

# Screen Updating and Cursor Movement Optimization: A Library Package

*Kenneth C. R. C. Arnold  
Elan Amir*

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:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the **termcap**(5) 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. *Ken Arnold 16 April 1986*

The help and/or support of Kirk McKusick and Keith Bostic (public vi!) was invaluable in bringing the package “into the 90’s”, which now includes completely new data structures and screen refresh optimization routines. *Elan Amir 29 December 1992*

## 1. Overview

In making available the generalized terminal descriptions in **termcap(5)**, 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 with nearly as much ease as is necessary to simply print or read things.

### 1.1. Terminology

In this document, the following terminology is used:

**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 Applications

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. Compilations should have the following form:

```
cc [ flags ] file ... -lcurses -ltermcap
```

### 1.3. Screen Updating

In order to update the screen optimally, it is necessary to use the `refresh` function. This function is defined in `termcap.h` and `term.h`. It is defined as follows:

## 1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are always available: *curscr*, which is the image of what the terminal looks like at present, and *stdscr*, which is the image of what the programmer wants the terminal to look like next. The user should not 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<sup>1</sup>. 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 clumsiness, most I/O routines can be preceded by the prefix “mv” and the desired (y, x) co-ordinates can then 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 a window pointer is needed, it is always the first parameter passed.

## 2. Variables

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

| 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 <i>Def_term</i> 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       | flag returned by routines upon success.   |

## 3. Usage

This is a description of how to actually use the screen package. For simplicity, we assume all updating, reading, etc. is applied to *stdscr*, although a different window can of course be specified.

<sup>1</sup> Actually, **addch()** is really a “#define” macro with arguments, as are most of the “functions” which act upon *stdscr*.

### 3.1. Initialization

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 **cbreak()** should be called after **initscr()**.

After the initial window allocation done by **initscr()**, specific window characteristics can be set. Scrolling can be enabled by calling **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. Additional windows can be created by using the functions **newwin()** and **subwin()**. **delwin()** allows you to delete an existing window. The variables *LINES* and *COLS* control the size of the terminal. They are initially implicitly set by **initscr()**, but can be altered explicitly by the user followed by a call to **initscr()**. Note that any call to **initscr()**, will always delete any existing *stdscr* and/or *curscr* before creating new ones so this change is best done before the initial call to **initscr()**.

### 3.2. Output

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. As mentioned above, you can combine the two into **mvaddch()** to do both things in one call.

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

After a change has been made to the window, you must call **refresh()**. when you want the portion of the terminal covered by the window to reflect the change. 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 routines **touchwin()**, **touchline()**, and **touchoverlap()** are provided to make it look like a desired part of window has been changed, thus forcing **refresh()** to check that whole subsection of the terminal for changes.

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

### 3.3. 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.

### 3.4. Termination

In order to perform 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 **gettmode()** 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. The terminal state module uses the variable *curses\_termios* to save the original terminal state which is then restored upon a call to **endwin()**. Thus, anytime after the call to **initscr**, **endwin()** should be called before exiting. Note however,

that `endwin()` should always be called **before** the final calls to `delwin()`, which free the storage of the windows.

#### 4. Cursor Movement Optimizations

One of the most difficult things to do properly is motion optimization. After using `gettmode()` and `setterm()` to get the terminal descriptions, the function `mvcur()` deals with this task. Its usage is simple: 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. 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. Character Output and Scrolling

The character output policy deals with the following problems. First, where is the location of the cursor after a character is printed, and secondly, when does the screen scroll if scrolling is enabled.

In the normal case the characters are output as expected, with the cursor occupying the position of the next character to be output. However, when the cursor is on the last column of the line, the cursor will remain on that position after the last character on the line is output and will only assume the position on the next line when the next character (the first on the next line) is output.

Likewise, if scrolling is enabled, a scroll will be invoked only when the first character on the first line

..

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 of 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**(*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.

**baudrate**();†

Returns the output baud rate of the terminal. This is a system dependent constant (defined in `<sys/tty.h>` on BSD systems, which is included by `<curses.h>`).

**box**(*WINDOW win, char vert, char 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.

**cbreak**();†

Set or the terminal to cbreak mode.

**clear**();†

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**(*WINDOW \*scr, int boolf*);†

Sets the clear flag for the screen *scr*. If *boolf* is non-zero, this will force a clear-screen to be printed on the next **refresh**(), or stop it from doing so if *boolf* is 0. 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.

**clrtoebot**();†

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**();†

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

**crmode**();†

Identical to **cbreak**(). The misnamed macro **crmode**() and **nocrmode**() is retained for backwards compatibility with earlier versions of the library.

**delch();**

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();**

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.

**delwin(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.

**echo();†**

Sets the terminal to echo characters.

**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 and before the final calls to **delwin()**. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal(2)**.

**erase();†**

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.

**erasechar();†**

Returns the erase character for the terminal, *i.e.*, the character used by the user to erase a single character from the input.

**flushok(WINDOW \*win, int boolf);**

Normally, **refresh()** **fflush('s); stdout** when it is finished. **flushok()** allows you to control this. if *boolf* is non-zero (*i.e.*, non-zero) it will do the **fflush()**, otherwise it will not.

**getch();†**

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(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()**) 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.

**gettmode();**

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

**getyx(WINDOW \*win, int y, int 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*.

**idlok(WINDOW \*win, int boolf);**

Reserved for future use. This will eventually signal to **refresh()** that it is all right to use the insert and delete line sequences when updating the window.

**inch();†**

Returns the character at the current position on the given window. This does not make any changes to the window.

**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* (initially "dumb"). If the boolean *My\_term* is non-zero, *Def\_term* is always used. If the system supports the **TIOCGWINSZ** *ioctl(2)* call, it is used to get the number of lines and columns for the terminal, otherwise it is taken from the **termcap** description.

**insch(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();**

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.

**killchar();†**

Returns the line kill character for the terminal, *i.e.*, the character used by the user to erase an entire line from the input.

**leaveok(WINDOW \*win, int boolf);†**

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is non-zero, 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 *boolf* is 0 the cursor will be moved to the current (y, x) co-ordinates. This flag (initially 0) retains its value until changed by the user.

**move**(*int y, int 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.

**mvcur**(*int lasty, int lastx, int newy, int 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.

**mvprintw**(*int y, int x, const char \*fmt, ...*);

Equivalent to:

```
move(y, x);
printw(fmt, ...);
```

**mvscanw**(*int y, int x, const char \*fmt, ...*);

Equivalent to:

```
move(y, x);
scanw(fmt, ...);
```

**mvwin**(*WINDOW \*win, int y, int 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. For subwindows, **mvwin**() also returns ERR if you attempt to move it off its main window. If you move a main window, all subwindows are moved along with it.

**mvwprintw**(*WINDOW \*win, int y, int x, const char \*fmt, ...*);

Equivalent to:

```
wmove(win, y, x);
printw(fmt, ...);
```

**mvwscanw**(*WINDOW \*win, int y, int x, const char \*fmt, ...*);

Equivalent to:

```
wmove(win, y, x);
scanw(fmt, ...);
```

**newwin**(*int lines, int cols, int begin\_y, int 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* × *COLS*, use **newwin**(0, 0, 0, 0).

**nl**();†

Set the terminal to 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.

**nocbreak();**†

Unset the terminal from cbreak mode.

**nocrmode();**†

Identical to **nocbreak()**. The misnamed macro **nocrmode()** is retained for backwards compatibility with earlier versions of the library.

**noecho();**†

Turn echoing of characters off.

**nonl();**†

Unset the terminal to from nl mode. See **nl()**.

**noraw();**†

Unset the terminal from raw mode. See **raw()**.

**overlay(WINDOW \*win1, WINDOW \*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. Note that all non-blank characters are overwritten destructively in the overlay.

**overwrite(WINDOW \*win1, WINDOW \*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(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.

**raw();**†

Set the terminal to raw mode. On version 7 **UNIX**<sup>2</sup> this also turns off newline mapping (see **nl()**).

**refresh();**†

---

<sup>2</sup> **UNIX** is a trademark of Unix System Laboratories.

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.

As a special case, if **wrefresh()** is called with the window *curscr* the screen is cleared and repainted as it is currently. This is very useful for allowing the redrawing of the screen when the user has garbage dumped on his terminal.

**resetty();**†

**resetty()** restores them to what **savetty()** stored. These functions are performed automatically by **initscr()** and **endwin()**. This function should not be used by the user.

**savetty();**†

**savetty()** saves the current tty characteristic flags. See **resetty()**. This function should not be used by the user.

**scanw(char \*fmt, ...);**

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

**scroll(WINDOW \*win);**

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

**scrollok(WINDOW \*win, int boolf);**†

Set the scroll flag for the given window. If *boolf* is 0, scrolling is not allowed. This is its default setting.

**standend();**†

End standout mode initiated by **standout()**.

**standout();**†

Causes any characters added to the window to be put in standout mode on the terminal (if it has that capability).

**subwin(WINDOW \*win, int lines, int cols, int begin\_y, int begin\_x);**

Create a new window with *lines* lines and *cols* columns starting at position (*begin\_y*, *begin\_x*) inside the window *win*. This means that any change made to either window in the area covered by the sub-window 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.

**touchline(WINDOW \*win, int y, int startx, int endx);**

This function performs a function similar to **touchwin()** on a single line. It marks the first change for the given line to be *startx*, if it is before the current first change mark, and the last change mark is set to be *endx* if it is currently less than *endx*.

**touchoverlap**(WINDOW \*win1, WINDOW \*win2);

Touch the window *win2* in the area which overlaps with *win1*. If they do not overlap, no changes are made.

**touchwin**(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.

**tstp**()

This function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the saved tty state and then calls **wrefresh**(*curscr*); to redraw the screen. **Initscr**() sets the signal SIGTSTP to trap to this routine.

**unctrl**(char \*ch);†

Returns a string which is an ASCII representation of *ch*. Characters are 8 bits long.

**unctrlllen**(char \*ch);†

Returns the length of the ASCII representation of *ch*.

**vwprintw**(WINDOW \*win, const char \*fmt, va\_list ap);

Identical to **printw**() except that it takes both a window specification and a pointer to a variable length argument list.

**wscanw**(WINDOW \*win, const char \*fmt, va\_list ap);

Identical to **scanw**() except that it takes both a window specification and a pointer to a variable length argument list.

**waddbytes**(WINDOW \*win, char \*str, int len);

This function is the low level character output function. *Len* characters of the string *str* are output to the current (y, x) co-ordinates position of the window specified by *win*.

*The following functions differ from the standard functions only in their specification of a window, rather than the use of the default stdscr.*

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

**waddstr**(WINDOW \*win, char \*str);

**wclear**(WINDOW \*win);

**wclrtoobot**(WINDOW \*win);

**wclrtoeol**(WINDOW \*win);

**wdelch**(WINDOW \*win);

**wdeleteln**(WINDOW \*win);

**werase**(WINDOW \*win);

**wgetch**(WINDOW \*win);

**wgetstr**(WINDOW \*win, char \*str);

**winch**(WINDOW \*win);†

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

```
winsertln(WINDOW *win);  
wmove(WINDOW *win, int y, int, x');  
wprintw(WINDOW *win, char *fmt, ...);  
wrefresh(WINDOW *win);  
wscanw(WINDOW *win, char *fmt, ...);  
wstandend(WINDOW *win);  
wstandout(WINDOW *win);
```

## 1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive. Further examples can be found in the games section of the source tree and in various utilities that use the screen such as *systat(1)*.

The following examples are intended to demonstrate the basic structure of a program using the package. An additional, more comprehensive, program can be found in the source code in the *examples* subdirectory.

### 1.1. Simple Character Output

This program demonstrates how to set up a window and output characters to it. Also, it demonstrates how one might control the output to the window. If you run this program, you will get a demonstration of the character output characteristics discussed in the above Character Output section.

```
#include <sys/types.h>
#include <curses.h>
#include <stdio.h>
#include <signal.h>

#define YSIZE 10
#define XSIZE 20

int quit();

main()
{
    int i, j, c;
    size_t len;
    char id[100];
    FILE *fp;
    char *s;

    initscr();                /* Always call initscr() first */
    signal(SIGINT, quit);     /* Make sure you have a 'cleanup' fn */
    crmode();                /* We want cbreak mode */
    noecho();                 /* We want to have control of chars */
    delwin(stdscr);          /* Create our own stdscr */
    stdscr = newwin(YSIZE, XSIZE, 10, 35);
    flushok(stdscr, TRUE);   /* Enable flushing of stdout */
    scrollok(stdscr, TRUE);  /* Enable scrolling */
    erase();                  /* Initially, clear the screen */

    standout();
    move(0,0);
    while (1) {
        c = getchar();
        switch(c) {
            case 'q':         /* Quit on 'q' */
                quit();
                break;
            case 's':         /* Go into standout mode on 's' */
                standout();
                break;
```

```

        case 'e':          /* Exit standout mode on 'e' */
            standend();
            break;
        case 'r':          /* Force a refresh on 'r' */
            wrefresh(curscr);
            break;
        default:           /* By default output the character */
            addch(c);
            refresh();
    }
}

int
quit()
{
    erase();              /* Terminate by erasing the screen */
    refresh();
    endwin();             /* Always end with endwin() */
    delwin(curscr);      /* Return storage */
    delwin(stdscr);
    putchar('^0');
    exit(0);
}

```

## 1.2. Twinkle

This is a moderately simple program which prints patterns on the screen. It switches between patterns of asterisks, 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 NLINES24
#define MAXPATTERNS4

typedef struct {
    int y, x;
} LOCS;

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

```

```

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

char    *getenv();

int     die();

main()
{
    srand(getpid());      /* initialize random sequence */

    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.
 */
die()
{
    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 characterR19 de90.t pattern
 * or not.
 */
makeboard()
{
    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;
                lp++;
            }
        Numstars = lp - Layout;
    }

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(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 */
}

puton(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();
    }
}

```

## 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;
    short      _ch_off;
    bool       _clear;
    bool       _leave;
    bool       _scroll;
    char       **_y;
    short      *_firstch;
    short      *_lastch;
    struct _win_st *_nextp, *_orig;
};

# define _ENDLINE      001
# define _FULLWIN      002
# define _SCROLLWIN    004
# define _FLUSH        010
# define _FULLLINE     020
# define _IDLINE       040
# define _STANDOUT     0200
# define _NOCHANGE     -1
```

`_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 *i*th line, and

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

is the *j*th character on the *i*th line. `_flags` can have one or more values or'd into it.

For windows that are not subwindows, `_orig` is NULL. For subwindows, it points to the main window to which the window is subsidiary. `_nextp` is a pointer in a circularly linked list of all the windows

---

<sup>3</sup> All variables not normally accessed directly by the user are named with an initial “\_” to avoid conflicts with the user's variables.

which are subwindows of the same main window, plus the main window itself.

`_firstch` and `_lastch` are `malloc(ed)`; arrays which contain the index of the first and last changed characters on the line. `_ch_off` is the x offset for the window in the `_firstch` and `_lastch` arrays for this window. For main windows, this is always 0; for subwindows it is the difference between the starting point of the main window and that of the subwindow, so that change markers can be set relative to the main window. This makes these markers global in scope.

All subwindows share the appropriate portions of `_y`, `_firstch`, `_lastch`, and `_insdel` with their main window.

`_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. `_FULLLINE` says that the width of a line is the same as the width of the terminal. If `_FLUSH` is set, it says that `fflush(, stdout)`; should be called at the end of each `refresh()` `_STANDOUT` says that all characters added to the screen are in standout mode. `_INSDEL` is reserved for future use, and is set by `idlok()`. `_firstch` is set to `_NOCHANGE` for lines on which there has been no change since the last `refresh()`.

## 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 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 asterisks, 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 NLINES24
# define MAXPATTERNS4

typedef struct {
    int      y, x;
} LOCS;

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

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

char   *getenv();

int    die();

main()
{
    srand(getpid());                /* initialize random sequence */

    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.
 */
die()
{
    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.
 */
makeboard()
{
    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;
                lp++;
            }
    Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(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 */
}

puton(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 fragment models 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 code, 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.
 */

typedef struct lst_st {
    int             y, x;
    /* linked list element */
    /* (y, x) position of piece */

```

```

        struct lst_st      *next, *last;      /* doubly linked */
    } LIST;

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

int     die();

main(ac, av)
int     ac;
char    *av[];
{
    evalargs(ac, av);        /* evaluate arguments */

    initscr();               /* initialize screen package */
    signal(SIGINT, die);     /* set to restore tty stats */
    cbreak();                /* 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.
 */
die()
{
    signal(SIGINT, SIG_IGN); /* ignore rubouts */
    mvcur(0, COLS - 1, LINES - 1, 0); /* go to bottom of screen */
    endwin();                 /* set terminal to good 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.
 */
getstart()
{
    reg char c;
    reg int   x, y;
    auto char buf[100];

```

```

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

for (;;) {
    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 '.':
            adjustx(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 list */
    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
 */
prboard() {

```

```

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):

```

extern int      _putchar();

main()
{
    reg char *sp;

    srand(getpid());      /* initialize random sequence */

    if (isatty(0)) {
        gettmode();
        if ((sp = getenv("TERM")) != NULL)
            setterm(sp);
        signal(SIGINT, die);
    }
    else {
        printf("Need a terminal on %d0, _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 */
    }
}

```

```
    }

    puton(ch)
    char    ch;
    {
        reg LOCS    *lp;
        reg int     r;
        reg LOCS    *end;
        LOCS        temp;
        static int  lasty, lastx;

        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*

|  |    |
|--|----|
| 1 Overview .....                       | 3  |
| 1.1 Terminology .....                  | 3  |
| 1.2 Compiling Applications .....       | 3  |
| 1.3 Screen Updating .....              | 3  |
| 1.4 Naming Conventions .....           | 4  |
| 2 Variables .....                      | 4  |
| 3 Usage .....                          | 4  |
| 3.1 Initialization .....               | 5  |
| 3.2 Output .....                       | 5  |
| 3.3 Input .....                        | 5  |
| 3.4 Termination .....                  | 5  |
| 4 Cursor Movement Optimizations .....  | 6  |
| 5 Character Output and Scrolling ..... | 6  |
| 6 Terminal State Handling .....        | 6  |
| 7 Subwindows .....                     | 6  |
| 8 The Functions .....                  | 6  |
| <b>Appendix A</b> .....                | 15 |
| 1 Examples .....                       | 15 |
| 1.1 Simple Character Output .....      | 15 |
| 1.2 Twinkle .....                      | 16 |
| <b>Appendix B</b> .....                | 19 |
| 1 The WINDOW structure .....           | 19 |
| <b>Appendix C</b> .....                | 21 |
| 1 Examples .....                       | 21 |
| 2 Screen Updating .....                | 21 |
| 2.1 Twinkle .....                      | 21 |
| 2.2 Life .....                         | 23 |
| 3 Motion optimization .....            | 26 |
| 3.1 Twinkle .....                      | 26 |