.

Screen Updating and Cursor Movement Optimization: A Library Package

Kenneth C. R. C. Arnold

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This document describes a package of C library functions which allow the user to:

1) update a screen with reasonable optimization,

2) get input from the terminal in a screen-oriented fashion, and

3) independent from the above, move the cursor optimally from one point to another.

These routines all use the /etc/termcap database to describe the capabilities of the terminal.

Acknowledgements

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

Screen Package

1. Overview

In making available the generalized terminal descriptions in /etc/termcap, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

window: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal: Sometimes called terminal screen. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special screen:

screen: This is a subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, stdscr, is automatically provided for the programmer.

1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

```
#include <curses.h>
```

at the top of the program source. The header file

<curses.h> needs to include <sgtty.h>, so the one should not do so oneself[1]. Also, compilations should have the following form:

cc [flags] file ... -lcurses -ltermlib

1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named WINDOW is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called curscr for current screen) is a screen image of what the terminal currently looks like. Another screen (called stdscr, for standard screen) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine refresh() (or wrefresh() if the window is not stdscr) is called. refresh() makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window does not change the terminal. Actual updates to the terminal screen are made only by calling refresh() or wrefresh(). This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: curscr, which knows what the terminal looks like, and stdscr, which is what the pro-

---

[1] The screen package also uses the Standard I/O library, so <curses.h> includes <stdio.h>. It is redundant (but harmless) for the programmer to do it, too.

grammer wants the terminal to look like next. The user should never really access curscr directly. Changes should be made to the appropriate screen, and then the routine refresh() (or wrefresh()) should be called.

Many functions are set up to deal with stdscr as a default screen. For example, to add a character to stdscr, one calls addch() with the desired character. If a different window is to be used, the routine waddch() (for window-specific addch()) is provided[2]. This convention of prepending function names with a "w" when they are to be applied to specific windows is consistent. The only routines which do not do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines move() and wmove() are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsyness, most I/O routines can be preceded by the prefix "mv" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (win) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

---

[2] Actually, addch() is really a "#define" macro with arguments, as are most of the "functions" which deal with stdscr as a default.

.

Screen Package

type	name	description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in Def term as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	storage class ``register'' (e.g., reg int i;)
bool	boolean type, actually a ``char'' (e.g., bool doneit;)
TRUE	boolean ``true'' flag (1).
FALSE	boolean ``false'' flag (0).

3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to stdscr. All instructions will work on any window, with changing the function name and parameters as mentioned above.

3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for curscr and stdscr must be allocated. These functions are performed by initscr(). Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, initscr() returns ERR. initscr() must always be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either curscr or stdscr are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like nl() and crmode() should be called after initscr().

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use scrollok(). If you want the cursor to

be left after the last change, use leaveok(). If this isn't done, refresh() will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions newwin() and subwin(). delwin() will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables LINES and COLS to be what you want, and then call initscr(). This is best done before, but can be done either before or after, the first call to initscr(), as it will always delete any existing stdscr and/or curscr before creating new ones.

3.2. The Nitty-Gritty

3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are addch() and move(). addch() adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, i.e., printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. move() changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into mvaddch() to do both things in one fell swoop.

The other output functions, such as addstr() and printw(), all call addch() to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call refresh().

In order to optimize finding changes, refresh() assumes that any part of the window not changed since the last refresh() of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine touchwin() is provided to make it look like the entire window has been changed, thus making refresh() check the whole subsection of the terminal for changes.

If you call wrefresh() with curscr, it will make the screen look like curscr thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

3.2.2. Input

Input is essentially a mirror image of output. The complementary function to addch() is getch() which, if echo

is set, will call addch() to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, getch() sets it to be cbreak, and then reads in the character.

3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in getttmode() and setterm(), which are called by initscr(). In order to clean up after the routines, the routine endwin() is provided. It restores tty modes to what they were when initscr() was first called. Thus, anytime after the call to initscr, endwin() should be called before exiting.

4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as eye and vi[3]. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead.

For such applications, such as some "crt hacks"[4] and optimizing cat(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

---

[3] Eye actually uses these functions, vi does not.

[4] Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as rocket and gun.

4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are[5]. The /etc/termcap database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from vi and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, HO is a string which moves the cursor to the "home" position[6]. As there are two types of variables involving ttys, there are two routines. The first, gettmode(), sets some variables based upon the tty modes accessed by gtty(2) and stty(2) The second, setterm(), a larger task by reading in the descriptions from the /etc/termcap database. This is the way these routines are used by initscr():

```
if (isatty(0)) {
        gettmode();
        if (sp=getenv("TERM"))
                setterm(sp);
}
else
        setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a termi-

nal[7]. If it is, gettmode() sets the terminal description modes from a gtty(2) getenv() is then called to get the name of the terminal, and that value (if there is one) is passed to setterm(), which reads in the variables from /etc/termcap associated with that terminal. (getenv() returns a pointer to a string containing the name of the terminal, which we save in the character pointer sp.) If isatty() returns

---

[5] If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

[6] These names are identical to those variables used in the /etc/termcap database to describe each capability. See Appendix A for a complete list of those read, and termcap(5) for a full description.

[7] isatty() is defined in the default C library function routines. It does a gtty(2) on the descriptor and checks the return value.

false, the default terminal Def term is used. The TI and VS sequences initialize the terminal (puts() is a macro which uses tputs() (see termcap(3)) to put out a string). It is these things which endwin() undoes.

4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it[8]. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor vi uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using gettmode() and setterm() to get the terminal descriptions, the function mvcur() deals with this task. It usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function tgoto() from the termlib(7) routines, or you can tell mvcur() that you are impossibly far away, like Cleveland. For example, to absolutely

address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

5. The Functions

In the following definitions, "[*]" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as addch(), it will show up as it's "w" counterpart. The arguments are given to show the order and type of each. Their names are

---

[8] Actually, it can be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

not mandatory, just suggestive.

5.1. Output Functions

```
addch(ch) [*]
char       ch;

waddch(win, ch)
WINDOW     *win;
char       ch;
```

Add the character ch on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

```
addstr(str) [*]
char       *str;

waddstr(win, str)
```

```
WINDOW     *win;
char       *str;
```

Add the string pointed to by str on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

```
box(win, vert, hor)
WINDOW     *win;
char       vert, hor;
```

Draws a box around the window using vert as the character for drawing the vertical sides, and hor for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

.

Screen Package

clear() [*]

```
wclear(win)
WINDOW     *win;
```

Resets the entire window to blanks. If win is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next refresh() call. This also moves the current (y, x) co-ordinates to (0, 0).

```
clearok(scr, boolf) [*]
WINDOW     *scr;
bool       boolf;
```

Sets the clear flag for the screen scr. If boolf is TRUE, this will force a clear-screen to be printed on the next refresh(), or stop it from doing so if boolf is FALSE. This only works on screens, and, unlike clear(), does not alter the contents of the screen. If scr is curscr, the next refresh() call will cause a clear-screen, even if the window passed to refresh() is not a screen.

clrtobot() [*]

wclrtobot(win)
WINDOW *win;

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "mv" command.

clrtoeol() [*]

wclrtoeol(win)
WINDOW *win;

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "mv" command.

delch()

wdelch(win)
WINDOW *win;

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
WINDOW *win;

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase() [*]

werase(win)
WINDOW *win;

Erases the window to blanks without setting the clear flag. This is analagous to clear(), except that it

never causes a clear-screen sequence to be generated on a refresh(). This has no associated "mv" command.

```
insch(c)
char        c;

winsch(win, c)
WINDOW      *win;
char        c;
```

Insert c at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

```
insertln()

winsertln(win)
WINDOW      *win;
```


Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged. This returns ERR if it would cause the screen to scroll illegally.

```
move(y, x) [*]
int         y, x;

wmove(win, y, x)
WINDOW      *win;
int         y, x;
```

Change the current (y, x) co-ordinates of the window to (y, x). This returns ERR if it would cause the screen to scroll illegally.

```
overlay(win1, win2)
WINDOW      *win1, *win2;
```

Overlay win1 on win2. The contents of win1, insofar as they fit, are placed on win2 at their starting (y, x) co-ordinates. This is done non-destructively, i.e.,

blanks on win1 leave the contents of the space on win2 untouched.

overwrite(win1, win2)
WINDOW *win1, *win2;

Overwrite win1 on win2. The contents of win1, insofar as they fit, are placed on win2 at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on win1 become blank on win2.

printw(fmt, arg1, arg2, ...)
char *fmt;

wprintw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;

Performs a printf() on the window starting at the current (y, x) co-ordinates. It uses addstr() to add the string on the window. It is often advisable to use

the field width options of printf() to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() [*]

wrefresh(win)
WINDOW *win;

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

standout() [*]

wstandout(win)
WINDOW *win;

standend() [*]

wstandend(win)

WINDOW *win;

Start and stop putting characters onto win in standout mode. standout() causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). standend() stops this. The sequences SO and SE (or US and UE if they are not defined) are used (see Appendix A).

5.2. Input Functions

crmode() [*]

nocrmode() [*]

Set or unset the terminal to/from cbreak mode.

echo() [*]

noecho() [*]

Sets the terminal to echo or not echo characters.

getch() [*]

wgetch(win)
WINDOW *win;

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If noecho has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of noecho, cbreak, or rawmode set. If you do not set one, whatever routine you call to read characters will set cbreak for you, and then reset to the original mode when finished.

getstr(str) [*]
char *str;

wgetstr(win, str)

WINDOW *win;
char *str;

Get a string through the window and put it in the location pointed to by str, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls getch() (or wgetch(win)) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

raw() [*]

noraw() [*]

Set or unset the terminal to/from raw mode. On version 7 UNIX[9] this also turns of newline mapping (see nl()).

---

[9] UNIX is a trademark of Bell Laboratories.

scanw(fmt, arg1, arg2, ...)
char *fmt;

wscanw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;

Perform a scanf() through the window using fmt. It does this using consecutive getch()'s (or wgetch(win)'s). This returns ERR if it would cause the screen to scroll illegally.

5.3. Miscellaneous Functions

delwin(win)
WINDOW *win;

Deletes the window from existence. All resources are freed for future use by calloc(3). If a window has a subwin() allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should

be deleted before their outer windows are.

endwin()

Finish up window routines before exit. This restores the terminal to the state it was before initscr() (or gettmode() and setterm()) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via signal(2).

getyx(win, y, x) [*]
WINDOW *win;
int y, x;

Puts the current (y, x) co-ordinates of win in the variables y and x. Since it is a macro, not a function, you do not pass the address of y and x.

inch() [*]

winch(win) [*]

Screen Package

WINDOW *win;

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "mv" command.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by Def term (initialy "dumb"). If the boolean My term is true, Def term is always used.

leaveok(win, boolf) [*]
WINDOW *win;

bool boolf;

Sets the boolean flag for leaving the cursor after the last change. If boolf is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for win will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initialy FALSE) retains its value until changed by the user.

longname(termbuf, name)
char *termbuf, *name;

Fills in name with the long (full) name of the terminal described by the termcap entry in termbuf. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable ttytype. Termbuf is usually set via the termlib routine tgetent().

mvwin(win, y, x)
WINDOW *win;
int y, x;

Move the home position of the window win from its current starting coordinates to (y, x). If that would put part or all of the window off the edge of the terminal screen, mvwin() returns ERR and does not change anything.

WINDOW *
newwin(lines, cols, begin y, begin x)
int lines, cols, begin y, begin x;

Create a new window with lines lines and cols columns starting at position (begin y, begin x). If either lines or cols is 0 (zero), that dimension will be set to (LINES - begin y) or (COLS - begin x) respectively. Thus, to get a new window of dimensions LINES x COLS, use newwin(0, 0, 0, 0).

nl() [*]

nonl() [*]

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping <RETURN> to <LINE-FEED>. If the mapping is not done, refresh() can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf) [*]
WINDOW *win;
bool boolf;

Set the scroll flag for the given window. If boolf is FALSE, scrolling is not allowed. This is its default setting.

touchwin(win)
WINDOW *win;

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

WINDOW *
subwin(win, lines, cols, begin y, begin x)
WINDOW *win;

.

int lines, cols, begin y, begin x;

Create a new window with lines lines and cols columns starting at position (begin y, begin x) in the middle of the window win. This means that any change made to either window in the area covered by the subwindow will be made on both windows. begin y, begin x are specified relative to the overall screen, not the relative (0, 0) of win. If either lines or cols is 0 (zero), that dimension will be set to (LINES - begin y) or (COLS - begin x) respectively.

unctrl(ch) [*]
char ch;

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of ch. Control characters become their upper-case equivalents preceded by a "^". Other letters stay just as they are. To use unctrl(), you

must have #include <unctrl.h> in your file.

5.4. Details

gettmode()

Get the tty stats. This is normally called by initscr().

mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;

Moves the terminal's cursor from (lasty, lastx) to (newy, newx) in an approximation of optimal fashion. This routine uses the functions borrowed from ex version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. move() and refresh() should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
WINDOW *win;

Scroll the window upward one line. This is normally not used by the user.

savetty() [*]

resetty() [*]

savetty() saves the current tty characteristic flags. resetty() restores them to what savetty() stored. These functions are performed automatically by initscr() and endwin().

setterm(name)
char *name;

Set the terminal characteristics to be those of the terminal named name. This is normally called by in-

itscr().

tstp()

If the new tty(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls wrefresh(curscr) to redraw the screen. initscr() sets the signal SIGTSTP to trap to this routine.

.

Appendix A

1. Capabilities from termcap

1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

1.2. Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a P at the front of their comment. This normally is a number of

milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by PC)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., 12*. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say P*.

1.3. Variables Set By setterm()

variables set by setterm()

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)

.

Appendix A

variables set by setterm()

Type	Name	Pad	Description
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non-Destructive space

bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence[10]
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with X are reserved for severely nauseous glitches

1.4. Variables Set By gettmode()

variables set by gettmode()

type	name	description

[10] US and UE, if they do not exist in the termcap entry, are copied from SO and SE in setterm()

Appendix A

variables set by gettmode()

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

.

Appendix B

1. The WINDOW structure

The WINDOW structure is defined as follows:

```
# define             WINDOW    struct _win_st

struct _win_st {
           short     _cury, _curx;
           short     _maxy, _maxx;
           short     _begy, _begx;
           short     _flags;
           bool      _clear;
           bool      _leave;
           bool      _scroll;
           char      **_y;
           short     *_firstch;
```

```
            short     *_lastch;
};

# define            _SUBWIN           01
# define            _ENDLINE          02
# define            _FULLWIN          04
# define            _SCROLLWIN        010
# define            _STANDOUT         0200
```

_cury and _curx are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. _maxy and _maxx are the maximum values allowed for (_cury, _curx). _begy and _begx are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. _cury, _curx, _maxy, and _maxx are measured relative to (_begy, _begx), not the terminal's home.

_clear tells if a clear-screen sequence is to be generated on the next refresh() call. This is only meaningful for screens. The initial clear-screen for the first refresh() call is generated by initially setting clear to be TRUE for curscr, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. _leave is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. _scroll is TRUE if scrolling is allowed.

_y is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the ith line, and

```
_y[i][j]
```

is the jth character on the ith line.

_flags can have one or more values or'd into it. _SUBWIN means that the window is a subwindow, which indicates to delwin() that the space for the lines is not to be freed. _ENDLINE says that the end of the line for this window is also the end of a screen. _FULLWIN says that this window is a screen. _SCROLLWIN indicates that the last character of this screen is at the lower right-hand corner of the terminal; i.e., if a character was put there, the terminal would scroll. _STANDOUT says that all characters added to the screen are in standout mode.

1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant of to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of as-

terisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
# include           <curses.h>
# include           <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */

# define            NCOLS     80
# define            NLINES    24
# define            MAXPATTERNS         4

struct locs {
          char      y, x;
};

typedef struct locs           LOCS;

LOCS      Layout[NCOLS * NLINES];       /* current board layout */

int       Pattern,                      /* current pattern number */
          Numstars;                     /* number of stars in pattern */

mainmain() {

          char                *getenv();
          int                 die();

          srand(getpid());                        /* initialize random seque
nce */

          initscr();
          signal(SIGINT, die);
          noecho();
          nonl();
          leaveok(stdscr, TRUE);
          scrollok(stdscr, FALSE);

          for (;;) {
                    makeboard();                  /* make the board setup */
                    puton('*');                   /* put on '*'s */
                    puton(' ');                   /* cover up with ' 's */
          }
}

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
diedie() {

          signal(SIGINT, SIG_IGN);
          mvcur(0, COLS-1, LINES-1, 0);
          endwin();
          exit(0);
}


/*
```

```
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboardmakeboard() {

            reg int             y, x;
            reg LOCS            *lp;

            Pattern = rand() % MAXPATTERNS;
            lp = Layout;
            for (y = 0; y < NLINES; y++)
                        for (x = 0; x < NCOLS; x++)
                                    if (ison(y, x)) {
                                                lp->y = y;
                                                lp++->x = x;
                                    }
            Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
isonison(y, x)
reg int     y, x; {

            switch (Pattern) {
              case 0:           /* alternating lines */
                        return !(y & 01);
              case 1:           /* box */
                        if (x >= LINES && y >= NCOLS)
                                    return FALSE;
                        if (y < 3 || y >= NLINES - 3)
                                    return TRUE;
                        return (x < 3 || x >= NCOLS - 3);


              case 2:           /* holy pattern! */
                        return ((x + y) & 01);
              case 3:           /* bar across center */
                        return (y >= 9 && y <= 15);
            }

            /* NOTREACHED */
}

putonputon(ch)
reg char            ch; {

            reg LOCS            *lp;
            reg int             r;
            reg LOCS            *end;
            LOCS                temp;

            end = &Layout[Numstars];
            for (lp = Layout; lp < end; lp++) {
                        r = rand() % Numstars;
                        temp = *lp;
                        *lp = Layout[r];
                        Layout[r] = temp;
            }

            for (lp = Layout; lp < end; lp++) {
                        mvaddch(lp->y, lp->x, ch);
                        refresh();
            }
```

```
}
```

2.2. Life

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
# include           <curses.h>
# include           <signal.h>

/*
 *          Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */

struct lst_st {                         /* linked list element */
          int               y, x;               /* (y, x) position of piec
e */
          struct lst_st     *next, *last;       /* doubly linked */
};

typedef struct lst_st       LIST;

LIST      *Head;                        /* head of linked list */

mainmain(ac, av)
int       ac;
char      *av[]; {

          int       die();

          evalargs(ac, av);                     /* evaluate arguments */

          initscr();                            /* initialize screen packa
ge */
          signal(SIGINT, die);                  /* set to restore tty stat
s */


          crmode();                             /* set for char-by-char */
          noecho();                             /*
                    input */
          nonl();                               /* for optimization */

          getstart();                           /* get starting position *
/
          for (;;) {
                    prboard();                  /* print out current board
 */
                    update();                   /* update board position *
/
          }
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */
```

```
diedie() {

        signal(SIGINT, SIG_IGN);                /* ignore rubouts */
        mvcur(0, COLS-1, LINES-1, 0);           /* go to bottom of screen
*/
        endwin();                               /* set terminal to initial
 state */
        exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key.  Thus, u move diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
getstartgetstart() {

        reg char            c;
        reg int             x, y;

        box(stdscr, '|', '_');                  /* box in the screen */
        move(1, 1);                             /* move to upper left corn
er */

        do {
                refresh();                      /* print current position
*/
                if ((c=getch()) == 'q')
                        break;
                switch (c) {
                  case 'u':
                  case 'i':
                  case 'o':
                  case 'j':
                  case 'l':
                  case 'm':
                  case ',':
                  case '.':
                        adjustyx(c);
                        break;
                  case 'f':
                        mvaddstr(0, 0, "File name: ");
                        getstr(buf);
                        readfile(buf);
                        break;
                  case 'x':
                        addch('X');
                        break;
                  case ' ':
                        addch(' ');
                        break;
                }
        }

        if (Head != NULL)                               /* start new lis
t */
                dellist(Head);
        Head = malloc(sizeof (LIST));

        /*
         * loop through the screen looking for 'x's, and add a list
         * element for each one
         */
```

```
                    for (y = 1; y < LINES - 1; y++)
                              for (x = 1; x < COLS - 1; x++) {
                                        move(y, x);
                                        if (inch() == 'x')
                                                  addlist(y, x);
                              }
          }

          /*
           * Print out the current board position from the linked list
           */
          prboardprboard() {

                    reg LIST            *hp;

                    erase();                                  /* clear out last position
*/
                    box(stdscr, '|', '_');                    /* box in the screen */

                    /*
                     * go through the list adding each piece to the newly
                     * blank board
                     */
                    for (hp = Head; hp; hp = hp->next)
                              mvaddch(hp->y, hp->x, 'X');

                    refresh();
          }
```

3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

3.1. Twinkle

The twinkle program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
          mainmain() {

                    reg char            *sp;
                    char                *getenv();
                    int                 _putchar(), die();

                    srand(getpid());                          /* initialize random seque
nce */

                    if (isatty(0)) {
                           gettmode();
                           if (sp=getenv("TERM"))
                                     setterm(sp);
                              signal(SIGINT, die);
                    }
                    else {
                              printf("Need a terminal on %d\n", _tty_ch);
                              exit(1);
                    }
                    _puts(TI);
                    _puts(VS);
```

```
            noecho();
            nonl();
            tputs(CL, NLINES, _putchar);
            for (;;) {
                    makeboard();                    /* make the board setup */
                    puton('*');                     /* put on '*'s */
                    puton(' ');                     /* cover up with ' 's */
            }
    }

    /*
     * _putchar defined for tputs() (and _puts())
     */
    _putchar_putchar(c)
    reg char            c; {

            putchar(c);
    }

    putonputon(ch)
    char        ch; {

            static int          lasty, lastx;
            reg LOCS            *lp;
            reg int             r;
            reg LOCS            *end;
            LOCS                temp;

            end = &Layout[Numstars];
            for (lp = Layout; lp < end; lp++) {
                    r = rand() % Numstars;
                    temp = *lp;
                    *lp = Layout[r];
                    Layout[r] = temp;
            }

            for (lp = Layout; lp < end; lp++)
                            /* prevent scrolling */
                    if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                            mvcur(lasty, lastx, lp->y, lp->x);
                            putchar(ch);
                            lasty = lp->y;
                            if ((lastx = lp->x + 1) >= NCOLS)
                                    if (AM) {
                                            lastx = 0;
                                            lasty++;
                                    }
                                    else
                                            lastx = NCOLS - 1;
                    }
    }
```

Contents