10. curses/terminfo

Introduction

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, or draw a display on the screen to help users enter and retrieve information from a database.

This tutorial explains how to use the Terminal Information Utilities package, commonly called **curses/terminfo**, to write screen management programs on SYSTEM V/68. This package includes a library of C routines, a database, and a set of operating system support tools. To start you writing screen management programs as soon as possible, the tutorial does not attempt to cover every part of the package. For instance, it covers only the most frequently used routines and then points you to **curses**(3X) and **terminfo**(4) in the *Programmer's Reference Manual* for more information. Keep the manual close at hand; you'll find it invaluable when you want to know more about one of these routines or about other routines not discussed here.

Because the routines are compiled C functions, you should be familiar with the C programming language before using **curses/terminfo**. You should also be familiar with the operating system/C language standard I/O package (see "System Calls and Subroutines" and "Input/Output" in Chapter 2 and **stdio**(3S)). With that knowledge, you can design screen management programs for many purposes.

This chapter has five sections:

- Overview

  This section briefly describes **curses**, **terminfo**, and the other components of the Terminal Information Utilities package.

- Working with **curses** Routines

  This section describes the basic routines making up the **curses**(3X) library. It covers the routines for writing to a screen, reading from a screen, and building windows. It also covers routines for more advanced screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time. Many examples are included to show the effect of using these routines.

- Working with **terminfo** Routines

  This section describes the routines in the **curses** library that deal directly with the **terminfo** database to handle certain terminal capabilities, such as programming function keys.

- Working with the **terminfo** Database

  This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

- **curses** Program Examples

  This section includes six programs that illustrate uses of **curses** routines.

Overview

curses

curses(3X) is the library of routines that you use to write screen management programs on the operating system. The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **printw**() that behaves much like **printf**(3S) and another routine **getch**() that behaves like **getc**(3S). The automatic teller program at your bank might use **printw**() to print its menus and **getch**() to accept your requests for deposits or withdrawals. A visual screen editor like the operating system screen editor **vi**(1) might also use these and other **curses** routines.

The **curses** routines are usually located in **/usr/lib/libcurses.a**. To compile a program using these routines, you must use the **cc**(1) command and include **−lcurses** on the command line so that the link editor can locate and load them:

10

cc *file*.**c** **−lcurses** **−o** *file*

The name **curses** comes from the cursor optimization that this library of routines provides. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you designed a screen editor program with **curses** routines and edited the sentence:

```
curses/terminfo is a great package for creating screens.
```

to read:

```
curses/terminfo is the best package for creating screens.
```

the program would output only `the best` in place of `a great`. The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen appropriately for the terminal on which a **curses** program is run. This means that the **curses** library can do whatever is required to update many different terminal types. It searches the **terminfo** database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your operating system's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple **curses** program. It uses some of the basic **curses** routines to move a cursor to the middle of a terminal screen and print the character string `BullsEye`. Each of these routines is described in the following section "Working with **curses** Routines" in this chapter. For now, just look at their names and you will get an idea of what each of them does.

```
#include <curses.h>

main()
{
    initscr();

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();
    addstr("Eye");
    refresh();
    endwin();
}
```

Figure 10-1. A Simple **curses** Program

terminfo

terminfo refers to both of the following:

- It is a group of routines within the **curses** library that handles certain terminal capabilities. For example, you can use these routines to program function keys (if your terminal has programmable keys) or to write filters. Shell programmers, as well as C programmers, can use the **terminfo** routines in their programs.
- It is a database containing the descriptions of many terminals that can be used with **curses** programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

  Each terminal description in the database is a separate, compiled file. You use the source code that **terminfo**(4) describes to create these files and the command **tic**(1M) to compile them.

  The compiled files are normally located in the directories **/usr/lib/terminfo/?**. These directories have single-character names, each of which is the first character in the name of a terminal. For example, an entry for the DEC vt100 is normally located in the file **/usr/lib/terminfo/v/vt100**.

Here's a simple shell script that uses the **terminfo** database:

```
#   Clear the screen and show the 0,0 position.
#
tput clear
tput cup 0 0              # or tput home
echo "<- this is 0 0"

#
#   Show the 5,10 position.
#
tput cup 5 10
echo "<- this is 5 10"
```

Figure 10-2. A Shell Script Using **terminfo** Routines

How curses and terminfo Work Together

A screen management program with **curses** routines refers to the **terminfo** database at run time to obtain the information it needs about the terminal being used—what we'll call the current terminal from here on.

For example, suppose you are using a DEC vt100 terminal to run the simple **curses** program shown in Figure 10-1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the `BullsEye` in the middle of it. The description of the DEC vt100 in the **terminfo** database has this information. All the **curses** program needs to know before it goes looking for the information is the name of your terminal. You tell the program the name by putting it in the environment variable **$TERM** when you log in or by setting and exporting **$TERM** in your **.profile** file (see **profile**(4)). Knowing **$TERM**, a **curses** program run on the current terminal can search the **terminfo** database to find the correct terminal description.

For example, assume that the following example lines are in a **.profile**:

```
TERM=vt100
export TERM
tput init
```

The first line names the terminal type, and the second line exports it. (See **profile**(4) in the *Programmer's Reference Manual.*) The third line of the example tells the operating system to initialize the current terminal. That is, it makes sure that the terminal is set up according to its description in the **terminfo** database. (The order of these lines is important. **$TERM** must be defined and exported first, so that when **tput** is called the proper initialization for the current terminal takes place.) If you had these lines in your **.profile** and you ran a **curses** program, the program would get the information that it needs about your terminal from the file **/usr/lib/terminfo/v/vt100**. which provides a match for **$TERM**.

Other Components of the Terminal Information Utilities

We said earlier that the Terminal Information Utilities package is commonly referred to as **curses/terminfo**. The package, however, has other components. We've mentioned some of them, **tic**(1M) for instance. Here's a complete list of the components discussed in this tutorial:

captoinfo(1M)	a tool for converting terminal descriptions developed on earlier releases of the operating system to **terminfo** descriptions
curses(3X)	

infocmp(1M)	a tool for printing and comparing compiled terminal descriptions
tabs(1)	a tool for setting non-standard tab stops
terminfo(4)	
tic(1M)	a tool for compiling terminal descriptions for the **terminfo** database
tput(1)	a tool for initializing the tab stops on a terminal and for outputting the value of a terminal capability

We also refer to **profile**(4), **scr_dump**(4), **term**(4), and **term**(5). For more information about any of these components, see the *Programmer's Reference Manual* and the *User's Reference Manual*.

Working with curses Routines

This section describes the basic **curses** routines for creating interactive screen management programs. It begins by describing the routines and other program components that every **curses** program needs to work properly. Then it tells you how to compile and run a **curses** program. Finally, it describes the most frequently used **curses** routines that:

- write output to and read input from a terminal screen
- control the data output and input — for example, to print output in bold type or prevent it from echoing (printing back on a screen)
- manipulate multiple screen images (windows)
- draw simple graphics
- manipulate soft labels on a terminal screen
- send output to and accept input from more than one terminal.

To illustrate the effect of using these routines, we include simple example programs as the routines are introduced. We also refer to a group of larger examples located in the section "**curses** Program Examples" in this chapter. These larger examples are more challenging; they sometimes make use of routines not discussed here. Keep the **curses**(3X) manual page handy.

What Every curses Program Needs

All **curses** programs need to include the header file <**curses.h**> and call the routines **initscr**(), **refresh**() or similar related routines, and **endwin**().

The Header File <curses.h>

The header file <**curses.h**> defines several global variables and data structures and defines several **curses** routines as macros.

To begin, let's consider the variables and data structures defined. <**curses.h**> defines all the parameters used by **curses** routines. It also defines the integer variables **LINES** and **COLS**; when a **curses** program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine **initscr**() described below. The header file defines the constants **OK** and **ERR**, too. Most **curses** routines have return values; the **OK** value is returned if a routine is properly completed, and the **ERR** value if some error occurs.

NOTE

> **LINES** and **COLS** are external (global) variables that represent the size of a terminal screen. Two similar variables, **$LINES** and **$COLUMNS**, may be set in a user's shell environment; a **curses** program uses the environment variables to determine the size of a screen. Whenever we refer to the environment variables in this chapter, we will use the **$** to distinguish them from the C declarations in the <**curses.h**> header file.
>
> For more information about these variables, see the following sections "The Routines **initscr**(), **refresh**(), and **endwin**()" and "More about **initscr**() and Lines and Columns."

Now let's consider the macro definitions. <**curses.h**> defines many **curses** routines as macros that call other macros or **curses** routines. For instance, the simple routine **refresh**() is a macro. The line

```
#define refresh() wrefresh(stdscr)
```

shows when **refresh** is called, it is expanded to call the **curses** routine **wrefresh**(). The latter routine in turn calls the two **curses** routines

wnoutrefresh() and **doupdate**(). Many other routines also group two or three routines together to achieve a particular result.

> **CAUTION**
>
> Macro expansion in **curses** programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about <**curses.h**>: it automatically includes <**stdio.h**> and the <**termio.h**> tty driver interface file. Including either file again in a program is harmless but wasteful.

The Routines initscr(), refresh(), endwin()

The routines **initscr**(), **refresh**(), and **endwin**() initialize a terminal screen to an "in **curses** state," update the contents of the screen, and restore the terminal to an "out of **curses** state," respectively. Use the simple program that we introduced earlier to learn about each of these routines:

```
#include <curses.h>

main()
{
   initscr();      /* initialize terminal settings and <curses.h>
                      data structures and variables */

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();      /* send output to (update) terminal screen */
   addstr("Eye");
   refresh();      /* send more output to terminal screen */
   endwin();       /* restore all terminal settings */
}
```

Figure 10-3. The Purposes of **initscr**(), **refresh**(), and **endwin**() in a Program

A **curses** program usually starts by calling **initscr**(); the program should call **initscr**() only once. Using the environment variable **$TERM** as the section "How **curses** and **terminfo** Work Together" describes, this routine determines what

terminal is being used. It then initializes all the declared data structures and other variables from **<curses.h>**. For example, **initscr()** would initialize **LINES** and **COLS** for the sample program on whatever terminal it was run. If the Teletype 5425 were used, this routine would initialize **LINES** to 24 and **COLS** to 80. Finally, this routine writes error messages to **stderr** and exits if errors occur.

During the execution of the program, output and input is handled by routines like **move()** and **addstr()** in the sample program. For example,

```
move( LINES/2 - 1, COLS/2 - 4 );
```

says to move the cursor to the left of the middle of the screen. Then the line:

```
addstr("Bulls");
```

says to write the character string `Bulls`. For example, if the Teletype 5425 were used, these routines would position the cursor and write the character string at (11,36).

NOTE

All **curses** routines that move the cursor move it from its home position in the upper left corner of a screen. The **(LINES,COLS)** coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the more common 'x,y' order of screen (or graph) coordinates. The `-1` in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

10

Routines like **move()** and **addstr()** do not actually change a physical terminal screen when they are called. The screen is updated only when **refresh()** is called. Before this, an internal representation of the screen called a window is updated. This is an important concept, which we discuss below under "More about **refresh()** and Windows."

Finally, a **curses** program ends by calling **endwin()**. This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling a curses Program

You compile programs that include **curses** routines as C language programs using the **cc**(1) command (documented in the *Programmer's Reference Manual*), which invokes the C compiler (see Chapter 2 in this guide for details).

The routines are usually stored in the library **/usr/lib/libcurses.a**. To direct the link editor to search this library, you must use the **-l** option with the **cc** command.

The general command line for compiling a **curses** program follows:

cc *file*.**c** **-lcurses** **-o** *file*

file.**c** is the name of the source program; and *file* is the executable object module.

Running a curses Program

curses programs count on certain information being in a user's environment to run properly. Specifically, users of a **curses** program should usually include the following three lines in their **.profile** files:

```
TERM=current terminal type
export TERM
tput init
```

For an explanation of these lines, see the section "How **curses** and **terminfo** Work Together" in this chapter. Users of a **curses** program could also define the environment variables **$LINES**, **$COLUMNS**, and **$TERMINFO** in their **.profile** files. However, unlike **$TERM**, these variables do not have to be defined.

If a **curses** program does not run as expected, you might want to debug it with **sdb**(1), which is documented in the *Programmer's Reference Manual*). When using **sdb**, you have to keep a few points in mind. First, a **curses** program is interactive and always has knowledge of where the cursor is located. An interactive debugger like **sdb**, however, may cause changes to the contents of the screen of which the **curses** program is not aware.

Second, a **curses** program outputs to a window until **refresh**() or a similar routine is called. Because output from the program may be delayed, debugging the output for consistency may be difficult.

Third, setting break points on **curses** routines that are macros, such as **refresh**(), does not work. You have to use the routines defined for these macros, instead; for example, you have to use **wrefresh**() instead of **refresh**(). See the above section, "The Header File <**curses.h**>," for more information about macros.

More about initscr() and Lines and Columns

After determining a terminal's screen dimensions, **initscr**() sets the variables **LINES** and **COLS**. These variables are set from the **terminfo** variables **lines** and **columns**. These, in turn, are set from the values in the **terminfo** database, unless these values are overridden by the values of the environment **$LINES** and **$COLUMNS**.

More about refresh() and Windows

As mentioned above, **curses** routines do not update a terminal until **refresh**() is called. Instead, they write to an internal representation of the screen called a window. When **refresh**() is called, all the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like a buffer does when you use an operating system editor. When you invoke **vi**(1), for instance, to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the **w** or **ZZ** command. Similarly, when you invoke a screen program made up of **curses** routines, they change the contents of a window. The changes become part of the current terminal screen only when **refresh**() is called.

<**curses.h**> supplies a default window named **stdscr** (standard screen), which is the size of the current terminal's screen, for all programs using **curses** routines. The header file defines **stdscr** to be of the type **WINDOW***, a pointer to a C structure that you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in **stdscr**. When **refresh**() is called, it compares the two screen images and sends a stream of characters to the terminal that make the current screen look like **stdscr**. A **curses** program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on the window. It optimizes output by printing as few characters as possible. Figure 10-4 illustrates what happens when you execute the sample curses program that prints `BullsEye` at the center of a terminal screen (see Figure 10-1). Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh**() is called. This **refresh**() clears the screen and updates it with the current contents of **stdscr**.

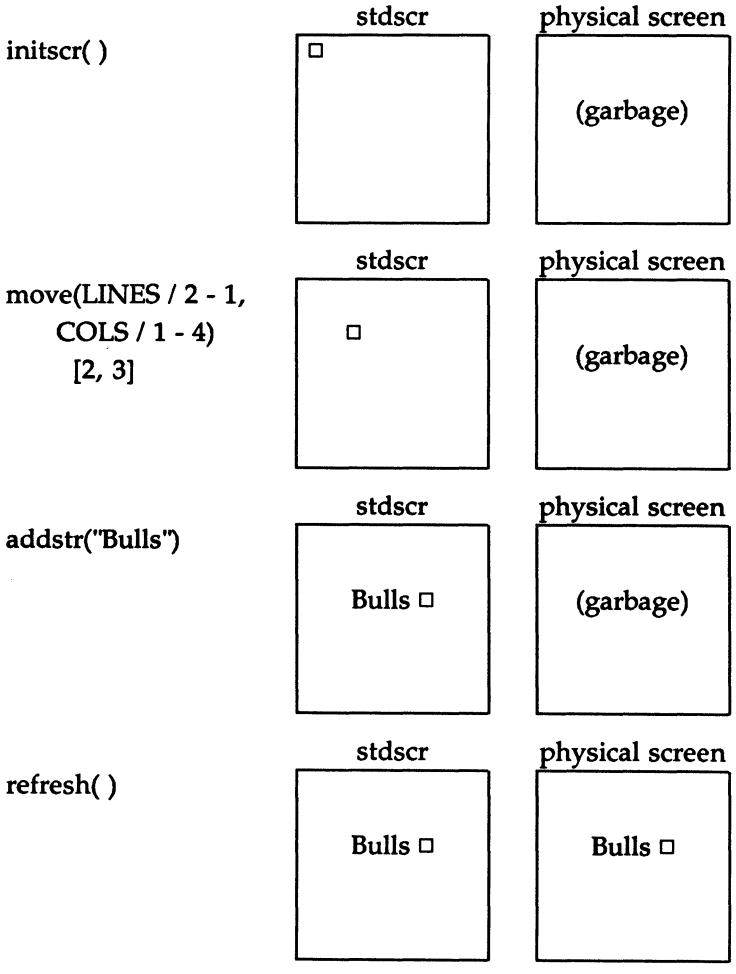


Figure 10-4. Relationship Between **stdscr** and a Terminal Screen (Sheet 1 of 2)

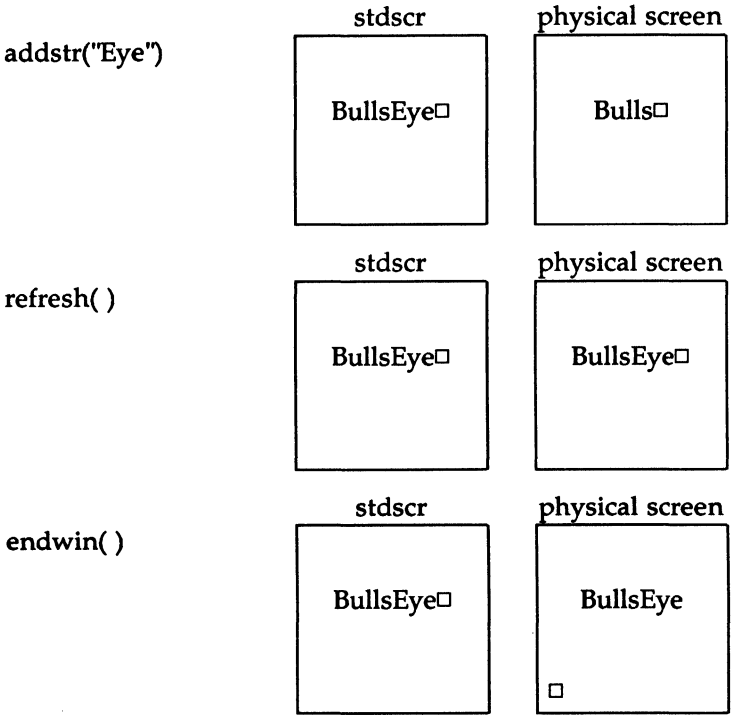


Figure 10-4. Relationship Between **stdscr** and a Terminal Screen (Sheet 2 of 2)

You can create other windows and use them instead of **stdscr**. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window.

It's possible to subdivide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It's also possible to create a window within a window; the smaller window is called a subwindow. Assume that you are designing an application that uses forms, for example, an expense voucher, as a user interface. You could use subwindows to control access to certain fields on the form.

Some **curses** routines are designed to work with a special type of window called a pad. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a

particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 10-5 represents what a pad, a subwindow, and some other windows could look like in comparison to a terminal screen.

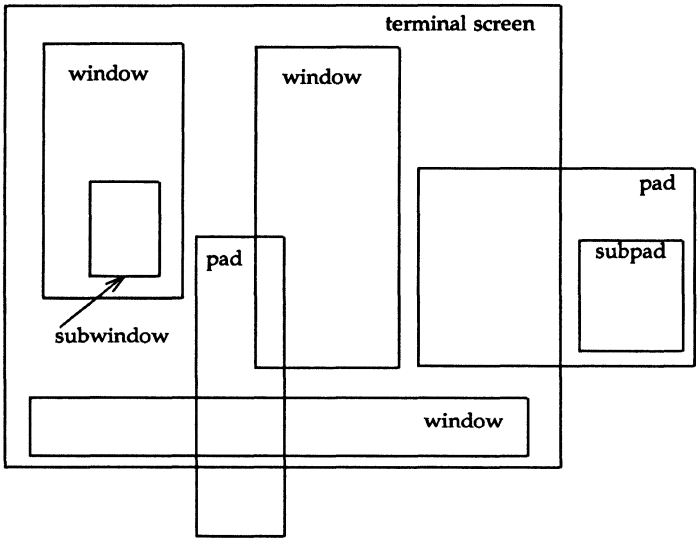


Figure 10-5. Multiple Windows and Pads Mapped to a Terminal Screen

The section "Building Windows and Pads" in this chapter describes the routines you use to create and use them. If you'd like to see a **curses** program with windows now, you can turn to the **window** program under the section "**curses** Program Examples" in this chapter.

Getting Simple Output and Input

Output

The routines that **curses** provides for writing to **stdscr** are similar to those provided by the **stdio**(3S) library for writing to a file. They let you

- write a character at a time — **addch()**
- write a string — **addstr()**
- format a string from a variety of input arguments — **printw()**
- move a cursor or move a cursor and print character(s) — **move()**, **mvaddch()**, **mvaddstr()**, **mvprintw()**
- clear a screen or a part of it — **clear()**, **erase()**, **clrtoeol()**, **clrtobot()**

Following are descriptions and examples of these routines.

> **CAUTION**
>
> The **curses** library provides its own set of output and input functions. You should not use other I/O routines or system calls, like **read**(2) and **write**(2), in a **curses** program. They may cause undesirable results when you run the program.

addch()

SYNOPSIS

#include <curses.h>

int addch(ch)
chtype ch;

NOTES

- **addch()** writes a single character to **stdscr**.
- The character is of the type **chtype**, which is defined in **<curses.h>**. **chtype** contains data and attributes (see "Output Attributes" in this chapter for information about attributes).
- When working with variables of this type, make sure you declare them as **chtype** and not as the basic type (for example, **short**) that **chtype** is declared to be in **<curses.h>**. This will ensure future compatibility.
- **addch()** does some translations. For example, it converts
  - the **<NL>** character to a clear to end of line and a move to the next line
  - the tab character to an appropriate number of blanks
  - other control characters to their ^X notation
- **addch()** normally returns **OK**. The only time **addch()** returns **ERR** is after adding a character to the lower right-hand corner of a window that does not scroll.
- **addch()** is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
  initscr();
  addch('a');
  refresh();
  endwin();
}
```

The output from this program will appear as follows:

```
a




$□
```

Also see the **show** program under "**curses** Example Programs" in this chapter.

addstr()

SYNOPSIS

```
#include <curses.h>

int addstr(str)
char *str;
```

NOTES

- **addstr()** writes a string of characters to **stdscr**.
- **addstr()** calls **addch()** to write each character.
- **addstr()** follows the same translation rules as **addch()**.
- **addstr()** returns **OK** on success and **ERR** on error.
- **addstr()** is a macro.

EXAMPLE

Recall the sample program that prints the character string `BullsEye`. See Figures 10-1, 10-2, and 10-4.

printw()

SYNOPSIS

#include <curses.h>

int printw(fmt [,arg...])
char *fmt

NOTES

- **printw**() handles formatted printing on **stdscr**.
- Like **printf**, **printw**() takes a format string and a variable number of arguments.
- Like **addstr**(), **printw**() calls **addch**() to write the string.
- **printw**() returns **OK** on success and **ERR** on error.

EXAMPLE

```
#include <curses.h>

main()
{
   char* title = "Not specified";
   int no = 0;

      /* Missing code. */

   initscr();

      /* Missing code. */

   printw("%s is not in stock.\n", title);
   printw("Please ask the cashier to order %d for you.\n", no);

   refresh();
   endwin();
}
```

The output from this program will appear as follows:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.



$ □
```

move()

SYNOPSIS

#include <curses.h>

int move(y, x);
int y, x;

NOTES

- **move()** positions the cursor for **stdscr** at the given row **y** and the given column **x**.
- Notice that **move()** takes the **y** coordinate before the **x** coordinate. The upper left-hand coordinates for **stdscr** are (0,0), the lower right-hand (**LINES** - 1, **COLS** - 1). See the section titled "The Routines **initscr()**, **refresh()**, and **endwin()**" for more information.
- **move()** may be combined with the write functions to form:
  - — **mvaddch(y, x, ch)**, which moves to a given position and prints a character
  - — **mvaddstr(y, x, str)**, which moves to a given position and prints a string of characters
  - — **mvprintw(y, x, fmt** [,*arg...*]),
    which moves to a given position and prints a formatted string.
- **move()** returns **OK** on success and **ERR** on error. Trying to move to a screen position of less than (0,0) or more than (**LINES** - 1, **COLS** - 1) causes an error.
- **move()** is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    addstr("Cursor should be here --> if move() works.");
    printw("\n\n\nPress <CR> to end test.");
    move(0,25);
    refresh();
    getch();      /* Gets <CR>; discussed below. */
    endwin();
}
```

Here's the output generated by running this program:

```
Cursor should be here -->□if move() works.


Press <CR> to end test.
```

After you press <**CR**>, the screen looks like this:

```
Cursor should be here -->


Press <CR> to end test.
$ □
```


See the **scatter** program under "**curses** Program Examples" in this chapter for another example of using **move**().

clear() and **erase()**

SYNOPSIS

#include <curses.h>

int clear()
int erase()

NOTES

- Both routines change **stdscr** to all blanks.
- **clear()** also assumes that the screen may have garbage that it doesn't know about; this routine first calls **erase()** and then **clearok()** which clears the physical screen completely on the next call to **refresh()** for **stdscr**. See the **curses**(3X) manual page for more information about **clearok()**.
- **initscr()** automatically calls **clear()**.
- **clear()** always returns **OK**; **erase()** returns no useful value.
- Both routines are macros.

clrtoeol() and **clrtobot**()

SYNOPSIS

#include <curses.h>

int clrtoeol()
int clrtobot()

NOTES

- **clrtoeol**() changes the remainder of a line to all blanks.
- **clrtobot**() changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

EXAMPLE

The following sample program uses **clrtobot**().

```
#include <curses.h>

main()
{
    initscr();
    addstr("Press <CR> to delete from here to the end of the line and on.");
    addstr("\nDelete this too.\nAnd this.");
    move(0,30);
    refresh();
    getch();
    clrtobot();
    refresh();
    endwin();
}
```

Here's the output generated by running this program:

```
Press <CR> to delete from here□to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to **refresh**(): one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press <**CR**>:

```
Press <CR> to delete from here

$ □
```


See the **show** and **two** programs under "**curses** Example Programs" for examples of uses for **clrtoeol**().

Input

curses routines for reading from the current terminal are similar to those provided by the **stdio**(3S) library for reading from a file. They let you:

- read a character at a time — **getch()**
- read a <**NL**>-terminated string — **getstr()**
- parse input, converting and assigning selected data to an argument list — **scanw()**

The primary routine is **getch()**, which processes a single input character and then returns that character. This routine is like the C library routine **getchar**()(3S) except that it makes several terminal- or system-dependent options available that are not possible with **getchar**(). For example, you can use **getch**() with the **curses** routine **keypad**(), which allows a **curses** program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key. See the descriptions of **getch**() and **keypad**() on the **curses**(3X) manual page for more information about **keypad**().

The following pages describe the basic routines for getting input in a screen program.

getch()

SYNOPSIS

#include <curses.h>

int getch()

NOTES

- **getch()** reads a single character from the current terminal.
- **getch()** returns the value of the character or **ERR** on 'end of file,' receipt of signals, or non-blocking read with no input.
- **getch()** is a macro.
- See the discussions about **echo(), noecho(), cbreak(), nocbreak(), raw(), noraw(), halfdelay(), nodelay(),** and **keypad()** below and in **curses**(3X).

EXAMPLE

```
#include <curses.h>

main()
{
   int ch;

   initscr();
   cbreak();          /* Explained later in the section "Input Options" */
   addstr("Press any character:  ");
   refresh();
   ch = getch();
   printw("\n\n\nThe character entered was a '%c'.\n", ch);
   refresh();
   endwin();
}
```

The output from this program follows. The first **refresh**() sends the **addstr**() character string from **stdscr** to the terminal:

```
Press any character:  □
```

Then assume that a **w** is typed at the keyboard. **getch**() accepts the character and assigns it to **ch**. Finally, the second **refresh**() is called and the screen appears as follows:


```
Press any character:  w

The character entered was a 'w'.

$□
```

For another example of **getch**(), see the **show** program under "**curses** Example Programs" in this chapter.

getstr()

SYNOPSIS

#include <curses.h>

int getstr(str)
char *str;

NOTES

- **getstr**() reads characters and stores them in a buffer until a <**CR**>, <**NL**>, or <**ENTER**> is received from **stdscr**. **getstr**() does not check for buffer overflow.
- The characters read and stored are in a character string.
- **getstr**() is a macro; it calls **getch**() to read each character.
- **getstr**() returns **ERR** if **getch**() returns **ERR** to it. Otherwise it returns **OK**.
- See the discussions about **echo**(), **noecho**(), **cbreak**(), **nocbreak**(), **raw**(), **noraw**(), **halfdelay**(), **nodelay**(), and **keypad**() below and in **curses**(3X).

EXAMPLE

```
#include <curses.h>

main()
{
char str[256];

   initscr();
   cbreak();       /* Explained later in the section "Input Options" */
   addstr("Enter a character string terminated by <CR>:\n\n");
   refresh()
   getstr(str);
   printw("\n\n\nThe string entered was \n'%s'\n", str);
   refresh();
   endwin();
}
```

Assume you entered the string 'I enjoy learning about the operating system.' The final screen (after entering <CR>) would appear as follows:

```
Enter a character string terminated by <CR>:

I enjoy learning about the operating system.


The string entered was
'I enjoy learning about the operating system.'

$□
```

scanw()

SYNOPSIS

```
#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;
```

NOTES

- **scanw()** calls **getstr()** and parses an input line.
- Like **scanf**(3S), **scanw()** uses a format string to convert and assign to a variable number of arguments.
- **scanw()** returns the same values as **scanf()**.
- See **scanf**(3S) for more information.

EXAMPLE

```
#include <curses.h>

main()
{
    char string[100];
    float number;

    initscr();
    cbreak();           /* Explained later in the  */
    echo();             /* section "Input Options" */
    addstr("Enter a number and a string separated by a comma: ");
    refresh();
    scanw("%f,%s",&number,string);
    clear();
    printw("The string was \"%s\" and the number was %f.",string,number);
    refresh();
    endwin();
}
```

Notice the two calls to **refresh**(). The first call updates the screen with the character string passed to **addstr**(), the second with the string returned from **scanw**(). Also notice the call to **clear**(). Assume you entered the following when prompted: **2,twin**. After running this program, your terminal screen would appear, as follows:

```
The string was "twin" and the number was 2.000000.


$□
```

10

Controlling Output and Input

Output Attributes

When we talked about **addch**(), we said that it writes a single character of the type **chtype** to **stdscr**. **chtype** has two parts: a part with information about the character itself and another part with information about a set of attributes associated with the character. The attributes allow a character to be printed in reverse video, bold, underlined, and so on.

stdscr always has a set of current attributes that it associates with each character as it is written. However, using the routine **attrset**() and related **curses** routines described below, you can change the current attributes. Below is a list of the attributes and what they mean:

- A_BLINK — blinking
- A_BOLD — extra bright or bold
- A_DIM — half bright
- A_REVERSE — reverse video
- A_STANDOUT — a terminal's best highlighting mode
- A_UNDERLINE — underlining
- A_ALTCHARSET — alternate character set (see the section "Drawing Lines and Other Graphics" in this chapter)

To use these attributes, you must pass them as arguments to **attrset**() and related routines; they can also be ORed with the bitwise OR (|) to **addch**().

NOTE

> Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, a **curses** program attempts to find a substitute attribute. If none is possible, the attribute is ignored.

Let's consider a use of one of these attributes. To display a word in bold, you would use the following code:


```
 ...
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
 ...
refresh();
```

Attributes can be turned on singly, such as **attrset**(A_BOLD) in the example, or in combination. To turn on blinking bold text, for example, you would use

attrset(A_BLINK | A_BOLD). Individual attributes can be turned on and off with the **curses** routines **attron**() and **attroff**() without affecting other attributes. **attrset(0)** turns all attributes off.

Notice the attribute called A_STANDOUT. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the A_STANDOUT attribute is recommended. Two convenient functions, **standout**() and **standend**() can be used to turn this attribute on and off; **standend**() turns off all attributes.

In addition to the attributes listed above, there are two bit masks called A_CHARTEXT and A_ATTRIBUTES. You can use these bit masks with the **curses** function **inch**() and the C logical AND (**&**) operator to extract the character or attributes of a position on a terminal screen. See the discussion of **inch**() on the **curses**(3X) manual page.

Following are descriptions of **attrset**() and the other **curses** routines that you can use to manipulate attributes.

attron(), attrset(), and attroff()

SYNOPSIS

```
#include <curses.h>

int attron( attrs )
chtype attrs;

int attrset( attrs )
chtype attrs;

int attroff( attrs )
chtype attrs;
```

NOTES

- **attron()** turns on the requested attribute **attrs** in addition to any that are currently on. **attrs** is of the type **chtype** and is defined in **<curses.h>**.
- **attrset()** turns on the requested attributes **attrs** instead of any that are currently turned on.
- **attroff()** turns off the requested attributes **attrs** if they are on.
- The attributes may be combined using the bitwise OR (|).
- All return **OK**.

EXAMPLE

See the **highlight** program under "**curses** Example Programs" in this chapter.

standout() and **standend()**

SYNOPSIS

#include <curses.h>

int standout()
int standend()

NOTES

- **standout()** turns on the preferred highlighting attribute, A_STANDOUT, for the current terminal. This routine is equivalent to **attron**(A_STANDOUT).
- **standend()** turns off all attributes. This routine is equivalent to **attrset**(0).
- Both always return **OK**.

EXAMPLE

See the **highlight** program under "**curses** Example Programs" in this chapter.

Bells and Flashing Screens

Occasionally, you may want to get a user's attention. Two **curses** routines were designed to help you do this. They let you ring the terminal's chimes and flash its screen.

flash() flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within earshot of the user. The routine **beep**() can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep**() will flash the screen.)

beep() and **flash**()

SYNOPSIS

#include <curses.h>

int flash()
int beep()

NOTES

- **flash**() tries to flash the terminals screen, if possible, and, if not, tries to ring the terminal bell.
- **beep**() tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.
- Neither returns any useful value.

Input Options

The operating system does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- echoes (prints back) characters to a terminal as they are typed
- interprets an erase character (typically #) and a line kill character (typically @)
- interprets a CTRL-D (control d) as end of file (EOF)
- interprets interrupt and quit characters
- strips the character's parity bit
- translates <**CR**> to <**NL**>

Because a **curses** program maintains total control over the screen, **curses** turns off echoing on the operating system and does echoing itself. At times, you may not want the operating system to process other characters in the standard way in an interactive screen management program. Some **curses** routines, **noecho**() and **cbreak**(), for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Table 10-1 shows some of the major routines for controlling input.

Every **curses** program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak**(), **raw**(), **nocbreak**(), or **noraw**() mode. Although the **curses** program starts up in **echo**() mode, as Table 10-1 shows, none of the other modes are guaranteed.

The combination of **noecho**() and **cbreak**() is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The **curses** routine **noecho**() is designed for this purpose. However, when **noecho**() turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak**() causes these characters to be uninterpreted.

TABLE 10-1. Input Option Settings for **curses** Programs

Input Options	Characters	
	Interpreted	**Uninterpreted**
Normal 'out of **curses** state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal **curses** 'start up state'	echoing (simulated)	All else undefined.
cbreak() and **echo()**	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and **noecho()**	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and **noecho()**	break, quit stripping erase, kill EOF	echoing
nocbreak() and **echo()**	See caution below.	
nl()	<CR> to <NL>	
nonl()		<CR> to <NL>
raw() (instead of **cbreak()**)		break, quit stripping

10

CAUTION

Do not use the combination **nocbreak**() and **noecho**(). If you use it in a program and also use **getch**(), the program will go in and out of **cbreak**() mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted in Table 10-1, you can use the **curses** routines **noraw(), halfdelay**(), and **nodelay**() to control input. See the **curses**(3X) manual page for discussions of these routines.

The next few pages describe **noecho**(), **cbreak**() and the related routines **echo**() and **nocbreak**() in more detail.

echo() and **noecho**()

SYNOPSIS

#include <curses.h>

int echo()
int noecho()

NOTES

- **echo**() turns on echoing of characters by **curses** as they are read in. This is the initial setting.
- **noecho**() turns off the echoing.
- Neither returns any useful value.
- **curses** programs may not run properly if you turn on echoing with **nocbreak**(). See Table 10-1 and accompanying caution. After you turn echoing off, you can still echo characters with **addch**().

EXAMPLE

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

cbreak() and **nocbreak**()

SYNOPSIS

```
#include < curses.h >
int cbreak()
int nocbreak()
```

NOTES

- **cbreak**() turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.
- **nocbreak**() returns to normal 'line at a time' processing. This is typically the initial setting.
- Neither returns any useful value.
- A **curses** program may not run properly if **cbreak**() is turned on and off within the same program or if the combination **nocbreak**() and **echo**() is used.
- See Table 10-1 and accompanying caution.

EXAMPLE

See the **editor** and **show** programs under "**curses** Program Examples" in this chapter.

Building Windows and Pads

An earlier section in this chapter, "More about **refresh**() and Windows" explained what windows and pads are and why you might want to use them. This section describes the **curses** routines you use to manipulate and create windows and pads.

Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with **stdscr**. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of a **stdscr** routine and adding the window name as the first parameter. For example, **addch**('c') would become **waddch(mywin, 'c')** if you wanted to write the character **c** to the window **mywin**. Here's a list of the window (or **w**) versions of the output routines discussed in "Getting Simple Output and Input."

- **waddch(***win, ch***)**
- **mvwaddch(***win, y, x, ch***)**
- **waddstr(***win, str***)**
- **mvwaddstr(***win, y, x, str***)**
- **wprintw(***win, fmt [, arg...]***)**
- **mvwprintw(***win, y, x, fmt [, arg...]***)**
- **wmove(***win, y, x***)**
- **wclear(***win***)** and **werase(***win***)**
- **wclrtoeol(***win***)** and **wclrtobot(***win***)**
- **wrefresh**()

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the addition of a *win* argument. Notice that the routines whose names begin with **mvw** take the *win* argument before the *y, x* coordinates, which is contrary to what the names imply. See **curses**(3X) for more information about these routines or the versions of the input routines **getch, getstr**(), and so on that you should use with windows.

All **w** routines can be used with pads except for **wrefresh**() and **wnoutrefresh**() (see below). In place of these two routines, you have to use **prefresh**() and **pnoutrefresh**() with pads.

The Routines wnoutrefresh() and doupdate()

If you recall from the earlier discussion about **refresh()**, we said that it sends the output from **stdscr** to the terminal screen. We also said that it was a macro that expands to **wrefresh(stdscr)** (see "What Every **curses** Program Needs" and "More about **refresh()** and Windows").

The **wrefresh()** routine is used to send the contents of a window (**stdscr** or one that you create) to a screen; it calls the routines **wnoutrefresh()** and **doupdate()**. Similarly, **prefresh()** sends the contents of a pad to a screen by calling **pnoutrefresh()** and **doupdate()**.

Using **wnoutrefresh()**—or **pnoutrefresh()** (this discussion will be limited to the former routine for simplicity)—and **doupdate()**, you can update terminal screens with more efficiency than using **wrefresh()** by itself. **wrefresh()** works by first calling **wnoutrefresh()**, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling **wnoutrefresh()**, **wrefresh()** then calls **doupdate()**, which compares the virtual screen to the physical screen and does the update. If you want to output several windows at once, calling **wrefresh()** will result in alternating calls to **wnoutrefresh()** and **doupdate()**, causing several bursts of output to a screen. However, by calling **wnoutrefresh()** for each window and then **doupdate()** only once, you can minimize the total number of characters transmitted and the processor time used. The following sample program uses only one **doupdate()**:

```
#include <curses.h>

main()
{
    WINDOW *w1, *w2;

    initscr();
    w1 = newwin(2,6,0,3);
    w2 = newwin(1,4,5,4);
    waddstr(w1, "Bulls");
    wnoutrefresh(w1);
    waddstr(w2, "Eye");
    wnoutrefresh(w2);
    doupdate();
    endwin();
}
```

Notice from the sample that you declare a new window at the beginning of a **curses** program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named `w1` and `w2` with the routine **newwin**() according to certain specifications. **newwin**() is discussed in more detail below.

Figure 10-7 illustrates the effect of **wnoutrefresh**() and **doupdate**() on these two windows, the virtual screen, and the physical screen.

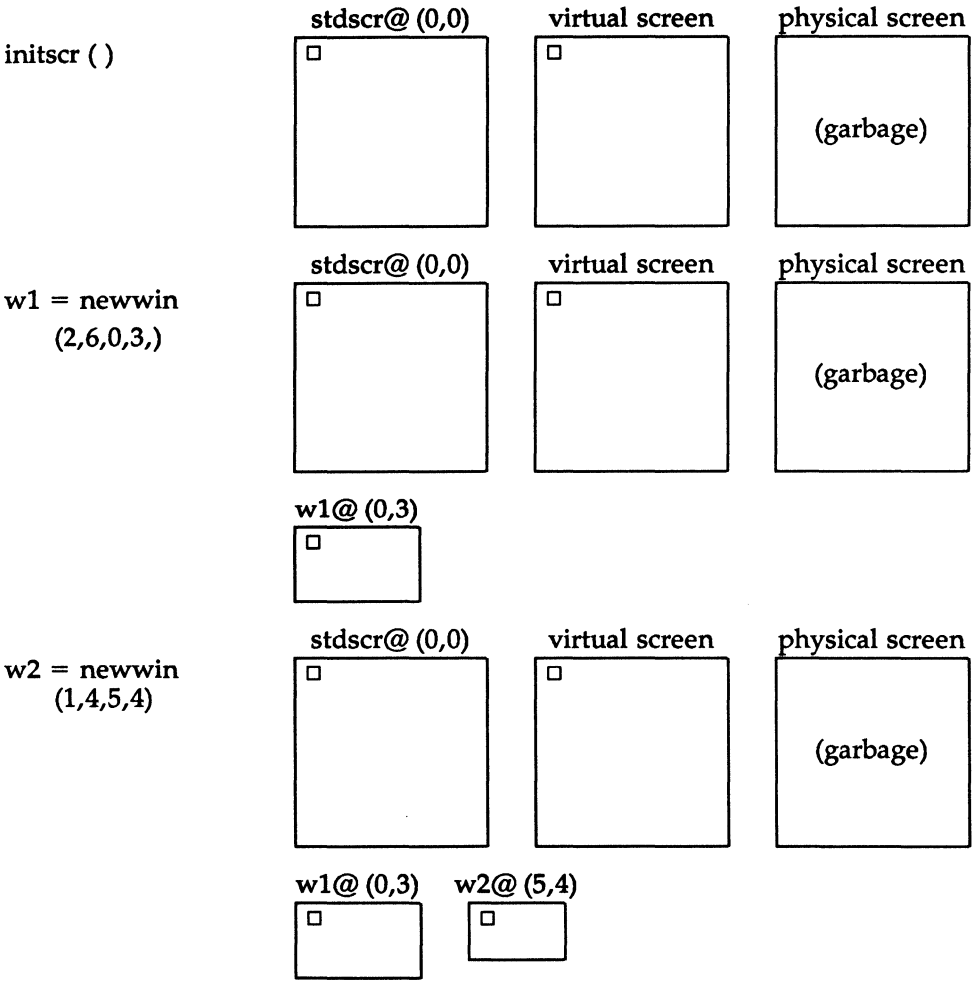


Figure 10-6. Relationship Between a Window and a Terminal Screen (Sheet 1 of 3)

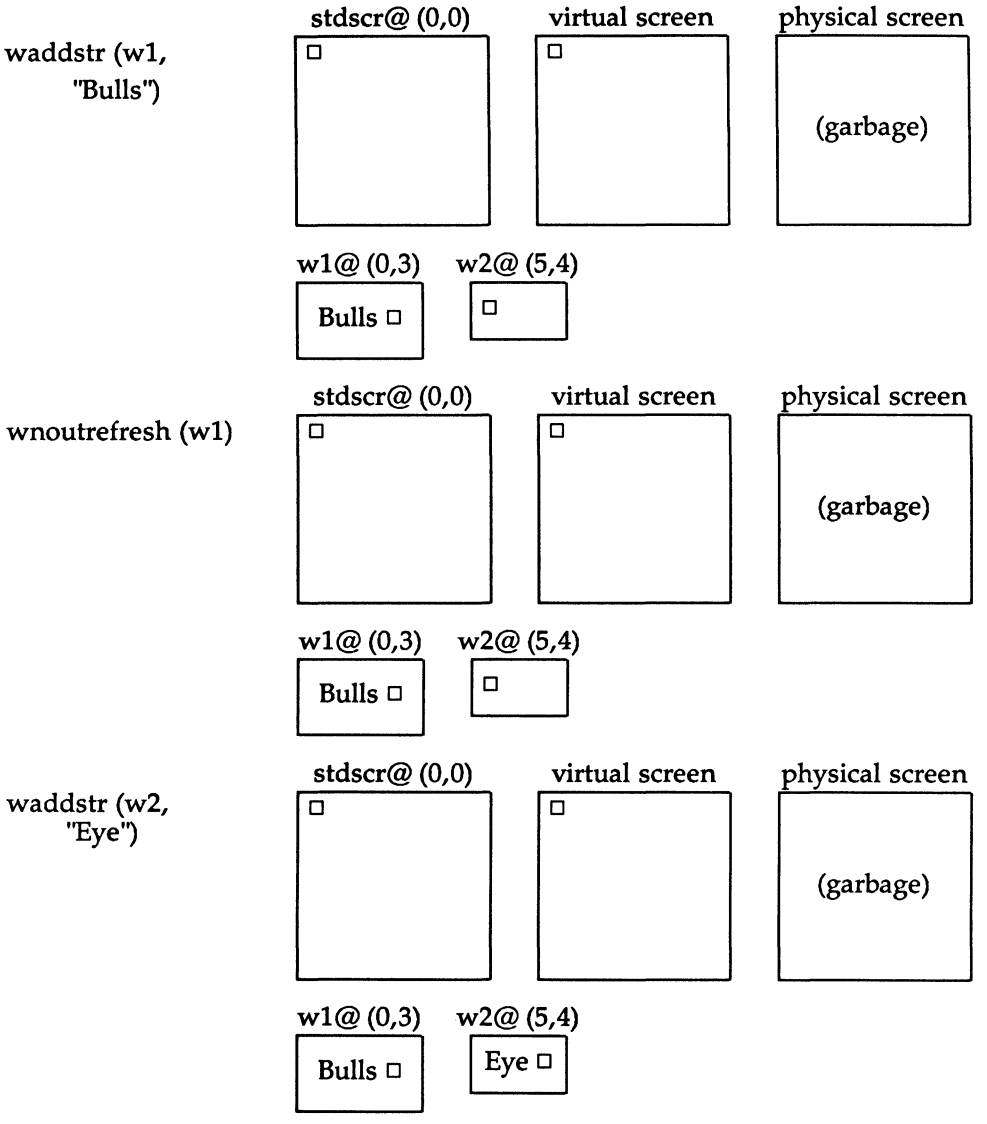


Figure 10-6. Relationship Between a Window and a Terminal Screen (Sheet 2 of 3)

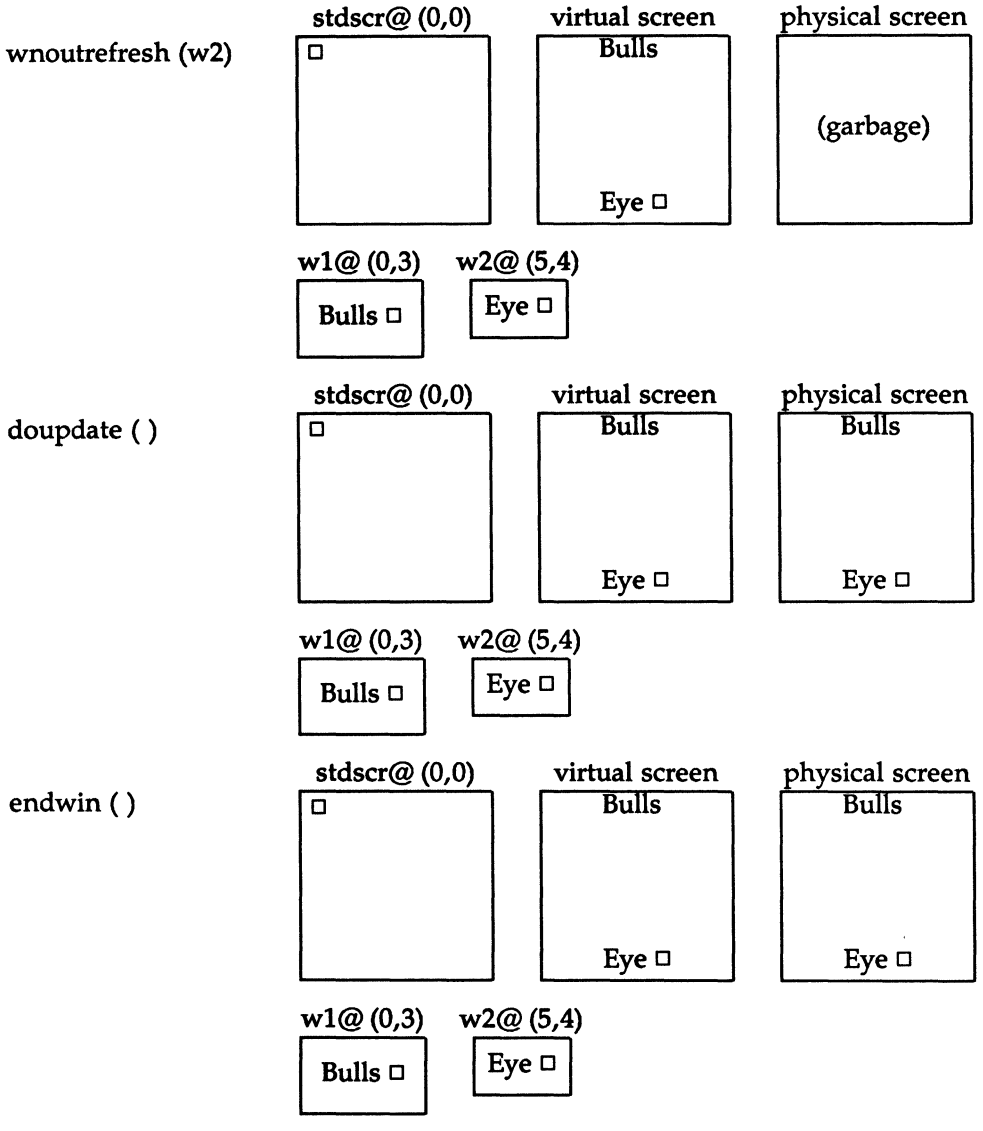


Figure 10-6. Relationship Between a Window and a Terminal Screen (Sheet 3 of 3)

New Windows

Following are descriptions of the routines **newwin**() and **subwin**(), which you use to create new windows. For information about creating new pads with **newpad**() and **subpad**(), see the **curses**(3X) manual page.

newwin()

SYNOPSIS

#include <curses.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;

NOTES

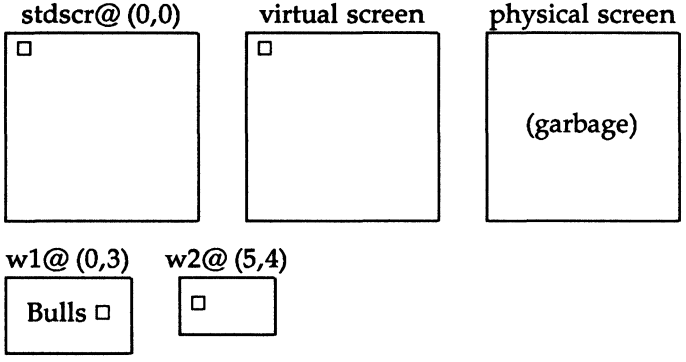
- **newwin**() returns a pointer to a new window with a new data area.
- The variables **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

EXAMPLE

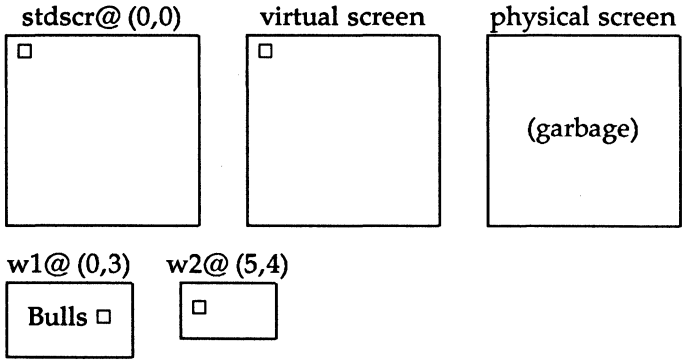
Recall the sample program using two windows; see Figure 10-6. Also see the **window** program under "**curses** Program Examples" in this chapter.

SYNOPSIS

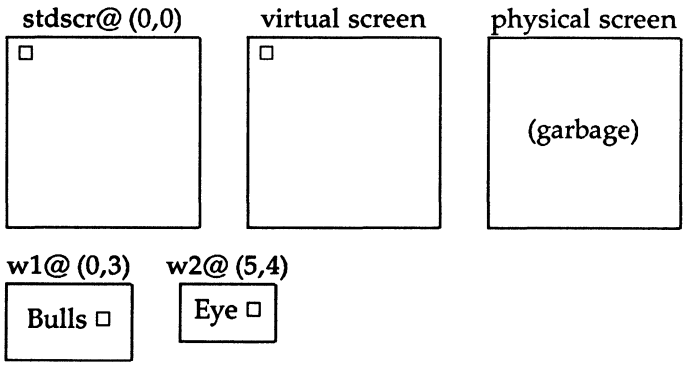
waddstr (w1, "Bulls")



wnoutrefresh (w1)



waddstr (w2, "Eye")



#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;

NOTES

- **subwin**() returns a new window that points to a section of another window, **orig**.
- **nlines** and **ncols** give the size of the new window.
- **begin_y** and **begin_x** give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.
- Subwindows and original windows can accidentally overwrite one another.

> **CAUTION**
>
> Subwindows of subwindows do not work (as of the copyright date of this *Programmer's Guide*).

EXAMPLE

```
#include <curses.h>

main()
{
 WINDOW *sub;

  initscr();
  box(stdscr,'w','w');      /* See the curses(3X) manual page for box() */
  mvwaddstr(stdscr,7,10,"------- this is 10,10");
  mvwaddch(stdscr,8,10,'|');
  mvwaddch(stdscr,9,10,'v');
  sub = subwin(stdscr,10,20,10,10);
  box(sub,'s','s');
  wnoutrefresh(stdscr);
  wrefresh(sub);
  endwin();
}
```

This program prints a border of **w**s around the stdscr (the sides of your terminal screen) and a border of s's around the subwindow **sub** when it is run. For another example, see the **window** program under "**curses** Program Examples" in this chapter.

Using Advanced curses Features

Knowing how to use the basic **curses** routines to get output and input and to work with windows, you can design screen management programs that meet the needs of many users. The **curses** library, however, has routines that let you do more in a program than handle I/O and multiple windows. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single **curses** program.

You should be comfortable using the routines previously discussed in this chapter and the other routines for I/O and window manipulation discussed on the **curses**(3X) manual page before you try to use the advanced **curses** features.

> CAUTION
>
> The routines described under "Routines for Drawing Lines and Other Graphics" and "Routines for Using Soft Labels" are features that are new for Release 3. If a program uses any of these routines, it may not run on earlier releases of the operating system. You must use the Release 3 version of the **curses** library on Release 3 to work with these routines.

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in **curses** programs. **curses** use the same names for glyphs as the VT100 line drawing character set.

To use the alternate character set in a **curses** program, you pass a set of variables whose names begin with ACS_ to the **curses** routine **waddch**() or a related routine. For example, ACS_ULCORNER is the variable for the upper left corner glyph. If a terminal has a line drawing character for this glyph, ACS_ULCORNER's value is the terminal's character for that glyph OR'd (|) with the bit-mask A_ALTCHARSET. If no line drawing character is available for that glyph, a standard ASCII character that approximates the glyph is stored in its place. For example, the default character for ACS_HLINE, a horizontal line, is a – (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard ACS_ names and their defaults are listed on the **curses**(3X) manual page.

Part of an example program that uses line drawing characters follows. The example uses the **curses** routine **box**() to draw a box around a menu on a screen. **box**() uses the line drawing characters by default or when | (the pipe) and – are chosen. (See **curses**(3X).) Up and down more indicators are drawn on the box border (using **ACS_UARROW** and **ACS_DARROW**) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);
  ...

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
   waddch(menuwin, ACS_UARROW);
else
   addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
   waddch(menuwin, ACS_DARROW);
else
   waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter v) isn't clearly discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.


```
if ( ! (ACS_DARROW & A_ALTCHARSET))
   ACS_DARROW = 'V';
```

For more information, see **curses**(3X) in the *Programmer's Reference Manual.*

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The **curses** library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for a **curses** program to make use of them.

Let's briefly discuss most of the **curses** routines needed to use soft labels: **slk_init()**, **slk_set()**, **slk_refresh()** and **slk_noutrefresh()**, **slk_clear**, and **slk_restore**.

When you use soft labels in a **curses** program, you have to call the routine **slk_int()** before **initscr()**. This sets an internal flag for **initscr()** to look at that says to use the soft labels. If **initscr()** discovers that there are fewer than eight soft labels on the screen, that their size is less than eight characters, or that there is no way to program them, then it will remove a line from the bottom of **stdscr** to use for the soft labels. The size of **stdscr** and the **LINES** variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the **LINES** and **COLS** variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from **stdscr**. The choices are between a 3-2-3 arrangement as appears on AT&T terminals, or a 4-4 arrangement as appears on Hewlett-Packard terminals. The **curses** routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine **slk_set()** takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left justified, 1 = centered, and 2 = right justified).

The routine **slk_noutrefresh()** is comparable to **wnoutrefresh()** in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a **wrefresh()** commonly follows, **slk_noutrefresh()** is the function that is most commonly used to output the labels.

Just as **wrefresh()** is equivalent to a **wnoutrefresh()** followed by a **doupdate()**, so too the function **slk_refresh()** is equivalent to a **slk_noutrefresh()** followed by a **doupdate()**.

To prevent the soft labels from getting in the way of a shell escape, **slk_clear**() may be called before doing the **endwin**(). This clears the soft labels off the screen and does a **doupdate**(). The function **slk_restore**() may be used to restore them to the screen. See the **curses**(3X) manual page for more information about the routines for using soft labels.

Working with More than One Terminal

A **curses** program can produce output on more than one terminal at the same time. This is useful for single process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the **curses** library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and the kind of terminal on each of those lines. The standard method, checking **$TERM** in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an inter-terminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

A **curses** program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals in a **curses** program have the type **SCREEN***. A new terminal is initialized by calling **newterm**(*type, outfd, infd*). **newterm** returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **stdio**(3S) file pointer (**FILE***) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to **initscr**(), which calls **newterm(getenv("TERM"), stdout, stdin)**.

To change the current terminal, call **set_term(***sp***)** where *sp* is the screen reference to be made current. **set_term**() returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**(). Options such as **cbreak**() and **noecho**() must be set separately for each terminal. The functions **endwin**() and **refresh**() must be called separately for each terminal. Figure 10-7 shows a typical scenario to output a message to several terminals.

```
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Figure 10-7. Sending a Message to Several Terminals

See the **two** program under "**curses** Program Examples" in this chapter for a more complete example.

Working with terminfo Routines

Some programs need to use lower-level routines (i.e., primitives) than those offered by the **curses** routines. For such programs, the **terminfo** routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use to manipulate the terminal.

There are three circumstances when it is proper to use **terminfo** routines. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the **terminfo** routines is worthwhile. The third is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines: the higher-level **curses** routines make your program more portable to other operating systems and to a wider class of terminals.

NOTE

You are discouraged from using **terminfo** routines, except for the purposes noted, because **curses** routines take care of all the glitches present in physical terminals. When you use the **terminfo** routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.

What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in Figure 10-8.

```
#include <curses.h>
#include <term.h>
...
    setupterm( (char*)0, 1, (int*)0 );
    ...
    putp(clear_screen);
    ...
    reset_shell_mode();
    exit(0);
```

Figure 10-8. Typical Framework of a **terminfo** Program

The header files **<curses.h>** and **<term.h>** are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm**() takes care of initialization. Passing this routine the values **(char*)0, 1,** and **(int*)0** invokes reasonable defaults. If **setupterm**() can't determine the kind of terminal you are on, it prints an error message and exits. **reset_shell_mode**() performs functions similar to **endwin**() and should be called before a **terminfo** program exits.

A global variable like **clear_screen** is defined by the call to **setupterm**(). It can be output using the **terminfo** routines **putp**() or **tputs**(), which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf**(3S), because it contains padding information. A program

that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher level routines like **addch**() and **getch**() are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4); see **curses**(3X) for a list of all the **terminfo** routines.

Compiling and Running a terminfo Program

The general command line for compiling and the guidelines for running a program with **terminfo** routines are the same as those for compiling any other **curses** program. See the sections "Compiling a **curses** Program" and "Running a **curses** Program" in this chapter for more information.

An Example terminfo Program

The example program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see "**curses** Program Examples") that does not use the higher level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */


#include <curses.h>
#include <term.h>

int ulmode = 0;                    /* Currently underlining */

main(argc, argv)
  int argc;
  char **argv;
{
   FILE *fd;
   int c, c2;
   int outch();

   if (argc > 2)
   {
      fprintf(stderr, "Usage: termhl [file]\n");
      exit(1);
   }
```

```
    if (argc == 2)
    {
        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
            perror(argv[1]);
            exit(2);
        }
    }
    else
    {
        fd = stdin;
    }
    setupterm((char*)0, 1, (int*)0);

    for (;;)
    {
        c = getc(fd);
        if (c == EOF)
        break;
        if (c == '\')
        {
            c2 = getc(fd);
            switch (c2)
            {
                case 'B':
                tputs(enter_bold_mode, 1, outch);
                continue;
                case 'U':
                tputs(enter_underline_mode, 1, outch);
                ulmode = 1;
                continue;
                case 'N':
                tputs(exit_attribute_mode, 1, outch);
                ulmode = 0;
                continue;
            }
            putch(c);
            putch(c2);
        }
        else
            putch(c);
    }
    fclose(fd);
    fflush(stdout);
    resetterm();
    exit(0);
}
```

```
/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
   int c;
{
   outch(c);
   if (ulmode && underline_char)
   {
      outch('\b');
      tputs(underline_char, 1, outch);
   }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
   int c;
{
   putchar(c);
}
```

A discussion of the use of the function **tputs**(*cap, affcnt, outc*) in this program will provide some insight into the **terminfo** routines. **tputs**() applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the **terminfo** database probably contain strings like **$<20>**, which means to pad for 20 milliseconds (see the following section "Specify Capabilities" in this chapter). **tputs** generates enough pad characters to delay for the appropriate time.

tput() has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls **putchar**. For these programs, the routine **putp**(*cap*) is a convenient abbreviation. **termhl** could be simplified by using **putp**().

Now to understand why you should use the **curses** level routines instead of **terminfo** level routines whenever possible, note the special check for the **underline_char** capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, outputs **underline_char**, if necessary. Low-level details such as this are precisely why the **curses** level is recommended over the **terminfo** level. **curses** takes care of terminals with different methods of underlining and other terminal functions. Programs at the **terminfo** level must handle such details themselves.

termhl was written to illustrate a typical use of the **terminfo** routines. It is more complex than it need be in order to illustrate some properties of **terminfo** programs. The routine **vidattr** (see **curses**(3X)) could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

Working with the terminfo Database

The **terminfo** database describes the many terminals with which **curses** programs, as well as some operating system tools, like **vi**(1), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

Writing Terminal Descriptions

Descriptions of many popular terminals are already included in the **terminfo** database. However, it is possible that you'll want to run a **curses** program on a terminal for which there is not currently a description. In that case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.
3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier.

In the next few pages, we follow each step required to build a terminal description for the fictitious terminal named "myterm."

Name the Terminal

The name of a terminal is the first information given in a **terminfo** terminal description. This string of names, assuming there is more than one name, is separated by pipe symbols (|). The first name given should be the most common abbreviation for the terminal. The last name given should be a long name that fully identifies the terminal. The long name is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description of the AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```


Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a **–w**) version of our fictitious terminal would be described as **myterm–w**. **term**(5) describes mode indicators in greater detail.

Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.
- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways — type:

  stty –echo; cat –vu
  Type in the keys you want to test;
  for example, see what right arrow (→) transmits.
  <CR>
  <CTRL-D>
  stty echo

  or:

  cat >dev/null
  Type in the escape sequences you want to test;
  for example, see what **\E[H** transmits.
  <CTRL-D>

- The first line in each of these testing methods sets up the terminal to carry out the tests. The **<CTRL-D>** helps return the terminal to its normal settings.
- See the **terminfo**(4) manual page. It lists all the capability names you have to use in a terminal description. The following section, "Specify Capabilities," gives details.

Specify Capabilities

Once you know the capabilities of your terminal, you have to describe them in your terminal description. You describe them with a string of comma-separated fields that contain the abbreviated **terminfo** name and, in some cases, the terminal's value for each capability. For example, **bel** is the abbreviated name for the beeping or ringing capability. On most terminals, a CTRL-G is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as **bel=^G,**.

The list of capabilities may continue onto multiple lines as long as white space (that is, tabs and spaces) begins every line but the first of the description. Comments can be included in the description by putting a # at the beginning of the line.

The **terminfo**(4) manual page has a complete list of the capabilities you can use in a terminal description. This list contains the name of the capability, the abbreviated name used in the database, the two-letter code that corresponds to the old **termcap** database name, and a short description of the capability. The abbreviated name that you will use in your database descriptions is shown in the column titled "Capname."

NOTE

For a **curses** program to run on any given terminal, its description in the **terminfo** database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

A terminal's character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters follow:

This shows a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as **cols#80,**.

= This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:

- **^** This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as **^G**.
- **\E or \e** These characters followed by another character show an escape instruction. An entry of \EC would transmit to the terminal as ESCAPE-C.
- **\n** These characters provide a <**NL**> character sequence.
- **\l** These characters provide a linefeed character sequence.

\r	These characters provide a return character sequence.
\t	These characters provide a tab character sequence.
\b	These characters provide a backspace character sequence.
\f	These characters provide a formfeed character sequence.
\s	These characters provide a space character sequence.
nnn	This is a character whose three-digit octal is *nnn*, where *nnn* can be one to three digits.
$< >	These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the "less than/greater than" symbols (< >). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as **$<20*>**. See the **terminfo**(4) manual page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as:

.bel=^G,

With this background information about specifying capabilities, let's add the capability string to our description of myterm. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

10

Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **terminfo** name for each capability in the parentheses following the capability description:

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (**am**).
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is ^G (**bel**).

- An 80-column wide screen (**cols**).
- A 30-line long screen (**lines**).
- Use of xon/xoff protocol (**xon**).

By combining the name string (see the section "Name the Terminal") and the capability descriptions that we now have, we get the following general **terminfo** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
      am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal myterm has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A <**CR**> is a CTRL-M (**cr**).
- A cursor up one line motion is a CTRL-K (**cuu1**).
- A cursor down one line motion is a CTRL-J (**cud1**).
- Moving the cursor to the left one space is a CTRL-H (**cub1**).
- Moving the cursor to the right one space is a CTRL-L (**cuf1**).
- Entering reverse video mode is an ESCAPE-D (**smso**).
- Exiting reverse video mode is an ESCAPE-Z (**rmso**).
- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (**el**).
- A terminal scrolls when receiving a <**NL**> at the bottom of a page (**ind**).

The revised terminal description for myterm including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
      am, bel=^G, cols#80, lines#30, xon,
      cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
      smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are, as follows:

- The backspace key generates a CTRL-H (**kbs**).
- The up arrow key generates an ESCAPE-[A (**kcuu1**).
- The down arrow key generates an ESCAPE-[B (**kcud1**).
- The right arrow key generates an ESCAPE-[C (**kcuf1**).
- The left arrow key generates an ESCAPE-[D (**kcub1**).
- The home key generates an ESCAPE-[H (**khome**).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
      am, bel=^G, cols#80, lines#30, xon,
      cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
      smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
      kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
      kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters — for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the **cup** capability is used and is passed two parameters: the row and column to address. String capabilities, such as **cup** and set attributes (**sgr**) capabilities, are passed arguments in a **terminfo** program by the **tparm**() routine.

The arguments to string capabilities are manipulated with special '%' sequences similar to those found in a **printf**(3S) statement. In addition, many of the features found on a simple stack-based RPN calculator are available. **cup**, as noted above, takes two arguments: the row and column. **sgr**, takes nine arguments, one for each of the nine video attributes. See **terminfo**(4) for the list and order of the attributes and further examples of **sgr**.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed

with H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence 'ESCAPE-[6 ; 19 H' would be output.

Integer arguments are pushed onto the stack with a '%p' sequence followed by the argument number, such as '%p2' to push the second argument. A shorthand sequence to increment the first two arguments is '%i'. To output the top number on the stack as a decimal, a '%d' sequence is used, exactly as in **printf**. Our terminal's **cup** sequence is built up as follows:

```
cup=\E[%i%p1%d;%p2%dH,
```

The elements of the sequence have the meanings listed below:

cup=	Meaning
\E[	output ESCAPE-[
%i	increment the two arguments
%p1	push the first argument (the row) onto the stack
%d	output the row as a decimal
;	output a semicolon
%p2	push the second argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
      am, bel=^G, cols#80, lines#30, xon,
      cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
      smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
      kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
      kcub1=\E[D, khome=\E[H,
      cup=\E[%i%p1%d;%p2%dH,
```


See **terminfo**(4) for more information about parameter string capabilities.

Compile the Description

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **.ti**. For example, the description of myterm would be in a source file named **myterm.ti**. The compiled description of myterm would usually be placed in **/usr/lib/terminfo/m/myterm**, since the first letter in the description entry is **m**. Links would also be made to synonyms of **myterm**, for example, to **/f/fancy**. If the environment variable **$TERMINFO** were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the **$TERMINFO** directory. All programs using the entry would then look in the new directory for the description file if **$TERMINFO** were set, before looking in the default **/usr/lib/terminfo**. The general format for the **tic** compiler is as follows:

tic [**–v**] [**–c**] *file*

The **–v** option causes the compiler to trace its actions and output information about its progress. The **–c** option causes a check for errors; it may be combined with the **–v** option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first use **cat**(1) to join them together. The following command line shows how to compile the **terminfo** source file for our fictitious terminal:

tic –v myterm.ti<CR>
(The trace information appears as the compilation
proceeds.)

Refer to the **tic**(1M) manual page in the *System Administrator's Reference Manual* for more information about the compiler.

Test the Description

Let's consider three ways to test a terminal description. First, you can test it by setting the environment variable **$TERMINFO** to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Second, you can test for correct insert line padding by commenting out **xon** in the description and then editing (using **vi**(1)) a large file (over 100 lines) at 9600 baud (if possible), and deleting about 15 lines from the middle of the screen. Type **u** (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **tput**(1) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and

produces no output. The general format for the **tput** command is as follows:

tput [**–T***type*] *capname*

The type of terminal you are requesting information about is identified with the **–T***type* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **$TERM**. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used.

tput clear
(The screen is cleared.)

The following command line shows how to output the number of columns for the terminal being used:

tput cols
(The number of columns used by the terminal appears here.)

The **tput**(1) manual page found in the *User's Reference Manual* contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **terminfo** source directory. The **infocmp**(1M) command was designed to help you with both of these tasks. For example:

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp –A /tmp/old –B /tmp/new –d 5420 5420
```


compares the old and new 5420 entries.

To print out the **terminfo** source for the 5420, type:

infocmp –I 5420

Converting a termcap Description to a terminfo Description

> **CAUTION**
>
> The **terminfo** database is designed to take the place of the **termcap** database. Because of the many programs and processes that have been written with and for the **termcap** database, it is not feasible to do a complete cutover at one time. Any conversion from **termcap** to **terminfo** requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The **captoinfo**(1M) command converts **termcap**(4) descriptions to **terminfo**(4) descriptions. When a file is passed to **captoinfo**, it looks for **termcap** descriptions and writes the equivalent **terminfo** descriptions on the standard output. For example, the command line:

captoinfo /etc/termcap

converts the file **/etc/termcap** to **terminfo** source, preserving comments and other extraneous information within the file. The command line:

captoinfo

looks up the current terminal in the **termcap** database, as specified by the **$TERM** and **$TERMCAP** environment variables and converts it to **terminfo**.

If you must have both **termcap** and **terminfo** terminal descriptions, keep the **terminfo** description only and use **infocmp –C** to get the **termcap** descriptions.

If you have been using cursor optimization programs with the **–ltermcap** or **–ltermlib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **–lcurses** option.

curses Program Examples

The following examples demonstrate uses of **curses** routines.

The editor Program

This program illustrates how to use **curses** routines to write a screen editor. For simplicity, **editor** keeps the buffer in **stdscr**; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the **move**(), **mvaddstr**(), **flash**(), **wnoutrefresh**() and **clrtoeol**() routines. These routines are all discussed in this chapter under "Working with **curses** Routines."

Second, it also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch**() routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the **insch**(), **delch**(), **insertln**(), and **deleteln**() routines. These functions insert and delete a character or line. See **curses**(3X) for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

NOTE

Because not all terminals have arrow keys, your **curses** programs will work on more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using **curses** routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line

noise affects the screen so much that the routines cannot keep track of it. A user invoking **editor** can type CTRL-L, causing the screen to be cleared and redrawn with a call to **wrefresh(curscr)**.

Finally, another important point is that the input command is terminated by CTRL-D, not the escape key. It is tempting to use escape as a command, since escape is one of the few special keys available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

editor and other **curses** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **curses** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your **curses** programs, avoid the escape key.


```
/* editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
 * the program.
 */

#include <stdio.h>
#include <curses.h>

#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
        extern void perror(), exit();
        int i, n, l;
```

```
        int c;
        int line = 0;
        FILE *fd;

        if (argc != 2)
        {
                fprintf(stderr, "Usage: %s file\n", argv[0]);
                exit(1);
        }

        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);
        /* Read in the file */
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\n')
                        line++;
                if (line > LINES - 2)
                        break;
                addch(c);
        }
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1], "w");
        for (l = 0; l < LINES - 1; l++)
        {
                n = len(l);
                for (i = 0; i < n; i++)
                        putc(mvinch(l, i) & A_CHARTEXT, fd);
                putc('\n', fd);
        }
        fclose(fd);

        endwin();
        exit(0);
}
```

```
len(lineno)
int lineno;
{
        int linelen = COLS - 1;

        while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;

        for (;;)
        {
                move(row, col);
                refresh();
                c = getch();

                /* Editor commands */
                switch (c)
                {

                /* hjkl and arrow keys: move cursor
                 * in direction indicated */
                case 'h':
                case KEY_LEFT:
                        if (col > 0)
                                col--;
                        else
                                flash();
                        break;

                case 'j':
                case KEY_DOWN:
                        if (row < LINES - 1)
                                row++;
                        else
                                flash();
                        break;

                case 'k':
                case KEY_UP:
                        if (row > 0)
                                row--;
                        else
                                flash();
                        break;
```

```
                case 'l':
                case KEY_RIGHT:
                        if (col < COLS - 1)
                                col++;
                        else
                                flash();
                        break;
                /* i: enter input mode */
                case KEY_IC:
                case 'i':
                        input();
                        break;

                /* x: delete current character */
                case KEY_DC:
                case 'x':
                        delch();
                        break;

                /* o: open up a new line and enter input mode */
                case KEY_IL:
                case 'o':
                        move(++row, col = 0);
                        insertln();
                        input();
                        break;

                /* d: delete current line */
                case KEY_DL:
                case 'd':
                        deleteln();
                        break;

                /* ^L: redraw screen */
                case KEY_CLEAR:
                case CTRL('L'):
                        wrefresh(curscr);
                        break;

                /* w: write and quit */
                case 'w':
                        return;
                /* q: quit without writing */
                case 'q':
                        endwin();
                        exit(2);
                default:
                        flash();
                        break;
                }
        }
}
```

```
/*
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
        int c;

        standout();
        mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
        standend();
        move(row, col);
        refresh();
        for (;;)
        {
                c = getch();
                if (c == CTRL('D') || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES - 1, COLS - 20);
        clrtoeol();
        move(row, col);
        refresh();
}
```

The highlight Program

This program illustrates a use of the routine **attrset**(). **highlight** reads a text file and uses embedded escape sequences to control attributes. **\U** turns on underlining, **\B** turns on bold, and **\N** restores the default output attributes.

Note the first call to **scrollok**(), a routine that we have not previously discussed (see **curses**(3X)). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **scrollok**() automatically scrolls the terminal up a line and calls **refresh**().

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
        FILE *fd;
        int c, c2;
        void exit(), perror();

        if (argc != 2)
        {
                fprintf(stderr, "Usage: highlight file\n");
                exit(1);
        }

        fd = fopen(argv[1], "r");

        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        scrollok(stdscr, TRUE);
        nonl();
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\\')
                {
                        c2 = getc(fd);
                        switch (c2)
                        {
                        case 'B':
                                attrset(A_BOLD);
                                continue;
                        case 'U':
                                attrset(A_UNDERLINE);
                                continue;
                        case 'N':
                                attrset(0);
                                continue;
                        }
```

```
                        addch(c);
                        addch(c2);
                }
                else
                        addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

The scatter Program

This program takes the first **LINES – 1** lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```
/*
 *      The scatter program.
 */

#include      <curses.h>
#include      <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS]; /* Screen Array */
int  T[MAXLINES][MAXCOLS]; /* Tag Array - Keeps track of    *
                            * the number of characters     *
                            * printed and their positions. */

main()
{
        register int row = 0,col = 0;
        register int c;
        int char_count = 0;
        time_t t;
        void exit(), srand();

        initscr();
        for(row = 0;row < MAXLINES;row++)
                for(col = 0;col < MAXCOLS;col++)
                        s[row][col]=' ';
```

```
        col = row = 0;
        /* Read screen in */
        while ((c=getchar()) != EOF && row < LINES ) {

                if(c != '\n')
                {
                        /* Place char in screen array */
                        s[row][col++] = c;
                        if(c != ' ')
                                char_count++;
                }
                else
                {
                        col = 0;
                        row++;
                }
        }

        time(&t);       /* Seed the random number generator */
        srand((unsigned)t);

        while (char_count)
        {
                row = rand() % LINES;
                col = (rand() >> 2) % COLS;
                if (T[row][col] != 1 && s[row][col] != ' ')
                {
                        move(row, col);
                        addch(s[row][col]);
                        T[row][col] = 1;
                        char_count--;
                        refresh();
                }
        }
        endwin();
        exit(0);
}
```

The show Program

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls **cbreak**() so that you can depress the space bar without having to hit return; it calls **noecho**() to prevent the space from echoing on the screen. The **nonl**() routine, which we have not previously discussed, is called to enable more cursor optimization. The **idlok**() routine, which we also have not discussed, is called to allow insert and delete line. (See **curses**(3X) for more information about these routines). Also notice that **clrtoeol**() and **clrtobot**() are called.

By creating an input file for **show** made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **curses**() program can be created. This type of input file is called a show script.

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

main(argc, argv)
int argc;
char *argv[];
{
        FILE *fd;
        char linebuf[BUFSIZ];
        int line;
        void done(), perror(), exit();

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ((fd=fopen(argv[1], "r")) == NULL)
        {

                perror(argv[1]);
                exit(2);
        }
        signal(SIGINT, done);

        initscr();
        noecho();
        cbreak();
        nonl();
        idlok(stdscr, TRUE);

        while(1)
        {
```

```
                move(0,0);
                for (line = 0; line < LINES; line++)
                {
                        if (!fgets(linebuf, sizeof linebuf, fd))
                        {
                                clrtobot();
                                done();
                        }
                        move(line, 0);
                        printw("%s", linebuf);
                }
                refresh();
                if (getch() == 'q')
                        done();
        }
}

void done()
{
        move(LINES - 1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "**sleep 100000**" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

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

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];
```

```
main(argc, argv)
int argc;
char **argv;
{
        void done(), exit();
        unsigned sleep();
        char *getenv();
        int c;

        if (argc != 4)
        {
             fprintf(stderr, "Usage: two othertty otherttytype inputfile\n");
             exit(1);
        }
        fd = fopen(argv[3], "r");
        fdyou = fopen(argv[1], "w+");
        signal(SIGINT, done);/* die gracefully */

        me = newterm(getenv("TERM"), stdout, stdin);  /* initialize my tty */
        you = newterm(argv[2], fdyou, fdyou);/* Initialize the other terminal */

        set_term(me); /* Set modes for my terminal */
        noecho();     /* turn off tty echo */
        cbreak();     /* enter cbreak mode */
        nonl();              /* Allow linefeed */
        nodelay(stdscr, TRUE);/* No hang on input */

        set_term(you);       /* Set modes for other terminal */
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screen full on my terminal */
        dump_page(me);

        /* Dump second screen full on the other terminal */
        dump_page(you);

        for (;;)      /* for each screen full */
        {
             set_term(me);
             c = getch();
             if (c == 'q')   /* wait for user to read it */
             done();
             if (c == ' ')
             dump_page(me);

             set_term(you);
             c = getch();
             if (c == 'q')   /* wait for user to read it */
             done();
             if (c == ' ')
             dump_page(you);
```

```
            sleep(1);
        }
}
dump_page(term)
  SCREEN *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line = 0; line < LINES - 1; line++) {
            if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrtobot();
            done();
            }
            mvaddstr(line, 0, linebuf);
        }
        standout();
        mvprintw(LINES - 1, 0, "--More--");
        standend();
        refresh();    /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
        /* Clean up first terminal */
        set_term(you);
        move(LINES - 1,0);   /* to lower left corner */

        clrtoeol();   /* clear bottom line */
        refresh();    /* flush out everything */
        endwin();     /* curses cleanup */

        /* Clean up second terminal */
        set_term(me);
        move(LINES - 1,0);   /* to lower left corner */
        clrtoeol();   /* clear bottom line */
        refresh();    /* flush out everything */
        endwin();     /* curses cleanup */
        exit(0);
}
```

The window Program

This example program demonstrates the use of multiple windows. The main display is kept in **stdscr**. When you want to put something other than what is in **stdscr** on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to **wrefresh**() for that window causes it to be written over the **stdscr** image on the terminal screen. Calling **refresh**() on **stdscr** results in the original window being redrawn on the screen. Note the calls to the **touchwin**() routine (which we have not discussed — see **curses**(3X)) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **curses** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call **touchwin**() for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()

{

        int i, c;
        char buf[120];
        void exit();

        initscr();
        nonl();
        noecho();
        cbreak();

        cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
        for (i = 0; i < LINES; i++)
                mvprintw(i, 0, "This is line %d of stdscr", i);

        for (;;)

        {
                refresh();
                c = getch();
                switch (c)

                {

                case 'c':       /* Enter command from keyboard */
                        werase(cmdwin);
                        wprintw(cmdwin, "Enter command:");
                        wmove(cmdwin, 2, 0);
                        for (i = 0; i < COLS; i++)
```

```
                        waddch(cmdwin, '-');
                    wmove(cmdwin, 1, 0);
                    touchwin(cmdwin);
                    wrefresh(cmdwin);
                    wgetstr(cmdwin, buf);
                    touchwin(stdscr);

                    /*
                     * The command is now in buf.
                     * It should be processed here.
                     */

            case 'q':
                    endwin();
                    exit(0);
            }

    }

}
```