New Features in Curses and Terminfo

Pavel Curtis

1. Introduction

This document describes new features that are being added to the Berkeley **curses** subroutine package. It also describes the new **terminfo** database, which replaces the Berkeley **termcap** database. The emphasis is on the new features.

2. New Features in Curses

This section describes the enhancements to curses. Briefly, the enhancements are:

a. Curses is smarter. It can take advantage of insert/delete line/character. (By default, it will not use insert/delete line. See *idlok()*.)

b. Curses uses the new **terminfo** data base, as described in the next section.

c. Curses works on more terminals.

d. It is possible to use more than one terminal at a time.

e. Video attributes can be displayed in any combination.

f. Curses handles terminals with the ‘‘magic cookie’’ video attribute glitch.

g. The function and arrow keys on terminals can be input as though they were a single character.

h. There is a user accessible scrolling region, like the DEC VT100.

i. If the programmer restricts his code to a subset of the full curses, the **MINICURSES** version can be used, which is smaller and faster.

j. Two routines are provided for setting the tty bits to the proper state for shell escapes and control-Z suspensions.

k. On systems that support it (currently only 4.1BSD), if the user types something during an update, the update will stop, pending a future update. This is useful when the user hits several keys, each of which causes a good deal of output.

l. The routine **getstr()** is smarter - it handles the users erase and kill characters, and echos its input.

m. The function **longname()** is now useful and actually works.

n. Nodelay mode allows ‘‘real time’’ programs to be written with the same interface on both systems. Setting the flag causes **getch** to return -1 if no input is waiting.

o. Several useful routines are provided to enhance portability.

2.1. Curses is Smarter

The algorithm used by curses has been replaced with an algorithm that takes into account insert and delete line and character functions, if available, in the terminal. By default, curses will not use insert/delete line. This was not done for performance reasons, since there is no speed penalty involved. Rather, it was found that some programs do not need this facility, and that if curses uses insert/delete line, the result on the screen can be visually annoying. Since most simple programs using curses do not need this, and since the old curses did not use it, the default is to avoid insert/delete line. Call the routine

idlok(stdscr, TRUE);

to enable insert/delete line, if your application needs it. Insert/delete character is always considered.

2.2. Additional Terminals

Curses works on a larger class of terminals than the previous version. Terminfo is able to address the cursor on more kinds of terminals. Curses will work even if absolute cursor addressing is not possible, as long as the cursor can be moved from any location to any other location. It considers local motions, parameterized motions, home, and carriage return.

Curses is still aimed at full duplex, alphanumeric, video terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bitmapped terminals.

Curses handles terminals with the ''magic cookie glitch'' in their video attributes.* This glitch means that a change in video attributes is implemented by storing a ''magic cookie'' in a location on the screen. This ''cookie'' takes up a space, preventing an exact implementation of what the programmer wanted. Curses takes the extra space into account, and moves part of the line to the right, as necessary. In some cases, this will unavoidably result in losing text from the right hand edge of the screen. Existing spaces are taken advantage of.

2.3. Multiple Terminals

Some applications need to display text on more than one terminal, controlled by the same process. Even if the terminals are different, the new curses can handle this.

All information about the current terminal is kept in a global variable

struct screen *SP;

Although the screen structure is hidden from the user, the C compiler will accept declarations of variables which are pointers. The user program should declare one screen pointer variable for each terminal it wishes to handle. The routine

struct screen *
newterm(type, fd)
char *type;
FILE *fp;

will set up a new terminal of the given terminal type which does output on file pointer fp. A call to *initscr()* is essentially *newterm*(*getenv*(''TERM''), **stdout**). A program wishing to use more than one terminal should use *newterm()* for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call

struct screen *
set_term(term)
struct screen *term;

The old value of SP will be returned. You should not assign directly to SP because certain other global variables must also be changed.

All curses routines always affect the current terminal. To handle several terminals, switch to each one in turn with *set_term()*, and then access it. Each terminal must be set up with *newterm()*, and closed down with *endwin()*.

2.4. Video Attributes

Video attributes can be displayed in any combination on terminals with this capability. They are treated as an extension of the standout capability, which is still present.

Each character position on the screen has 16 bits of information associated with it. 7 of these bits are the character to be displayed, leaving separate bits for 9 video attributes. These bits are used for standout, underline, reverse video, blink, dim, bold, blank, protect, and alternate character set. Standout is taken to be whatever highlighting works best on the terminal, and should be used by any program that does not need specific or combined attributes. Underlining, reverse video, blink, dim, and bold are the usual video

* This feature is not supported in the current test release. It will be implemented in the official distribution.

attributes. Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set are dependent on the particular terminal. The use of these last three bits is subject to change and not recommended.

The routines to use these attributes include

attrset(attrs)	*wattrset*(attrs)
attron(attrs)	*wattron*(attrs)
attroff(attrs)	*wattroff*(attrs)
standout()	*wstandout*()
standend()	*wstandend*()

Attributes, if given, can be any combination of A_STANDOUT, A_UNDERLINE, A_REVERSE, A_BLINK, A_DIM, A_BOLD, A_INVIS, A_PROTECT, and A_ALTCHARSET. These constants, defined in curses.h, can be combined with the C | (or) operator to get multiple attributes. *Attrset()* sets the current attributes to the given *attr*; *attron()* turns on the given *attrs* in addition to any attributes that are already on; *attroff()* turns off the given attributes, without affecting any others. *standout()* and *standend()* are equivalent to *attron*(A_STANDOUT) and *attroff*(A_STANDOUT).

Since standout is stored in the 8th bit of the text byte, it is possible to recompile curses so that only 8 bits are stored for each character, making a smaller curses, and still be able to use standout. Also, programs that restrict themselves to the routines *standout()* and *standend()* will work with both the new and old curses.

If the particular terminal does not have the particular attribute or combination requested, curses will attempt to use some other attribute in its place. If the terminal has no highlighting at all, all attributes will be ignored.

2.5. Function Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these terminals send differs from terminal to terminal. Curses allows the programmer to handle these keys.

A program using function keys should turn on the keypad by calling

keypad(**stdscr**, TRUE)

at initialization. This will cause special characters to be passed through to the program by the function *getch()*. These keys have constants which are defined in curses.h. They have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program using function keys should avoid using the ESCAPE key, since most sequences start with escape, creating an ambiguity. Curses will set a one second alarm to deal with this ambiguity, which will cause delayed response to the escape key. It is a good idea to avoid escape in any case, since there is eventually pressure for nearly *any* screen oriented program to accept arrow key input.

2.6. Scrolling Region

There is a user accessible scrolling region, like the DEC VT100. Normally, it is set to the entire window, but the calls

setscrreg(top, bot)
wsetscrreg(win, top, bot)

set the scrolling region for **stdscr** or the given window to any combination of top and bottom margins. If scrolling has been enabled with *scrollok*, scrolling will take place only within that window. See the *Curses Reference Manual* for the detailed semantics of this construct.

2.7. Mini-Curses*

* This feature is not supported in the current test release. It will be implemented in the official distribution.

The new curses is bigger than the old one, and has to copy from the current window to an internal screen image for every call to *refresh()*. If the programmer is only interested in screen output optimization, and does not want the windowing or input functions, an interface to the lower level routines is available. This will make the program somewhat smaller and faster. The interface is a subset of full curses, so that conversion between the levels is not necessary to switch from mini-curses to full curses.

The subset mainly requires you to avoid use of more than the one window **stdscr**. Thus, all functions beginning with ''w'' are generally undefined. Certain high level functions that are convenient but not essential are also not available, including *printw()* and *scanw()* Also, the input routine *getch()* cannot be used with mini-curses. Features implemented at a low level, such as use of hardware insert/delete line and video attributes, are available in both versions. Also, mode setting routines such as *cbreak()* and *noecho()* are allowed. See the manual page for the exact list of routines allowed with mini-curses.

To access mini-curses, add **-DMINICURSES** to the CFLAGS in your makefile. If you ask for routines that are not in the subset, the loader will print error messages such as

```
Undefined:
no_getch
no_waddch
```

to tell you that the routines *getch()* and *waddch()* were used but are not available in the subset. Since the preprocessor is involved in the implementation of mini-curses, you must recompile the entire program if you change from one version to the other. Similarly, programs compiled with the old curses must be recompiled for the new curses.

2.8. TTY Mode Functions

In addition to the save/restore routines *savetty()* and *resetty()*, standard routines are available for going into and out of normal tty mode. These routines are *resetterm()*, which puts the terminal back in the mode it was in when curses was started, and *fixterm()*, which undoes the effects of *resetterm()*, that is, restores the ''current curses mode''. *endwin()* automatically calls *resetterm()*, and the routine to handle control-Z (on 4.1BSD systems with process control) also uses *resetterm()* and *fixterm()*. The programmer should use these routines before and after shell escapes, and also if he writes his own routine to handle control-Z. These routines are also available at the *terminfo* level.

2.9. Typeahead Check*

On systems that support it (current only 4.1BSD), if the user types something during an update, the update will stop, pending a future update. This is useful when the user rapidly hits several keys, each of which causes a good deal of output. This feature is automatic and cannot be disabled.

2.10. Getstr()

The routine *getstr()* is smarter. The semantics are slightly different from the old *getstr()*, but no incompatibilities are anticipated. No matter what the setting of *echo* is, strings typed in here are echoed at the current cursor location. The users erase and kill characters are understood and handled. This makes it unnecessary for an interactive program to deal with erase, kill, and echoing when the user is typing a line of text.

2.11. Longname()

The function *longname()* is now useful and actually works. The previous version required the programmer to call *tgetent()* directly and pass the resulting string, along with a buffer, to *longname()*. The string actually returned was the second alias for the terminal, not the long name.

The new *longname()* function does not take any arguments. It returns a pointer to a static area containing the actual long name of the terminal. No call to *tgetent()* is needed, in fact, that routine no longer exists.

---

* This feature is not supported in the current test release. It will be implemented in the official distribution.

2.12. Nodelay Mode

The call

nodelay(**stdscr**, TRUE)

will put the terminal in ‘‘nodelay mode’’. While in this mode, any call to *getch()* will return -1 if there is nothing waiting to be read immediately. This is useful for writing programs requiring ‘‘real time’’ behavior where the user watches action on the screen and presses a key when he wants something to happen. For example, the cursor can be moving across the screen, and the user can press an arrow key to change direction. This mode is especially useful for games such as PacMan and Space Invaders.

2.13. Portability

Several useful routines are provided to enhance portability. While these routines do not directly relate to terminal handling, their implementation is different from system to system, and the differences can be isolated from the user program by including them in curses.

Functions *erasechar()* and *killchar()* return the characters which erase one character, and kill the entire input line, respectively. The function *baudrate()* will return the current baud rate, as an integer. (For example, at 9600 baud, the integer 9600 will be returned, not the value B9600 from <sgtty.h>.) The routine *flushinp()* will cause all typeahead to be thrown away.

2.14. Features No Longer Supported

In general, an effort has been made to support old features where possible. However, there are some features of the old curses that cannot be supported, due to the change to terminfo, or due to other miscelaneous causes.

The old curses defined a number of two letter variables, such as CM, containing termcap capabilities. These variables are no longer accessible to the user. In general, their semantics are different, as are their names. A program using primarily these variables is really written at the termcap level. Also unavailable are the related variables NONL, GT, and UPPERCASE.

Such programs should be recoded to avoid these capabilities, if at all possible, instead using the higher level curses functions. If this is not possible, recode at the terminfo level. A program making only light use can probably be easily changed to avoid these variables completely. A program at the terminfo level that only needs motion optimization should probably still be recoded to use the high level routines, in order to work on more terminals. If this is not possible, recode at the terminfo level, continuing to use *mvcur()*, which is still supported. It is not necessary to call *mvcur()* to move to the lower left corner of the screen before calling *endwin()*.

Some programs (notably rogue) use varibles in <curses.h> which begin with an underline. Use of these variables and fields is to be avoided. Most of the internal structures used by curses are hidden from the user. The variables _tty and _tty_ch are no longer accessible. (Since _tty was a version 7 dependent structure, it was not portable to use it anyway.) Useful fields, such as the erase and kill characters, and the baud rate, can be discovered using the portable functions described above.

3. Termlib-Level Changes

The termcap(3) (termlib) library has been consolidated with the curses(3) library to form a new curses(3) library. The termlib level is very different in the new version. The routines *tgetent()*, *tgetnum()*, *tgetstr()*, and *tgetflag()* are gone. Initialization is instead done by calling

setupterm(termtype, filedes, errret)
char *termtype;
int filedes;
int *errret;

This routine takes care of all reading in of capabilities, and any other system dependent initialization. The terminal type can be passed as 0, causing *setupterm()* to use *getenv*(‘‘TERM’’) as a default. *errret* is a pointer to an integer used to return a status value. The value returned is 0 if there is no such terminal type,

1 if all went well, or -1 for some trouble. A null pointer can be passed for this value, telling *setupterm()* to print an error message and exit if the terminal cannot be found.

When exiting, or calling a shell escape, the user program should call *resetterm()* to restore the tty modes. After the shell escape, *fixterm()* can be called to set the tty modes back to their internal settings. These calls are now **required**, since they perform system dependent processing. They do not output the **enter_ca_mode** and **exit_ca_mode** strings (**ti** and **te** in termcap) but should be called at the same times. *Setupterm()* calls *fixterm()*.

tgoto() has been replaced by *tparm()*, which is a more powerful parameterized string mechanism. The *tgoto()* routine is still available for compatibility. *tputs()* is unchanged.

The external variables **UP**, **BC**, **PC**, and **ospeed** no longer exist. The programmer need not worry about these, as their function is now handled internally.

4. Changes from Termcap to Terminfo

This section describes the extensions in terminfo that were not present in termcap, and the incompatible changes that were made. It is intended for a programmer or termcap author who is familiar with termcap and wishes to become familiar with terminfo. The emphasis is on the database, not on the programmer interface.

4.1. Syntax

The first thing you will notice upon scanning terminfo is that it looks cosmetically different from termcap. All the backslashes are gone from ends of lines. Fields are separated with commas instead of colons, and white space after the commas makes them more readable. Continuation lines are now defined as lines beginning with a blank or tab, not lines following a backslash. These changes make terminfo easier to read and to modify.

4.2. Names

The names of the capabilities are no longer limited to two letters. There is no longer a hard limit to the names, but an informal limit of 5 characters is used. Since the two letter limit is gone, many of the capabilities have been renamed. They now correspond as closely as possible the the ANSI standard X3.64. While learning the new set of names will be tricky at first, eventually life will be simpler, since most new terminals use the ANSI abbreviations.

4.3. Defaults

A change that is perhaps not so obvious is that certain defaults are no longer implied. In termcap, \r was assumed to be a carriage return unless **nc** was present, indicating that it did not work, or **cr** was present, indicating an alternative. In terminfo, if **cr** is present, the string so given works, otherwise it should be assumed *not* to work. The **bs** and **bc** capabilities are replaced by **cub** and **cub1**. (The former takes a parameter, moving left that many spaces. The latter is probably more common in terminals and moves left one space.) **nl** (linefeed) has been split into two functions: **cud1** (moves the cursor down one line) and **ind** (scroll forward). **cud1** applies when the cursor is not on the bottom line, **ind** applies when it is on the bottom line. The bell capability is now explicitly given as **bel**.

4.4. Compilation

The terminfo database is compiled, unlike termcap. This means that a terminfo source file (describing some set of terminals) is processed by the terminfo compiler, producing a binary description of the terminal in a file under /etc/term. The setupterm routine reads in this file.

The advantage to compilation is that starting up a program using terminfo is faster. It is no longer necessary to carry around the variable TERMCAP in the environment. It is actually faster to start up a compiled terminfo *without* the environment variable, than it is to start up an uncompiled termcap *with* the environment variable. The increase in speed comes partly from not having to skip past other terminal descriptions, and partly from the compiler having sorted the capabilities into order so that a linear scan can read them in. (The termcap initialization algorithm is quadratic on the size of the capability. The more

capabilities you are interested in, the worse this gets. It had gotten to the point where it took 2 CPU seconds on a VAX 11/750 to start up a process using an uncompiled terminfo!)

There exists an environment variable TERMINFO which is taken by the compiler to be the destination directory of the new object files. It is also used by *setupterm()* to find an entry for a given terminal. First it looks in the directory given in TERMINFO and, if not found there, checks /etc/term. **Note**, however, that, unlike the old TERMCAP variable, you may not put the source for an entry in the TERMINFO variable. *All* terminfo entries must be compiled.

4.5. Parameterised Strings

The old *tgoto()* mechanism, which was designed for cursor addressing only, has been replaced by a more general parameter mechanism, accessed through the function *tparm()*. Since the parameters are not compatible in the terminfo database, a termcap **cm** *description must be converted manually to terminfo.*

The new mechanism is based on a stack. % operations are used to push parameters and constants onto the stack, do arithmetic and other operations on the top of the stack, and print out values in various formats. This makes it possible to handle a larger class of terminals, such as the AED 512, which addresses the cursor in terms of pixels, not character positions, and the TEC scope, which numbers the rows and columns from the lower right hand corner of the screen. Any number of parameters from 1 to 9 is possible, whereas *tgoto()* allowed only two parameters. If-then-else testing is possible, as is storage in a limited number of variables. There is no provision for loops or printing strings in any format other than %s. The full details are described in terminfo(5).

A few brief examples are included here to show common conversions. For more examples, compare the termcap **cm** and terminfo **cup** entries for your favorite terminal. ‘‘%+ ’’ (add space and print as a character) would be treated as ‘‘%p1%’ ’%+%c’’, that is, push the first parameter, push space, add the top two numbers on the stack, and output the top item on the stack using character (%c) format. (Of course, for the second parameter, the %p1 must be changed to %p2.) ‘‘%.’’ (print as a character) would be ‘‘%p1%c’’. ‘‘%d’’ (print in decimal) would be ‘‘%p1%d’’. As with *tgoto()*, characters standing by themselves (no % sign) are output as is.

4.6. More Capabilities

There are a number of new capabilities. The set of new capabilities may vary, depending on the version of termcap you are used to. It is probably worthwhile to read terminfo(5) for a complete list. This section describes capabilities new to terminfo that were never put in termcap.

There are provisions for dealing with more video attributes. Termcap had strings to turn on and off standout and underline modes. Terminfo has these and several more. There are strings to turn on bold, inverse video, blinking, dim, protected, and blanking. Rather than have separate string for turning off each of these, a single capability: **sgr0**, turns them all off.

The effect of turning on more than one attribute at a time with the separate strings is undefined. A parameterized string, **sgr**, can be used to turn them on in combination.

More function keys are defined now. There are provisions for f0 through f10 as well as keys such as erase, insert mode, insert line, delete line, delete character, print, and so on. All of these keys can be accessed through curses as if they were single characters. Also, **vi** version 3.8 has default meanings for many of them.

Several new uses are made of parameterized strings. For example, capabilities exist to move the cursor to a particular column in the current row, a particular row in the current column, and to move left, right, up, or down a given number of spaces. These capabilities make a big difference on some terminals, such as the Tektronix 4025. Also, column addressing is useful for filters that do not know what row they are in, or as a shorter form of cursor addressing when the target is in the same row.

There are now capabilities to turn on and off a local printer, and to print the current page. Also, there are provisions for moving the cursor to and from a status line. These capabilities can be used by a background status program, such as *sysline*, to keep status information in the status line without bothering foreground processes. This only works on terminals with a writable status line, such as the h19 or tvi950, or on

terminals where one can be simulated, such as the hp2626, vt100, or ambassador, by allocating one of the ordinary screen lines for a status line.

4.7. How to Convert from Termcap to Terminfo

This section is intended for programmers who need to convert programs that use termcap to the new terminfo database. It describes the steps needed for the conversion.

If you must make the conversion, you are strongly urged to convert to curses, rather than converting to terminfo. The curses interface is higher level and will probably do a better job of optimizing your output. Your program will work on a wider range of terminals if you use curses. It will also become more portable. The effort to convert to curses is probably about the same as to convert to terminfo.

There are some programs for which curses is not a possibility. Curses takes over the CRT screen, and this implies initially clearing the screen. For some programs, such as filters, this may not make sense. Also, if you are writing a special purpose program which uses some terminfo capability that curses does not use, it will probably be necessary to use the terminfo level interface.

4.8. Conversion

The first step is to include the headers <curses.h> and <term.h> (in that order). These headers will bring into existence a set of ''variables'' (actually macros) that contain values of capabilities. For example, the macro **cursor_address** will be defined, replacing the termcap **cm** capability. You should remove the declarations for all variables you use for capabilities returned by *tgetflag()*, *tgetnum()*, and *tgetstr()*.

The most difficult step is that all variables removed in the previous step must be renamed the standard names. For example, if you stored **cm** in the variable CM, you would change *tputs*(*tgoto*(**CM**, i, j), 1, outch) to *tputs*(*tgoto*(**cursor_address**, i, j), 1, outch). Consult terminfo(5) for a list of standard names. A **sed** script is often useful for this step. Care must be taken to avoid mention of the variable as part of a longer word (a version of **sed** supporting the **ex** <word> convention is useful here.) Also, you should proofread the results, since sometimes comments and strings get substituted that shouldn't have been.

Remove all your termcap initialization code. This code typically calls *tgetent()*, *tgetstr()*, *tgetflag()*, and *tgetnum()*. You can also remove declarations used only for this initialization, usually including buffers for the entry and string values. Replace it with a single call to *setupterm*(0, 1, 0). This call will never return if something goes wrong, that is, if there is no $TERM in the environment or there is no such terminal, the routine will print an error and exit. If you need an error indication passed back for more sophisticated error recovery, pass an integer variable in the third parameter, i.e. setupterm(0, 1, &i). The value returned in **i** will be the same as that previously returned by *tgetent()*. Other more sophisticated calls to *setupterm()* are possible, see the documentation if some terminal other than $TERM or some file descriptor other than **stdout** are involved.

Before the program exits, insert a call to *resetterm()*. This will restore the tty modes to their state before setupterm was called, and do any other system dependent exit processing. This routine can also be called before a shell escape, you should call *fixterm()* after the shell escape to restore the tty modes to those needed by terminfo. (Currently *setupterm()* will turn off the XTABS bit in the tty driver, since some terminals need to send control I for escape sequences. You should be sure to expand any tabs in your software if necessary.)

From the programmers viewpoint, the routine *tputs()* is exactly as in termcap. The padding syntax in the capability is different, but this only affects the capabilities in the terminfo database. No change to a program will be needed for *tputs()*.

The *tgoto()* routine is kept around for upward compatibility, but you should probably replace calls to *tgoto()* by calls to *tparm()*. The call *tgoto*(cap, y, x) will call *tparm*(cap, x, y). Note that the order of the last two arguments is reversed - it was backwards in *tgoto()* from what it probably should have been. In addition to the capability, *tparm()* can now take up to nine parameters, or as few as one.

If you use certain capabilities, there are a few convention changes you should be aware of. These do not affect very many programs, but will require some minor recoding of a few programs. In termcap, the cursor is moved left by control-H if **bs** is present, otherwise, if **bc** is present, that character is used. In

terminfo, the cursor is moved left with **cub1**, if present, or by **cub**, if present. If neither is there, there is no implied control-H. Similarly, termcap assumed that control-M was carriage return unless **nc** or **cr** was specified. In terminfo, carriage return is always the string specified by **cr**, and if not present, there is no carriage return capability. In termcap, linefeed is assumed to both move the cursor down (if it is not on the bottom line) and to scroll one line (if it is on the bottom line), unless **ns** is present. **sf** and **do** capabilities were present but little used, and some software assumed that **sf** worked with the cursor anywhere on the screen. In terminfo, there is no implied linefeed - moving the cursor down is done with **cud1** or **cud** and scrolling is done with **ind**. **ind** is only defined when the cursor is at the bottom of the screen. Finally, the implied control G used to ring the bell unless **vb** was present has been replaced with an explicit **bel**.

Replace references in your makefile from -ltermcap or -ltermlib with references to -lcurses.

Now recompile your program. It should run properly using terminfo.

4.9. Space Conditions

The expansion of a macro name into a structure reference will probably make your program a bit bigger. If space is a problem, one thing you can do is add **-DSINGLE** to the CFLAGS in your makefile. This causes the macros to expand to a static reference instead of a dynamic reference, resulting in smaller code. It cannot be used if you intend to involve more than one terminal from a single process. Since very few programs talk to two terminals at once, it is almost always safe to define SINGLE.

If your program was pushing the limit on a small machine, it may not fit with terminfo unless you trim it down some. While the startup routines are faster, they tend to generate larger code than those of termcap. Also, *tputs()* and *tparm()* are more sophisticated and larger.