '`so`' magic cookie on the screen. (It may also be possible to remove a cookie which is not at the beginning of a line by clearing that line.) The '`xt`' capability also has implications for the use of tab characters, but in that regard it is obsolete (See Section 3.3 [Cursor Motion], page 25).

'`so`' String of commands to enter standout mode.

'`se`' String of commands to leave standout mode.

'`sg`' Numeric capability, the width on the screen of the magic cookie. This capability is absent in terminals that record appearance modes character by character.

'`ms`' Flag whose presence means that it is safe to move the cursor while the appearance modes are not in the normal state. If this flag is absent, programs should always reset the appearance modes to normal before moving the cursor.

'`xs`' Flag whose presence means that the only way to reset appearance modes already on the screen is to clear to end of line. On a per-character terminal, you must clear the area where the modes are set. On a magic cookie terminal, you must clear an area containing the cookie. See the discussion above.

'`xt`' Flag whose presence means that the cursor cannot be positioned right in front of a magic cookie, and that '`se`' is a command to delete the next magic cookie following the cursor. See discussion above.

'`mb`' String of commands to enter blinking mode.

'`md`' String of commands to enter double-bright mode.

'`mh`' String of commands to enter half-bright mode.

'`mk`' String of commands to enter invisible mode.

'`mp`' String of commands to enter protected mode.

'`mr`' String of commands to enter reverse-video mode.

'`me`' String of commands to turn off all appearance modes, including standout mode and underline mode. On some terminals it also turns off alternate character set mode; on others, it may not. This capability must be present if any of '`mb`' ... '`mr`' is present.

'`as`' String of commands to turn on alternate character set mode. This mode assigns some or all graphic characters an alternate picture on the screen. There is no standard as to what the alternate pictures look like.

'ae' String of commands to turn off alternate character set mode.

'sa' String of commands to turn on an arbitrary combination of appearance modes. It accepts 9 parameters, each of which controls a particular kind of appearance mode. A parameter should be 1 to turn its appearance mode on, or zero to turn that mode off. Most terminals do not support the 'sa' capability, even among those that do have various appearance modes.

The nine parameters are, in order, *standout*, *underline*, *reverse*, *blink*, *half-bright*, *double-bright*, *blank*, *protect*, *alt char set*.

3.11 Underlining

Underlining on most terminals is a kind of appearance mode, much like standout mode. Therefore, it may be implemented using magic cookies or as a flag in the terminal whose current state affects each character that is output. See Section 3.10 [Standout], page 39, for a full explanation.

The 'ug' capability is a numeric capability whose presence indicates that the terminal uses magic cookies for underlining. Its value is the number of character positions that a magic cookie for underlining occupies; it is used for underlining just as 'sg' is used for standout. Aside from the simplest applications, it is impossible to use underlining correctly without paying attention to the value of 'ug'.

'us' String of commands to turn on underline mode or to output a magic cookie to start underlining.

'ue' String of commands to turn off underline mode or to output a magic cookie to stop underlining.

'ug' Width of magic cookie that represents a change of underline mode; or missing, if the terminal does not use a magic cookie for this.

'ms' Flag whose presence means that it is safe to move the cursor while the appearance modes are not in the normal state. Underlining is an appearance mode. If this flag is absent, programs should always turn off underlining before moving the cursor.

There are two other, older ways of doing underlining: there can be a command to underline a single character, or the output of '_', the ASCII underscore character, as an overstrike could cause a character to be underlined. New programs need not bother to handle these capabilities unless the author cares strongly about the obscure terminals which support them. However, terminal descriptions should provide these capabilities when appropriate.

'uc' String of commands to underline the character under the cursor, and move the cursor right.

'ul' Flag whose presence means that the terminal can underline by overstriking an underscore character ('_'); some terminals can do this even though they do not support overstriking in general. An implication of this flag is that when outputting new text to overwrite old text, underscore characters must be treated specially lest they underline the old text instead.

3.12 Cursor Visibility

Some terminals have the ability to make the cursor invisible, or to enhance it. Enhancing the cursor is often done by programs that plan to use the cursor to indicate to the user a position of interest that may be anywhere on the screen—for example, the Emacs editor enhances the cursor on entry. Such programs should always restore the cursor to normal on exit.

'vs' String of commands to enhance the cursor.

'vi' String of commands to make the cursor invisible.

've' String of commands to return the cursor to normal.

If you define either 'vs' or 'vi', you must also define 've'.

3.13 Bell

Here we describe commands to make the terminal ask for the user to pay attention to it.

'bl' String of commands to cause the terminal to make an audible sound. If this capability is absent, the terminal has no way to make a suitable sound.

'vb' String of commands to cause the screen to flash to attract attention ("visible bell"). If this capability is absent, the terminal has no way to do such a thing.

3.14 Keypad and Function Keys

Many terminals have arrow and function keys that transmit specific character sequences to the computer. Since the precise sequences used depend on the terminal, termcap defines capabilities used to say what the sequences are. Unlike most termcap string-valued capabilities, these are not strings of commands to be sent to the terminal, rather strings that are received from the terminal.

Programs that expect to use keypad keys should check, initially, for a 'ks' capability and send it, to make the keypad actually transmit. Such programs should also send the 'ke' string when exiting.

'ks' String of commands to make the keypad keys transmit. If this capability is not provided, but the others in this section are, programs may assume that the keypad keys always transmit.

'ke' String of commands to make the keypad keys work locally. This capability is provided only if 'ks' is.

'kl' String of input characters sent by typing the left-arrow key. If this capability is missing, you cannot expect the terminal to have a left-arrow key that transmits anything to the computer.

'kr' String of input characters sent by typing the right-arrow key.

'ku' String of input characters sent by typing the up-arrow key.

'kd' String of input characters sent by typing the down-arrow key.

'kh' String of input characters sent by typing the "home-position" key.

'K1' ... 'K5'
Strings of input characters sent by the five other keys in a 3-by-3 array that includes the arrow keys, if the keyboard has such a 3-by-3 array. Note that one of these keys may be the "home-position" key, in which case one of these capabilities will have the same value as the 'kh' key.

'k0' String of input characters sent by function key 10 (or 0, if the terminal has one labeled 0).

'k1' ... 'k9'
Strings of input characters sent by function keys 1 through 9, provided for those function keys that exist.

'kn' Number: the number of numbered function keys, if there are more than 10.

'l0' ... 'l9'
Strings which are the labels appearing on the keyboard on the keys described by the capabilities 'k0' ... 'l9'. These capabilities should be left undefined if the labels are 'f0' or 'f10' and 'f1' ... 'f9'.

'kH' String of input characters sent by the "home down" key, if there is one.

'kb' String of input characters sent by the "backspace" key, if there is one.

`ka` String of input characters sent by the "clear all tabs" key, if there is one.

`kt` String of input characters sent by the "clear tab stop this column" key, if there is one.

`kC` String of input characters sent by the "clear screen" key, if there is one.

`kD` String of input characters sent by the "delete character" key, if there is one.

`kL` String of input characters sent by the "delete line" key, if there is one.

`kM` String of input characters sent by the "exit insert mode" key, if there is one.

`kE` String of input characters sent by the "clear to end of line" key, if there is one.

`kS` String of input characters sent by the "clear to end of screen" key, if there is one.

`kI` String of input characters sent by the "insert character" or "enter insert mode" key, if there is one.

`kA` String of input characters sent by the "insert line" key, if there is one.

`kN` String of input characters sent by the "next page" key, if there is one.

`kP` String of input characters sent by the "previous page" key, if there is one.

`kF` String of input characters sent by the "scroll forward" key, if there is one.

`kR` String of input characters sent by the "scroll reverse" key, if there is one.

`kT` String of input characters sent by the "set tab stop in this column" key, if there is one.

`ko` String listing the other function keys the terminal has. This is a very obsolete way of describing the same information found in the `kH` ... `kT` keys. The string contains a list of two-character termcap capability names, separated by commas. The meaning is that for each capability name listed, the terminal has a key which sends the string which is the value of that capability. For example, the value `:ko=cl,ll,sf,sr:` says that the terminal has four function keys which mean "clear screen", "home down", "scroll forward" and "scroll reverse".

3.15 Meta Key

A Meta key is a key on the keyboard that modifies each character you type by controlling the 0200 bit. This bit is on if and only if the Meta key is held down when the character is typed. Characters typed using the Meta key are called Meta characters. Emacs uses Meta characters as editing commands.

'`km`' Flag whose presence means that the terminal has a Meta key.

'`mm`' String of commands to enable the functioning of the Meta key.

'`mo`' String of commands to disable the functioning of the Meta key.

If the terminal has '`km`' but does not have '`mm`' and '`mo`', it means that the Meta key always functions. If it has '`mm`' and '`mo`', it means that the Meta key can be turned on or off. Send the '`mm`' string to turn it on, and the '`mo`' string to turn it off. I do not know why one would ever not want it to be on.

3.16 Initialization

'`ti`' String of commands to put the terminal into whatever special modes are needed or appropriate for programs that move the cursor nonsequentially around the screen. Programs that use termcap to do full-screen display should output this string when they start up.

'`te`' String of commands to undo what is done by the '`ti`' string. Programs that output the '`ti`' string on entry should output this string when they exit.

'`is`' String of commands to initialize the terminal for each login session.

'`if`' String which is the name of a file containing the string of commands to initialize the terminal for each session of use. Normally '`is`' and '`if`' are not both used.

'`i1`'
'`i3`' Two more strings of commands to initialize the terminal for each login session. The '`i1`' string (if defined) is output before '`is`' or '`if`', and the '`i3`' string (if defined) is output after.

The reason for having three separate initialization strings is to make it easier to define a group of related terminal types with slightly different initializations. Define two or three of the strings in the basic type; then the other types can override one or two of the strings.

'rs' String of commands to reset the terminal from any strange mode it may be in. Normally this includes the 'is' string (or other commands with the same effects) and more. What would go in the 'rs' string but not in the 'is' string are annoying or slow commands to bring the terminal back from strange modes that nobody would normally use.

'it' Numeric value, the initial spacing between hardware tab stop columns when the terminal is powered up. Programs to initialize the terminal can use this to decide whether there is a need to set the tab stops. If the initial width is 8, well and good; if it is not 8, then the tab stops should be set; if they cannot be set, the kernel is told to convert tabs to spaces, and other programs will observe this and do likewise.

'ct' String of commands to clear all tab stops.

'st' String of commands to set tab stop at current cursor column on all lines.

'NF' Flag whose presence means that the terminal does not support XON/XOFF flow control. Programs should not send XON (*C-q*) or XOFF (*C-s*) characters to the terminal.

3.17 Padding Capabilities

There are two terminal capabilities that exist just to explain the proper way to obey the padding specifications in all the command string capabilities. One, 'pc', must be obeyed by all termcap-using programs.

'pb' Numeric value, the lowest baud rate at which padding is actually needed. Programs may check this and refrain from doing any padding at lower speeds.

'pc' String of commands for padding. The first character of this string is to be used as the pad character, instead of using null characters for padding. If 'pc' is not provided, use null characters. Every program that uses termcap must look up this capability and use it to set the variable `PC` that is used by `tputs`. See Section 1.5 [Padding], page 8.

Some termcap capabilities exist just to specify the amount of padding that the kernel should give to cursor motion commands used in ordinary sequential output.

'dC' Numeric value, the number of msec of padding needed for the carriage-return character.

'dN' Numeric value, the number of msec of padding needed for the newline (linefeed) character.

`dB` Numeric value, the number of msec of padding needed for the backspace character.

`dF` Numeric value, the number of msec of padding needed for the formfeed character.

`dT` Numeric value, the number of msec of padding needed for the tab character.

In some systems, the kernel uses the above capabilities; in other systems, the kernel uses the paddings specified in the string capabilities `cr`, `sf`, `le`, `ff` and `ta`. Descriptions of terminals which require such padding should contain the `dC` ... `dT` capabilities and also specify the appropriate padding in the corresponding string capabilities. Since no modern terminals require padding for ordinary sequential output, you probably won't need to do either of these things.

3.18 Status Line

A *status line* is a line on the terminal that is not used for ordinary display output but instead used for a special message. The intended use is for a continuously updated description of what the user's program is doing, and that is where the name "status line" comes from, but in fact it could be used for anything. The distinguishing characteristic of a status line is that ordinary output to the terminal does not affect it; it changes only if the special status line commands of this section are used.

`hs` Flag whose presence means that the terminal has a status line. If a terminal description specifies that there is a status line, it must provide the `ts` and `fs` capabilities.

`ts` String of commands to move the terminal cursor into the status line. Usually these commands must specifically record the old cursor position for the sake of the `fs` string.

`fs` String of commands to move the cursor back from the status line to its previous position (outside the status line).

`es` Flag whose presence means that other display commands work while writing the status line. In other words, one can clear parts of it, insert or delete characters, move the cursor within it using `ch` if there is a `ch` capability, enter and leave standout mode, and so on.

`ds` String of commands to disable the display of the status line. This may be absent, if there is no way to disable the status line display.

'ws' Numeric value, the width of the status line. If this capability is absent in a terminal that has a status line, it means the status line is the same width as the other lines.

Note that the value of 'ws' is sometimes as small as 8.

3.19 Half-Line Motion

Some terminals have commands for moving the cursor vertically by half-lines, useful for outputting subscripts and superscripts. Mostly it is hardcopy terminals that have such features.

'hu' String of commands to move the cursor up half a line. If the terminal is a display, it is your responsibility to avoid moving up past the top line; however, most likely the terminal that supports this is a hardcopy terminal and there is nothing to be concerned about.

'hd' String of commands to move the cursor down half a line. If the terminal is a display, it is your responsibility to avoid moving down past the bottom line, etc.

3.20 Controlling Printers Attached to Terminals

Some terminals have attached hardcopy printer ports. They may be able to copy the screen contents to the printer; they may also be able to redirect output to the printer. Termcap does not have anything to tell the program whether the redirected output appears also on the screen; it does on some terminals but not all.

'ps' String of commands to cause the contents of the screen to be printed. If it is absent, the screen contents cannot be printed.

'po' String of commands to redirect further output to the printer.

'pf' String of commands to terminate redirection of output to the printer. This capability must be present in the description if 'po' is.

'pO' String of commands to redirect output to the printer for next *n* characters of output, regardless of what they are. Redirection will end automatically after *n* characters of further output. Until then, nothing that is output can end redirection, not even the 'pf' string if there is one. The number *n* should not be more than 255.

One use of this capability is to send non-text byte sequences (such as bit-maps) to the printer.

Most terminals with printers do not support all of '`ps`', '`po`' and '`pO`'; any one or two of them may be supported. To make a program that can send output to all kinds of printers, it is necessary to check for all three of these capabilities, choose the most convenient of the ones that are provided, and use it in its own appropriate fashion.

4 Summary of Capability Names

Here are all the terminal capability names in alphabetical order with a brief description of each. For cross references to their definitions, see the index of capability names (see [Cap Index], page 61).

`ae`	String to turn off alternate character set mode.
`al`	String to insert a blank line before the cursor.
`AL`	String to insert *n* blank lines before the cursor.
`am`	Flag: output to last column wraps cursor to next line.
`as`	String to turn on alternate character set mode.like.
`bc`	Very obsolete alternative name for the `le` capability.
`bl`	String to sound the bell.
`bs`	Obsolete flag: ASCII backspace may be used for leftward motion.
`bt`	String to move the cursor left to the previous hardware tab stop column.
`bw`	Flag: `le` at left margin wraps to end of previous line.
`CC`	String to change terminal's command character.
`cd`	String to clear the line the cursor is on, and following lines.
`ce`	String to clear from the cursor to the end of the line.
`ch`	String to position the cursor at column *c* in the same line.
`cl`	String to clear the entire screen and put cursor at upper left corner.
`cm`	String to position the cursor at line *l*, column *c*.
`CM`	String to position the cursor at line *l*, column *c*, relative to display memory.
`co`	Number: width of the screen.
`cr`	String to move cursor sideways to left margin.
`cs`	String to set the scroll region.
`cS`	Alternate form of string to set the scroll region.
`ct`	String to clear all tab stops.
`cv`	String to position the cursor at line *l* in the same column.
`da`	Flag: data scrolled off top of screen may be scrolled back.

`db` Flag: data scrolled off bottom of screen may be scrolled back.

`dB` Obsolete number: msec of padding needed for the backspace character.

`dc` String to delete one character position at the cursor.

`dC` Obsolete number: msec of padding needed for the carriage-return character.

`DC` String to delete *n* characters starting at the cursor.

`dF` Obsolete number: msec of padding needed for the formfeed character.

`dl` String to delete the line the cursor is on.

`DL` String to delete *n* lines starting with the cursor's line.

`dm` String to enter delete mode.

`dN` Obsolete number: msec of padding needed for the newline character.

`do` String to move the cursor vertically down one line.

`DO` String to move cursor vertically down *n* lines.

`ds` String to disable the display of the status line.

`dT` Obsolete number: msec of padding needed for the tab character.

`ec` String of commands to clear *n* characters at cursor.

`ed` String to exit delete mode.

`ei` String to leave insert mode.

`eo` Flag: output of a space can erase an overstrike.

`es` Flag: other display commands work while writing the status line.

`ff` String to advance to the next page, for a hardcopy terminal.

`fs` String to move the cursor back from the status line to its previous position (outside the status line).

`gn` Flag: this terminal type is generic, not real.

`hc` Flag: hardcopy terminal.

`hd` String to move the cursor down half a line.

`ho` String to position cursor at upper left corner.

`hs` Flag: the terminal has a status line.

'hu' String to move the cursor up half a line.

'hz' Flag: terminal cannot accept '~' as output.

'i1' String to initialize the terminal for each login session.

'i3' String to initialize the terminal for each login session.

'ic' String to insert one character position at the cursor.

'IC' String to insert *n* character positions at the cursor.

'if' String naming a file of commands to initialize the terminal.

'im' String to enter insert mode.

'in' Flag: outputting a space is different from moving over empty positions.

'ip' String to output following an inserted character in insert mode.

'is' String to initialize the terminal for each login session.

'it' Number: initial spacing between hardware tab stop columns.

'k0' String of input sent by function key 0 or 10.

'k1 ... k9' Strings of input sent by function keys 1 through 9.

'K1 ... K5' Strings sent by the five other keys in 3-by-3 array with arrows.

'ka' String of input sent by the "clear all tabs" key.

'kA' String of input sent by the "insert line" key.

'kb' String of input sent by the "backspace" key.

'kC' String of input sent by the "clear screen" key.

'kd' String of input sent by typing the down-arrow key.

'kD' String of input sent by the "delete character" key.

'ke' String to make the function keys work locally.

'kE' String of input sent by the "clear to end of line" key.

'kF' String of input sent by the "scroll forward" key.

'kh' String of input sent by typing the "home-position" key.

'kH' String of input sent by the "home down" key.

'kI' String of input sent by the "insert character" or "enter insert mode" key.

'kl' String of input sent by typing the left-arrow key.

'kL' String of input sent by the "delete line" key.

`km` Flag: the terminal has a Meta key.

`kM` String of input sent by the "exit insert mode" key.

`kn` Numeric value, the number of numbered function keys.

`kN` String of input sent by the "next page" key.

`ko` Very obsolete string listing the terminal's named function keys.

`kP` String of input sent by the "previous page" key.

`kr` String of input sent by typing the right-arrow key.

`kR` String of input sent by the "scroll reverse" key.

`ks` String to make the function keys transmit.

`kS` String of input sent by the "clear to end of screen" key.

`kt` String of input sent by the "clear tab stop this column" key.

`kT` String of input sent by the "set tab stop in this column" key.

`ku` String of input sent by typing the up-arrow key.

`l0` String on keyboard labelling function key 0 or 10.

`l1 ... l9` Strings on keyboard labelling function keys 1 through 9.

`le` String to move the cursor left one column.

`LE` String to move cursor left *n* columns.

`li` Number: height of the screen.

`ll` String to position cursor at lower left corner.

`lm` Number: lines of display memory.

`LP` Flag: writing to last column of last line will not scroll.

`mb` String to enter blinking mode.

`md` String to enter double-bright mode.

`me` String to turn off all appearance modes

`mh` String to enter half-bright mode.

`mi` Flag: cursor motion in insert mode is safe.

`mk` String to enter invisible mode.

`mm` String to enable the functioning of the Meta key.

`mo` String to disable the functioning of the Meta key.

`mp` String to enter protected mode.

'`mr`' String to enter reverse-video mode.

'`ms`' Flag: cursor motion in standout mode is safe.

'`nc`' Obsolete flag: do not use ASCII carriage-return on this terminal.

'`nd`' String to move the cursor right one column.

'`NF`' Flag: do not use XON/XOFF flow control.

'`nl`' Obsolete alternative name for the '`do`' and '`sf`' capabilities.

'`ns`' Flag: the terminal does not normally scroll for sequential output.

'`nw`' String to move to start of next line, possibly clearing rest of old line.

'`os`' Flag: terminal can overstrike.

'`pb`' Number: the lowest baud rate at which padding is actually needed.

'`pc`' String containing character for padding.

'`pf`' String to terminate redirection of output to the printer.

'`po`' String to redirect further output to the printer.

'`pO`' String to redirect *n* characters ofoutput to the printer.

'`ps`' String to print the screen on the attached printer.

'`rc`' String to move to last saved cursor position.

'`RI`' String to move cursor right *n* columns.

'`rp`' String to output character *c* repeated *n* times.

'`rs`' String to reset the terminal from any strange modes.

'`sa`' String to turn on an arbitrary combination of appearance modes.

'`sc`' String to save the current cursor position.

'`se`' String to leave standout mode.

'`sf`' String to scroll the screen one line up.

'`SF`' String to scroll the screen *n* lines up.

'`sg`' Number: width of magic standout cookie. Absent if magic cookies are not used.

'`so`' String to enter standout mode.

'`sr`' String to scroll the screen one line down.

'`SR`' String to scroll the screen *n* line down.

'st' String to set tab stop at current cursor column on all lines. programs.

'ta' String to move the cursor right to the next hardware tab stop column.

'te' String to return terminal to settings for sequential output.

'ti' String to initialize terminal for random cursor motion.

'ts' String to move the terminal cursor into the status line.

'uc' String to underline one character and move cursor right.

'ue' String to turn off underline mode

'ug' Number: width of underlining magic cookie. Absent if underlining doesn't use magic cookies.

'ul' Flag: underline by overstriking with an underscore.

'up' String to move the cursor vertically up one line.

'UP' String to move cursor vertically up *n* lines.

'us' String to turn on underline mode

'vb' String to make the screen flash.

've' String to return the cursor to normal.

'vi' String to make the cursor invisible.

'vs' String to enhance the cursor.

'wi' String to set the terminal output screen window.

'ws' Number: the width of the status line.

'xb' Flag: superbee terminal.

'xn' Flag: cursor wraps in a strange way.

'xs' Flag: clearing a line is the only way to clear the appearance modes of positions in that line (or, only way to remove magic cookies on that line).

'xt' Flag: Teleray 1061; several strange characteristics.

Variable and Function Index

B

BC 16

O

ospeed 11

P

PC 11

T

`tgetent` 3
`tgetflag` 5
`tgetnum` 5
`tgetstr` 5
`tgoto` 16
`tparam` 15
`tputs` 10

U

UP 16

Capability Index

A

ae 43
al 34
AL 35
am 29
as 42

B

bc 29
bl 44
bs 29
bt 28
bw 27

C

CC 24
cd 33
ce 33
ch 28
cl 33
cm 26
CM 27
co 25
cr 26
cs 31
cS 32
ct 48
cv 28

D

da 32
db 32
dB 49
dc 39
dC 48
DC 39
dF 49
dl 34
DL 35
dm 39
dN 48
do 27
DO 27
ds 49
dT 49

E

ec 33
ed 39
ei 37
eo 23
es 49

F

ff 28
fs 49

G

gn 24

H

hc 24
hd 50
ho 26
hs 49
hu 50
hz 24

I

i1 47
i3 47
ic 37
IC 37
if 47
im 37
in 38
ip 37
is 47
it 48

K

K1...K5 45
k1...k9 45
kA...kT 45
ka...ku 45
km 47

L

l0...l9 45
le 27
LE 27
li 25
ll 26
lm 32
LP 30

M

mb 42
md 42
me 42
mh 42
mi 38
mk 42
mm 47
mo 47
mp 42
mr 42
ms 42, 43

N

nc 28
nd 27
NF 48
nl 29
ns 32
nw 27

O

os 23

P

pb 48
pc 48
pf 50
po 50
pO 50
ps 50

R

rc 28
RI 27
rp 24
rs 48

S

sa 43
sc 28
se 42
sf 31
SF 31
sg 42
so 42
sr 31
SR 31
st 48

T

ta 28
te 47
ti 47
ts 49

U

uc 44
ue 43
ug 43
ul 44
up 27
UP 27
us 43

V

vb 44
ve 44
vi 44
vs 44

W

`wi` 33
`ws` 50

X

`xb` 25
`xn` 30
`xs` 42
`xt` 29, 42

Concept Index

%

% 12

A

appearance modes 39

B

bell 44

C

clearing the screen 33
command character 24
cursor motion 25

D

delete character 36
delete line 34
delete mode 36
description format 17

E

erasing 33

G

generic terminal type 24

H

home position 26

I

inheritance 20
initialization 47
insert character 36
insert line 34
insert mode 36

L

line speed 10

M

magic cookie 39
meta key 47

N

names of terminal types 19

O

overstrike 23

P

padding 8, 48
parameters 12
printer 50

R

repeat output 24
reset 47

S

screen size 19, 25
scrolling 30
standout 39
status line 49
Superbee 25

T

tab stops 47
termcap 1
terminal flags (kernel) 7

U

underlining 43

V

visibility 44
visible bell 44

W

window 33
wrapping 19, 29

Table of Contents

Introduction **1**

1 The Termcap Library **3**

1.1 Preparing to Use the Termcap Library 3
1.2 Finding a Terminal Description: `tgetent` 3
1.3 Interrogating the Terminal Description 5
1.4 Initialization for Use of Termcap 7
1.5 Padding 8
1.5.1 Why Pad, and How 8
1.5.2 When There Is Not Enough Padding 9
1.5.3 Specifying Padding in a Terminal Description 10
1.5.4 Performing Padding with `tputs` 10
1.6 Filling In Parameters 12
1.6.1 Describing the Encoding 12
1.6.2 Sending Display Commands with Parameters 15
1.6.2.1 `tparam` 15
1.6.2.2 `tgoto` 16

2 The Format of the Data Base **17**

2.1 Terminal Description Format 17
2.2 Writing the Capabilities 18
2.3 Terminal Type Name Conventions 19
2.4 Inheriting from Related Descriptions 20
2.5 When Changes in the Data Base Take Effect 21

3 Definitions of the Terminal Capabilities **23**

3.1 Basic Characteristics 23
3.2 Screen Size 25
3.3 Cursor Motion 25
3.4 Wrapping 29
3.5 Scrolling 30
3.6 Windows 33
3.7 Clearing Parts of the Screen 33
3.8 Insert/Delete Line 34
3.9 Insert/Delete Character 36
3.10 Standout and Appearance Modes 39
3.11 Underlining 43
3.12 Cursor Visibility 44
3.13 Bell 44

3.14 Keypad and Function Keys 44
3.15 Meta Key 47
3.16 Initialization 47
3.17 Padding Capabilities 48
3.18 Status Line 49
3.19 Half-Line Motion 50
3.20 Controlling Printers Attached to Terminals 50

4 Summary of Capability Names 53

Variable and Function Index 59

Capability Index 61

Concept Index 65