



INTERIX Curses (Release 2.2)

Softway Systems Inc.

Writing Programs with Ncurses

This document originally written by
Eric S. Raymond
Zeyd M. Ben-Halim

It has been reformatted and modified slightly to indicate
which features are not part of the INTERIX distribution
of **ncurses** but is otherwise intact.

1.1 Introduction

This document is an introduction to programming with **curses**. It is not an exhaustive reference for the curses Application Programming Interface (API); that role is filled by the **curses** manual pages. Rather, it is intended to help C programmers ease into using the package.

This document is aimed at C applications programmers not yet specifically familiar with **ncurses**. If you are already an experienced **curses** programmer, you should nevertheless read the sections on Mouse Interfacing, Debugging, Compatibility with Older Versions, and Hints, Tips, and Tricks. These will bring you up to speed on the special features and quirks of the **ncurses** implementation. If you are not so experienced, keep reading.

The **curses** package is a subroutine library for terminal-independent screen-painting and input-event handling which presents a high level screen model to the programmer, hiding differences between terminal types and doing automatic optimization of output to change one screenful of text into another. **Curses** uses terminfo, which is a database format that can describe the capabilities of thousands of different terminals.

The **curses** API may seem something of an archaism on desktops increasingly dominated by X, Motif, and Tcl/Tk. Nevertheless, systems still support tty lines and X supports *xterm*; the **curses** API has the advantage of (a) back-portability to character-cell terminals, and (b) simplicity. For an application that does not require bit-mapped graphics and multiple fonts, an interface implementation using **curses** will typically be a great deal simpler and less expensive than one using an X toolkit.

1.1.1 A Brief History of Curses

The first ancestor of **curses** was the routines written to provide screen-handling for the game *rogue*; these used the already-existing **termcap** database facility for describing terminal capabilities. These routines were abstracted into a documented library and first released with the early BSD versions.

System III from Bell Labs featured a rewritten and much-improved **curses** library. It introduced the terminfo format. Terminfo is based on Berkeley's termcap database, but contains a number of improvements and extensions. Parameterized capabilities strings were introduced, making it possible to describe multiple video attributes, and colors and to handle far more unusual terminals than possible with termcap. In the later AT&T System V releases, **curses** evolved to use more facilities and offer more capabilities, going far beyond BSD curses in power and flexibility.

1.1.2 Scope of This Document

This document describes **ncurses**, a freeware implementation of the System V **curses** API with some clearly marked extensions. It includes the following System V **curses** features:

- Support for multiple screen highlights (BSD curses could only handle one ‘standout’ highlight, usually reverse-video).
- Support for line- and box-drawing using forms characters.
- Recognition of function keys on input.
- Color support.
- Support for pads (windows of larger than screen size on which the screen or a subwindow defines a viewport).

Also, this package makes use of the insert and delete line and character features of terminals so equipped, and determines how to optimally use these features with no help from the programmer. It allows arbitrary combinations of video attributes to be displayed, even on terminals that leave “magic cookies” on the screen to mark changes in attributes.

The **ncurses** package can also capture and use event reports from a mouse in some environments (notably, xterm under the X window system). This document includes tips for using the mouse.

The **ncurses** package was originated by Pavel Curtis. The primary maintainer of the package is Zeyd Ben-Halim <zmbenhal@netcom.com>. Eric S. Raymond <esr@snark.thyrsus.com> wrote many of the new features in versions after 1.8.1 and wrote most of this introduction.

This document also describes the extension library, similarly modeled on the SVr4 panels facility. This library allows you to associate backing store with each of a stack or deck of overlapping windows, and provides operations for moving windows around in the stack that change their visibility in the natural way (handling window overlaps).

Finally, this document describes in detail the menus and forms extension libraries, also cloned from System V, which support easy construction and sequences of menus and fill-in forms. This code was contributed to the project by Jurgen Pfeifer.

1.1.3 Terminology

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

window

A data structure describing a sub-rectangle of the screen (possibly the entire screen). You can write to a window as though it were a miniature screen, scrolling independently of other windows on the physical screen.

screens

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

terminal screen

The package’s idea of what the terminal display currently looks like, i.e., what the user sees now. This is a special screen.

1.2 An Overview of Curses

1.2.1 Compiling Programs Using Curses

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

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

at the top of the program source. The screen package uses the Standard I/O library, so **<curses.h>** includes **<stdio.h>**. **<curses.h>** also includes **<termios.h>**, **<termio.h>**, or **<sgtty.h>** depending on your system. It is redundant (but harmless) for the programmer to do these includes, too. In linking with **curses** you need to have **-lcurses** in your LDFLAGS or on the command line. There is no need for any other libraries.

1.2.2 Updating the Screen

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

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen; it's more like a scratchpad or write buffer.

To make the section of physical screen corresponding to a window reflect the contents of the window structure, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called.

A given physical screen section may be within the scope of any number of overlapping windows. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package implementation determine the most efficient way to repaint the screen.

1.2.3 Standard Windows and Function Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never actually access *curscr* directly. Changes should be made to through the API, and then the routine *refresh()* (or *wrefresh()*) called.

Many functions are defined to use *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character as argument. To write to a different window, use the routine *waddch()* (for 'w'indow-specific *addch()*) is provided. This convention of prepending function names with a 'w' when they are to be applied to specific windows is consistent. The only routines which do not follow it are those for which a window must always be specified.

In order to move the current (y, x) coordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then

perform some I/O operation. In order to avoid clumsiness, most I/O routines can be preceded by the prefix 'mv' and the desired (y, x) coordinates prepended to the arguments to the function. For example, the calls

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

can be replaced by

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

and

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

can be replaced by

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

Note that the window description pointer (win) comes before the added (y, x) coordinates. If a function requires a window pointer, it is always the first parameter passed.

1.2.4 Variables

The **curses** library sets some variables describing the terminal capabilities.

type	name	description
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal

The **<curses.h>** also introduces some **#define** constants and types of general usefulness:

bool boolean type, actually a **char** (e.g., `bool doneit;`)

TRUE boolean 'true' flag (1).

FALSE boolean 'false' flag (0).

ERR error flag returned by routines on a fail (-1).

OK error flag returned by routines when things go right.

1.3 Using the Curses Library

Now we describe how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. These instructions will work on any window, providing you change the function names and parameters as mentioned above.

Here is a sample program to motivate the discussion:

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

static void finish(int sig);

main(int argc, char *argv[])
{
    /* initialize your non-curses data structures here */

    (void) signal(SIGINT, finish);          /* arrange interrupts to terminate */

    (void) initscr();                      /* initialize the curses library */
    keypad(stdscr, TRUE);                  /* enable keyboard mapping */
    (void) nonl();                          /* tell curses not to do NL->CR/NL on output */
    (void) cbreak();                       /* take input chars one at a time, no wait for \n */
    (void) noecho();                       /* don't echo input */

    if (has_colors())
    {
        start_color();

        /*
         * Simple color assignment, often all we need.
         */
        init_pair(COLOR_BLACK, COLOR_BLACK, COLOR_BLACK);
        init_pair(COLOR_GREEN, COLOR_GREEN, COLOR_BLACK);
        init_pair(COLOR_RED, COLOR_RED, COLOR_BLACK);
        init_pair(COLOR_CYAN, COLOR_CYAN, COLOR_BLACK);
        init_pair(COLOR_WHITE, COLOR_WHITE, COLOR_BLACK);
        init_pair(COLOR_MAGENTA, COLOR_MAGENTA, COLOR_BLACK);
        init_pair(COLOR_BLUE, COLOR_BLUE, COLOR_BLACK);
        init_pair(COLOR_YELLOW, COLOR_YELLOW, COLOR_BLACK);
    }

    for (;;)
    {
        int c = getch();                  /* refresh, accept single keystroke of input */

        /* process the command keystroke */
    }

    finish(0);                            /* we're done */
}

static void finish(int sig)
```

```

{
endwin();

/* do your non-curses wrapup here */

exit(0);
}

```

1.3.1 Starting Up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These function *initscr()* does both these things. Since it must allocate space for the windows, it can overflow memory when attempting to do so. On the rare occasions this happens, *initscr()* will terminate the program with an error message. *initscr()* must always be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *cbreak()* should be called after *initscr()*.

Once the screen windows have been allocated, you can set them up for your program. If you want to, say, allow a screen to scroll, use *scrollok()*. If you want the cursor to be left in place after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) coordinates after updating it.

You can create new windows of your own using the functions *newwin()*, *derwin()*, and *subwin()*. The routine *delwin()* will allow you to get rid of old windows. All the options described above can be applied to any window.

1.3.2 Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*.

addch() adds a character at the current (y, x) coordinates. *move()* changes the current (y, x) coordinates to whatever you want them to be. It returns ERR if you try to move off the window. As mentioned above, you can combine the two into *mvaddch()* to do both things at once.

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

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

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

1.3.3 Input

The complementary function to *addch()* is *getch()* which, if *echo()* is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. Since initially the terminal has echoing enabled and is in ordinary “cooked” mode, one or the other has to be changed before calling *getch()*; otherwise, the program’s output will be unpredictable.

When you need to accept line-oriented input in a window, the functions *wgetstr()* and friends are available. There is even a *wscanw()* function that can do *scanf()*-style multi-field parsing on window input. These pseudo-line-oriented functions turn on echoing while they execute.

The example code above uses the call *keypad(stdscr, TRUE)* to enable support for function-key mapping. With this feature, the *getch()* code watches the input stream for character sequences that correspond to arrow and function keys. These sequences are returned as pseudo-character values. The **#define** values returned are listed in the **<ncurses.h>**. The mapping from sequences to **#define** values is determined by *key_* capabilities in the terminal’s terminfo entry.

1.3.4 Using Forms Characters

The *addch()* function (and some others, including *box()* and *border()*) can accept some pseudo-character arguments which are specially defined by **ncurses**. These are **#define** values set up in the **<ncurses.h>** header; see there for a complete list (look for the prefix *ACS_*).

The most useful of the ACS defines are the forms-drawing characters. You can use these to draw boxes and simple graphs on the screen. If the terminal does not have such characters, **<ncurses.h>** will map them to a recognizable (though ugly) set of ASCII defaults.

1.3.5 Character Attributes and Color

The **ncurses** package supports screen highlights including standout, reverse-video, underline, and blink. It also supports color, which is treated as another kind of highlight.

Highlights are encoded, internally, as high bits of the pseudo-character type (**chtype**) that **<ncurses.h>** uses to represent the contents of a screen cell. See the **<ncurses.h>** header file for a complete list of highlight mask values (look for the prefix *A_*).

There are two ways to make highlights. One is to logical-or the value of the highlights you want into the character argument of an *addch()* call, or any other output call that takes a **chtype** argument.

The other is to set the current-highlight value. This is logical-or’ed with any highlight you specify the first way. You do this with the functions *attron()*, *attroff()*, and *attrset()*; see the manual pages for details.

Color is a special kind of highlight. The package actually thinks in terms of color pairs, combinations of foreground and background colors. The sample code above sets up eight color pairs, all of the guaranteed-available colors on black. Note that each color pair is, in effect, given the name of its foreground color. Any other range of eight non-conflicting values could have been used as the first arguments of the *init_pair()* values.

Once you've done an `init_pair()` that creates color-pair `N`, you can use `COLOR_PAIR(N)` as a highlight that invokes that particular color combination. Note that `COLOR_PAIR(N)`, for constant `N`, is itself a compile-time constant and can be used in initializers.

1.3.6 Mouse Interfacing

The **ncurses** library also provides a mouse interface. Note: this facility is original to **ncurses**, it is not part of either the XSI Curses standard, nor of System V Release 4, nor BSD curses. Thus, we recommend that you wrap mouse-related code in an `#ifdef` using the feature macro `NCURSES_MOUSE_VERSION` so it will not be compiled and linked on non-ncurses systems.

Presently, mouse event reporting works only under xterm and does not work at all with INTERIX. In the future, ncurses will detect the presence of **gpm**(1), Alessandro Rubini's freeware mouse server for Linux systems, and accept mouse reports through it.

The mouse interface is very simple. To activate it, you use the function `mousemask()`, passing it as first argument a bit-mask that specifies what kinds of events you want your program to be able to see. It will return the bit-mask of events that actually become visible, which may differ from the argument if the mouse device is not capable of reporting some of the event types you specify.

Once the mouse is active, your application's command loop should watch for a return value of `KEY_MOUSE` from `wgetch()`. When you see this, a mouse event report has been queued. To pick it off the queue, use the function `getmouse()` (you must do this before the next `wgetch()`, otherwise another mouse event might come in and make the first one inaccessible).

Each call to `getmouse()` fills a structure (the address of which you'll pass it) with mouse event data. The event data includes zero-origin, screen-relative character-cell coordinates of the mouse pointer. It also includes an event mask. Bits in this mask will be set, corresponding to the event type being reported.

The mouse structure contains two additional fields which may be significant in the future as ncurses interfaces to new kinds of pointing device. In addition to `x` and `y` coordinates, there is a slot for a `z` coordinate; this might be useful with touchscreens that can return a pressure or duration parameter. There is also a device ID field, which could be used to distinguish between multiple pointing devices.

The class of visible events may be changed at any time via `mousemask()`. Events that can be reported include presses, releases, single-, double- and triple-clicks (you can set the maximum button-down time for clicks). If you don't make clicks visible, they will be reported as press-release pairs. In some environments, the event mask may include bits reporting the state of shift, alt, and ctrl keys on the keyboard during the event.

A function to check whether a mouse event fell within a given window is also supplied. You can use this to see whether a given window should consider a mouse event relevant to it.

Because mouse event reporting will not be available in all environments, it would be unwise to build **ncurses** applications that *require* the use of a mouse. Rather, you should use the mouse as a shortcut for point-and-shoot commands your application would normally accept from the keyboard. Two of the test games in the **ncurses** distribution (*bs* and *knight*) contain code that illustrates how this can be done.

See the manual page `curs_mouse` for full details of the mouse-interface functions.

1.3.7 Finishing Up

In order to clean up after the **ncurses** routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called, and moves the cursor down to the lower-left corner. Thus, anytime after the call to *initscr()*, *endwin()* should be called before exiting.

1.4 Function Descriptions

We describe the detailed behavior of some important curses functions here, as a supplement to the manual page descriptions.

1.4.1 Initialization and Wrapup

initscr()

The first function called should almost always be *initscr()*. This will determine the terminal type and initialize curses data structures. *initscr()* also arranges that the first call to *refresh()* will clear the screen. If an error occurs a message is written to standard error and the program exits. Otherwise it returns a pointer to *stdscr*. A few functions may be called before *initscr()* (*slk_init()*, *filter()*, *ripofflines()*, *use_env()*, and, if you are using multiple terminals, *newterm()*.)

endwin()

Your program should always call *endwin()* before exiting or shelling out of the program. This function will restore tty modes, move the cursor to the lower left corner of the screen, reset the terminal into the proper non-visual mode. Calling *refresh()* or *doupdate()* after a temporary escape from the program will restore the ncurses screen from before the escape.

newterm(type, ofp, ifp)

A program which outputs to more than one terminal should use *newterm()* instead of *initscr()*. *newterm()* should be called once for each terminal. It returns a variable of type **SCREEN *** which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string) and **FILE** pointers for the output and input of the terminal. If type is NULL then the environment variable *TERM* is used. *endwin()* should be called once at wrapup time for each terminal opened using this function.

set_term(new)

This function is used to switch to a different terminal previously opened by *newterm()*. The screen reference for the new terminal is passed as the parameter. The previous terminal is returned by the function. All other calls affect only the current terminal.

delscreen(sp)

The inverse of *newterm()*; deallocates the data structures associated with a given **SCREEN** reference.

1.4.2 Causing Output to the Terminal**refresh() and wrefresh(*win*)**

These functions must be called to actually get any output on the terminal, as other routines merely manipulate data structures. *wrefresh()* copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. *refresh()* does a refresh of *stdscr*. Unless *leaveok()* has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

doupdate() and wnoutrefresh(*win*)

These two functions allow multiple updates with more efficiency than *wrefresh*. To use them, it is important to understand how curses works. In addition to all the window structures, curses keeps two data structures representing the terminal screen: a physical screen, describing what is actually on the screen, and a virtual screen, describing what the programmer wants to have on the screen. *wrefresh* works by first copying the named window to the virtual screen (*wnoutrefresh()*), and then calling the routine to update the screen (*doupdate()*). If the programmer wishes to output several windows at once, a series of calls to *wrefresh()* will result in alternating calls to *wnoutrefresh()* and *doupdate()*, causing several bursts of output to the screen. By calling *wnoutrefresh()* for each window, it is then possible to call *doupdate()* once, resulting in only one burst of output, with fewer total characters transmitted (this also avoids a visually annoying flicker at each update).

1.4.3 Low-Level Capability Access**setupterm(*term*, *filenum*, *errret*)**

This routine is called to initialize a terminal's description, without setting up the curses screen structures or changing the tty-driver mode bits. *term* is the character string representing the name of the terminal being used. *filenum* is the file descriptor of the terminal to be used for output. *errret* is a pointer to an integer, in which a success or failure indication is returned. The values returned can be 1 (all is well), 0 (no such terminal), or -1 (some problem locating the terminfo database).

The value of *term* can be given as NULL, which will cause the value of *TERM* in the environment to be used. The *errret* pointer can also be given as NULL, meaning no error code is wanted. If *errret* is defaulted, and something goes wrong, *setupterm()* will print an appropriate error message and exit, rather than returning. Thus, a simple program can call *setupterm(0, 1, 0)* and not worry about initialization errors.

After the call to *setupterm()*, the global variable *cur_term* is set to point to the current structure of terminal capabilities. By calling *setupterm()* for each terminal, and saving and restoring *cur_term*, it is possible for a program to use two or more terminals at once. *Setupterm()* also stores the names section of the terminal description in the global character array *ttytype[]*. Subsequent calls to *setupterm()* will overwrite this array, so you'll have to save it yourself if need be.

1.4.4 Debugging

NOTE: These functions are not part of the standard curses SPI!

`trace()`

This function can be used to explicitly set a trace level. If the trace level is nonzero, execution of your program will generate a file called ‘trace’ in the current working directory containing a report on the library’s actions. Higher trace levels enable more detailed (and verbose) reporting — see comments attached to `TRACE_` defines in the `<ncurses.h>` file for details. (It is also possible to set a trace level by assigning a trace level value to the environment variable `NCURSES_TRACE`).

`_tracef()`

This function can be used to output your own debugging information. It is only available only if you link with `-lncurses_g` (not shipped with INTERIX). It can be used the same way as `printf()`, only it outputs a newline after the end of arguments. The output goes to a file called **trace** in the current directory.

Trace logs can be difficult to interpret due to the sheer volume of data dumped in them. There is a script called *tracemunch* included with the **ncurses** distribution that can alleviate this problem somewhat; it compacts long sequences of similar operations into more succinct single-line pseudo-operations. These pseudo-ops can be distinguished by the fact that they are named in capital letters.

1.5 Hints, Tips, and Tricks

The **ncurses** manual pages are a complete reference for this library. In the remainder of this document, we discuss various useful methods that may not be obvious from the manual page descriptions.

1.5.1 Some Notes of Caution

If you find yourself thinking you need to use `noraw()` or `nocbreak()`, think again and move carefully. It’s probably better design to use `getstr()` or one of its relatives to simulate cooked mode. The `noraw()` and `nocbreak()` functions try to restore cooked mode, but they may end up clobbering some control bits set before you started your application. Also, they have always been poorly documented, and are likely to hurt your application’s usability with other curses libraries.

Bear in mind that `refresh()` is a synonym for `wrefresh(stdscr)`, and don’t try to mix use of `stdscr` with use of windows declared by `newwin()`; a `refresh()` call will blow them off the screen. The right way to handle this is to use `subwin()`, or not touch `stdscr` at all and tile your screen with declared windows which you then `wnoutrefresh()` somewhere in your program event loop, with a single `doupdate()` call to trigger actual repainting.

You are much less likely to run into problems if you design your screen layouts to use tiled rather than overlapping windows. Historically, curses support for overlapping windows has been weak, fragile, and poorly documented. The **ncurses** library is not yet an exception to this rule.

There is a freeware panels library included in the **ncurses** distribution that does a pretty good job of strengthening the overlapping-windows facilities.

Try to avoid using the global variables `LINES` and `COLS`. Use `getmaxyx()` on the `stdscr` context instead. Reason: your code may be ported to run in an environment with window resizes, in which case several screens could be open with different sizes.

1.5.2 Temporarily Leaving Ncurses Mode

Sometimes you will want to write a program that spends most of its time in screen mode, but occasionally returns to ordinary ‘cooked’ (canonical) mode. A common reason for this is to support shell-out. This behavior is simple to arrange in **ncurses**.

To leave **ncurses** mode, call `endwin()` as you would if you were intending to terminate the program. This will take the screen back to cooked mode; you can do your shell-out. When you want to return to `ncurses()` mode, simply call `refresh()` or `doupdate()`. This will repaint the screen.

There is a boolean function, `isendwin()`, which code can use to test whether **ncurses** screen mode is active. It returns `TRUE` in the interval between an `endwin()` call and the following `refresh()`, `FALSE` otherwise.

Here is some sample code for shellout:

```
addstr("Shelling out...");
def_prog_mode();           /* save current tty modes */
endwin();                  /* restore original tty modes */
system("sh");              /* run shell */
addstr("returned.\n");     /* prepare return message */
refresh();                 /* restore save modes, repaint screen */
```

1.5.3 Handling Multiple Terminal Screens

The `initscr()` function actually calls a function named `newterm()` to do most of its work. If you are writing a program that opens multiple terminals, use `newterm()` directly.

For each call, you will have to specify a terminal type and a pair of file pointers; each call will return a screen reference, and `stdscr` will be set to the last one allocated. You will switch between screens with the `set_term()` call. Note that you will also have to call `def_shell_mode()` and `def_prog_mode()` on each tty yourself.

1.5.4 Testing for Terminal Capabilities

Sometimes you may want to write programs that test for the presence of various capabilities before deciding whether to go into **ncurses** mode. An easy way to do this is to call `setupterm()`, then use the functions `tigetflag()`, `tigetnum()`, and `tigetstr()` to do your testing.

A particularly useful case of this often comes up when you want to test whether a given terminal type should be treated as ‘smart’ (cursor-addressable) or ‘stupid’. The right way to test this is to see if the return value of `tigetstr("cup")` is non-NULL. Alternatively, you can include the `<term.h>` file and test the value of the macro `cursor_address`.

1.5.5 Tuning for Speed

Use the *addchstr()* family of functions for fast screen-painting of text when you know the text doesn't contain any control characters. Try to make attribute changes infrequent on your screens. Don't use the *immedok()* option!

1.5.6 Special Features of Ncurses

When running on PC-clones, **ncurses** has enhanced support for the IBM high-half and ROM characters. The *A_ALTCHARSET* highlight, enables display of both high-half ACS graphics and the PC ROM graphics 0-31 that are normally interpreted as control characters.

The *wresize()* function allows you to resize a window in place.

1.6 Compatibility with Older Versions

Despite our best efforts, there are some differences between **ncurses** and the (undocumented!) behavior of older curses implementations. These arise from ambiguities or omissions in the documentation of the API.

1.6.1 Refresh of Overlapping Windows

If you define two windows A and B that overlap, and then alternately scribble on and refresh them, the changes made to the overlapping region under historic **curses** versions were often not documented precisely.

To understand why this is a problem, remember that screen updates are calculated between two representations of the *entire* display. The documentation says that when you refresh a window, it is first copied to the virtual screen, and then changes are calculated to update the physical screen (and applied to the terminal). But "copied to" is not very specific, and subtle differences in how copying works can produce different behaviors in the case where two overlapping windows are each being refreshed at unpredictable intervals.

What happens to the overlapping region depends on what *wnoutrefresh()* does with its argument — what portions of the argument window it copies to the virtual screen. Some implementations do "change copy", copying down only locations in the window that have changed (or been marked changed with *wtouchln()* and friends). Some implementations do "entire copy", copying *all* window locations to the virtual screen whether or not they have changed.

The **ncurses** library itself has not always been consistent on this score. Due to a bug, versions 1.8.7 to 1.9.8a did entire copy. Versions 1.8.6 and older, and versions 1.9.9 and newer, do change copy.

For most commercial curses implementations, it is not documented and not known for sure (at least not to the **ncurses** maintainers) whether they do change copy or entire copy. We know that System V release 3 curses has logic in it that looks like an attempt to do change copy, but the surrounding logic and data representations are sufficiently complex, and our knowledge sufficiently indirect, that it's hard to know whether this is reliable.

It is not clear what the SVr4 documentation and XSI standard intend. The XSI Curses standard barely mentions *wnoutrefresh()*; the SVr4 documents seem to be describing

entire-copy, but it is possible with some effort and straining to read them the other way.

It might therefore be unwise to rely on either behavior in programs that might have to be linked with other curses implementations. Instead, you can do an explicit *touchwin()* before the *wnoutrefresh()* call to guarantee an entire-contents copy anywhere.

The really clean way to handle this is to use the panels library. If, when you want a screen update, you do *update_panels()*, it will do all the necessary *wnoutrefresh()* calls for whatever panel stacking order you have defined. Then you can do one *doupdate()* and there will be a *single* burst of physical I/O that will do all your updates.

1.6.2 Background Erase

If you have been using a very old versions of **ncurses** (1.8.7 or older) you may be surprised by the behavior of the erase functions. In older versions, erased areas of a window were filled with a blank modified by the window's current attribute (as set by *wattrset()*, *wattron()*, *wattroff()* and friends).

In newer versions, this is not so. Instead, the attribute of erased blanks is normal unless and until it is modified by the functions *bkgdset()* or *wbkgdset()*.

This change in behavior conforms **ncurses** to System V Release 4 and the XSI Curses standard.

1.6.3 XSI Curses Conformance

The **ncurses** library is intended to be base-level conformant with the XSI Curses standard from X/Open. May extended-level features (in fact, almost all features not directly concerned with wide characters and internationalization) are also supported.

One effect of XSI conformance is the change in behavior described under "Background Erase — Compatibility with Old Versions".

Also, **ncurses** meets the XSI requirement that every macro entry point have a corresponding function which may be linked (and will be prototype-checked) if the macro definition is disabled with **#undef**.

The Panels Library

The **ncurses** library by itself provides good support for screen displays in which the windows are tiled (non-overlapping). In the more general case that windows may overlap, you have to use a series of *wnoutrefresh()* calls followed by a *doupdate()*, and be careful about the order you do the window refreshes in. It has to be bottom-upwards, otherwise parts of windows that should be obscured will show through.

When your interface design is such that windows may dive deeper into the visibility stack or pop to the top at runtime, the resulting book-keeping can be tedious and difficult to get right. Hence the panels library.

The **panel** library first appeared in AT&T System V. The version documented here is the freeware **panel** code distributed with **ncurses**.

2.1 Compiling with the Panels Library

Your panels-using modules must import the panels library declarations with

```
#include <panel.h>
```

and must be linked explicitly with the panels library using an **-lpanel** argument. Note that they must also link the **ncurses** library with **-lncurses**. Most modern linkers are two-pass and will accept either order, but it is still good practice to put **-lpanel** first and **-lncurses** second.

2.2 Overview of Panels

A panel object is a window that is implicitly treated as part of a **deck** including all other panel objects. The deck has an implicit bottom-to-top visibility order. The panels library includes an update function (analogous to *refresh()*) that displays all panels in the deck in the proper order to resolve overlaps. The standard window, *stdscr*, is considered below all panels.

Details on the panels functions are available in the man pages. We'll just hit the highlights here.

You create a panel from a window by calling *new_panel()* on a window pointer. It then becomes the top of the deck. The panel's window is available as the value of *panel_window()* called with the panel pointer as argument.

You can delete a panel (removing it from the deck) with *del_panel()*. This will not deallocate the associated window; you have to do that yourself.

You can replace a panel's window with a different window by calling *replace_window()*. The new window may be of different size; the panel code will re-compute all overlaps. This operation doesn't change the panel's position in the deck.

To move a panel's window, use *move_panel()*. The *mvwin()* function on the panel's window isn't sufficient because it doesn't update the panels library's representation of where the windows are. This operation leaves the panel's depth, contents, and size

unchanged.

Two functions (*top_panel()*, *bottom_panel()*) are provided for rearranging the deck. The first pops its argument window to the top of the deck; the second sends it to the bottom. Either operation leaves the panel's screen location, contents, and size unchanged.

The function *update_panels()* does all the *wnoutrefresh()* calls needed to prepare for *doupdate()* (which you must call yourself, afterwards).

Typically, you will want to call *update_panels()* and *doupdate()* just before accepting command input, once in each cycle of interaction with the user. If you call *update_panels()* after each and every panel write, you'll generate a lot of unnecessary refresh activity and screen flicker.

2.3 Panels, Input, and the Standard Screen

You shouldn't mix *wnoutrefresh()* or *wrefresh()* operations with panels code; this will work only if the argument window is either in the top panel or un-obscured by any other panels.

The *stdscr* window is a special case. It is considered below all panels. Because changes to panels may obscure parts of *stdscr*, though, you should call *update_panels()* before *doupdate()* even when you only change *stdscr*.

Note that *wgetch()* automatically calls *wrefresh()*. Therefore, before requesting input from a panel window, you need to be sure that the panel is totally un-obscured.

There is presently no way to display changes to one obscured panel without repainting all panels.

2.4 Hiding Panels

It's possible to remove a panel from the deck temporarily; use *hide_panel()* for this. You can un-hide a panel with *show_panel()*. The predicate function *panel_hidden()* tests whether or not a panel is hidden.

The *panel_update()* code ignores hidden panels. You cannot do *top_panel()* or *bottom_panel()* on a hidden *panel()*. Other panels operations are applicable.

2.5 Miscellaneous Other Facilities

It's possible to navigate the deck using the functions *panel_above()* and *panel_below()*. Handed a panel pointer, they return the panel above or below that panel. Handed NULL, they return the bottom-most or top-most panel.

Every panel has an associated user pointer, not used by the panel code, to which you can attach application data. See the man page documentation of *set_panel_userptr()* and *panel_userptr()* for details.

The Menu Library

A menu is a screen display that assists the user to choose some subset of a given set of items. The **menu** library is a curses extension that supports easy programming of menu hierarchies with a uniform but flexible interface.

The **menu** library first appeared in AT&T System V. The version documented here is the freeware **menu** code distributed with **ncurses**.

3.1 Compiling with the Menu Library

Your menu-using modules must import the menu library declarations with

```
#include <menu.h>
```

and must be linked explicitly with the menus library using an **-lmenu** argument. Note that they must also link the **ncurses** library with **-lncurses**. Most modern linkers are two-pass and will accept either order, but it is still good practice to put **-lmenu** first and **-lncurses** second.

3.2 Overview of Menus

The menus created by this library consist of collections of **items** including a name string part and a description string part. To make menus, you create groups of these items and connect them with menu frame objects.

The menu can then be **posted**, that is written to an associated window. Actually, each menu has two associated windows; a containing window in which the programmer can scribble titles or borders, and a subwindow in which the menu items proper are displayed. If this subwindow is too small to display all the items, it will be a scrollable viewport on the collection of items.

A menu may also be **unposted** (that is, undisplayed), and finally freed to make the storage associated with it and its items available for re-use.

The general flow of control of a menu program looks like this:

1. Initialize **curses**.
2. Create the menu items, using *new_item()*.
3. Create the menu using *new_menu()*.
4. Post the menu using *menu_post()*.
5. Refresh the screen.
6. Process user requests via an input loop.
7. Unpost the menu using *menu_unpost()*.
8. Free the menu, using *free_menu()*.

9. Free the items using *free_item()*.
10. Terminate **curses**.

3.3 Selecting Items

Menus may be multi-valued or (the default) single-valued (see the manual page *mitem_opts* to see how to change the default). Both types always have a **current item**.

From a single-valued menu you can read the selected value simply by looking at the current item. From a multi-valued menu, you get the selected set by looping through the items applying the *item_value()*

predicate function. Your menu-processing code can use the function *set_item_value()* to flag the items in the select set.

Menu items can be made un-selectable using *set_item_opts()* or *item_opts_off()* with the *O_SELECTABLE* argument. This is the only option so far defined for menus, but it is good practice to code as though other option bits might be on.

3.4 Menu Display

The menu library calculates a minimum display size for your window, based on the following variables:

- The number and maximum length of the menu items
- Whether the *O_ROWMAJOR* option is enabled
- Whether display of descriptions is enabled
- Whatever menu format may have been set by the programmer
- The length of the menu mark string used for highlighting selected items

The function *set_menu_format()* allows you to set the maximum size of the viewport or **menu page** that will be used to display menu items. You can retrieve any format associated with a menu with *menu_format()*. The default format is rows=16, columns=1.

The actual menu page may be smaller than the format size. This depends on the item number and size and whether *O_ROWMAJOR* is on. This option (on by default) causes menu items to be displayed in a ‘raster-scan’ pattern, so that if more than one item will fit horizontally the first couple of items are side-by-side in the top row. The alternative is column-major display, which tries to put the first several items in the first column.

As mentioned above, a menu format not large enough to allow all items to fit on-screen will result in a menu display that is vertically scrollable.

You can scroll it with requests to the menu driver, which will be described in the section on menu input handling.

Each menu has a **mark string** used to visually tag selected items; see the *menu_mark* manual page for details. The mark string length also influences the menu page size.

The function *scale_menu()* returns the minimum display size that the menu code computes from all these factors.

There are other menu display attributes including a select attribute, an attribute for selectable items, an attribute for unselectable items, and a pad character used to separate item name text from description text. These have reasonable defaults which the library allows you to change (see the *menu_attrbs* manual page).

3.5 Menu Windows

Each menu has, as mentioned previously, a pair of associated windows. Both these windows are painted when the menu is posted and erased when the menu is unposted.

The outer or frame window is not otherwise touched by the menu routines. It exists so the programmer can associate a title, a border, or perhaps help text with the menu and have it properly refreshed or erased at post/unpost time. The inner window or **subwindow** is where the current menu page is displayed.

By default, both windows are *stdscr*. You can set them with the functions in *menu_win*.

When you call *menu_post()*, you write the menu to its subwindow. When you call *menu_unpost()*, you erase the subwindow. However, neither of these actually modifies the screen. To do that, call *wrefresh()* or some equivalent.

3.6 Processing Menu Input

The main loop of your menu-processing code should call *menu_driver()* repeatedly. The first argument of this routine is a menu pointer; the second is a menu command code. You should write an input-fetching routine that maps input characters to menu command codes, and pass its output to *menu_driver()*. The menu command codes are fully documented in *menu_driver*.

The simplest group of command codes is REQ_NEXT_ITEM, REQ_PREV_ITEM, REQ_FIRST_ITEM, REQ_LAST_ITEM, REQ_UP_ITEM, REQ_DOWN_ITEM, REQ_LEFT_ITEM, REQ_RIGHT_ITEM. These change the currently selected item. These requests may cause scrolling of the menu page if it only partially displayed.

There are explicit requests for scrolling which also change the current item (because the select location does not change, but the item there does). These are REQ_SCR_DLINE, REQ_SCR_ULINE, REQ_SCR_DPAGE, and REQ_SCR_UPAGE.

The REQ_TOGGLE_ITEM selects or deselects the current item. It is for use in multi-valued menus; if you use it with O_ONEVALUE on, you'll get an error return (E_REQUEST_DENIED).

Each menu has an associated pattern buffer. The *menu_driver()* logic tries to accumulate printable ASCII characters passed in in that buffer; when it matches a prefix of an item name, that item (or the next matching item) is selected. If appending a character yields no new match, that character is deleted from the pattern buffer, and *menu_driver()* returns E_NO_MATCH.

Some requests change the pattern buffer directly: REQ_CLEAR_PATTERN, REQ_BACK_PATTERNP, REQ_NEXT_MATCH, REQ_PREV_MATCH. The latter two are useful when pattern buffer input matches more than one item in a multi-valued menu.

Each successful scroll or item navigation request clears the pattern buffer. It is also possible to set the pattern buffer explicitly with *set_menu_pattern()*.

Finally, menu driver requests above the constant `MAX_COMMAND` are considered application-specific commands. The *menu_driver()* code ignores them and returns `E_UNKNOWN_COMMAND`.

3.7 Miscellaneous Other Features

Various menu options can affect the processing and visual appearance and input processing of menus. See *menu_opts* for details.

It is possible to change the current item from application code; this is useful if you want to write your own navigation requests. It is also possible to explicitly set the top row of the menu display. See *mitem_current*.

If your application needs to change the menu subwindow cursor for any reason, *pos_menu_cursor()* will restore it to the correct location for continuing menu driver processing.

It is possible to set hooks to be called at menu initialization and wrapup time, and whenever the selected item changes. See *menu_hook*.

Each item, and each menu, has an associated user pointer on which you can hang application data. See *mitem_userptr* and *menu_userptr*.

The Form Library

The **form** library is a curses extension that supports easy programming of on-screen forms for data entry and program control.

The **form** library first appeared in AT&T System V. The version documented here is the freeware **form** code distributed with **ncurses**.

4.1 Compiling with the Form Library

Your form-using modules must import the form library declarations with

```
#include <form.h>
```

and must be linked explicitly with the forms library using an **-lform** argument. Note that they must also link the **ncurses** library with **-lncurses**. Most modern linkers are two-pass and will accept either order, but it is still good practice to put **-lform** first and **-lncurses** second.

4.2 Overview of Forms

A form is a collection of fields; each field may be either a label (explanatory text) or a data-entry location. Long forms may be segmented into pages; each entry to a new page clears the screen.

To make forms, you create groups of fields and connect them with form frame objects; the form library makes this relatively simple.

Once defined, a form can be **posted**, that is written to an associated window. Actually, each form has two associated windows; a containing window in which the programmer can scribble titles or borders, and a subwindow in which the form fields proper are displayed.

As the form user fills out the posted form, navigation and editing keys support movement between fields, editing keys support modifying field, and plain text adds to or changes data in a current field. The form library allows you (the forms designer) to bind each navigation and editing key to any keystroke accepted by **curses**

Fields may have validation conditions on them, so that they check input data for type and value. The form library supplies a rich set of pre-defined field types, and makes it relatively easy to define new ones.

Once its transaction is completed (or aborted), a form may be **unposted** (that is, undisplayed), and finally freed to make the storage associated with it and its items available for re-use.

The general flow of control of a form program looks like this:

1. Initialize **curses**.
2. Create the form fields, using `new_field()`.

3. Create the form using *new_form()*.
4. Post the form using *form_post()*.
5. Refresh the screen.
6. Process user requests via an input loop.
7. Unpost the form using *form_unpost()*.
8. Free the form, using *free_form()*.
9. Free the fields using *free_field()*.
10. Terminate **curses**.

Note that this looks much like a menu program; the form library handles tasks which are in many ways similar, and its interface was obviously designed to resemble that of the menu library wherever possible.

In forms programs, however, the ‘process user requests’ is somewhat more complicated than for menus. Besides menu-like navigation operations, the menu driver loop has to support field editing and data validation.

4.3 Creating and Freeing Fields and Forms

The basic function for creating fields is *new_field()*:

```
FIELD *new_field(int height, int width,      /* new field size */
                 int top, int left,         /* upper left corner */
                 int offscreen,             /* number of offscreen rows */
                 int nbuf);                 /* number of working buffers */
```

Menu items always occupy a single row, but forms fields may have multiple rows. So *new_field()* requires you to specify a width and height (the first two arguments, which must both be greater than zero).

You must also specify the location of the field’s upper left corner on the screen (the third and fourth arguments, which must be zero or greater). Note that these coordinates are relative to the form subwindow, which will coincide with *stdscr* by default but need not be *stdscr* if you’ve done an explicit *set_form_window()* call.

The fifth argument allows you to specify a number of off-screen rows. If this is zero, the entire field will always be displayed. If it is nonzero, the form will be scrollable, with only one screen-full (initially the top part) displayed at any given time. If you make a field dynamic and grow it so it will no longer fit on the screen, the form will become scrollable even if the *offscreen* argument was initially zero.

The forms library allocates one working buffer per field; the size of each buffer is $((\text{height} + \text{offscreen}) * \text{width} + 1)$, one character for each position in the field plus a NUL terminator. The sixth argument is the number of additional data buffers to allocate for the field; your application can use them for its own purposes.

```
FIELD *dup_field(FIELD *field,              /* field to copy */
                 int top, int left);         /* location of new copy */
```

The function *dup_field()* duplicates an existing field at a new location. Size and buffering information are copied; some attribute flags and status bits are not (see the Fn

`form_field_new` for details).

```
FIELD *link_field(FIELD *field,          /* field to copy */
                  int top, int left);    /* location of new copy */
```

The function `link_field()` also duplicates an existing field at a new location. The difference from `dup_field()` is that it arranges for the new field's buffer to be shared with the old one.

Besides the obvious use in making a field editable from two different form pages, linked fields give you a way to hack in dynamic labels. If you declare several fields linked to an original, and then make them inactive, changes from the original will still be propagated to the linked fields.

As with duplicated fields, linked fields have attribute bits separate from the original.

As you might guess, all these field-allocations return NULL if the field allocation is not possible due to an out-of-memory error or out-of-bounds arguments.

To connect fields to a form, use

```
FORM *new_form(FIELD **fields);
```

This function expects to see a NULL-terminated array of field pointers. Said fields are connected to a newly-allocated form object; its address is returned (or else NULL if the allocation fails).

Note that `new_field()` does *not* copy the pointer array into private storage; if you modify the contents of the pointer array during forms processing, all manner of bizarre things might happen. Also note that any given field may only be connected to one form.

The functions `free_field()` and `free_form()` are available to free field and form objects. It is an error to attempt to free a field connected to a form, but not vice-versa; thus, you will generally free your form objects first.

4.4 Fetching and Changing Field Attributes

Each form field has a number of location and size attributes associated with it. There are other field attributes used to control display and editing of the field. Some (for example, the `O_STATIC` bit) involve sufficient complications to be covered in sections of their own later on. We cover the functions used to get and set several basic attributes here.

When a field is created, the attributes not specified by the `new_field()` function are copied from an invisible system default field. In attribute-setting and -fetching functions, the argument NULL is taken to mean this field. Changes to it persist as defaults until your forms application terminates.

4.4.1 Fetching Size and Location Data

You can retrieve field sizes and locations through:

```
int field_info(FIELD *field,          /* field from which to fetch */
               int *height, int *width, /* field size */
               int *top, int *left,    /* upper left corner */
               int *offscreen,         /* number of offscreen rows */
               int *nbuf);             /* number of working buffers */
```

This function is a sort of inverse of *new_field()*; instead of setting size and location attributes of a new field, it fetches them from an existing one.

4.4.2 Changing the Field Location

If is possible to move a field's location on the screen:

```
int move_field(FIELD *field,          /* field to alter */
               int top, int left);     /* new upper-left corner */
```

You can, of course, query the current location through *field_info()*.

4.4.3 Setting the Justification Attribute

One-line fields may be unjustified, justified right, justified left, or centered. Here is how you manipulate this attribute:

```
int set_field_just(FIELD *field,      /* field to alter */
                   int justmode);     /* mode to set */

int field_just(FIELD *field);          /* fetch mode of field */
```

The mode values accepted and returned by this functions are preprocessor macros NO_JUSTIFICATION, JUSTIFY_RIGHT, JUSTIFY_LEFT, or JUSTIFY_CENTER.

4.4.4 Field Display Attributes

For each field, you can set a foreground attribute for entered characters, a background attribute for the entire field, and a pad character for the unfilled portion of the field. You can also control pagination of the form.

This group of four field attributes controls the visual appearance of the field on the screen, without affecting in any way the data in the field buffer.

```
int set_field_fore(FIELD *field,      /* field to alter */
                   chtype attr);      /* attribute to set */

chtype field_fore(FIELD *field);      /* field to query */

int set_field_back(FIELD *field,      /* field to alter */
                   chtype attr);      /* attribute to set */

chtype field_back(FIELD *field);      /* field to query */

int set_field_pad(FIELD *field,       /* field to alter */
                  int pad);           /* pad character to set */
```

```

ctype field_pad(FIELD *field);

int set_new_page(FIELD *field,      /* field to alter */
                 int flag);        /* TRUE to force new page */

ctype new_page(FIELD *field);      /* field to query */

```

The attributes set and returned by the first four functions are normal **curses** display attribute values (A_STANDOUT, A_BOLD, A_REVERSE etc).

The page bit of a field controls whether it is displayed at the start of a new form screen.

4.4.5 Field Option Bits

There is also a large collection of field option bits you can set to control various aspects of forms processing. You can manipulate them with these functions:

```

int set_field_opts(FIELD *field,      /* field to alter */
                  int *field,);      /* attribute to set */

int field_opts_on(FIELD *field,      /* field to alter */
                 int *field,);       /* attributes to turn on */

int field_opts_off(FIELD *field,     /* field to alter */
                  int *field,);      /* attributes to turn off */

int field_opts(FIELD *field);        /* field to query */

```

By default, all options are on. Here are the available option bits:

O_VISIBLE

Controls whether the field is visible on the screen. Can be used during form processing to hide or pop up fields depending on the value of parent fields.

O_ACTIVE

Controls whether the field is active during forms processing (i.e. visited by form navigation keys). Can be used to make labels or derived fields with buffer values alterable by the forms application, not the user.

O_PUBLIC

Controls whether data is displayed during field entry. If this option is turned off on a field, the library will accept and edit data in that field, but it will not be displayed and the visible field cursor will not move. You can turn off the O_PUBLIC bit to define password fields.

O_EDIT

Controls whether the field's data can be modified. When this option is off, all editing requests except REQ_PREV_CHOICE and REQ_NEXT_CHOICE will fail. Such read-only fields may be useful for help messages.

O_WRAP

Controls word-wrapping in multi-line fields. Normally, when any character of a (blank-separated) word reaches the end of the current line, the entire word is wrapped to the next line (assuming there is one). When this option is off, the word will be split across the line break.

O_BLANK

Controls field blanking. When this option is on, entering a character at the first field position erases the entire field (except for the just-entered character).

O_AUTOSKIP

Controls automatic skip to next field when this one fills. Normally, when the forms user tries to type more data into a field than will fit, the editing location jumps to next field. When this option is off, the user's cursor will hang at the end of the field. This option is ignored in dynamic fields that have not reached their size limit.

O_NULLOK

Controls whether validation is applied to blank fields. Normally, it is not; the user can leave a field blank without invoking the usual validation check on exit. If this option is off on a field, exit from it will invoke a validation check.

O_PASSOK

Controls whether validation occurs on every exit, or only after the field is modified. Normally the latter is true. Setting O_PASSOK may be useful if your field's validation function may change during forms processing.

O_STATIC

Controls whether the field is fixed to its initial dimensions. If you turn this off, the field becomes dynamic and will stretch to fit entered data.

A field's options cannot be changed while the field is currently selected. However, options may be changed on posted fields that are not current.

The option values are bit-masks and can be composed with logical-or in the obvious way.

4.5 Field Status

Every field has a status flag, which is set to FALSE when the field is created and TRUE when the value in field buffer 0 changes. This flag can be queried and set directly:

```
int set_field_status(FIELD *field,      /* field to alter */
                    int status);        /* mode to set */

int field_status(FIELD *field);         /* fetch mode of field */
```

Setting this flag under program control can be useful if you use the same form repeatedly, looking for modified fields each time.

Calling *field_status()* on a field not currently selected for input will return a correct value. Calling *field_status()* on a field that is currently selected for input may not necessarily give a correct field status value, because entered data isn't necessarily copied to buffer zero before the exit validation check.

To guarantee that the returned status value reflects reality, call *field_status()* either (1) in the field's exit validation check routine, (2) from the field's or form's initialization or termination hooks, or (3) just after a REQ_VALIDATION request has been processed by the forms driver.

4.6 Field User Pointer

Each field structure contains one character pointer slot that is not used by the forms library. It is intended to be used by applications to store private per-field data. You can manipulate it with:

```
int set_field_userptr(FIELD *field,      /* field to alter */
                     char *userptr);    /* mode to set */

char *field_userptr(FIELD *field);      /* fetch mode of field */
```

(Properly, this user pointer field ought to have **(void *)** type. The **(char *)** type is retained for System V compatibility.)

It is valid to set the user pointer of the default field (with a *set_field_userptr()* call passed a NULL field pointer.) When a new field is created, the default-field user pointer is copied to initialize the new field's user pointer.

4.7 Variable-Sized Fields

Normally, a field is fixed at the size specified for it at creation time. If, however, you turn off its `O_STATIC` bit, it becomes **dynamic** and will automatically resize itself to accommodate data as it is entered. If the field has extra buffers associated with it, they will grow right along with the main input buffer.

A one-line dynamic field will have a fixed height (1) but variable width, scrolling horizontally to display data within the field area as originally dimensioned and located. A multi-line dynamic field will have a fixed width, but variable height (number of rows), scrolling vertically to display data within the field area as originally dimensioned and located.

Normally, a dynamic field is allowed to grow without limit. But it is possible to set an upper limit on the size of a dynamic field. You do it with this function:

```
int set_max_field(FIELD *field,      /* field to alter (may not be NULL) */
                  int max_size);    /* upper limit on field size */
```

If the field is one-line, *max_size* is taken to be a column size limit; if it is multi-line, it is taken to be a line size limit. To disable any limit, use an argument of zero. The growth limit can be changed whether or not the `O_STATIC` bit is on, but has no effect until it is.

The following properties of a field change when it becomes dynamic:

- If there is no growth limit, there is no final position of the field; therefore `O_AUTOSKIP` and `O_NL_OVERLOAD` are ignored.
- Field justification will be ignored (though whatever justification is set up will be retained internally and can be queried).
- The *dup_field()* and *link_field()* calls copy dynamic-buffer sizes. If the `O_STATIC` option is set on one of a collection of links, buffer resizing will occur only when the field is edited through that link.
- The call *field_info()* will retrieve the original static size of the field; use *dynamic_field_info()* to get the actual dynamic size.

4.8 Field Validation

By default, a field will accept any data that will fit in its input buffer. However, it is possible to attach a validation type to a field. If you do this, any attempt to leave the field while it contains data that doesn't match the validation type will fail. Some validation types also have a character-validity check for each time a character is entered in the field.

A field's validation check (if any) is not called when `set_field_buffer()` modifies the input buffer, nor when that buffer is changed through a linked field.

The **form** library provides a rich set of pre-defined validation types, and gives you the capability to define custom ones of your own. You can examine and change field validation attributes with the following functions:

```
int set_field_type(FIELD *field,          /* field to alter */
                  FIELDTYPE *ftype,      /* type to associate */
                  ...);                  /* additional arguments */

FIELDTYPE *field_type(FIELD *field);    /* field to query */
```

The validation type of a field is considered an attribute of the field. As with other field attributes, Also, doing `set_field_type()` with a NULL field default will change the system default for validation of newly-created fields.

Here are the pre-defined validation types:

4.8.1 TYPE_ALPHA

This field type accepts alphabetic data; no blanks, no digits, no special characters (this is checked at character-entry time). It is set up with:

```
int set_field_type(FIELD *field,          /* field to alter */
                  TYPE_ALPHA,             /* type to associate */
                  int width);             /* maximum width of field */
```

The *width* argument sets a minimum width of data. Typically you'll want to set this to the field width; if it's greater than the field width, the validation check will always fail. A minimum width of zero makes field completion optional.

4.8.2 TYPE_ALNUM

This field type accepts alphabetic data and digits; no blanks, no special characters (this is checked at character-entry time). It is set up with:

```
int set_field_type(FIELD *field,          /* field to alter */
                  TYPE_ALNUM,             /* type to associate */
                  int width);             /* maximum width of field */
```

The *width* argument sets a minimum width of data. As with `TYPE_ALPHA`, typically you'll want to set this to the field width; if it's greater than the field width, the validation check will always fail. A minimum width of zero makes field completion optional.

4.8.3 TYPE_ENUM

This type allows you to restrict a field's values to be among a specified set of string values (for example, the two-letter postal codes for U.S. states). It is set up with:

```
int set_field_type(FIELD *field,          /* field to alter */
                  TYPE_ENUM,             /* type to associate */
                  char **valuelist,       /* list of possible values */
                  int checkcase,          /* case-sensitive? */
                  int checkunique);       /* must specify uniquely? */
```

The *valuelist* parameter must point at a NULL-terminated list of valid strings. The *checkcase* argument, if true, makes comparison with the string case-sensitive.

When the user exits a TYPE_ENUM field, the validation procedure tries to complete the data in the buffer to a valid entry. If a complete choice string has been entered, it is of course valid. But it is also possible to enter a prefix of a valid string and have it completed for you.

By default, if you enter such a prefix and it matches more than one value in the string list, the prefix will be completed to the first matching value. But the *checkunique* argument, if true, requires prefix matches to be unique in order to be valid.

The REQ_NEXT_CHOICE and REQ_PREV_CHOICE input requests can be particularly useful with these fields.

4.8.4 TYPE_INTEGER

This field type accepts an integer. It is set up as follows:

```
int set_field_type(FIELD *field,          /* field to alter */
                  TYPE_INTEGER,           /* type to associate */
                  int padding,            /* # places to zero-pad to */
                  int vmin, int vmax);    /* valid range */
```

Valid characters consist of an optional leading minus and digits. The range check is performed on exit. If the range maximum is less than or equal to the minimum, the range is ignored.

If the value passes its range check, it is padded with as many leading zero digits as necessary to meet the padding argument.

A TYPE_INTEGER value buffer can conveniently be interpreted with the C library function *atoi*.

4.8.5 TYPE_NUMERIC

This field type accepts a decimal number. It is set up as follows:

```
int set_field_type(FIELD *field,          /* field to alter */
                  TYPE_NUMERIC,           /* type to associate */
                  int padding,            /* # places of precision */
                  int vmin, int vmax);    /* valid range */
```

Valid characters consist of an optional leading minus and digits, possibly including a decimal point. The range check is performed on exit. If the range maximum is less than or equal to the minimum, the range is ignored.

If the value passes its range check, it is padded with as many trailing zero digits as necessary to meet the padding argument.

A `TYPE_NUMERIC` value buffer can conveniently be interpreted with the C library function `atof`.

4.8.6 TYPE_REGEX

This field type accepts data matching a regular expression. It is set up as follows:

```
int set_field_type(FIELD *field,          /* field to alter */
                  TYPE_REGEX,           /* type to associate */
                  char *regex);         /* expression to match */
```

The syntax for regular expressions is that of *regcomp*. The check for regular-expression match is performed on exit.

4.9 Direct Field Buffer Manipulation

The most central attribute of a field is its buffer contents. When a form has been completed, your application usually needs to know the state of each field buffer. You can find this out with:

```
char *field_buffer(FIELD *field,          /* field to query */
                  int bufindex);         /* number of buffer to query */
```

Normally, the state of the zero-numbered buffer for each field is set by the user's editing actions on that field. It's sometimes useful to be able to set the value of the zero-numbered (or some other) buffer from your application:

```
int set_field_buffer(FIELD *field,        /* field to alter */
                    int bufindex,        /* number of buffer to alter */
                    char *value);        /* string value to set */
```

If the field is not large enough and cannot be resized to a sufficiently large size to contain the specified value, the value will be truncated to fit.

Calling `field_buffer()` with a null field pointer will raise an error. Calling `field_buffer()` on a field not currently selected for input will return a correct value. Calling `field_buffer()` on a field that is currently selected for input may not necessarily give a correct field buffer value, because entered data isn't necessarily copied to buffer zero before the exit validation check.

To guarantee that the returned buffer value reflects on-screen reality, call `field_buffer()` either (1) in the field's exit validation check routine, (2) from the field's or form's initialization or termination hooks, or (3) just after a `REQ_VALIDATION` request has been processed by the forms driver.

4.10 Attributes of Forms

As with field attributes, form attributes inherit a default from a system default form structure. These defaults can be queried or set by of these functions using a form-pointer argument of NULL.

The most important attribute of a form is its field list. You can query and change this list with:

```
int  set_form_fields(FORM *form,           /* form to alter */
                    FIELD **fields);      /* fields to connect */

char *form_fields(FORM *form);           /* fetch fields of form */

int  field_count(FORM *form);            /* count connect fields */
```

The second argument of *set_form_fields()* may be a NULL-terminated field pointer array like the one required by *new_form()*. In that case, the old fields of the form are disconnected but not freed (and eligible to be connected to other forms), then the new fields are connected.

It may also be null, in which case the old fields are disconnected (and not freed) but no new ones are connected.

The *field_count()* function simply counts the number of fields connected to a given form. It returns -1 if the form-pointer argument is NULL.

4.11 Control of Form Display

In the overview section, you saw that to display a form you normally start by defining its size (and fields), posting it, and refreshing the screen. There is an hidden step before posting, which is the association of the form with a frame window (actually, a pair of windows) within which it will be displayed. By default, the forms library associates every form with the full-screen window *stdscr*.

By making this step explicit, you can associate a form with a declared frame window on your screen display. This can be useful if you want to adapt the form display to different screen sizes, dynamically tile forms on the screen, or use a form as part of an interface layout managed by panels.

The two windows associated with each form have the same functions as their analogues in the menu library. Both these windows are painted when the form is posted and erased when the form is unposted.

The outer or frame window is not otherwise touched by the form routines. It exists so the programmer can associate a title, a border, or perhaps help text with the form and have it properly refreshed or erased at post/unpost time. The inner window or subwindow is where the current form page is actually displayed.

In order to declare your own frame window for a form, you'll need to know the size of the form's bounding rectangle. You can get this information with:

```
int  scale_form(FORM *form,           /* form to query */
                int  *rows,           /* form rows */
                int  *cols);          /* form cols */
```

The form dimensions are passed back in the locations pointed to by the arguments. Once you have this information, you can use it to declare of windows, then use one of these functions:

```
int  set_form_win(FORM *form,          /* form to alter */
                  WINDOW *win);       /* frame window to connect */

WINDOW *form_win(FORM *form);         /* fetch frame window of form */

int  set_form_sub(FORM *form,          /* form to alter */
                  WINDOW *win);       /* form subwindow to connect */

WINDOW *form_sub(FORM *form);         /* fetch form subwindow of form */
```

Note that curses operations, including *refresh()*, on the form, should be done on the frame window, not the form subwindow.

It is possible to check from your application whether all of a scrollable field is actually displayed within the menu subwindow. Use these functions:

```
int  data_ahead(FORM *form);          /* form to be queried */

int  data_behind(FORM *form);         /* form to be queried */
```

The function *data_ahead()* returns TRUE if (a) the current field is one-line and has undisplayed data off to the right, (b) the current field is multi-line and there is data off-screen below it.

The function *data_behind()* returns TRUE if the first (upper left hand) character position is off-screen (not being displayed).

Finally, there is a function to restore the form window's cursor to the value expected by the forms driver:

```
int  pos_form_cursor(FORM *form);     /* form to be queried */
```

If your application changes the form window cursor, call this function before handing control back to the forms driver in order to re-synchronize it.

4.12 Input Processing in the Forms Driver

The function *form_driver()* handles virtualized input requests for form navigation, editing, and validation requests, just as *menu_driver()* does for menus (see the section on menu input handling).

```
int  form_driver(FORM *form,          /* form to pass input to */
                 int  request);       /* form request code */
```

Your input virtualization function needs to take input and then convert it to either an alphanumeric character (which is treated as data to be entered in the currently-selected field), or a forms processing request.

The forms driver provides hooks (through input-validation and field-termination functions) with which your application code can check that the input taken by the driver matched what was expected.

4.12.1 Page Navigation Requests

These requests cause page-level moves through the form, triggering display of a new form screen.

REQ_NEXT_PAGE

Move to the next form page.

REQ_PREV_PAGE

Move to the previous form page.

REQ_FIRST_PAGE

Move to the first form page.

REQ_LAST_PAGE

Move to the last form page.

These requests treat the list as cyclic; that is, REQ_NEXT_PAGE from the last page goes to the first, and REQ_PREV_PAGE from the first page goes to the last.

4.12.2 Inter-Field Navigation Requests

These requests handle navigation between fields on the same page.

REQ_NEXT_FIELD

Move to next field.

REQ_PREV_FIELD

Move to previous field.

REQ_FIRST_FIELD

Move to the first field.

REQ_LAST_FIELD

Move to the last field.

REQ_SNEXT_FIELD

Move to sorted next field.

REQ_SPREV_FIELD

Move to sorted previous field.

REQ_SFIRST_FIELD

Move to the sorted first field.

REQ_SLAST_FIELD

Move to the sorted last field.

REQ_LEFT_FIELD

Move left to field.

REQ_RIGHT_FIELD

Move right to field.

REQ_UP_FIELD

Move up to field.

REQ_DOWN_FIELD

Move down to field.

These requests treat the list of fields on a page as cyclic; that is, `REQ_NEXT_FIELD()` from the last field goes to the first, and `REQ_PREV_FIELD()` from the first field goes to the last. The order of the fields for these (and the `REQ_FIRST_FIELD` and `REQ_LAST_FIELD()` requests) is simply the order of the field pointers in the form array (as set up by `new_form()` or `set_form_fields()`).

It is also possible to traverse the fields as if they had been sorted in screen-position order, so the sequence goes left-to-right and top-to-bottom. To do this, use the second group of four sorted-movement requests.

Finally, it is possible to move between fields using visual directions up, down, right, and left. To accomplish this, use the third group of four requests. Note, however, that the position of a form for purposes of these requests is its upper-left corner.

For example, suppose you have a multi-line field B, and two single-line fields A and C on the same line with B, with A to the left of B and C to the right of B. A `REQ_MOVE_RIGHT` from A will go to B only if A, B, and C *all* share the same first line; otherwise it will skip over B to C.

4.12.3 Intra-Field Navigation Requests

These requests drive movement of the edit cursor within the currently selected field.

`REQ_NEXT_CHAR`

Move to next character.

`REQ_PREV_CHAR`

Move to previous character.

`REQ_NEXT_LINE`

Move to next line.

`REQ_PREV_LINE`

Move to previous line.

`REQ_NEXT_WORD`

Move to next word.

`REQ_PREV_WORD`

Move to previous word.

`REQ_BEG_FIELD`

Move to beginning of field.

`REQ_END_FIELD`

Move to end of field.

`REQ_BEG_LINE`

Move to beginning of line.

`REQ_END_LINE`

Move to end of line.

`REQ_LEFT_CHAR`

Move left in field.

`REQ_RIGHT_CHAR`

Move right in field.

REQ_UP_CHAR

Move up in field.

REQ_DOWN_CHAR

Move down in field.

Each *word* is separated from the previous and next characters by whitespace. The commands to move to beginning and end of line or field look for the first or last non-pad character in their ranges.

4.12.4 Scrolling Requests

Fields that are dynamic and have grown and fields explicitly created with offscreen rows are scrollable. One-line fields scroll horizontally; multi-line fields scroll vertically. Most scrolling is triggered by editing and intra-field movement (the library scrolls the field to keep the cursor visible). It is possible to explicitly request scrolling with the following requests:

REQ_SCR_FLINE

Scroll vertically forward a line.

REQ_SCR_BLINE

Scroll vertically backward a line.

REQ_SCR_FPAGE

Scroll vertically forward a page.

REQ_SCR_BPAGE

Scroll vertically backward a page.

REQ_SCR_FHPAGE

Scroll vertically forward half a page.

REQ_SCR_BHPAGE

Scroll vertically backward half a page.

REQ_SCR_FCHAR

Scroll horizontally forward a character.

REQ_SCR_BCHAR

Scroll horizontally backward a character.

REQ_SCR_HFLINE

Scroll horizontally one field width forward.

REQ_SCR_HBLINE

Scroll horizontally one field width backward.

REQ_SCR_HFHALF

Scroll horizontally one half field width forward.

REQ_SCR_HBHALF

Scroll horizontally one half field width backward.

For scrolling purposes, a *page* of a field is the height of its visible part.

4.12.5 Editing Requests

When you pass the forms driver an ASCII character, it is treated as a request to add the character to the field's data buffer. Whether this is an insertion or a replacement depends on the field's edit mode (insertion is the default).

The following requests support editing the field and changing the edit mode:

REQ_INS_MODE

Set insertion mode.

REQ_OVL_MODE

Set overlay mode.

REQ_NEW_LINE

New line request (see below for explanation).

REQ_INS_CHAR

Insert space at character location.

REQ_INS_LINE

Insert blank line at character location.

REQ_DEL_CHAR

Delete character at cursor.

REQ_DEL_PREV

Delete previous word at cursor.

REQ_DEL_LINE

Delete line at cursor.

REQ_DEL_WORD

Delete word at cursor.

REQ_CLR_EOL

Clear to end of line.

REQ_CLR_EOF

Clear to end of field.

REQ_CLEAR_FIELD

Clear entire field.

The behavior of the REQ_NEW_LINE and REQ_DEL_PREV requests is complicated and partly controlled by a pair of forms options. The special cases are triggered when the cursor is at the beginning of a field, or on the last line of the field.

First, we consider REQ_NEW_LINE:

The normal behavior of REQ_NEW_LINE in insert mode is to break the current line at the position of the edit cursor, inserting the portion of the current line after the cursor as a new line following the current and moving the cursor to the beginning of that new line (you may think of this as inserting a newline in the field buffer).

The normal behavior of REQ_NEW_LINE in overlay mode is to clear the current line from the position of the edit cursor to end of line. The cursor is then moved to the beginning of the next line.

However, REQ_NEW_LINE at the beginning of a field, or on the last line of a field, instead does a REQ_NEXT_FIELD. If the O_NL_OVERLOAD option is off, this special

action is disabled.

Now, let us consider REQ_DEL_PREV:

The normal behavior of REQ_DEL_PREV is to delete the previous character. If insert mode is on, and the cursor is at the start of a line, and the text on that line will fit on the previous one, it instead appends the contents of the current line to the previous one and deletes the current line (you may think of this as deleting a newline from the field buffer).

However, REQ_DEL_PREV at the beginning of a field is instead treated as a REQ_PREV_FIELD.

If the O_BS_OVERLOAD option is off, this special action is disabled and the forms driver just returns E_REQUEST_DENIED.

See Form Options for discussion of how to set and clear the overload options.

4.12.6 Order Requests

If the type of your field is ordered, and has associated functions for getting the next and previous values of the type from a given value, there are requests that can fetch that value into the field buffer:

REQ_NEXT_CHOICE

Place the successor value of the current value in the buffer.

REQ_PREV_CHOICE

Place the predecessor value of the current value in the buffer.

Of the built-in field types, only TYPE_ENUM has built-in successor and predecessor functions. When you define a field type of your own (see Custom Validation Types), you can associate our own ordering functions.

4.12.7 Application Commands

Form requests are represented as integers above the **curses** value greater than KEY_MAX and less than or equal to the constant MAX_COMMAND. If your input-virtualization routine returns a value above MAX_COMMAND, the forms driver will ignore it.

4.13 Field Change Hooks

It is possible to set function hooks to be executed whenever the current field or form changes. Here are the functions that support this:

```
typedef void (*HOOK)();          /* pointer to function returning void */

int set_form_init(FORM *form,    /* form to alter */
                  HOOK hook);    /* initialization hook */

HOOK form_init(FORM *form);      /* form to query */

int set_form_term(FORM *form,    /* form to alter */
                  HOOK hook);    /* termination hook */
```

```

HOOK form_term(FORM *form);          /* form to query */

int set_field_init(FORM *form,        /* form to alter */
                  HOOK hook);        /* initialization hook */

HOOK field_init(FORM *form);          /* form to query */

int set_field_term(FORM *form,        /* form to alter */
                  HOOK hook);        /* termination hook */

HOOK field_term(FORM *form);          /* form to query */

```

These functions allow you to either set or query four different hooks. In each of the set functions, the second argument should be the address of a hook function. These functions differ only in the timing of the hook call.

form_init

This hook is called when the form is posted; also, just after each page change operation.

field_init

This hook is called when the form is posted; also, just after each field change

field_term

This hook is called just after field validation; that is, just before the field is altered. It is also called when the form is unposted.

form_term

This hook is called when the form is unposted; also, just before each page change operation.

Calls to these hooks may be triggered

1. When user editing requests are processed by the forms driver
2. When the current page is changed by *set_current_field()* call
3. When the current field is changed by a *set_form_page()* call

See Field Change Commands for discussion of the latter two cases.

You can set a default hook for all fields by passing one of the set functions a NULL first argument.

You can disable any of these hooks by (re)setting them to NULL, the default value.

4.14 Field Change Commands

Normally, navigation through the form will be driven by the user's input requests. But sometimes it is useful to be able to move the focus for editing and viewing under control of your application, or ask which field it currently is in. The following functions help you accomplish this:

```
int set_current_field(FORM *form,          /* form to alter */
                    FIELD *field);        /* field to shift to */

FIELD *current_field(FORM *form);         /* form to query */

int field_index(FORM *form,               /* form to query */
               FIELD *field);             /* field to get index of */
```

The function *field_index()* returns the index of the given field in the given form's field array (the array passed to *new_form()* or *set_form_fields()*).

The initial current field of a form is the first active field on the first page. The function *set_form_fields()* resets this.

It is also possible to move around by pages.

```
int set_form_page(FORM *form,             /* form to alter */
                 int page);               /* page to go to (0-origin) */

int form_page(FORM *form);                /* return form's current page */
```

The initial page of a newly-created form is 0. The function *set_form_fields()* resets this.

4.15 Form Options

Like fields, forms may have control option bits. They can be changed or queried with these functions:

```
int set_form_opts(FORM *form,             /* form to alter */
                 int attr);               /* attribute to set */

int form_opts_on(FORM *form,              /* form to alter */
                int attr);               /* attributes to turn on */

int form_opts_off(FORM *form,             /* form to alter */
                 int attr);              /* attributes to turn off */

int form_opts(FORM *form);                /* form to query */
```

By default, all options are on. Here are the available option bits:

O_NL_OVERLOAD

Enable overloading of REQ_NEW_LINE as described in Editing Requests. The value of this option is ignored on dynamic fields that have not reached their size limit; these have no last line, so the circumstances for triggering a REQ_NEXT_FIELD never arise.

O_BS_OVERLOAD

Enable overloading of REQ_DEL_PREV as described in Editing Requests.

The option values are bit-masks and can be composed with logical-or in the obvious way.

4.16 Custom Validation Types

The **form** library gives you the capability to define custom validation types of your own. Further, the optional additional arguments of *set_field_type()* effectively allow you to parameterize validation types. Most of the complications in the validation-type interface have to do with the handling of the additional arguments within custom validation functions.

4.16.1 Union Types

The simplest way to create a custom data type is to compose it from two preexisting ones:

```
FIELD *link_fielddtype(FIELDTYPE *type1,
                      FIELDTYPE *type2);
```

This function creates a field type that will accept any of the values legal for either of its argument field types (which may be either predefined or programmer-defined).

If a *set_field_type()* call later requires arguments, the new composite type expects all arguments for the first type, then all arguments for the second. Order functions (see Order Requests) associated with the component types will work on the composite; what it does is check the validation function for the first type, then for the second, to figure what type the buffer contents should be treated as.

4.16.2 New Field Types

To create a field type from scratch, you need to specify one or both of the following things:

- A character-validation function, to check each character as it is entered.
- A field-validation function to be applied on exit from the field.

Here's how you do that:

```
typedef int (*HOOK)();          /* pointer to function returning int */
```

```
FIELDTYPE *new_fielddtype(HOOK f_validate, /* field validator */
                        HOOK c_validate); /* character validator */
```

```
int free_fielddtype(FIELDTYPE *ftype);    /* type to free */
```

At least one of the arguments of *new_fielddtype()* must be non-NULL. The forms driver will automatically call the new type's validation functions at appropriate points in processing a field of the new type.

The function *free_fielddtype()* deallocates the argument fieldtype, freeing all storage associated with it.

Normally, a field validator is called when the user attempts to leave the field. Its first argument is a field pointer, from which it can get to field buffer 0 and test it. If the function returns TRUE, the operation succeeds; if it returns FALSE, the edit cursor stays in the field.

A character validator gets the character passed in as a first argument. It too should return TRUE if the character is valid, FALSE otherwise.

4.16.3 Validation Function Arguments

Your field- and character- validation functions will be passed a second argument as well. This second argument is the address of a structure (which we'll call a *pile*) built from any of the field-type-specific arguments passed to *set_field_type()*. If no such arguments are defined for the field type, this pile pointer argument will be NULL.

In order to arrange for such arguments to be passed to your validation functions, you must associate a small set of storage-management functions with the type. The forms driver will use these to synthesize a pile from the trailing arguments of each *set_field_type()* argument, and a pointer to the pile will be passed to the validation functions.

Here is how you make the association:

```
typedef char  *(*PTRHOOK)();      /* pointer to function returning (char *) */
typedef void  (*VOIDHOOK)();      /* pointer to function returning void */

int set_fieldtype_arg(FIELDTYPE *ftype,      /* type to alter */
                     PTRHOOK  fmake_str,    /* make structure from args */
                     PTRHOOK  fcopy_str,    /* make copy of structure */
                     VOIDHOOK  ffree_str);  /* free structure storage */
```

Here is how the storage-management hooks are used:

make_str

This function is called by *set_field_type()*. It gets one argument, a **va_list** of the type-specific arguments passed to *set_field_type()*. It is expected to return a pile pointer to a data structure that encapsulates those arguments.

copy_str

This function is called by form library functions that allocate new field instances. It is expected to take a pile pointer, copy the pile to allocated storage, and return the address of the pile copy.

free_str

This function is called by field- and type-deallocation routines in the library. It takes a pile pointer argument, and is expected to free the storage of that pile.

The *make_str* and *copy_str* functions may return NULL to signal allocation failure. The library routines will that call them will return error indication when this happens. Thus, your validation functions should never see a NULL pile pointer and need not check specially for it.

4.16.4 Order Functions For Custom Types

Some custom field types are simply ordered in the same well-defined way that `TYPE_ENUM` is. For such types, it is possible to define successor and predecessor functions to support the `REQ_NEXT_CHOICE` and `REQ_PREV_CHOICE` requests. Here's how:

```
typedef int  (*INTHOOK)();      /* pointer to function returning int */

int set_fielddtype_arg(FIELDTYPE *type,      /* type to alter */
                      INTHOOK succ,         /* get successor value */
                      INTHOOK pred);        /* get predecessor value */
```

The successor and predecessor arguments will each be passed two arguments; a field pointer, and a pile pointer (as for the validation functions). They are expected to use the function `field_buffer()` to read the current value, and `set_field_buffer()` on buffer 0 to set the next or previous value. Either hook may return `TRUE` to indicate success (a legal next or previous value was set) or `FALSE` to indicate failure.

4.16.5 Avoiding Problems

The interface for defining custom types is complicated and tricky. Rather than attempting to create a custom type entirely from scratch, you should start by studying the library source code for whichever of the pre-defined types seems to be closest to what you want.

Use that code as a model, and evolve it towards what you really want. You will avoid many problems and annoyances that way. The code in the **ncurses** library has been specifically un-copyrighted to support this.

If your custom type defines order functions, have do something intuitive with a blank field. A useful convention is to make the successor of a blank field the types minimum value, and its predecessor the maximum.

Full descriptions of the form library entry points are available in the *form_** manual pages included with the **ncurses** distributions.

Eric S. Raymond <esr@snark.thyrsus.com>

Curses Reference Pages

NAME

ncurses – CRT screen handling and optimization package

SYNOPSIS

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

DESCRIPTION

The **curses** library routines give the user a terminal-independent method of updating character screens with reasonable optimization. This implementation is “new curses” (ncurses) and is the approved replacement for 4.4BSD classic curses, which is being discontinued.

The **ncurses** routines emulate the *curses(3X)* library of System V Release 4, and the XPG4 curses standard (XSI curses) but the **ncurses** library is freely redistributable in source form. Differences from the SVr4 curses are summarized under the EXTENSIONS and BUGS sections below and described in detail in the EXTENSIONS and BUGS sections of individual man pages.

A program using these routines must be linked with the **-lncurses** option, or (if it has been generated) with the debugging library **-lncurses_g**. The ncurses_g library generates trace logs (in a file called 'trace' in the current directory) that describe curses actions.

The **ncurses** package supports: overall screen, window and pad manipulation; output to windows and pads; reading terminal input; control over terminal and **curses** input and output options; environment query routines; color manipulation; use of soft label keys; terminfo capabilities; and access to low-level terminal-manipulation routines.

To initialize the routines, the routine **initscr** or **newterm** must be called before any of the other routines that deal with windows and screens are used. The routine **endwin** must be called before exiting. To get character-at-a-time input without echoing (most interactive, screen oriented programs want this), the following sequence should be used:

```
initscr();
cbreak();
noecho();
```

Most programs would additionally use the sequence:

```
nonl();
intrflush(stdscr, FALSE);
keypad(stdscr, TRUE);
```

Before a **curses** program is run, the tab stops of the terminal should be set and its initialization strings, if defined, must be output. This can be done by executing the

```
$ tput init
```

command after the shell environment variable *TERM* has been exported. *tset* is usually responsible for doing this. (See *terminfo* for further details.)

The **curses** library permits manipulation of data structures, called **windows**, which can be thought of as two-dimensional arrays of characters representing all or part of a CRT screen. A default window called *stdscr*, which is the size of the terminal screen, is supplied. Others may be created with *newwin()*.

Note that **curses** does not handle overlapping windows, that's done by the **panels(3x)** library. This means that you can either use *stdscr* or divide the screen into tiled windows

and not using *stdscr* at all. Mixing the two will result in unpredictable, and undesired, effects.

Windows are referred to by variables declared as **WINDOW ***. These data structures are manipulated with routines described here and elsewhere in the **ncurses** manual pages. Among which the most basic routines are *move()* and *addch()*. More general versions of these routines are included with names beginning with **w**, allowing the user to specify a window. The routines not beginning with **w** affect *stdscr*.)

After using routines to manipulate a window, *refresh()* is called, telling **curses** to make the user's CRT screen look like *stdscr*. The characters in a window are actually of type **chtype** (character and attribute data), so that other information about the character may also be stored with each character.

Special windows called **pads** may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be completely displayed. See *curs_pad* for more information.

In addition to drawing characters on the screen, video attributes and colors may be supported, causing the characters to show up in such modes as underlined, in reverse video, or in color on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, **curses** is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in **<curses.h>**, such as **A_REVERSE**, **ACS_HLINE**, and **KEY_LEFT**.

If the environment variables *LINES* and *COLUMNS* are set, or if the program is executing in a window environment, line and column information in the environment will override information read by **terminfo**. This would effect a program running in an AT&T 630 layer, for example, where the size of a screen is changeable.

If the environment variable *TERMINFO* is defined, any program using **curses** checks for a local terminal definition before checking in the standard place. For example, if *TERM* is set to **att4424**, then the compiled terminal definition is found in

\$TERMINFO/a/att4424.

(The **a** is copied from the first letter of **att4424** to avoid creation of huge directories.)

However, if *TERMINFO* is set to **\$HOME/myterms**, **curses** first checks

\$HOME/myterms/a/att4424

and if that fails, it then checks

/a/att4424

This is useful for developing experimental definitions or when write permission in *is* not available.

The integer variables *LINES* and *COLS* are defined in **<curses.h>** and will be filled in by *initscr()* with the size of the screen. The constants **TRUE** and **FALSE** have the values 1 and 0, respectively.

The **curses** routines also define the **WINDOW *** variable *curscr* which is used for certain low-level operations like clearing and redrawing a screen containing garbage. The *curscr* can be used in only a few routines.

Routine and Argument Names

Many **curses** routines have two or more versions. The routines prefixed with **w** require a window argument. The routines prefixed with **p** require a pad argument. Those without a prefix generally use *stdscr*.

The routines prefixed with **mv** require a *y* and *x* coordinate to move to before performing the appropriate action. The **mv** routines imply a call to *move()* before the call to the other routine. The coordinate *y* always refers to the row (of the window), and *x* always refers to the column. The upper left-hand corner is always (0,0), not (1,1).

The routines prefixed with **mvw** take both a window argument and *x* and *y* coordinates. The window argument is always specified before the coordinates.

In each case, *win* is the window affected, and *pad* is the pad affected; *win* and *pad* are always pointers to type **WINDOW**.

Option setting routines require a Boolean flag *bf* with the value TRUE or FALSE; *bf* is always of type **bool**. The variables *ch* and *attrs* below are always of type **chtype**. The types **WINDOW**, **SCREEN**, **bool**, and **chtype** are defined in **<curses.h>**. The type **TERMINAL** is defined in **<term.h>**. All other arguments are integers.

Routine Name Index

The following table lists each **curses** routine and the name of the manual page on which it is described. Routines flagged with “*” are **ncurses**-specific, not described by XPG4 or present in SVr4.

curses Routine Name	Manual Page Name
<i>addch()</i>	<i>curs_addch</i>
<i>addchnstr()</i>	<i>curs_addchstr</i>
<i>addchstr()</i>	<i>curs_addchstr</i>
<i>addnstr()</i>	<i>curs_addstr</i>
<i>addstr()</i>	<i>curs_addstr</i>
<i>attroff()</i>	<i>curs_attr</i>
<i>attron()</i>	<i>curs_attr</i>
<i>attrset()</i>	<i>curs_attr</i>
<i>baudrate()</i>	<i>curs_termattrs</i>
<i>beep()</i>	<i>curs_beep</i>
<i>bkgd()</i>	<i>curs_bkgd</i>
<i>bkgdset()</i>	<i>curs_bkgd</i>
<i>border()</i>	<i>curs_border</i>
<i>box()</i>	<i>curs_border</i>
<i>can_change_color()</i>	<i>curs_color</i>
<i>cbreak()</i>	<i>curs_inopts</i>
<i>clear()</i>	<i>curs_clear</i>
<i>clearok()</i>	<i>curs_outopts</i>
<i>clrtoebot()</i>	<i>curs_clear</i>
<i>clrtoeol()</i>	<i>curs_clear</i>
<i>color_content()</i>	<i>curs_color</i>

curses Routine Name	Manual Page Name
<code>copywin()</code>	<code>curs_overlay</code>
<code>curs_set()</code>	<code>curs_kernel</code>
<code>def_prog_mode()</code>	<code>curs_kernel</code>
<code>def_shell_mode()</code>	<code>curs_kernel</code>
<code>del_curterm()</code>	<code>curs_terminfo</code>
<code>delay_output()</code>	<code>curs_util</code>
<code>delch()</code>	<code>curs_delch</code>
<code>deleteln()</code>	<code>curs_deleteln</code>
<code>delscreen()</code>	<code>curs_initscr</code>
<code>delwin()</code>	<code>curs_window</code>
<code>derwin()</code>	<code>curs_window</code>
<code>doupdate()</code>	<code>curs_refresh</code>
<code>dupwin()</code>	<code>curs_window</code>
<code>echo()</code>	<code>curs_inopts</code>
<code>echochar()</code>	<code>curs_addch</code>
<code>endwin()</code>	<code>curs_initscr</code>
<code>erase()</code>	<code>curs_clear</code>
<code>erasechar()</code>	<code>curs_termattrs</code>
<code>filter()</code>	<code>curs_util</code>
<code>flash()</code>	<code>curs_beep</code>
<code>flushinp()</code>	<code>curs_util</code>
<code>getbegyx()</code>	<code>curs_getyx</code>
<code>getch()</code>	<code>curs_getch</code>
<code>getmaxyx()</code>	<code>curs_getyx</code>
<code>getmouse()</code>	<code>curs_mouse*</code>
<code>getparyx()</code>	<code>curs_getyx</code>
<code>getstr()</code>	<code>curs_getstr</code>
<code>getsyx()</code>	<code>curs_kernel</code>
<code>getwin()</code>	<code>curs_util</code>
<code>getyx()</code>	<code>curs_getyx</code>
<code>halfdelay()</code>	<code>curs_inopts</code>
<code>has_colors()</code>	<code>curs_color</code>
<code>has_ic()</code>	<code>curs_termattrs</code>
<code>has_il()</code>	<code>curs_termattrs</code>
<code>hline()</code>	<code>curs_border</code>
<code>idcok()</code>	<code>curs_outopts</code>
<code>idlok()</code>	<code>curs_outopts</code>
<code>immedok()</code>	<code>curs_outopts</code>
<code>inch()</code>	<code>curs_inch</code>
<code>inchnstr()</code>	<code>curs_inchstr</code>
<code>inchstr()</code>	<code>curs_inchstr</code>
<code>init_color()</code>	<code>curs_color</code>
<code>init_pair()</code>	<code>curs_color</code>
<code>initscr()</code>	<code>curs_initscr</code>
<code>innstr()</code>	<code>curs_instr</code>
<code>insch()</code>	<code>curs_insch</code>

curses Routine Name	Manual Page Name
insertln()	curs_deleteln
insnstr()	curs_insstr
insstr()	curs_insstr
instr()	curs_instr
intrflush()	curs_inopts
is_linetouched()	curs_touch
is_wintouched()	curs_touch
isendwin()	curs_initscr
keyname()	curs_util
keypad()	curs_inopts
killchar()	curs_termattrs
leaveok()	curs_outopts
longname()	curs_termattrs
meta()	curs_inopts
mouseinterval()	curs_mouse*
mousemask()	curs_mouse*
move()	curs_move
mvaddch()	curs_addch
mvaddchnstr()	curs_addchstr
mvaddchstr()	curs_addchstr
mvaddnstr()	curs_addstr
mvaddstr()	curs_addstr
mvcur()	curs_terminfo
mvdelch()	curs_delch
mvderwin()	curs_window
mvgetch()	curs_getch
mvgetstr()	curs_getstr
mvinch()	curs_inch
mvinchnstr()	curs_inchstr
mvinchstr()	curs_inchstr
mvinnstr()	curs_instr
mvinsch()	curs_insch
mvinsnstr()	curs_insstr
mvinsstr()	curs_insstr
mvinstr()	curs_instr
mvprintw()	curs_printw
mvscanw()	curs_scanw
mvwaddch()	curs_addch
mvwaddchnstr()	curs_addchstr
mvwaddchstr()	curs_addchstr
mvwaddnstr()	curs_addstr
mvwaddstr()	curs_addstr
mvwdelch()	curs_delch
mvwgetch()	curs_getch
mvwgetstr()	curs_getstr
mvwin()	curs_window

curses Routine Name	Manual Page Name
<code>mvwinchnstr()</code>	<code>curs_inchstr</code>
<code>mvwinchstr()</code>	<code>curs_inchstr</code>
<code>mvwinnstr()</code>	<code>curs_instr</code>
<code>mvwinsch()</code>	<code>curs_insch</code>
<code>mvwinsnstr()</code>	<code>curs_insstr</code>
<code>mvwinsstr()</code>	<code>curs_insstr</code>
<code>mvwinstr()</code>	<code>curs_instr</code>
<code>mvwprintw()</code>	<code>curs_printw</code>
<code>mvwscanw()</code>	<code>curs_scanw</code>
<code>napms()</code>	<code>curs_kernel</code>
<code>newpad()</code>	<code>curs_pad</code>
<code>newterm()</code>	<code>curs_initscr</code>
<code>newwin()</code>	<code>curs_window</code>
<code>nl()</code>	<code>curs_outopts</code>
<code>nocbreak()</code>	<code>curs_inopts</code>
<code>nodelay()</code>	<code>curs_inopts</code>
<code>noecho()</code>	<code>curs_inopts</code>
<code>nonl()</code>	<code>curs_outopts</code>
<code>noqiflush()</code>	<code>curs_inopts</code>
<code>noraw()</code>	<code>curs_inopts</code>
<code>notimeout()</code>	<code>curs_inopts</code>
<code>overlay()</code>	<code>curs_overlay</code>
<code>overwrite()</code>	<code>curs_overlay</code>
<code>pair_content()</code>	<code>curs_color</code>
<code>pechochar()</code>	<code>curs_pad</code>
<code>pnoutrefresh()</code>	<code>curs_pad</code>
<code>prefresh()</code>	<code>curs_pad</code>
<code>printw()</code>	<code>curs_printw</code>
<code>putp()</code>	<code>curs_terminfo</code>
<code>putwin()</code>	<code>curs_util</code>
<code>qiflush()</code>	<code>curs_inopts</code>
<code>raw()</code>	<code>curs_inopts</code>
<code>redrawwin()</code>	<code>curs_refresh</code>
<code>refresh()</code>	<code>curs_refresh</code>
<code>reset_prog_mode()</code>	<code>curs_kernel</code>
<code>reset_shell_mode()</code>	<code>curs_kernel</code>
<code>resetty()</code>	<code>curs_kernel</code>
<code>restartterm()</code>	<code>curs_terminfo</code>
<code>ripoffline()</code>	<code>curs_kernel</code>
<code>savetty()</code>	<code>curs_kernel</code>
<code>scanw()</code>	<code>curs_scanw</code>
<code>scr_dump()</code>	<code>curs_scr_dump</code>
<code>scr_init()</code>	<code>curs_scr_dump</code>
<code>scr_restore()</code>	<code>curs_scr_dump</code>
<code>scr_set()</code>	<code>curs_scr_dump</code>
<code>scrI()</code>	<code>curs_scroll</code>

curses Routine Name	Manuscript Page Name
scrolllok()	curs_outopts
set_curterm()	curs_terminfo
set_term()	curs_initscr
setscrreg()	curs_outopts
setsyx()	curs_kernel
setterm()	curs_terminfo
setupterm()	curs_terminfo
slk_attroff()	curs_slk
slk_attron()	curs_slk
slk_attrset()	curs_slk
slk_clear()	curs_slk
slk_init()	curs_slk
slk_label()	curs_slk
slk_noutrefresh()	curs_slk
slk_refresh()	curs_slk
slk_restore()	curs_slk
slk_set()	curs_slk
slk_touch()	curs_slk
standend()	curs_attr
standout()	curs_attr
start_color()	curs_color
subpad()	curs_pad
subwin()	curs_window
syncok()	curs_window
termattrs()	curs_termattrs
termname()	curs_termattrs
tigetflag()	curs_terminfo
tigetnum()	curs_terminfo
tigetstr()	curs_terminfo
timeout()	curs_inopts
touchline()	curs_touch
touchwin()	curs_touch
tparm()	curs_terminfo
tputs()	curs_terminfo
typeahead()	curs_inopts
unctrl()	curs_util
ungetch()	curs_getch
ungetmouse()	curs_mouse*
untouchwin()	curs_touch
use_env()	curs_util
vidattr()	curs_terminfo
vidputs()	curs_terminfo
vline()	curs_border
vwprintw()	curs_printw
vwscanw()	curs_scanw
waddch()	curs_addch

curses Routine Name	Manual Page Name
waddchstr()	curs_addchstr
waddnstr()	curs_addstr
waddstr()	curs_addstr
wattroff()	curs_attr
wattron()	curs_attr
wattrset()	curs_attr
wbkgd()	curs_bkgd
wbkgdset()	curs_bkgd
wborder()	curs_border
wclear()	curs_clear
wclrtoobot()	curs_clear
wclrtoeol()	curs_clear
wcursyncup()	curs_window
wdelch()	curs_delch
wdeleteln()	curs_deleteln
wechochar()	curs_addch
wenclose()	curs_mouse*
werase()	curs_clear
wgetch()	curs_getch
wgetnstr()	curs_getstr
wgetstr()	curs_getstr
whline()	curs_border
winch()	curs_inch
winchnstr()	curs_inchstr
winchstr()	curs_inchstr
winnstr()	curs_instr
winsch()	curs_insch
winsdelln()	curs_deleteln
winsertln()	curs_deleteln
winsnstr()	curs_insstr
winsstr()	curs_insstr
winstr()	curs_instr
wmove()	curs_move
wnoutrefresh()	curs_refresh
wprintw()	curs_printw
wredrawln()	curs_refresh
wrefresh()	curs_refresh
wresize()	curs_resize*
wscanw()	curs_scanw
wscr1()	curs_scroll
wsetscrreg()	curs_outopts
wstandend()	curs_attr
wstandout()	curs_attr
wsyncdown()	curs_window
wsyncup()	curs_window
wtimeout()	curs_inopts

curses Routine Name	Macros Routine Name
wvline()	curs_border

RETURN VALUE

Routines that return an integer return ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the routine descriptions.

All macros return the value of the w version, except *setscrreg()*, *wsetscrreg()*, *getyx()*, *getbegyx()*, *getmaxyx()*. The return values of *setscrreg()*, *wsetscrreg()*, *getyx()*, *getbegyx()*, and *getmaxyx()* are undefined (i.e., these should not be used as the right-hand side of assignment statements).

Routines that return pointers return NULL on error.

PORTABILITY

The **curses** library is intended to be BASE-level conformant with the XSI Curses standard. Certain portions of the EXTENDED XSI Curses functionality (including color support) are supported. The following EXTENDED XSI Curses calls in support of wide (multibyte) characters are not yet implemented (in alphabetical order):

add_wch(), *add_wchnstr()*, *add_wchstr()*, *addnwstr()*, *addwstr()*, *bkgrnd()*, *bkgrndset()*, *border_set()*, *box_set()*, *echo_wchar()*, *erasewchar()*, *get_wch()*, *get_wstr()*, *getbkgrnd()*, *getn_wstr()*, *getwctype()*, *hline_set()*, *in_wch()*, *innwstr()*, *ins_nwstr()*, *ins_wch()*, *ins_wstr()*, *inwchnstr()*, *inwchstr()*, *inwstr()*, *killwchar()*, *mvadd_wch()*, *mvadd_wchnstr()*, *mvadd_wchstr()*, *mvaddnwstr()*, *mvaddwstr()*, *mvget_wch()*, *mvget_wstr()*, *mvgetn_wstr()*, *mvhline_set()*, *mvin_wch()*, *mvinnwstr()*, *mvins_nwstr()*, *mvins_wch()*, *mvins_wstr()*, *mvinwchnstr()*, *mvinwchstr()*, *mvinwchstr()*, *mvinwstr()*, *mvvline_set()*, *mvwadd_wch()*, *mvwadd_wchnstr()*, *mvwadd_wchstr()*, *mvwaddnwstr()*, *mvwget_ch()*, *mvwget_wstr()*, *mvwgetn_wstr()*, *mvwhline_set()*, *mvwin_wch()*, *mvwinnwstr()*, *mvwins_nwstr()*, *mvwins_wch()*, *mvwins_wstr()*, *mvwinwchnstr()*, *mvwinwstr()*, *mvwvline_set()*, *vhline_set()*, *wadd_wch()*, *wadd_wchnstr()*, *wadd_wchstr()*, *waddnwstr()*, *waddwstr()*, *wbkgrnd()*, *wbkgrndset()*, *wborder_set()*, *wecho_wchar()*, *wget_wch()*, *wget_wstr()*, *wgetbkgrnd()*, *wgetn_wstr()*, *whline_set()*, *win_wch()*, *winnwstr()*, *wins_nwstr()*, *wins_wch()*, *wins_wstr()*, *winwchnstr()*, *winwchstr()*, *winwstr()*, *wvline_set()*.

A small number of local differences (that is, individual differences between the XSI Curses and **curses** calls) are described in **PORTABILITY** sections of the library man pages.

The terminfo format supported by **curses** is binary-compatible with SVr4, but not conformant with XSI curses. This is because the XSI Curses drafters, in a remarkable fit of braindamage, changed the undocumented SVr4 capability **getm** from a string to a boolean, changing the binary offsets of all capabilities after it in the SVr4 order and making it impossible for any SVr4-compatible implementation to be fully conformant.

The routines *getmouse()*, *mousemask()*, *ungetmouse()*, *mouseinterval()*, and *wenclose()* relating to mouse interfacing are not part of XPG4, nor are they present in SVr4. They are also not supported under INTERIX. See the *curs_mouse* manual page for details.

The routine *wresize()* is not part of XPG4, nor is it present in SVr4. See the *curs_resize* manual page for details.

In historic curses versions, delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff** and **tab** activated corresponding delay bits in the tty driver. In this implementation, all padding is done by NUL sends. This method is slightly more expensive, but narrows the interface to the kernel significantly and increases the package's portability.

correspondingly.

In the XSI standard and SVr4 manual pages, many entry points have prototype arguments of the form **char *const** (or **cchar_t *const**, or **wchar_t *const**, or **void *const**). Depending on one's interpretation of the ANSI C standard (see section 3.5.4.1), these declarations are either (a) meaningless, or (b) meaningless and illegal. The declaration **const char *x** is a modifiable pointer to unmodifiable data, but **char *const x** is an unmodifiable pointer to modifiable data. Given that C passes arguments by value, **<type> *const** as a formal type is at best dubious. Some compilers choke on the prototypes. Therefore, in this implementation, they have been changed to **const <type> *** globally.

NOTES

The header file **<curses.h>** automatically includes the header files **<stdio.h>** and **<unctrl.h>**.

If standard output from a **curses** program is re-directed to something which is not a tty, screen updates will be directed to standard error. This was an undocumented feature of AT&T System V Release 3 curses.

AUTHORS

Zeyd M. Ben-Halim, Eric S. Raymond. Descends from the original pcurses by Pavel Curtis.

SEE ALSO

terminfo and pages whose names begin *curs_* for detailed routine descriptions.

NAME

`addch()`, `waddch()`, `mvaddch()`, `mvwaddch()`, `echochar()`, `wechochar()` – add a character with attributes to a curses window then advance the cursor

SYNOPSIS

```
#include <curses.h>

int addch(chtype ch);
int waddch(WINDOW *win, chtype ch);
int mvaddch(int y, int x, chtype ch);
int mvwaddch(WINDOW *win, int y, int x, chtype ch);
int echochar(chtype ch);
int wechochar(WINDOW *win, chtype ch);
```

DESCRIPTION

The `addch()`, `waddch()`, `mvaddch()` and `mvwaddch()` routines put the character `ch` into the given window at its current window position, which is then advanced. They are analogous to `putchar()` in the stdio package.

If the advance is at the right margin, the cursor automatically wraps to the beginning of the next line. At the bottom of the current scrolling region, if **scrollok** is enabled, the scrolling region is scrolled up one line.

If `ch` is a tab, newline, or backspace, the cursor is moved appropriately within the window. Backspace moves the cursor one character left; at the left edge of a window it does nothing. Newline does a `clrtoeol()`, then moves the cursor to the window left margin on the next line, scrolling the window if on the last line). Tabs are considered to be at every eighth column.

If `ch` is any control character other than tab, newline, or backspace, it is drawn in `^X` notation. Calling `winch()` after adding a control character does not return the character itself, but instead returns the `^`-representation of the control character. (To emit control characters literally, use `echochar()`.)

Video attributes can be combined with a character argument passed to `addch()` or related functions by logical-ORing them into the character. (Thus, text, including attributes, can be copied from one place to another using `inch()` and `addch()`.) See the `curs_attr()` page for values of predefined video attribute constants that can be usefully ORed into characters.

The `echochar()` and `wechochar()` routines are equivalent to a call to `addch()` followed by a call to `refresh()`, or a call to `waddch()` followed by a call to `wrefresh()`. The knowledge that only a single character is being output is used and, for non-control characters, a considerable performance gain may be seen by using these routines instead of their equivalents.

Line Graphics

The following variables may be used to add line drawing characters to the screen with routines of the *addch* family. The default character listed below is used if the *acsc* capability doesn't define a terminal-specific replacement for it (but see the EXTENSIONS section below). The names are taken from VT100 nomenclature.

<i>Name</i>	<i>Default</i>	<i>Description</i>
ACS_ULCORNER	+	upper left-hand corner
ACS_LLCORNER	+	lower left-hand corner
ACS_URCORNER	+	upper right-hand corner
ACS_LRCORNER	+	lower right-hand corner
ACS_RTEE	+	right tee
ACS_LTEE	+	left tee
ACS_BTEE	+	bottom tee
ACS_TTEE	+	top tee
ACS_HLINE	-	horizontal line
ACS_VLINE		vertical line
ACS_PLUS	+	plus
ACS_S1	-	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus / minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block
ACS_S3	-	scan line 3
ACS_S7	-	scan line 7
ACS_LEQUAL	<	less-than-or-equal-to
ACS_GEQUAL	>	greater-than-or-equal-to
ACS_PI	*	greek pi
ACS_NEQUAL	!	not-equal
ACS_STERLING	f	pound-sterling symbol

RETURN VALUE

All routines return the integer ERR upon failure and OK on success (the SVr4 manuals specify only "an integer value other than ERR") upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

Note that *addch()*, *mvaddch()*, *mvwaddch()*, and *echochar()* may be macros.

EXTENSIONS

The following extended *curses()* features are available only on PC-clone consoles and compatible terminals obeying the ANSI.SYS de-facto standard for terminal control sequences. They are not part of XSI curses.

- The attribute `A_ALTCHARSET` actually forces literal display of PC ROM characters including the high-half graphics. Your console driver may still capture or translate a few (such as ESC) but this feature should give you access to the card-suit characters, up and down-arrow, and most others in the range 0-32. (In a terminfo entry designed for use with `ncurses()`, the high-half characters are obtained using this attribute with an `acsc()` string in which the second of each pair is a high-half character.)
- Giving `wechochar()` an argument with its high bit set will produce the corresponding high-half ASCII graphic (SVr4 curses also has this feature but does not document it). A control-character argument, however, will not typically produce the corresponding graphic; characters such as CR, NL, FF and TAB are typically interpreted by the console driver itself, and ESC will be interpreted as the leader of a control sequence.

PORTABILITY

All these functions are described in the XSI Curses standard, Issue 4. The defaults specified for forms-drawing characters apply in the POSIX locale.

The seven ACS symbols starting with `ACS_S3` were not documented in any publicly released System V. However, many publicly available terminfos include **acsc** strings in which their key characters (`pryz{|}`) are embedded, and a second-hand list of their character descriptions has come to light. The ACS-prefixed names for them were invented for **ncurses()**

SEE ALSO

curses, *curs_attr()*, *curs_clear()*, *curs_inch()*, *curs_outopts()*, *curs_refresh()*, *putc()*

NAME

addchstr(), addchnstr(), waddchstr(), waddchnstr(), mvaddchstr(), mvaddchnstr(), mvwaddchstr(), mvwaddchnstr() – add a string of characters (and attributes) to a curses window

SYNOPSIS

```
#include <curses.h>

int addchstr(const chtype *chstr);
int addchnstr(const chtype *chstr, int n);
int waddchstr(WINDOW *win, const chtype *chstr);
int waddchnstr(WINDOW *win, const chtype *chstr, int n);
int mvaddchstr(int y, int x, const chtype *chstr);
int mvaddchnstr(int y, int x, const chtype *chstr, int n);
int mvwaddchstr(WINDOW *win, int y, int x, const chtype *chstr);
int mvwaddchnstr(WINDOW *win, int y, int x, const chtype *chstr,
                  int n);
```

DESCRIPTION

These routines copy *chstr* into the window image structure at and after the current cursor position. The four routines with *n* as the last argument copy at most *n* elements, but no more than will fit on the line. If *n*=-1 then the whole string is copied, to the maximum number of characters that will fit on the line.

The window cursor is *not* advanced, and these routines work faster than *waddnstr()*. On the other hand, they don't perform any kind of checking (such as for the newline, backspace, or carriage return characters), they don't advance the current cursor position, they don't expand other control characters to ^-escapes, and they truncate the string if it crosses the right margin, rather than wrapping it around to the new line.

RETURN VALUES

All routines return the integer ERR upon failure and OK on success (the SVr4 manuals specify only "an integer value other than ERR") upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

All routines except *waddchnstr()* may be macros.

PORTABILITY

All these entry points are described in the XSI Curses standard, Issue 4.

SEE ALSO

curses

NAME

addstr(), addnstr(), waddstr(), waddnstr(), mvaddstr(), mvaddnstr(), mvwaddstr(), mvwaddnstr() – add a string of characters to a curses window and advance cursor

SYNOPSIS

```
#include <curses.h>

int addstr(const char *str);
int addnstr(const char *str, int n);
int waddstr(WINDOW *win, const char *str);
int waddnstr(WINDOW *win, const char *str, int n);
int mvaddstr(int y, int x, const char *str);
int mvaddnstr(int y, int x, const char *str, int n);
int mvwaddstr(WINDOW *win, int y, int x, const char *str);
int mvwaddnstr(WINDOW *win, int y, int x, const char *str, int n);
```

DESCRIPTION

These routines write the characters of the (null-terminated) character string *str* on the given window. It is similar to calling *waddch()* once for each character in the string. The four routines with *n* as the last argument write at most *n* characters. If *n* is *-1*, then the entire string will be added.

RETURN VALUE

All routines return the integer *ERR* upon failure and *OK* on success (the SVr4 manuals specify only "an integer value other than *ERR*") upon successful completion.

NOTES

All of these routines except *waddstr()* and *waddnstr()* may be macros.

PORTABILITY

All these entry points are described in the XSI Curses standard, Issue 4. The XSI errors *EILSEQ* and *E_OVERFLOW*, associated with extended-level conformance, are not yet detected.

SEE ALSO

curses, *curs_addch*

NAME

`attroff()`, `wattroff()`, `attron()`, `wattron()`, `attrset()`, `wattrset()`, `standend()`, `wstandend()`, `standout()`, `wstandout()` – curses character and window attribute control routines

SYNOPSIS

```
#include <curses.h>

int attroff(int attrs);
int wattroff(WINDOW *win, int attrs);
int attron(int attrs);
int wattron(WINDOW *win, int attrs);
int attrset(int attrs);
int wattrset(WINDOW *win, int attrs);
int standend(void);
int wstandend(WINDOW *win);
int standout(void);
int wstandout(WINDOW *win);

attr_t attr_get(void);
attr_t wattr_get(WINDOW *win);
int attr_off(attr_t attrs);
int wattr_off(WINDOW *win, attr_t attrs);
int attr_on(attr_t attrs);
int wattr_on(WINDOW *win, attr_t attrs);
int attr_set(attr_t attrs);
int wattr_set(WINDOW *win, attr_t attrs);

int chgat(int n, attr_t attr, short color, const void *opts)
int wchgat(WINDOW *win, int n, attr_t attr, short color,
            const void *opts)
int mvchgat(int y, int x, int n, attr_t attr, short color,
            const void *opts)
int mvwchgat(WINDOW *win, int y, int x, int n, attr_t attr,
             short color, const void *opts)
```

DESCRIPTION

These routines manipulate the current attributes of the named window. The current attributes of a window apply to all characters that are written into the window with `waddch()`, `waddstr()` and `wprintw()`. Attributes are a property of the character, and move with the character through any scrolling and insert / delete line / character operations. To the extent possible, they are displayed as appropriate modifications to the graphic rendition of characters put on the screen.

The routine `attrset()` sets the current attributes of the given window to *attrs*. The routine `attroff()` turns off the named attributes without turning any other attributes on or off. The routine `attron()` turns on the named attributes without affecting any others. The routine `standout()` is the same as `attron(A_STANDOUT)`. The routine `standend()` is the same as `attrset(A_NORMAL)` or `attrset(0)` that is, it turns off all attributes.

The routine `wattr_get()` returns the current attribute for the given window; `attr_get()` returns the current attribute for `stdscr`. The remaining `attr_*` functions operate exactly like the corresponding `attr*` functions, except that they take arguments of type **attr_t**

rather than **int**.

The routine *chgat()* changes the attributes of a given number of characters starting at the current cursor location of *stdscr*. It does not update the cursor and does not perform wrapping. A character count of -1 or greater than the remaining window width means to change attributes all the way to the end of the current line. The *wchgat()* function generalizes this to any window; the *mvwchgat()* function does a cursor move before acting. In these functions, the color argument is a color-pair index (as in the first argument of *init_pair()*, see *curs_color()*). The *opts* argument is not presently used, but is reserved for the future (leave it NULL).

Attributes

The following video attributes, defined in **<curses.h>**, can be passed to the routines *attron()*, *attroff()* and *attrset()* or OR-ed with the characters passed to *addch()*.

A_NORMAL	Normal display (no highlight)
A_STANDOUT	Best highlighting mode of the terminal.
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_PROTECT	Protected mode
A_INVIS	Invisible or blank mode
A_ALTCHARSET	Alternate character set
A_CHARTEXT	Bit-mask to extract a character
COLOR_PAIR(<i>n</i>)	Color-pair number <i>n</i>

The following macro is the reverse of *COLOR_PAIR(n)*:

PAIR_NUMBER(attrs)

Returns the pair number associated with the *COLOR_PAIR(n)* attribute.

The return values of many of these routines are not meaningful (they are implemented as macro-expanded assignments and simply return their argument). The SVr4 manual page claims (falsely) that these routines always return 1.

NOTES

attroff(), *wattroff()*, *attron()*, *wattron()*, *attrset()*, *wattrset()*, *standend()* and *standout()* may be macros.

PORTABILITY

All these functions are supported in the XSI Curses standard, Issue 4. The standard defined the dedicated type for highlights, **attr_t**, which is not defined in SVr4 curses. The functions taking **attr_t** arguments are not supported under SVr4.

The XSI Curses standard states that whether the traditional functions *attron()*, *attroff()*, *attrset()* can manipulate attributes other than A_BLINK, A_BOLD, A_DIM, A_REVERSE, A_STANDOUT, or A_UNDERLINE is "unspecified". Under this implementation as well as SVr4 curses, these functions correctly manipulate all other highlights (specifically, A_ALTCHARSET, A_PROTECT, and A_INVIS).

XSI Curses added the new entry points, *attr_get()*, *attr_on()*, *attr_off()*, *attr_set()*, *wattr_on()*, *wattr_off()*, *wattr_get()*, *wattr_set()*. These are intended to work with a new

series of highlight macros prefixed with “WA_”.

WA_NORMAL	Normal display (no highlight)
WA_STANDOUT	Best highlighting mode of the terminal.
WA_UNDERLINE	Underlining
WA_REVERSE	Reverse video
WA_BLINK	Blinking
WA_DIM	Half bright
WA_BOLD	Extra bright or bold
WA_ALTCHARSET	Alternate character set

The XSI curses standard specifies that each pair of corresponding “A_” and “WA_”-using functions operates on the same current-highlight information.

The XSI standard extended conformance level adds new highlights A_HORIZONTAL, A_LEFT, A_LOW, A_RIGHT, A_TOP, A_VERTICAL (and corresponding “WA_” macros for each) which this curses does not yet support.

SEE ALSO

curses, curs_addch, curs_addstr, curs_printw

NAME

beep(), flash() – curses bell and screen flash routines

SYNOPSIS

```
#include <curses.h>

int beep(void);
int flash(void);
```

DESCRIPTION

The *beep()* and *flash()* routines are used to alert the terminal user. The routine *beep()* sounds an audible alarm on the terminal, if possible; otherwise it flashes the screen (visible bell). The routine *flash()* flashes the screen, and if that is not possible, sounds the alert. If neither alert is possible, nothing happens. Nearly all terminals have an audible alert (bell or beep), but only some can flash the screen.

RETURN VALUE

These routines return OK if they succeed in beeping or flashing, ERR otherwise.

EXTENSIONS

SVr4's beep and flash routines always returned OK, so it was not possible to tell when the beep or flash failed.

PORTABILITY

These functions are defined in the XSI Curses standard, Issue 4. Like SVr4, it specifies that they always return OK.

SEE ALSO

curses

NAME

bkgdset(), wbkgdset(), bkgd(), wbkgd() – curses window background manipulation routines

SYNOPSIS

```
#include <curses.h>

void bkgdset(const chtype ch);
void wbkgdset(WINDOW *win, const chtype ch);
int bkgd(const chtype ch);
int wbkgd(WINDOW *win, const chtype ch);
chtype wgetbkgd(WINDOW *win);
```

DESCRIPTION

The *bkgdset()* and *wbkgdset()* routines manipulate the background of the named window. Background is a **chtype** consisting of any combination of attributes and a character. The attribute part of the background is combined (ORed) with all non-blank characters that are written into the window with *waddch()*. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert / delete line / character operations. To the extent possible on a particular terminal, the attribute part of the background is displayed as the graphic rendition of the character put on the screen.

The *bkgd()* and *wbkgd()* routines combine the new background with every position in the window. Background is any combination of attributes and a character. Only the attribute part is used to set the background of non-blank characters, while both character and attributes are used for blank positions. To the extent possible on a particular terminal, the attribute part of the background is displayed as the graphic rendition of the character put on the screen.

The *wgetbkgd()* function returns the given window's current background character / attribute pair.

RETURN VALUE

The routines *bkgd()* and *wbkgd()* return the integer OK. The SVr4.0 manual says "or a non-negative integer if *immedok()* is set", but this appears to be an error.

NOTES

Note that *bkgdset()* and *bkgd()* may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. The draft does not include **const** qualifiers on the arguments. The standard specifies that *bkgd()* and *wbkgd()* return ERR on failure, but gives no failure conditions. The draft describes the new function *wgetbkgd()*, which is not in SVr4.

SEE ALSO

curses, *curs_addch*, *curs_outopts*

NAME

`border()`, `wborder()`, `box()`, `hline()`, `whline()`, `vline()`, `wvline()` – create curses borders, horizontal and vertical lines

SYNOPSIS

```
#include <curses.h>

int border(chtype ls, chtype rs, chtype ts, chtype bs,
           chtype tl, chtype tr, chtype bl, chtype br);
int wborder(WINDOW *win, chtype ls, chtype rs, chtype ts, chtype bs,
           chtype tl, chtype tr, chtype bl, chtype br);
int box(WINDOW *win, chtype verch, chtype horch);
int hline(chtype ch, int n);
int whline(WINDOW *win, chtype ch, int n);
int vline(chtype ch, int n);
int wvline(WINDOW *win, chtype ch, int n);
```

DESCRIPTION

The `border()`, `wborder()` and `box()` routines draw a box around the edges of a window. The argument *ls* is a character and attributes used for the left side of the border, *rs* – right side, *ts* – top side, *bs* – bottom side, *tl* – top left-hand corner, *tr* – top right-hand corner, *bl* – bottom left-hand corner, and *br* – bottom right-hand corner. If any of these arguments is zero, then the following default values (defined in `<curses.h>`) are used instead: `ACS_VLINE`, `ACS_VLINE`, `ACS_HLINE`, `ACS_HLINE`, `ACS_ULCORNER`, `ACS_URCORNER`, `ACS_BLCORNER`, `ACS_BRCORNER`.

The call

```
box(win, verch, horch)
```

is a shorthand for the following call:

```
wborder(win, verch, verch, horch, horch, 0, 0, 0, 0)
```

The `hline()` and `whline()` functions draw a horizontal (left to right) line using *ch* starting at the current cursor position in the window. The current cursor position is not changed. The line is at most *n* characters long, or as many as fit into the window.

The `vline()` and `wvline()` functions draw a vertical (top to bottom) line using *ch* starting at the current cursor position in the window. The current cursor position is not changed. The line is at most *n* characters long, or as many as fit into the window.

RETURN VALUE

All routines return the integer OK. The SVr4.0 manual says "or a non-negative integer if `immedok()` is set", but this appears to be an error.

NOTES

The borders generated by these functions are *inside* borders (this is also true of SVr4 curses, though the fact is not documented).

`border()` and `box()` may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. Additional functions `mvhline()`, `mvvline()`, `mvwhline()`, and `mvwvline()` are described there which this implementation does not yet support. The standard specifies that they return ERR on

failure, but specifies no error conditions.

SEE ALSO

curses, *curs_outopts*

NAME

`erase()`, `werase()`, `clear()`, `wclear()`, `clrtobot()`, `wclrtobot()`, `clrtoeol()`, `wclrtoeol()` – clear all or part of a curses window

SYNOPSIS

```
#include <curses.h>

int erase(void);
int werase(WINDOW *win);
int clear(void);
int wclear(WINDOW *win);
int clrtobot(void);
int wclrtobot(WINDOW *win);
int clrtoeol(void);
int wclrtoeol(WINDOW *win);
```

DESCRIPTION

The `erase()` and `werase()` routines copy blanks to every position in the window, clearing the screen.

The `clear()` and `wclear()` routines are like `erase()` and `werase()`, but they also call `clearok()`, so that the screen is cleared completely on the next call to `wrefresh()` for that window and repainted from scratch.

The `clrtobot()` and `wclrtobot()` routines erase from the cursor to the end of screen. That is, they erase all lines below the cursor in the window. Also, the current line to the right of the cursor, inclusive, is erased.

The `clrtoeol()` and `wclrtoeol()` routines erase the current line to the right of the cursor, inclusive, to the end of the current line.

Blanks created by erasure have the current background rendition (as set by `wbkgdset()`) merged into them.

RETURN VALUE

All routines return the integer OK. The SVr4.0 manual says "or a non-negative integer if `immedok()` is set", but this appears to be an error.

NOTES

Note that `erase()`, `werase()`, `clear()`, `wclear()`, `clrtobot()`, and `clrtoeol()` may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. The standard specifies that they return ERR on failure, but specifies no error conditions.

Some historic curses implementations had, as an undocumented feature, the ability to do the equivalent of `clearok(..., 1)` by saying `touchwin(stdscr)` or `clear(stdscr)`. This will not work under ncurses.

SEE ALSO

curses, *curs_outopts* *curs_refresh*

NAME

`start_color()`, `init_pair()`, `init_color()`, `has_colors()`, `can_change_color()`, `color_content()`, `pair_content` – curses color manipulation routines

SYNOPSIS

```
#include <curses.h>

int start_color(void);
int init_pair(short pair, short f, short b);
int init_color(short color, short r, short g, short b);
bool has_colors(void);
bool can_change_color(void);
int color_content(short color, short *r, short *g, short *b);
int pair_content(short pair, short *f, short *b);
```

DESCRIPTION**Overview**

curses support color attributes on terminals with that capability. To use these routines `start_color()` must be called, usually right after `initscr()`. Colors are always used in pairs (referred to as color-pairs). A color-pair consists of a foreground color (for characters) and a background color (for the blank field on which the characters are displayed). A programmer initializes a color-pair with the routine `init_pair()`. After it has been initialized, `COLOR_PAIR(n)`, a macro defined in `<curses.h>`, can be used as a new video attribute.

If a terminal is capable of redefining colors, the programmer can use the routine `init_color()` to change the definition of a color. The routines `has_colors()` and `can_change_color()` return TRUE or FALSE, depending on whether the terminal has color capabilities and whether the programmer can change the colors. The routine `color_content()` allows a programmer to extract the amounts of red, green, and blue components in an initialized color. The routine `pair_content()` allows a programmer to find out how a given color-pair is currently defined.

Routine Descriptions

The `start_color()` routine requires no arguments. It must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after `initscr()`. `start_color()` initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables, `COLORS` and `COLOR_PAIRS` (respectively defining the maximum number of colors and color-pairs the terminal can support). It also restores the colors on the terminal to the values they had when the terminal was just turned on.

The `init_pair()` routine changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of the first argument must be between 1 and `COLOR_PAIRS-1`. The value of the second and third arguments must be between 0 and `COLORS` (the 0 color pair is wired to white on black and cannot be changed). If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair is changed to the new definition.

The *init_color()* routine changes the definition of a color. It takes four arguments: the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of the first argument must be between 0 and *COLORS*. (See the section **Colors** for the default color index.) Each of the last three arguments must be a value between 0 and 1000. When *init_color()* is used, all occurrences of that color on the screen immediately change to the new definition.

The *has_colors()* routine requires no arguments. It returns TRUE if the terminal can manipulate colors; otherwise, it returns FALSE. This routine facilitates writing terminal-independent programs. For example, a programmer can use it to decide whether to use color or some other video attribute.

The *can_change_color()* routine requires no arguments. It returns TRUE if the terminal supports colors and can change their definitions; other, it returns FALSE. This routine facilitates writing terminal-independent programs.

The *color_content()* routine gives programmers a way to find the intensity of the red, green, and blue (RGB) components in a color. It requires four arguments: the color number, and three addresses of **shorts** for storing the information about the amounts of red, green, and blue components in the given color. The value of the first argument must be between 0 and *COLORS*. The values that are stored at the addresses pointed to by the last three arguments are between 0 (no component) and 1000 (maximum amount of component).

The *pair_content()* routine allows programmers to find out what colors a given color-pair consists of. It requires three arguments: the color-pair number, and two addresses of **shorts** for storing the foreground and the background color numbers. The value of the first argument must be between 1 and *COLOR_PAIRS*-1. The values that are stored at the addresses pointed to by the second and third arguments are between 0 and *COLORS*.

Colors

In **<curses.h>** the following macros are defined. These are the default colors. **curses** also assumes that *COLOR_BLACK* is the default background color for all terminals.

```
COLOR_BLACK
COLOR_RED
COLOR_GREEN
COLOR_YELLOW
COLOR_BLUE
COLOR_MAGENTA
COLOR_CYAN
COLOR_WHITE
```

RETURN VALUE

The routines *can_change_color()* and *has_colors()* return TRUE or FALSE.

All other routines return the integer ERR upon failure and an OK (SVr4 specifies only "an integer value other than ERR") upon successful completion. Note that setting an implicit background color via a color pair affects only character cells that a character write operation explicitly touches. To change the background color used when parts of a window are blanked by erasing or scrolling operations, see *curs_bkgd*.

Several caveats apply on 386 and 486 machines with VGA-compatible graphics:

COLOR_YELLOW is actually brown. To get yellow, use COLOR_YELLOW combined with the A_BOLD attribute.

The A_BLINK attribute should in theory cause the background to go bright. This often fails to work, and even some cards for which it mostly works (such as the Paradise and compatibles) do the wrong thing when you try to set a bright "yellow" background (you get a blinking yellow foreground instead).

Color RGB values are not settable. This implementation satisfies XSI Curses's minimum maximums for *COLORS* and *COLOR_PAIRS*.

SEE ALSO

curses, *curs_attr*, *curs_initscr*

NAME

`delch()`, `wdelch()`, `mvdelch()`, `mvwdelch()` – delete character under the cursor in a curses window

SYNOPSIS

```
#include <curses.h>

int delch(void);
int wdelch(WINDOW *win);
int mvdelch(int y, int x);
int mvwdelch(WINDOW *win, int y, int x);
```

DESCRIPTION

These routines delete the character under the cursor; all characters to the right of the cursor on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to *y*, *x*, if specified). (This does not imply use of the hardware delete character feature.)

RETURN VALUE

All routines return the integer `ERR` upon failure and an OK (SVr4 specifies only "an integer value other than `ERR`") upon successful completion.

NOTES

Note that *delch()*, *mvdelch()*, and *mvwdelch()* may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. The standard specifies that they return `ERR` on failure, but specifies no error conditions.

SEE ALSO

curses

NAME

`deleteln()`, `wdeleteln()`, `insdelln()`, `winsdelln()`, `insertln()`, `wininsertln()` – delete and insert lines in a curses window

SYNOPSIS

```
#include <curses.h>

int deleteln(void);
int wdeleteln(WINDOW *win);
int insdelln(int n);
int winsdelln(WINDOW *win, int n);
int insertln(void);
int wininsertln(WINDOW *win);
```

DESCRIPTION

The `deleteln()` and `wdeleteln()` routines delete the line under the cursor in the window; all lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change.

The `insdelln()` and `winsdelln()` routines, for positive *n*, insert *n* lines into the specified window above the current line. The *n* bottom lines are lost. For negative *n*, delete *n* lines (starting with the one under the cursor), and move the remaining lines up. The bottom *n* lines are cleared. The current cursor position remains the same.

The `insertln()` and `wininsertln()` routines, insert a blank line above the current line and the bottom line is lost.

RETURN VALUE

All routines return the integer `ERR` upon failure and an `OK` (SVr4 specifies only "an integer value other than `ERR`") upon successful completion.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. The standard specifies that they return `ERR` on failure, but specifies no error conditions.

NOTES

Note that all but `winsdelln()` may be a macros.

These routines do not require a hardware line delete or insert feature in the terminal. In fact, they won't use hardware line delete/insert unless

```
idlok(..., TRUE)
```

has been set on the current window.

SEE ALSO

curses

NAME

`getch()`, `wgetch()`, `mvgetch()`, `mvwgetch()`, `ungetch()` – get (or push back) characters from curses terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getch(void);
int wgetch(WINDOW *win);
int mvgetch(int y, int x);
int mvwgetch(WINDOW *win, int y, int x);
int ungetch(int ch);
```

DESCRIPTION

The `getch()`, `wgetch()`, `mvgetch()` and `mvwgetch()`, routines read a character from the window. In no-delay mode, if no input is waiting, the value `ERR` is returned. In delay mode, the program waits until the system passes text through to the program. Depending on the setting of `cbreak()`, this is after one character (`cbreak` mode), or after the first newline (`nocbreak` mode). In half-delay mode, the program waits until a character is typed or the specified timeout has been reached.

If `noecho()` has been set, then the character will also be echoed into the designated window according to the following rules: If the character is the current erase character, left arrow, or backspace, the cursor is moved one space to the left and that screen position is erased as if `delch()` had been called. If the character value is any other `KEY_` define, the user is alerted with a `beep()` call. Otherwise the character is simply output to the screen.

If the window is not a pad, and it has been moved or modified since the last call to `wrefresh()`, `wrefresh()` will be called before another character is read.

If `keypad()` is `TRUE`, and a function key is pressed, the token for that function key is returned instead of the raw characters. Possible function keys are defined in **<curses.h>** as macros with values outside the range of 8-bit characters whose names begin with `KEY_`. Thus, a variable intended to hold the return value of a function key must be of **short** size or larger.

When a character that could be the beginning of a function key is received (which, on modern terminals, means an escape character), **curses** sets a timer. If the remainder of the sequence does not come in within the designated time, the character is passed through; otherwise, the function key value is returned. For this reason, many terminals experience a delay between the time a user presses the escape key and the escape is returned to the program.

The `ungetch()` routine places `ch` back onto the input queue to be returned by the next call to `wgetch()`. Note that there is, in effect, just one input queue for all windows.

Function Keys

The following function keys, defined in `< curses.h >`, might be returned by `getch()` if `keypad()` has been enabled. Not all of these are necessarily supported on any particular terminal.

Name	Key name
KEY_BREAK	Break key
KEY_DOWN	The four arrow keys ...
KEY_UP	
KEY_LEFT	
KEY_RIGHT	
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace
KEY_F0	Function keys; space for 64 keys is reserved.
KEY_F(n)	For $0 < n < 63$
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send
KEY_SRESET	Soft (partial) reset
KEY_RESET	Reset or hard reset
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom (lower left).
	Keypad is arranged like this:
	A1 up A3
	left B2 right
	C1 down C3
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab key
KEY_BEG	Beg(inning) key
KEY_CANCEL	Cancel key

Name	Key name
KEY_CLOSE	Close key
KEY_COMMAND	Cmd (command) key
KEY_COPY	Copy key
KEY_CREATE	Create key
KEY_END	End key
KEY_EXIT	Exit key
KEY_FIND	Find key
KEY_HELP	Help key
KEY_MARK	Mark key
KEY_MESSAGE	Message key
KEY_MOVE	Move key
KEY_NEXT	Next object key
KEY_OPEN	Open key
KEY_OPTIONS	Options key
KEY_PREVIOUS	Previous object key
KEY_REDO	Redo key
KEY_REFERENCE	Ref(erence) key
KEY_REFRESH	Refresh key
KEY_REPLACE	Replace key
KEY_RESTART	Restart key
KEY_RESUME	Resume key
KEY_SAVE	Save key
KEY_SBEG	Shifted beginning key
KEY_SCANCEL	Shifted cancel key
KEY_SCOMMAND	Shifted command key
KEY_SCOPY	Shifted copy key
KEY_SCREATE	Shifted create key
KEY_SDC	Shifted delete char key
KEY_SDL	Shifted delete line key
KEY_SELECT	Select key
KEY_SEND	Shifted end key
KEY_SEOL	Shifted clear line key
KEY_SEXIT	Shifted exit key
KEY_SFIND	Shifted find key
KEY_SHELP	Shifted help key
KEY_SHOME	Shifted home key
KEY_SIC	Shifted input key
KEY_SLEFT	Shifted left arrow key
KEY_SMESSAGE	Shifted message key
KEY_SMOVE	Shifted move key
KEY_SNEXT	Shifted next key
KEY_SOPTIONS	Shifted options key
KEY_SPREVIOUS	Shifted prev key
KEY_SPRINT	Shifted print key
KEY_SREDO	Shifted redo key
KEY_SREPLACE	Shifted replace key

Name	Key name
KEY_SRIGHT	Shifted right arrow
KEY_SRSUME	Shifted resume key
KEY_SSAVE	Shifted save key
KEY_SSUSPEND	Shifted suspend key
KEY_SUNDO	Shifted undo key
KEY_SUSPEND	Suspend key
KEY_UNDO	Undo key

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR" (OK in the case of *ungetch()*) upon successful completion.

NOTES

Use of the escape key by a programmer for a single character function is discouraged, as it will cause a delay of up to one second while the keypad code looks for a following function-key sequence.

When using *getch()*, *wgetch()*, *mvgetch()*, or *mvwgetch()*, *nocbreak* mode (*nocbreak()*) and *echo* mode (*echo()*) should not be used at the same time. Depending on the state of the tty driver when each character is typed, the program may produce undesirable results.

Note that *getch()*, *mvgetch()*, and *mvwgetch()* may be macros.

Historically, the set of keypad macros was largely defined by the extremely function-key-rich keyboard of the AT&T 7300, aka 3B1, aka Safari 4. Modern personal computers usually have only a small subset of these. IBM PC-style consoles typically support little more than KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT, KEY_HOME, KEY_END, KEY_NPAGE, KEY_PPAGE, and function keys 1 through 12. The Ins key is usually mapped to KEY_IC.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. They read single-byte characters only. The standard specifies that they return ERR on failure, but specifies no error conditions.

The echo behavior of these functions on input of KEY_ or backspace characters was not specified. This description is adopted from the XSI Curses standard.

SEE ALSO

curses, *curs_inopts*, *curs_move*, *curs_refresh*

NAME

`getstr()`, `wgetstr()`, `mvgetstr()`, `mvwgetstr()`, `wgetnstr()` – accept character strings from curses terminal keyboard

SYNOPSIS

```
#include <curses.h>

int getstr(char *str);
int wgetstr(WINDOW *win, char *str);
int mvgetstr(int y, int x, char *str);
int mvwgetstr(WINDOW *win, int y, int x, char *str);
int wgetnstr(WINDOW *win, char *str, int n);
```

DESCRIPTION

The function `getstr()` is equivalent to a series of calls to `getch()`, until a newline or carriage return is received (the terminating character is not included in the returned string). The resulting value is placed in the area pointed to by the character pointer `str`.

`wgetnstr()` reads at most `n` characters, thus preventing a possible overflow of the input buffer. Any attempt to enter more characters (other than the terminating newline or carriage return) causes a beep. Function keys also cause a beep and are ignored.

The user's erase and kill characters are interpreted. If keypad mode is on for the window, `KEY_LEFT` and `KEY_BACKSPACE` are both considered equivalent to the user's kill character.

Characters input are echoed only if `echo()` is currently on. In that case, backspace is echoed as deletion of the previous character (typically a left motion).

RETURN VALUE

All routines return the integer `ERR` upon failure and an OK (SVr4 specifies only "an integer value other than `ERR`") upon successful completion.

NOTES

Note that `getstr()`, `mvgetstr()`, and `mvwgetstr()` may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. They read single-byte characters only. The standard specifies that they return `ERR` on failure, but the single error condition `[Eoverflow]` associated with extended-level conformance is not yet returned (the XSI curses support for multi-byte characters is not yet present).

SVr3 and early SVr4 curses implementations did not reject function keys; the SVr4.0 documentation claimed that "special keys" (such as function keys, "home" key, "clear" key, etc.) are interpreted" without giving details. It lied. In fact, the 'character' value appended to the string by those implementations was predictable but not useful (being, in fact, the low-order eight bits of the key's `KEY_` value).

The functions `getnstr()`, `mvgetnstr()`, and `mvwgetnstr()` were present but not documented in SVr4.

SEE ALSO

`curses`, `curs_getch`

NAME

`getyx()`, `getparyx()`, `getbegyx()`, `getmaxyx()` – get curses cursor and window coordinates

SYNOPSIS

```
#include <curses.h>

void getyx(WINDOW *win, int y, int x);
void getparyx(WINDOW *win, int y, int x);
void getbegyx(WINDOW *win, int y, int x);
void getmaxyx(WINDOW *win, int y, int x);
```

DESCRIPTION

The `getyx()` macro places the current cursor position of the given window in the two integer variables `y` and `x`.

If `win` is a subwindow, the `getparyx()` macro places the beginning coordinates of the subwindow relative to the parent window into two integer variables `y` and `x`. Otherwise, -1 is placed into `y` and `x`.

Like `getyx()`, the `getbegyx()` and `getmaxyx()` macros store the current beginning coordinates and size of the specified window.

RETURN VALUE

The return values of these macros are undefined (i.e., they should not be used as the right-hand side of assignment statements).

NOTES

All of these interfaces are macros, so **&** is not necessary before the variables `y` and `x`.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4.

SEE ALSO

curses

NAME

inch(), winch(), mvinch(), mvwinch() – get a character and attributes from a curses window

SYNOPSIS

```
#include <curses.h>

chtype inch(void);
chtype winch(WINDOW *win);
chtype mvinch(int y, int x);
chtype mvwinch(WINDOW *win, int y, int x);
```

DESCRIPTION

These routines return the character, of type **chtype**, at the current position in the named window. If any attributes are set for that position, their values are OR-ed into the value returned. Constants defined in **<curses.h>** can be used with the **&** (logical AND) operator to extract the character or attributes alone.

Attributes

The following bit-masks may be AND-ed with characters returned by *winch()*.

A_CHARTEXT	Bit-mask to extract character
A_ATTRIBUTES	Bit-mask to extract attributes
A_COLOR	Bit-mask to extract color-pair field information

NOTES

Note that all of these routines may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4.

SEE ALSO

curses

NAME

inchstr(), inchnstr(), winchstr(), winchnstr(), mvinchstr(), mvwinchnstr(), mvwinchstr(), mvwinchnstr() – get a string of characters (and attributes) from a curses window

SYNOPSIS

```
#include <curses.h>

int inchstr(chtype *chstr);
int inchnstr(chtype *chstr, int n);
int winchstr(WINDOW *win, chtype *chstr);
int winchnstr(WINDOW *win, chtype *chstr, int n);
int mvinchstr(int y, int x, chtype *chstr);
int mvwinchnstr(int y, int x, chtype *chstr, int n);
int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);
int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
```

DESCRIPTION

These routines return a NULL-terminated array of **chtype** quantities, starting at the current cursor position in the named window and ending at the right margin of the window. The four functions with *n* as the last argument, return a leading substring at most *n* characters long (exclusive of the trailing (chtype)0). Constants defined in **<curses.h>** can be used with the **&** (logical AND) operator to extract the character or the attribute alone from any position in the *chstr* (see *curs_inch*).

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion (the number of characters retrieved, exclusive of the trailing 0).

NOTES

Note that all routines except *winchnstr()* may be macros. SVr4 does not document whether the result string is 0-terminated; it does not document whether a length limit argument includes any trailing 0; and it does not document the meaning of the return value.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. It is no more specific than the SVr4 documentation on the trailing 0. It does specify that the successful return of the functions is OK.

SEE ALSO

curses, *curs_inch*

NAME

`initscr()`, `newterm()`, `endwin()`, `isendwin()`, `set_term()`, `delscreen()` – curses screen initialization and manipulation routines

SYNOPSIS

```
#include <curses.h>

WINDOW *initscr(void);
int endwin(void);
int isendwin(void);
SCREEN *newterm(const char *type, FILE *outfd, FILE *infd);
SCREEN *set_term(SCREEN *new);
void delscreen(SCREEN *sp);
```

DESCRIPTION

`initscr()` is normally the first curses routine to call when initializing a program. A few special routines sometimes need to be called before it; these are `slk_init()`, `filter()`, `ripcoffline()`, and `use_env()`. For multiple-terminal applications, `newterm()` may be called before `initscr()`.

The `initscr` code determines the terminal type and initializes all curses data structures. `initscr` also causes the first call to `refresh()` to clear the screen. If errors occur, `initscr()` writes an appropriate error message to standard error and exits; otherwise, a pointer is returned to `stdscr`.

A program that outputs to more than one terminal should use the `newterm()` routine for each terminal instead of `initscr()`. A program that needs to inspect capabilities, so it can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use `newterm()`. The routine `newterm()` should be called once for each terminal. It returns a variable of type **SCREEN** * which should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the value of *TERM*, a file pointer for output to the terminal, and another file pointer for input from the terminal (if *type* is NULL, the value of *TERM* will be used). The program must also call `endwin()` for each terminal being used before exiting from curses. If `newterm()` is called more than once for the same terminal, the first terminal referred to must be the last one for which `endwin()` is called.

A program should always call `endwin()` before exiting or escaping from curses mode temporarily. This routine restores tty modes, moves the cursor to the lower left-hand corner of the screen and resets the terminal into the proper non-visual mode. Calling `refresh()` or `doupdate()` after a temporary escape causes the program to resume visual mode.

The `isendwin()` routine returns TRUE if `endwin()` has been called without any subsequent calls to `wrefresh()`, and FALSE otherwise.

The `set_term()` routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. The previous terminal is returned by the routine. This is the only routine which manipulates **SCREEN** pointers; all other routines affect only the current terminal.

The `delscreen()` routine frees storage associated with the **SCREEN** data structure. The `endwin()` routine does not do this, so `delscreen()` should be called after `endwin()` if a particular **SCREEN** is no longer needed.

RETURN VALUE

endwin() returns the integer ERR upon failure and OK upon successful completion.

Routines that return pointers always return NULL on error.

NOTES

Note that *initscr()* and *newterm()* may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. It specifies that portable applications must not call *initscr()* more than once.

SEE ALSO

curses, *curs_kernel*, *curs_refresh*, *curs_slk*, *curs_util*

NAME

`cbreak()`, `nocbreak()`, `echo()`, `noecho()`, `halfdelay()`, `intrflush()`, `keypad()`, `meta()`, `nodelay()`, `notimeout()`, `raw()`, `noraw()`, `noqiflush()`, `qiflush()`, `timeout()`, `wttimeout()`, `typeahead()` – curses input options

SYNOPSIS

```
#include <curses.h>

int cbreak(void);
int nocbreak(void);
int echo(void);
int noecho(void);
int halfdelay(int tenths);
int intrflush(WINDOW *win, bool bf);
int keypad(WINDOW *win, bool bf);
int meta(WINDOW *win, bool bf);
int nodelay(WINDOW *win, bool bf);
int raw(void);
int noraw(void);
void noqiflush(void);
void qiflush(void);
int notimeout(WINDOW *win, bool bf);
void timeout(int delay);
void wttimeout(WINDOW *win, int delay);
int typeahead(int fd);
```

DESCRIPTION

Normally, the tty driver buffers typed characters until a newline or carriage return is typed. The `cbreak()` routine disables line buffering and erase/kill character-processing (interrupt and flow control characters are unaffected), making characters typed by the user immediately available to the program. The `nocbreak()` routine returns the terminal to normal (cooked) mode.

Initially the terminal may or may not be in `cbreak()` mode, as the mode is inherited; therefore, a program should call `cbreak()` or `nocbreak()` explicitly. Most interactive programs using `curses()` set the `cbreak()` mode. Note that `cbreak()` overrides `raw()`. (See `curs_getch` for a discussion of how these routines interact with `echo()` and `noecho()`.)

The `echo()` and `noecho()` routines control whether characters typed by the user are echoed by `getch()` as they are typed. Echoing by the tty driver is always disabled, but initially `getch()` is in echo mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling `noecho()`. (See `curs_getch` for a discussion of how these routines interact with `cbreak()` and `nocbreak()`.)

The `halfdelay()` routine is used for half-delay mode, which is similar to `cbreak()` mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR is returned if nothing has been typed. The value of **tenths** must be a number between 1 and 255. Use `nocbreak()` to leave half-delay mode.

If the `intrflush()` option is enabled (*bf* is TRUE), when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving

the effect of faster response to the interrupt, but causing *curses()* to have the wrong idea of what is on the screen. Disabling the option (*bf* is FALSE) prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

The *keypad()* option enables the keypad of the user's terminal. If enabled (*bf* is TRUE), the user can press a function key (such as an arrow key) and *wgetch()* returns a single value representing the function key, as in KEY_LEFT. If disabled (*bf* is FALSE), *curses()* does not treat function keys specially and the program has to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option causes the terminal keypad to be turned on when *wgetch()* is called. The default value for *keypad()* is false.

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the tty driver (see *termio*). To force 8 bits to be returned, invoke *meta(win, TRUE)*; this is equivalent, under POSIX, to setting the CS8 flag on the terminal. To force 7 bits to be returned, invoke *meta(win, FALSE)*; this is equivalent, under POSIX, to setting the CS8 flag on the terminal. The window argument, *win*, is always ignored. If the terminfo capabilities **smm** (meta_on) and **rmm** (meta_off) are defined for the terminal, **smm** is sent to the terminal when *meta(win, TRUE)* is called and **rmm** is sent when *meta(win, FALSE)* is called.

The *nodelay()* option causes *getch()* to be a non-blocking call. If no input is ready, *getch()* returns ERR. If disabled (*bf* is FALSE), *getch()* waits until a key is pressed.

While interpreting an input escape sequence, *wgetch()* sets a timer while waiting for the next character. If *notimeout(win, TRUE)* is called, then *wgetch()* does not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

The *raw()* and *noraw()* routines place the terminal into or out of raw mode. Raw mode is similar to *cbreak()* mode, in that characters typed are immediately passed through to the user program. The differences are that in raw mode, the interrupt, quit, suspend, and flow control characters are all passed through uninterpreted, instead of generating a signal. The behavior of the BREAK key depends on other bits in the tty driver that are not set by **curses**.

When the *noqiflush()* routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done [see *termio*]. When *qiflush()* is called, the queues will be flushed when these control characters are read. You may want to call *noqiflush()* in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

The *timeout()* and *wtimeout()* routines set blocking or non-blocking read for a given window. If *delay* is negative, blocking read is used (i.e., waits indefinitely for input). If *delay* is zero, then non-blocking read is used (i.e., read returns ERR if no input is waiting). If *delay* is positive, then read blocks for *delay* milliseconds, and returns ERR if there is still no input. Hence, these routines provide the same functionality as *nodelay()*, plus the additional capability of being able to block for only *delay* milliseconds (where *delay* is positive).

The *curses()* library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until *refresh()* or *doupdate()* is called again. This allows faster response to commands typed in advance. Normally, the input FILE pointer

passed to *newterm()*, or *stdin* in the case that *initscr()* was used, will be used to do this typeahead checking. The *typeahead()* routine specifies that the file descriptor *fd* is to be used to check for typeahead instead. If *fd* is *-1*, then no typeahead checking is done.

RETURN VALUE

All routines that return an integer return *ERR* upon failure and *OK* (SVr4 specifies only "an integer value other than *ERR*") upon successful completion, unless otherwise noted in the preceding routine descriptions.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4.

The ncurses library obeys the XPG4 standard and the historical practice of the AT&T curses implementations, in that the echo bit is cleared when curses initializes the terminal state. BSD curses differed from this slightly; it left the echo bit on at initialization, but the BSD *raw()* call turned it off as a side-effect. For best portability, set *echo* or *noecho* explicitly just after initialization, even if your program remains in cooked mode.

NOTES

Note that *echo()*, *noecho()*, *halfdelay()*, *intrflush()*, *meta()*, *nodelay()*, *notimeout()*, *noqiflush()*, *qiflush()*, *timeout()*, and *wtimeout()* may be macros.

The *noraw()* and *nocbreak()* calls follow historical practice in that they attempt to restore to normal ('cooked') mode from raw and cbreak modes respectively. Mixing raw / noraw and cbreak / nocbreak calls leads to tty driver control states that are hard to predict or understand; it is not recommended.

SEE ALSO

curses, *curs_getch*, *curs_initscr*, *termio*

NAME

`insch()`, `winsch()`, `mvinsch()`, `mvwinsch()` – insert a character before cursor in a curses window

SYNOPSIS

```
#include <curses.h>

int insch(chtype ch);
int winsch(WINDOW *win, chtype ch);
int mvinsch(int y, int x, chtype ch);
int mvwinsch(WINDOW *win, int y, int x, chtype ch);
```

DESCRIPTION

These routines insert the character *ch* before the character under the cursor. All characters to the right of the cursor are moved one space to the right, with the possibility of the rightmost character on the line being lost. The insertion operation does not change the cursor position.

RETURN VALUE

All routines that return an integer return ERR upon failure and OK (SVr4 specifies only "an integer value other than ERR") upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

These routines do not necessarily imply use of a hardware insert character feature.

Note that `insch()`, `mvinsch()`, and `mvwinsch()` may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4.

SEE ALSO

curses

NAME

insstr(), insnstr(), winsstr(), winsnstr(), mvinsstr(), mvinsnstr(), mvwinsstr(), mvwinsnstr() – insert string before cursor in a curses window

SYNOPSIS

```
#include <curses.h>

int insstr(const char *str);
int insnstr(const char *str, int n);
int winsstr(WINDOW *win, const char *str);
int winsnstr(WINDOW *win, const char *str, int n);
int mvinsstr(int y, int x, const char *str);
int mvinsnstr(int y, int x, const char *str, int n);
int mvwinsstr(WINDOW *win, int y, int x, const char *str);
int mvwinsnstr(WINDOW *win, int y, int x, const char *str,
               int n);
```

DESCRIPTION

These routines insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the possibility of the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified). The four routines with *n* as the last argument insert a leading substring of at most *n* characters. If *n* ≤ 0, then the entire string is inserted.

If a character in *str* is a tab, newline, carriage return or backspace, the cursor is moved appropriately within the window. A newline also does a *clrtoeol()* before moving. Tabs are considered to be at every eighth column. If a character in *str* is another control character, it is drawn in the ^X notation. Calling *winch()* after adding a control character (and moving to it, if necessary) does not return the control character, but instead returns a character in the ^-representation of the control character.

RETURN VALUE

All routines that return an integer return ERR upon failure and OK (SVr4 specifies only "an integer value other than ERR") upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

All but *winsnstr()* may be macros.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4, which adds *const* qualifiers to the arguments. The XSI Curses error conditions [EILSEQ] and [EILOVERFLOW] associated with extended-level conformance are not yet detected (this implementation does not yet support XPG4 multibyte characters).

SEE ALSO

curses, *curs_clear*, *curs_inch*

NAME

`instr()`, `innstr()`, `winstr()`, `winnstr()`, `mvinstr()`, `mvinnstr()`, `mvwinstr()`, `mvwinnstr()` – get a string of characters from a curses window

SYNOPSIS

```
#include <curses.h>

int instr(char *str);
int innstr(char *str, int n);
int winstr(WINDOW *win, char *str);
int winnstr(WINDOW *win, char *str, int n);
int mvinstr(int y, int x, char *str);
int mvinnstr(int y, int x, char *str, int n);
int mvwinstr(WINDOW *win, int y, int x, char *str);
int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
```

DESCRIPTION

These routines return a string of characters in *str*, extracted starting at the current cursor position in the named window and ending at the right margin of the window. Attributes are stripped from the characters. The four functions with *n* as the last argument return a leading substring at most *n*() characters long (exclusive of the trailing NUL).

RETURN VALUE

All routines that return an integer return ERR upon failure and OK (SVr4 specifies only "an integer value other than ERR") upon successful completion, unless otherwise noted in the preceding routine descriptions.

NOTES

Note that all routines except `winnstr()` may be macros. SVr4 does not document whether a length limit includes or excludes the trailing NUL, and does not document the meaning of the return values.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. The standard specifies that they always return OK on success. The XSI Curses error conditions [EILSEQ] and [EILOVERFLOW] associated with extended-level conformance are not yet detected (this implementation does not yet support XPG4 multibyte characters).

SEE ALSO

curses

NAME

`def_prog_mode()`, `def_shell_mode()`, `reset_prog_mode()`, `reset_shell_mode()`, `resetty()`, `savetty()`, `getsyx()`, `setsyx()`, `ripline()`, `curs_set()`, `napms()` – low-level xscurses routines

SYNOPSIS

```
#include <curses.h>

int def_prog_mode(void);
int def_shell_mode(void);
int reset_prog_mode(void);
int reset_shell_mode(void);
int resetty(void);
int savetty(void);
void getsyx(int y, int x);
void setsyx(int y, int x);
int ripline(int line, int (*init)(WINDOW *, int));
int curs_set(int visibility);
int napms(unsigned int ms);
```

DESCRIPTION

The following routines give low-level access to various **curses** capabilities. These routines typically are used inside library routines.

The `def_prog_mode()` and `def_shell_mode()` routines save the current terminal modes as the “program” (in **curses**) or “shell” (not in **curses**) state for use by the `reset_prog_mode()` and `reset_shell_mode()` routines. This is done automatically by `initscr()`. There is one such save area for each screen context allocated by `newterm()`.

The `reset_prog_mode()` and `reset_shell_mode()` routines restore the terminal to “program” or “shell” state. These are done automatically by `endwin()` and, after an `endwin()`, by `doupdate()`, so they normally are not called.

The `resetty()` and `savetty()` routines save and restore the state of the terminal modes. `savetty()` saves the current state in a buffer and `resetty()` restores the state to what it was at the last call to `savetty()`.

The `getsyx()` routine returns the current coordinates of the virtual screen cursor in `y` and `x`. If `leaveok()` is currently TRUE, then `-1,-1` is returned. If lines have been removed from the top of the screen, using `ripline()`, `y` and `x` include these lines; therefore, `y` and `x` should be used only as arguments for `setsyx()`.

The `setsyx()` routine sets the virtual screen cursor to `y`. If `y` and `x` are both `-1`, then `leaveok()` is set. The two routines `getsyx()` and `setsyx()` are designed to be used by a library routine, which manipulates curses windows but does not want to change the current position of the program’s cursor. The library routine would call `getsyx()` at the beginning, do its manipulation of its own windows, do a `wnoutrefresh()` on its windows, call `setsyx()`, and then call `doupdate()`.

The `ripline()` routine provides access to the same facility that `slk_init()` (see `curs_slk`) uses to reduce the size of the screen. `ripline()` must be called before `initscr()` or `newterm()` is called. If `line` is positive, a line is removed from the top of `stdscr`; if `line` is negative, a line is removed from the bottom. When this is done inside `initscr()`, the routine `init()` (supplied by the user) is called with two arguments: a window pointer to

the one-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables *LINES* and *COLS* (defined in `<curses.h>`) are not guaranteed to be accurate and *wrefresh()* or *doupdate()* must not be called. It is allowable to call *wnoutrefresh()* during the initialization routine.

ripoffline() can be called up to five times before calling *initscr()* or *newterm()*.

The *curs_set()* routine sets the cursor state is set to invisible, normal, or very visible for *visibility()* equal to 0, 1, or 2 respectively. If the terminal supports the *visibility* requested, the previous *cursor* state is returned; otherwise, ERR is returned.

The *napms()* routine is used to sleep for *ms* milliseconds.

RETURN VALUE

Except for *curs_set()*, these routines always return OK. *curs_set()* returns the previous cursor state, or ERR if the requested *visibility* is not supported.

NOTES

getsyx() is a macro, so **&** is not necessary before the variables *y* and *x*.

The SVr4 man pages warn that the return value of *curs_set()* "is currently incorrect". This implementation gets it right, but it may be unwise to count on the correctness of the return value anywhere else.

PORTABILITY

The functions *setsyx()* and *getsyx()* are not described in the XSI Curses standard, Issue 4. All other functions are as described in XSI Curses.

The SVr4 documentation describes *setsyx()* and *getsyx()* as having return type `int`. This is misleading, as they are macros with no documented semantics for the return value.

SEE ALSO

curses, *curs_initscr*, *curs_outopts*, *curs_refresh*, *curs_scr_dump*, *curs_slk*

NAME

getmouse(), ungetmouse(), mousemask() – mouse interface through curses

SYNOPSIS

```
#include <curses.h>

typedef unsigned long mmask_t;

typedef struct
{
    short id;           /* ID to distinguish multiple devices */
    int x, y, z;        /* event coordinates */
    mmask_t bstate;     /* button state bits */
}
MEVENT;

int getmouse(MEVENT *event);
int ungetmouse(MEVENT *event);
mmask_t mousemask(mmask_t newmask, mmask_t *oldmask);
bool wenclose(WINDOW *win, int y, int x)
int mouseinterval(int erval)
```

DESCRIPTION

These functions provide an interface to mouse events from **ncurses**. Mouse events are represented by KEY_MOUSE pseudo-key values in the *wgetch()* input stream.

To make mouse events visible, use the *mousemask()* function. This will set the mouse events to be reported. By default, no mouse events are reported. The function will return a mask to indicate which of the specified mouse events can be reported; on complete failure it returns 0. If oldmask is non-NULL, this function fills the indicated location with the previous value of the given window's mouse event mask.

As a side effect, setting a zero mousemask may turn off the mouse pointer; setting a nonzero mask may turn it on. Whether this happens is device-dependent.

Here are the mouse event type masks:

<i>Name</i>	<i>Description</i>
BUTTON1_PRESSED	mouse button 1 down
BUTTON1_RELEASED	mouse button 1 up
BUTTON1_CLICKED	mouse button 1 clicked
BUTTON1_DOUBLE_CLICKED	mouse button 1 double clicked
BUTTON1_TRIPLE_CLICKED	mouse button 1 triple clicked
BUTTON2_PRESSED	mouse button 2 down
BUTTON2_RELEASED	mouse button 2 up
BUTTON2_CLICKED	mouse button 2 clicked
BUTTON2_DOUBLE_CLICKED	mouse button 2 double clicked
BUTTON2_TRIPLE_CLICKED	mouse button 2 triple clicked
BUTTON3_PRESSED	mouse button 3 down
BUTTON3_RELEASED	mouse button 3 up

<i>Name</i>	<i>Description</i>
BUTTON3_CLICKED	mouse button 3 clicked
BUTTON3_DOUBLE_CLICKED	mouse button 3 double clicked
BUTTON3_TRIPLE_CLICKED	mouse button 3 triple clicked
BUTTON4_PRESSED	mouse button 4 down
BUTTON4_RELEASED	mouse button 4 up
BUTTON4_CLICKED	mouse button 4 clicked
BUTTON4_DOUBLE_CLICKED	mouse button 4 double clicked
BUTTON4_TRIPLE_CLICKED	mouse button 4 triple clicked
BUTTON_SHIFT	shift was down during button state change
BUTTON_CTRL	control was down during button state change
BUTTON_ALT	alt was down during button state change
ALL_MOUSE_EVENTS	report all button state changes
REPORT_MOUSE_POSITION	report mouse movement

Once a class of mouse events have been made visible in a window, calling the *wgetch()* function on that window may return *KEY_MOUSE* as an indicator that a mouse event has been queued. To read the event data and pop the event off the queue, call *getmouse()*. This function will return OK if a mouse event is actually visible in the given window, ERR otherwise. When *getmouse()* returns OK, the data deposited as y and x in the event structure coordinates will be screen-relative character-cell coordinates. The returned state mask will have exactly one bit set to indicate the event type.

The *ungetmouse()* function behaves analogously to *ungetch()*. It pushes a *KEY_MOUSE()* event onto the input queue, and associates with that event the given state data and screen-relative character-cell coordinates.

The *wenclose()* function tests whether a given pair of screen-relative character-cell coordinates is enclosed by a given window, returning TRUE if it is and FALSE otherwise. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

The *mouseinterval()* function sets the maximum time (in thousands of a second) that can elapse between press and release events in order for them to be recognized as a click. This function returns the previous interval value. The default is one fifth of a second.

Note that mouse events will be ignored when input is in cooked mode, and will cause an error beep when cooked mode is being simulated in a window by a function such as *getstr()* that expects a linefeed for input-loop termination.

RETURN VALUE

All routines return the integer ERR upon failure or OK upon successful completion.

PORTABILITY

These calls were designed for **ncurses**, and are not found in SVr4 curses, 4.4BSD curses, or any other previous version of curses.

The feature macro *NCURSES_MOUSE_VERSION* is provided so the preprocessor can be used to test whether these features are present (its value is 1). NOTE: THIS INTERFACE IS EXPERIMENTAL AND IS SUBJECT TO CHANGE WITHOUT NOTICE! If the interface is changed, the value of *NCURSES_MOUSE_VERSION()* will be incremented.

These functions are not provided under INTERIX. *NCURSES_MOUSE_VERSION* is set to 0. This reference page is provided for information value only.

The order of the *MEVENT()* structure members is not guaranteed. Additional fields may be added to the structure in the future.

Under **ncurses**, these calls are implemented using either *xterm*'s built-in mouse-tracking API or Alessandro Rubini's *gpm* server. If you are using something other than *xterm* there is no *gpm* daemon running on your machine, mouse events will not be visible to **ncurses** (and the *wmousemask()* function will always return 0).

The **z** member in the event structure is not presently used. It is intended for use with touchscreens (which may be pressure-sensitive) or with 3D-mice / trackballs / power gloves.

BUGS

Mouse events under *xterm* will not in fact be ignored during cooked mode, if they have been enabled by *wmousemask()*. Instead, the *xterm* mouse report sequence will appear in the string read.

Mouse events under *xterm* will not be detected correctly in a window with its keypad bit off.

SEE ALSO

curses

NAME

`move()`, `wmove()` – move curses window cursor

SYNOPSIS

```
#include <curses.h>

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

DESCRIPTION

These routines move the cursor associated with the window to line *y* and column *x*. This routine does not move the physical cursor of the terminal until *refresh()* is called. The position specified is relative to the upper left-hand corner of the window, which is (0,0).

RETURN VALUE

These routines return ERR upon failure and OK (SVr4 specifies only "an integer value other than ERR") upon successful completion.

NOTES

Note that *move()* may be a macro.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4. The standard specifies that if (*y*, *x*) is within a multi-column character, the cursor is moved to the first column of that character; however, this implementation does not yet support the extended-level XSI multi-byte characters.

SEE ALSO

curses, *curs_refresh*

NAME

clearok(), idlok(), idcok(), immedok(), leaveok(), setscrreg(), wsetscrreg(), scrolllok(), nl(), nonl() – curses output options

SYNOPSIS

```
#include <curses.h>

int clearok(WINDOW *win, bool bf);
int idlok(WINDOW *win, bool bf);
void idcok(WINDOW *win, bool bf);
void immedok(WINDOW *win, bool bf);
int leaveok(WINDOW *win, bool bf);
int setscrreg(int top, int bot);
int wsetscrreg(WINDOW *win, int top, int bot);
int scrolllok(WINDOW *win, bool bf);
int nl(void);
int nonl(void);
```

DESCRIPTION

These routines set options that change the style of output within **curses**. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling *endwin()*.

If *clearok()* is called with TRUE as argument, the next call to *wrefresh()* with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect. If the *win* argument to *clearok()* is the global variable *curscr()*, the next call to *wrefresh()* with any window causes the screen to be cleared and repainted from scratch.

If *idlok()* is called with TRUE as argument, **curses** considers using the hardware insert / delete line feature of terminals so equipped. If disabled (*bf* is FALSE), **curses** very seldom uses this feature. (The insert / delete character feature is always considered.) This option should be enabled only if the application needs insert / delete line, for example, for a screen editor. It is disabled by default because insert / delete line tends to be visually annoying when used in applications where it isn't really needed. If insert / delete line cannot be used, *curses()* redraws the changed portions of all lines.

If *idcok()* is called with TRUE as argument, **curses** considers using the hardware insert / delete character feature of terminals so equipped. This is enabled by default.

If *immedok()* is called with TRUE as argument, any change in the window image, such as the ones caused by *waddch()*, *wclrtoeol()*, *wscrl()*, etc., automatically cause a call to *wrefresh()*. However, it may degrade the performance considerably, due to repeated calls to *wrefresh()*. It is disabled by default.

Normally, the hardware cursor is left at the location of the window cursor being refreshed. The *leaveok()* option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

The *setscrreg()* and *wsetscrreg()* routines allow the application programmer to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and

bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and *scrollok()* are enabled, an attempt to move off the bottom margin line causes all lines in the scrolling region to scroll one line in the direction of the first line. Only the text of the window is scrolled. (Note that this has nothing to do with the use of a physical scrolling region capability in the terminal, like that in the VT100. If *idlok()* is enabled and the terminal has either a scrolling region or insert / delete line capability, they will probably be used by the output routines.)

The *scrollok()* option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If disabled (*bf* is FALSE), the cursor is left on the bottom line. If enabled (*bf* is TRUE), the window is scrolled up one line (Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call *idlok()*).

The *nl()* and *nonl()* routines control whether the underlying display device translates the return key into newline on input, and whether it translates newline into return and line-feed on output (in either case, the call `addch( '\n' )` does the equivalent of return and line feed on the virtual screen). Initially, these translations do occur. If you disable them using *nonl()*, **curses** will be able to make better use of the line-feed capability, resulting in faster cursor motion. Also, **curses** will then be able to detect the return key.

RETURN VALUE

The functions *setscrreg()* and *wsetscrreg()* return OK upon success and ERR upon failure. All other routines that return an integer always return OK.

PORTABILITY

These functions are described in the XSI Curses standard, Issue 4.

The XSI Curses standard is ambiguous on the question of whether *raw()* should disable the CRLF translations controlled by *nl()* and *nonl()*. BSD curses did turn off these translations; AT&T curses (at least as late as SVr1) did not. We choose to do so, on the theory that a programmer requesting raw input wants a clean (ideally 8-bit clean) connection that the operating system does not mess with.

Some historic curses implementations had, as an undocumented feature, the ability to do the equivalent of `clearok( ..., 1)` by saying `touchwin(stdscr)` or `clear(stdscr)`. This will not work under ncurses.

Earlier System V curses implementations specified that with *scrollok()* enabled, any window modification triggering a scroll also forced a physical refresh. XSI Curses does not require this, and *ncurses()* avoids doing it in order to perform better vertical-motion optimization at *wrefresh()* time.

NOTES

Note that *clearok()*, *leaveok()*, *scrollok()*, *idcok()*, *nl()*, *nonl()* and *setscrreg()* may be macros.

The *immedok()* routine is useful for windows that are used as terminal emulators.

SEE ALSO

curses, *curs_addch*, *curs_clear*, *curs_initscr*, *curs_scroll*, *curs_refresh*

NAME

overlay(), overwrite(), copywin() – overlay and manipulate overlapped curses windows

SYNOPSIS

```
#include <curses.h>

int overlay(const WINDOW *srcwin, WINDOW *dstwin);
int overwrite(const WINDOW *srcwin, WINDOW *dstwin);
int copywin(WINDOW *srcwin, WINDOW *dstwin, int sminrow,
            int smincol, int dminrow, int dmincol, int dmaxrow,
            int dmaxcol, int overlay);
```

DESCRIPTION

The *overlay()* and *overwrite()* routines overlay *srcwin* on top of *dstwin*. *srcwin* and *dstwin* are not to be the same size; only text where the two windows overlap is copied. The difference is that *overlay()* is non-destructive (blanks are not copied) whereas *overwrite()* is destructive.

The *copywin()* routine provides a finer granularity of control over the **overlay** and *overwrite()* routines. Like in the *prefresh()* routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is TRUE, then copying is non-destructive, as in *overlay()*.

RETURN VALUE

Routines that return an integer return ERR upon failure, and OK (SVr4 only specifies "an integer value other than ERR") upon successful completion.

NOTES

overlay() and *overwrite()* may be macros.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions (adding the *const* qualifiers). It further specifies their behavior in the presence of characters with multi-byte renditions (not yet supported in this implementation).

SEE ALSO

curses, *curs_pad*, *curs_refresh*

NAME

`newpad()`, `subpad()`, `prefresh()`, `pnoutrefresh()`, `pechochar()` – create and display curses pads

SYNOPSIS

```
#include <curses.h>

WINDOW *newpad(int nlines, int ncols);
WINDOW *subpad(WINDOW *orig, int nlines, int ncols,
               int begin_y, int begin_x);
int prefresh(WINDOW *pad, int pminrow, int pmincol,
             int sminrow, int smincol, int smaxrow, int smaxcol);
int pnoutrefresh(WINDOW *pad, int pminrow, int pmincol,
                int sminrow, int smincol, int smaxrow, int smaxcol);
int pechochar(WINDOW *pad, chtype ch);
```

DESCRIPTION

The `newpad()` routine creates and returns a pointer to a new pad data structure with the given number of lines, *nlines*, and columns, *ncols*. A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g., from scrolling or echoing of input) do not occur. It is not legal to call `wrefresh()` with a *pad* as an argument; the routines `prefresh()` or `pnoutrefresh()` should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for the display.

The `subpad()` routine creates and returns a pointer to a subwindow within a pad with the given number of lines, *nlines*, and columns, *ncols*. Unlike `subwin()`, which uses screen coordinates, the window is at position (*begin_x*, *begin_y*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window affect both windows. During the use of this routine, it will often be necessary to call `touchwin()` or `touchline()` on *orig* before calling `prefresh()`.

The `prefresh()` and `pnoutrefresh()` routines are analogous to `wrefresh()` and `wnoutrefresh()` except that they relate to pads instead of windows. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

The `pechochar` routine is functionally equivalent to a call to `addch()` followed by a call to `refresh()`, a call to `waddch()` followed by a call to `wrefresh()`, or a call to `waddch()` followed by a call to `prefresh()`. The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain might be seen by using these routines instead of their equivalents. In the case of `pechochar()`, the last location of the pad on the screen is reused for the arguments to `prefresh()`.

RETURN VALUE

Routines that return an integer return ERR upon failure and OK (SVr4 only specifies "an integer value other than ERR") upon successful completion.

Routines that return pointers return NULL on error, and set *errno* to [ENOMEM].

NOTES

Note that *pechochar()* may be a macro.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions.

SEE ALSO

curses, *curs_refresh()*, *curs_touch()*, *curs_addch()*

NAME

`printw()`, `wprintw()`, `mvprintw()`, `mvwprintw()`, `vwprintw()` – print formatted output in curses windows

SYNOPSIS

```
#include <curses.h>

int printw(char *fmt [, arg] ...);
int wprintw(WINDOW *win, char *fmt [, arg] ...);
int mvprintw(int y, int x, char *fmt [, arg] ...);
int mvwprintw(WINDOW *win, int y, int x,
               char *fmt [, arg] ...);

#include <stdarg.h>
int vwprintw(WINDOW *win, char *fmt, varglist);
int vw_printw(WINDOW *win, char *fmt, varglist);
```

DESCRIPTION

The `printw()`, `wprintw()`, `mvprintw()` and `mvwprintw()` routines are analogous to `printf()`. In effect, the string that would be output by `printf()` is output instead as though `waddstr()` were used on the given window.

The `vwprintw()` routine is analogous to `vprintf()` and performs a `wprintw()` using a variable argument list. The third argument is a **va_list**, a pointer to a list of arguments, as defined in **<stdarg.h>**. The `vw_printw()` routine is a **#define** for `vwprintw()`. `vw_printw()` is planned for the XSI Curses standard. Traditionally, `vwprintw()` made use of **<varargs.h>** and `vw_printw()` made use of **<stdarg.h>**, but both of the routines in this package use **<stdarg.h>**.

RETURN VALUE

Routines that return an integer return ERR upon failure and OK (SVr4 only specifies “an integer value other than ERR”) upon successful completion.

SEE ALSO

`curses`, `printf()`, `vprintf()`

NAME

`refresh()`, `wrefresh()`, `wnoutrefresh()`, `doupdate()`, `redrawwin()`, `wredrawln()` – refresh curses windows and lines

SYNOPSIS

```
#include <curses.h>

int refresh(void);
int wrefresh(WINDOW *win);
int wnoutrefresh(WINDOW *win);
int doupdate(void);
int redrawwin(WINDOW *win);
int wredrawln(WINDOW *win, int beg_line, int num_lines);
```

DESCRIPTION

The `refresh()` and `wrefresh()` routines (or `wnoutrefresh()` and `doupdate()`) must be called to get actual output to the terminal, as other routines merely manipulate data structures. The routine `wrefresh()` copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. The `refresh()` routine is the same, using `stdscr` as the default window. Unless `leaveok()` has been enabled, the physical cursor of the terminal is left at the location of the cursor for that window.

The `wnoutrefresh()` and `doupdate()` routines allow multiple updates with more efficiency than `wrefresh()` alone. In addition to all the window structures, **curses** keeps two data structures representing the terminal screen: a physical screen, describing what is actually on the screen, and a virtual screen, describing what the programmer wants to have on the screen.

The routine `wrefresh()` works by first calling `wnoutrefresh()`, which copies the named window to the virtual screen, and then calling **doupdate**, which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to `wrefresh()` results in alternating calls to `wnoutrefresh()` and `doupdate()`, causing several bursts of output to the screen. By first calling `wnoutrefresh()` for each window, it is then possible to call `doupdate()` once, resulting in only one burst of output, with fewer total characters transmitted and less CPU time used. If the `win` argument to `wrefresh()` is the global variable `curscr`, the screen is immediately cleared and repainted from scratch.

The phrase "copies the named window to the virtual screen" above is ambiguous. What actually happens is that all **touched** (changed) lines in the window are copied to the virtual screen. This affects programs that use overlapping windows; it means that if two windows overlap, you can refresh them in either order and the overlap region will be modified only when it is explicitly changed. (But see the section on **PORTABILITY** below for a warning about exploiting this behavior.)

The `wredrawln()` routine indicates to **curses** that some screen lines are corrupted and should be thrown away before anything is written over them it touches the indicated lines (marking them changed) then does a refresh of the window. The routine `redrawwin()` touches the entire window and then refreshes it.

RETURN VALUE

Routines that return an integer return `ERR` upon failure, and `OK` (SVr4 only specifies "an integer value other than `ERR`") upon successful completion.

NOTES

Note that *refresh()* and *redrawwin()* may be macros.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions.

Whether *wnoutrefresh()* copies to the virtual screen the entire contents of a window or just its changed portions has never been well-documented in historic curses versions (including SVr4). It might be unwise to rely on either behavior in programs that might have to be linked with other curses implementations. Instead, you can do an explicit *touchwin()* before the *wnoutrefresh()* call to guarantee an entire-contents copy anywhere.

SEE ALSO

curses, *curs_outopts*

NAME

wresize() – resize a curses window

SYNOPSIS

```
#include <curses.h>

int wresize(WINDOW *win, int lines, int columns);
```

DESCRIPTION

The *wresize()* function adjusts the dimensions of an **ncurses** window to the specified values, reallocating storage as necessary.

If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by *wbkgndset()*) merged into them. If the current cursor position is outside the retained area of the window, the cursor is moved to within the nearest window edge. Window data is otherwise preserved.

This function will fail if either of the new dimensions less than or equal to zero, or if memory cannot be allocated or reallocated for the window.

Note that *wresize()* performs non-reentrant *malloc()* operations, so it isn't advisable to resize from within an interrupt handler. Instead, have your handler set an event flag to be interpreted next time around your program's command loop.

PORTABILITY

This call was designed for **ncurses**, and is not found in SVr4 curses, 4.4BSD curses, or any other previous version of curses. It is recommended that you conditionalize all code depending on it with the symbol NCURSES.

RETURN VALUE

The function returns the integer ERR upon failure and OK on success.

NAME

`scanw()`, `wscanw()`, `mvscanw()`, `mvwscanw()`, `vwscanw()` – convert formatted input from a curses window

SYNOPSIS

```
#include <curses.h>

int scanw(char *fmt [, arg] ...);
int wscanw(WINDOW *win, char *fmt [, arg] ...);
int mvscanw(int y, int x, char *fmt [, arg] ...);
int mvwscanw(WINDOW *win, int y, int x, char *fmt [, arg] ...);

#include <stdarg.h>

int vwscanw(WINDOW *win, char *fmt, va_list varglist);
int vw_scanw(WINDOW *win, char *fmt, va_list varglist);
```

DESCRIPTION

The `scanw()`, `wscanw()` and `mvscanw()` routines are analogous to `scanf()` (see `scanf()`). The effect of these routines is as though `wgetstr()` were called on the window, and the resulting line used as input for `sscanf()`. Fields which do not map to a variable in the `fmt` field are lost.

The `vwscanw()` routine is similar to `vwprintw()` in that it performs a `wscanw()` using a variable argument list. The third argument is a **va_list**, a pointer to a list of arguments, as defined in `<stdarg.h>`. The `vw_scanw()` routine is a **#define** for `vwscanw()`. `vw_scanw()` is planned for the XSI Curses standard. Traditionally, `vwscanw()` made use of `<varargs.h>` and `vw_scanw()` made use of `<stdarg.h>`, but both of the routines in this package use `<stdarg.h>`.

RETURN VALUE

`vwscanw()` returns `ERR` on failure and an integer equal to the number of fields scanned on success.

Applications may use the return value from the `scanw()`, `wscanw()`, `mvscanw()` and `mvwscanw()` routines to determine the number of fields which were mapped in the call.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions. The function `vwscanw()` is marked TO BE WITHDRAWN, and is to be replaced by a function `vw_scanw()` using the `<stdarg.h>` interface.

SEE ALSO

`curses`, `curs_getstr`, `curs_printw`, `scanf()`

NAME

`scr_dump()`, `scr_restore()`, `scr_init()`, `scr_set()` – read or write a curses screen from or to a file

SYNOPSIS

```
#include <curses.h>

int scr_dump(const char *filename);
int scr_restore(const char *filename);
int scr_init(const char *filename);
int scr_set(const char *filename);
```

DESCRIPTION

The `scr_dump()` routine dumps the current contents of the virtual screen to the file *filename*.

The `scr_restore()` routine sets the virtual screen to the contents of *filename*, which must have been written using `scr_dump()`. The next call to `doupdate()` restores the screen to the way it looked in the dump file.

The `scr_init()` routine reads in the contents of *filename* and uses them to initialize the **curses** data structures about what the terminal currently has on its screen. If the data is determined to be valid, **curses** bases its next update of the screen on this information rather than clearing the screen and starting from scratch. `scr_init()` is used after `initscr()` or a `system()` call to share the screen with another process which has done a `scr_dump()` after its `endwin()` call. The data is declared invalid if the terminfo capabilities **rmcup** and **nrnmc** exist; also if the terminal has been written to since the preceding `scr_dump()` call.

The `scr_set()` routine is a combination of `scr_restore()` and `scr_init()`. It tells the program that the information in *filename* is what is currently on the screen, and also what the program wants on the screen. This can be thought of as a screen inheritance function.

To read (write) a window from (to) a file, use the `getwin()` and `putwin()` routines (see the *curs_util* reference page).

RETURN VALUE

All routines return the integer ERR upon failure and OK upon success.

NOTES

`Scr_init()`, `scr_set()`, and `scr_restore()` may be macros.

PORTABILITY

The XSI Curses standard, Issue 4, describes these functions (adding the **const** qualifiers).

The SVr4 docs merely say under `scr_init()` that the dump data is also considered invalid "if the time-stamp of the tty is old" but don't define "old".

SEE ALSO

curses, *curs_initscr*, *curs_refresh*, *curs_util*, *system()*

NAME

`scroll()`, `srl()`, `wscrl()` – scroll a curses window

SYNOPSIS

```
#include <curses.h>

int scroll(WINDOW *win);
int srl(int n);
int wscrl(WINDOW *win, int n);
```

DESCRIPTION

The `scroll()` routine scrolls the window up one line. This involves moving the lines in the window data structure. As an optimization, if the scrolling region of the window is the entire screen, the physical screen may be scrolled at the same time.

For positive *n*, the `srl()` and `wscrl()` routines scroll the window up *n* lines (line *i*+*n* becomes *i*); otherwise scroll the window down *n* lines. This involves moving the lines in the window character image structure. The current cursor position is not changed.

For these functions to work, scrolling must be enabled via `scrollok()`.

RETURN VALUE

These routines return ERR upon failure, and OK (SVr4 only specifies "an integer value other than ERR") upon successful completion.

NOTES

Note that `srl()` and `scroll()` may be macros.

The SVr4 documentation says that the optimization of physically scrolling immediately if the scroll region is the entire screen "is" performed, not "may be" performed. This implementation deliberately does not guarantee that this will occur, in order to leave open the possibility of smarter optimization of multiple scroll actions on the next update.

Neither the SVr4 documentation specifies whether the current attribute or current color-pair of blanks generated by the scroll function is zeroed. Under this implementation it is.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions.

SEE ALSO

curses, *curs_outopts*

NAME

`slk_init()`, `slk_set()`, `slk_refresh()`, `slk_noutrefresh()`, `slk_label()`, `slk_clear()`, `slk_restore()`, `slk_touch()`, `slk_attron()`, `slk_attrset()`, `slk_attroff()` – curses soft label routines

SYNOPSIS

```
#include <curses.h>

int slk_init(int fmt);
int slk_set(int labnum, const char *label, int fmt);
int slk_refresh(void);
int slk_noutrefresh(void);
char *slk_label(int labnum);
int slk_clear(void);
int slk_restore(void);
int slk_touch(void);
int slk_attron(attr_t attrs);
int slk_attrset(attr_t attrs);
int slk_attroff(attr_t attrs);
```

DESCRIPTION

The *slk** functions manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, **curses** takes over the bottom line of *stdscr*, reducing the size of *stdscr* and the variable *LINES*. **curses** standardizes on eight labels of up to eight characters each.

The *slk_init()* routine must be called before *initscr()* or *newterm()* is called. If *initscr()* eventually uses a line from *stdscr* to emulate the soft labels, then *fmt* determines how the labels are arranged on the screen. Setting *fmt* to 0 indicates a 3-2-3 arrangement of the labels; 1 indicates a 4-4 arrangement.

The *slk_set()* routine requires *labnum* to be a label number, from 1 to 8; *label* must be the string to be put on the label, up to eight characters in length. A null string or a null pointer sets up a blank label. *fmt* is either 0, 1, or 2, indicating whether the label is to be left-justified, centered, or right-justified, respectively, within the label.

The *slk_refresh()* and *slk_noutrefresh()* routines correspond to the *wrefresh()* and *wnoutrefresh()* routines.

The *slk_label()* routine returns the current label for label number *labnum*, with leading and trailing blanks stripped.

The *slk_clear()* routine clears the soft labels from the screen.

The *slk_restore()* routine restores the soft labels to the screen after a *slk_clear()* has been performed.

The *slk_touch()* routine forces all the soft labels to be output the next time a *slk_noutrefresh()* is performed.

The *slk_attron()*, *slk_attrset()* and *slk_attroff()* routines correspond to *attron()*, *attrset()*, and *attroff()*. They have an effect only if soft labels are simulated on the bottom line of the screen. The default highlight for soft keys is A_STANDOUT (as in System V curses, which does not document this fact).

RETURN VALUE

These routines return ERR upon failure and OK (SVr4 specifies only "an integer value

other than ERR") upon successful completion.

slk_label() returns NULL on error.

NOTES

Most applications would use *slk_noutrefresh()* because a *wrefresh()* is likely to follow soon.

PORTABILITY

The XSI Curses standard, Issue 4, describes these functions. It changes the argument type of the attribute-manipulation functions *slk_attron()*, *slk_attroff()*, *slk_attrset()* to be **attr_t**, and adds **const** qualifiers.

SEE ALSO

curses, *curs_attr*, *curs_initscr*, *curs_refresh*

NAME

baudrate(), erasechar(), has_ic(), has_il(), killchar(), longname(), termattrs(), termname()
– curses environment query routines

SYNOPSIS

```
#include <curses.h>

int baudrate(void);
char erasechar(void);
int has_ic(void);
int has_il(void);
char killchar(void);
char *longname(void);
attr_t termattrs(void);
char *termname(void);
```

DESCRIPTION

The *baudrate()* routine returns the output speed of the terminal. The number returned is in bits per second, for example 9600, and is an integer.

The *erasechar()* routine returns the user's current erase character.

The *has_ic()* routine is true if the terminal has insert- and delete- character capabilities.

The *has_il()* routine is true if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to determine if it would be appropriate to turn on physical scrolling using *scrollok()*.

The *killchar()* routine returns the user's current line kill character.

The *longname()* routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *initscr()* or *newterm()*. The area is overwritten by each call to *newterm()* and is not restored by *set_term()*, so the value should be saved between calls to *newterm()* if *longname()* is going to be used with multiple terminals.

If a given terminal doesn't support a video attribute that an application program is trying to use, *curses()* may substitute a different video attribute for it. The *termattrs()* function returns a logical OR of all video attributes supported by the terminal. This information is useful when a *curses()* program needs complete control over the appearance of the screen.

The *termname()* routine returns the value of the environmental variable *TERM()* (truncated to 14 characters).

RETURN VALUE

longname() and *termname()* return NULL on error.

Routines that return an integer return ERR upon failure and OK (SVr4 only specifies "an integer value other than ERR") upon successful completion.

NOTES

Note that *termattrs()* may be a macro.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions. It changes the return type of *termattrs()* to the new type **attr_t**.

SEE ALSO

curses, curs_initscr, curs_outopts

NAME

setupterm(), setterm(), set_curterm(), del_curterm(), restartterm(), tparm(), tputs(), putp(), vidputs(), vidattr(), mvcur(), tigetflag(), tigetnum(), tigetstr() – curses interfaces to terminfo database

SYNOPSIS

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

int setupterm(const char *term, int ldes, int *errret);
int setterm(const char *term);
TERMINAL *set_curterm(TERMINAL *nterm);
int del_curterm(TERMINAL *oterm);
int restartterm(const char *term, int ldes, int *errret);
char *tparm(const char *str, ...);
char *tparam(const char *str, char *buffer, int size, ...);
int tputs(const char *str, int acnt, int (*putc)(int));
int putp(const char *str);
int vidputs(chtype attrs, int (*putc)(char *));
int vidattr(chtype attrs);
int mvcur(int oldrow, int oldcol, int newrow, int newcol);
int tigetag(const char *capname);
int tigetnum(const char *capname);
char *tigetstr(const char *capname);
```

DESCRIPTION

These low-level routines must be called by programs that have to deal directly with the **terminfo** database to handle certain terminal capabilities, such as programming function keys. For all other functionality, **curses** routines are more suitable and their use is recommended.

Initially, *setupterm()* should be called. Note that *setupterm()* is automatically called by *initscr()* and *newterm()*. This defines the set of terminal-dependent variables (listed in *terminfo*). The **terminfo** variables **lines** and **columns** are initialized by *setupterm()* as follows: If *use_env(FALSE)* has been called, values for **lines** and **columns** specified in **terminfo** are used. Otherwise, if the environment variables *LINES* and *COLUMNS* exist, their values are used. If these environment variables do not exist and the program is running in a window, the current window size is used. Otherwise, if the environment variables do not exist, the values for **lines** and **columns** specified in the **terminfo** database are used.

The header files *<curses.h>* and *<term.h>* should be included (in this order) to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through *tparm()* to instantiate them. All **terminfo** strings (including the output of *tparm()*) should be printed with *tputs()* or *putp()*. Call the *reset_shell_mode()* to restore the tty modes before exiting (see the *curs_kernel* reference page). Programs which use cursor addressing should output *enter_ca_mode()* upon startup and should output *exit_ca_mode()* before exiting. Programs desiring shell escapes should call *reset_shell_mode()* and output *exit_ca_mode()* before the shell is called and should output *enter_ca_mode()* and call *reset_prog_mode()* after returning from the shell.

The *setupterm()* routine reads in the **terminfo** database, initializing the **terminfo** structures, but does not set up the output virtualization structures used by **curses**. The

terminal type is the character string *term*; if *term* is null, the environment variable *TERM* is used. All output is to file descriptor *fildes* which is initialized for output. If *errret* is not null, then *setupterm()* returns OK or ERR and stores a status value in the integer pointed to by *errret*. A status of 1 in *errret* is normal, 0 means that the terminal could not be found, and -1 means that the **terminfo** database could not be found. If *errret* is null, *setupterm()* prints an error message upon finding an error and exits. Thus, the simplest call is:

```
setupterm((char *)0, 1, (int *)0);
```

which uses all the defaults and sends the output to stdout.

The *setterm()* routine is being replaced by *setupterm()*. The call:

```
setupterm(term, 1, (int *)0)
```

provides the same functionality as *setterm(*term*)*. The *setterm()* routine is included here for BSD compatibility, and is not recommended for new programs.

The *set_curterm()* routine sets the variable *cur_term* to *nterm*, and makes all of the **terminfo** boolean, numeric, and string variables use the values from *nterm*. It returns the old value of *cur_term*.

The *del_curterm()* routine frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as *cur_term*, references to any of the **terminfo** boolean, numeric, and string variables thereafter may refer to invalid memory locations until another *setupterm()* has been called.

The *restartterm()* routine is similar to *setupterm()* and *initscr()*, except that it is called after restoring memory to a previous state (for example, when reloading a game saved as a core image dump). It assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different. Accordingly, it saves various tty state bits, does a *setupterm*, and then restores the bits.

The *tparm()* routine instantiates the string *str* with parameters *pi*. A pointer is returned to the result of *str* with the parameters applied.

The *tputs()* routine applies padding information to the string *str* and outputs it. The *str* must be a terminfo string variable or the return value from *tparm()*, *tgetstr()*, or *tgoto()*. *affcnt* is the number of lines affected, or 1 if not applicable. *putc()* is a *putchar()*-like routine to which the characters are passed, one at a time.

The *putp()* routine calls

```
tputs(str, 1, putchar).
```

Note that the output of *putp()* always goes to *stdout*, not to the *fildes* specified in *setupterm()*.

The *vidputs()* routine displays the string on the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed in *curses*. The characters are passed to the *putchar()*-like routine *putc()*.

The *vidattr()* routine is like the *vidputs()* routine, except that it outputs through *putchar()*.

The *mvcur()* routine provides low-level cursor motion. It takes effect immediately (rather than at the next refresh).

The *tigetflag()*, *tigetnum()* and *tigetstr()* routines return the value of the capability corresponding to the **terminfo** *capname* passed to them, such as **xenl**.

The *tigetflag()* routine returns the value -1 if *capname* is not a boolean capability.

The *tigetnum()* routine returns the value -2 if *capname* is not a numeric capability.

The *tigetstr()* routine returns the value (char *)-1 if *capname* is not a string capability.

The *capname* for each capability is given in the table column entitled *capname* code in the capabilities section of *terminfo*.

```
char *boolnames, *boolcodes, *boolfnames
char *numnames, *numcodes, *numfnames
char *strnames, *strcodes, *strfnames
```

These null-terminated arrays contain the *capnames*, the **termcap** codes, and the full C names, for each of the **terminfo** variables.

RETURN VALUE

Routines that return an integer return ERR upon failure and OK (SVr4 only specifies "an integer value other than ERR") upon successful completion, unless otherwise noted in the preceding routine descriptions.

Routines that return pointers always return NULL on error.

NOTES

The *setupterm()* routine should be used in place of *setterm()*. It may be useful when you want to test for terminal capabilities without committing to the allocation of storage involved in *initscr()*.

Note that *vidattr()* and *vidputs()* may be macros.

PORTABILITY

The function *setterm()* is not described in the XSI Curses standard and must be considered non-portable. All other functions are as described in the XSI curses standard.

In System V Release 4, *set_curterm()* has an **int** return type and returns OK or ERR. We have chosen to implement the XSI Curses semantics.

In System V Release 4, the third argument of *tputs()* has the type **int (*putc)(char)**.

SEE ALSO

curses, *curs_initscr*, *curs_kernel*, *termcap*, *putc()*, *terminfo*

NAME

`touchwin()`, `touchline()`, `untouchwin()`, `wtouchln()`, `is_linetouched()`, `is_wintouched()` – curses refresh control routines

SYNOPSIS

```
#include <curses.h>

int touchwin(WINDOW *win);
int touchline(WINDOW *win, int start, int count);
int untouchwin(WINDOW *win);
int wtouchln(WINDOW *win, int y, int n, int changed);
int is_linetouched(WINDOW *win, int line);
int is_wintouched(WINDOW *win);
```

DESCRIPTION

The `touchwin()` and `touchline()` routines throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window affects the other window, but the records of which lines have been changed in the other window do not reflect the change. The routine `touchline()` only pretends that *count* lines have been changed, beginning with line *start*.

The `untouchwin()` routine marks all lines in the window as unchanged since the last call to `wrefresh()`.

The `wtouchln()` routine makes *n* lines in the window, starting at line *y*, look as if they have been changed (*changed*=1) or have not been changed (*changed*=0) since the last call to `wrefresh()`.

The `is_linetouched()` and `is_wintouched()` routines return TRUE if the specified line / window was modified since the last call to `wrefresh()`; otherwise they return FALSE. In addition, `is_linetouched()` returns ERR if *line* is not valid for the given window.

RETURN VALUE

All routines return the integer ERR upon failure and an integer value other than ERR upon successful completion, unless otherwise noted in the preceding routine descriptions.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions.

Some historic curses implementations had, as an undocumented feature, the ability to do the equivalent of `clearok(..., 1)` by saying `touchwin(stdscr)` or `clear(stdscr)`. This will not work under **ncurses**.

NOTES

All routines except `wtouchln()` may be macros.

SEE ALSO

curses, *curs_refresh*

NAME

unctrl(), keyname(), filter(), use_env(), putwin(), getwin(), delay_output(), flushinp() – miscellaneous curses utility routines

SYNOPSIS

```
#include <curses.h>

char *unctrl(chtype c);
char *keyname(int c);
void filter(void);
void use_env(char bool);
int putwin(WINDOW *win, FILE *filep);
WINDOW *getwin(FILE *filep);
int delay_output(int ms);
int flushinp(void);
```

DESCRIPTION

The *unctrl()* macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

The *keyname()* routine returns a character string corresponding to the key *c*.

The *filter()* routine, if used, must be called before *initscr()* or *newterm()* are called. The effect is that, during those calls, *LINES* is set to 1; the capabilities **clear**, **cup**, **cud**, **cud1**, **cuu1**, **cuu**, **vpa** are disabled; and the **home** string is set to the value of **cr**.

The *use_env()* routine, if used, is called before *initscr()* or *newterm()* are called. When called with FALSE as an argument, the values of **lines** and **columns** specified in the **terminfo** database will be used, even if the environment variables *LINES* and *COLUMNS* (used by default) are set, or if **curses** is running in a window (in which case default behavior would be to use the window size if *LINES* and *COLUMNS* are not set).

The *putwin()* routine writes all data associated with window *win* into the file to which *filep* points. This information can be later retrieved using the *getwin()* function.

The *getwin()* routine reads window related data stored in the file by *putwin()*. The routine then creates and initializes a new window using that data. It returns a pointer to the new window.

The *delay_output()* routine inserts an *ms* millisecond pause in output. This routine should not be used extensively because padding characters are used rather than a CPU pause.

The *flushinp()* routine throws away any typeahead that has been typed by the user and has not yet been read by the program.

RETURN VALUE

Except for *flushinp()*, routines that return an integer return ERR upon failure and OK (SVr4 specifies only "an integer value other than ERR") upon successful completion.

flushinp() always returns OK.

Routines that return pointers return NULL on error.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions.

The SVr4 documentation describes the action of *filter()* only in the vaguest terms. The description here is adapted from the XSI Curses standard (which erroneously fails to describe the disabling of **cuu**).

NOTES

Note that *unctrl()* is a macro, which is defined in **<unctrl.h>**.

SEE ALSO

curses, *curs_initscr*, *curs_scr_dump*

NAME

`newwin()`, `delwin()`, `mvwin()`, `subwin()`, `derwin()`, `mvderwin()`, `dupwin()`, `wsyncup()`, `syncok()`, `wcursyncup()`, `wsyncdown()` – create curses windows

SYNOPSIS

```
#include <curses.h>

WINDOW *newwin(int nlines, int ncols, int begin_y,
               int begin_x);
int delwin(WINDOW *win);
int mvwin(WINDOW *win, int y, int x);
WINDOW *subwin(WINDOW *orig, int nlines, int ncols,
               int begin_y, int begin_x);
WINDOW *derwin(WINDOW *orig, int nlines, int ncols,
               int begin_y, int begin_x);
int mvderwin(WINDOW *win, int par_y, int par_x);
WINDOW *dupwin(WINDOW *win);
void wsyncup(WINDOW *win);
int syncok(WINDOW *win, bool bf);
void wcursyncup(WINDOW *win);
void wsyncdown(WINDOW *win);
```

DESCRIPTION

Calling `newwin()` creates and returns a pointer to a new window with the given number of lines and columns. The upper left-hand corner of the window is at line `begin_y`, column `begin_x`. If either `nlines` or `ncols` is zero, they default to `LINES – begin_y` and `COLS – begin_x`. A new full-screen window is created by calling `newwin(0, 0, 0, 0)`.

Calling `delwin()` deletes the named window, freeing all memory associated with it (it does not actually erase the window's screen image). Subwindows must be deleted before the main window can be deleted.

Calling `mvwin()` moves the window so that the upper left-hand corner is at position (`x`, `y`). If the move would cause the window to be off the screen, it is an error and the window is not moved. Moving subwindows is allowed, but should be avoided.

Calling `subwin()` creates and returns a pointer to a new window with the given number of lines, `nlines`, and columns, `ncols`. The window is at position (`begin_y`, `begin_x`) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. The subwindow shares memory with the window *orig*. When using this routine, it is necessary to call `touchwin()` or `touchline()` on *orig* before calling `wrefresh()` on the subwindow.

Calling `derwin()` is the same as calling `subwin()`, except that `begin_y` and `begin_x` are relative to the origin of the window *orig* rather than the screen. There is no difference between the subwindows and the derived windows.

Calling `mvderwin()` moves a derived window (or subwindow) inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

Calling `dupwin()` creates an exact duplicate of the window *win*.

Calling *wsyncup()* touches all locations in ancestors of *win* that are changed in *win*. If *syncok()* is called with second argument TRUE then is called automatically whenever there is a change in the window.

The *wsyncdown()* routine touches each location in *win* that has been touched in any of its ancestor windows. This routine is called by *wrefresh()*, so it should almost never be necessary to call it manually.

The routine *wcursyncup()* updates the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

RETURN VALUE

Routines that return an integer return the integer ERR upon failure and OK (SVr4 only specifies "an integer value other than ERR") upon successful completion.

delwin() returns the integer ERR upon failure and OK upon successful completion.

Routines that return pointers return NULL on error.

NOTES

If many small changes are made to the window, the *wsyncup()* option could degrade performance.

Note that *syncok()* may be a macro.

BUGS

The subwindow functions (*subwin()*, *derwin()*, *mvderwin()*, *wsyncup()*, *wsyncdown()*, *wcursyncup()*, *syncok()*) are flaky, incompletely implemented, and not well tested.

The System V curses documentation is very unclear about what *wsyncup()* and *wsyncdown()* actually do. It seems to imply that they are only supposed to touch exactly those lines that are affected by ancestor changes. The language here, and the behavior of the **curses** implementation, is patterned on the XPG4 curses standard. The weaker XPG4 spec may result in slower updates.

PORTABILITY

The XSI Curses standard, Issue 4 describes these functions.

SEE ALSO

curses, *curs_refresh*, *curs_touch*

NAME

forms – curses extension for programming forms

SYNOPSIS

```
#include <form.h>
```

DESCRIPTION

The **forms** library provides terminal-independent facilities for composing form screens on character-cell terminals. The library includes: field routines, which create and modify form fields; and form routines, which group fields into forms, display forms on the screen, and handle interaction with the user.

The **forms** library uses the **curses** libraries, and a curses initialization routine such as *initscr()* must be called before using any of these functions. To use the **forms** library, link with the options **-lform -lcurses**.

Current Default Values for Field Attributes

The **forms** library maintains a default value for field attributes. You can get or set this default by calling the appropriate *get_* or *set_* routine with a NULL field pointer. Changing this default with a *set_* function affects future field creations, but does not change the rendering of fields already created.

Routine Name Index

The following table lists each form routine and the name of the manual page on which it is described.

curses Routine Name	Manual Page Name
<i>current_field()</i>	form_page
<i>data_ahead()</i>	form_data
<i>data_behind()</i>	form_data
<i>dup_field()</i>	form_field_new
<i>dynamic_fieldinfo()</i>	form_field_info
<i>field_arg()</i>	form_field_validation
<i>field_back()</i>	form_field_attributes
<i>field_buffer()</i>	form_field_buffer
<i>field_count()</i>	form_field
<i>field_fore()</i>	form_field_attributes
<i>field_index()</i>	form_page
<i>field_info()</i>	form_field_info
<i>field_init()</i>	form_hook
<i>field_just()</i>	form_field_just
<i>field_opts()</i>	form_field_opts
<i>field_opts_off()</i>	form_field_opts
<i>field_opts_on()</i>	form_field_opts
<i>field_pad()</i>	form_field_attributes

curses Routine Name	Manual Page Name
<code>field_status()</code>	<code>form_field_buffer</code>
<code>field_term()</code>	<code>form_hook</code>
<code>field_type()</code>	<code>form_field_validation</code>
<code>field_userptr()</code>	<code>form_field_userptr</code>
<code>form_driver()</code>	<code>form_driver</code>
<code>form_fields()</code>	<code>form_field</code>
<code>form_init()</code>	<code>form_hook</code>
<code>form_opts()</code>	<code>form_opts</code>
<code>form_opts_off()</code>	<code>form_opts</code>
<code>form_opts_on()</code>	<code>form_opts</code>
<code>form_page()</code>	<code>form_page</code>
<code>form_sub()</code>	<code>form_win</code>
<code>form_term()</code>	<code>form_hook</code>
<code>form_userptr()</code>	<code>form_userptr</code>
<code>form_win()</code>	<code>form_win</code>
<code>free_field()</code>	<code>form_field_new</code>
<code>free_form()</code>	<code>form_new</code>
<code>link_field()</code>	<code>form_field_new</code>
<code>link_fieldtype()</code>	<code>link_fieldtype</code>
<code>move_field()</code>	<code>form_field</code>
<code>new_field()</code>	<code>form_field_new</code>
<code>new_form()</code>	<code>form_new</code>
<code>new_page()</code>	<code>form_new_page</code>
<code>pos_form_cursor()</code>	<code>form_cursor</code>
<code>post_form()</code>	<code>form_post</code>
<code>scale_form()</code>	<code>form_win</code>
<code>set_current_field()</code>	<code>form_page</code>
<code>set_field_back()</code>	<code>form_field_attributes</code>
<code>set_field_buffer()</code>	<code>form_field_buffer</code>
<code>set_field_fore()</code>	<code>form_field_attributes</code>
<code>set_field_init()</code>	<code>form_hook</code>
<code>set_field_just()</code>	<code>form_field_just</code>
<code>set_field_opts()</code>	<code>form_field_opts</code>
<code>set_field_pad()</code>	<code>form_field_attributes</code>
<code>set_field_status()</code>	<code>form_field_buffer</code>
<code>set_field_term()</code>	<code>form_hook</code>
<code>set_field_type()</code>	<code>form_field_validation</code>
<code>set_field_userptr()</code>	<code>form_field_userptr</code>
<code>set_fieldtype_arg()</code>	<code>form_fieldtype</code>
<code>set_fieldtype_choice()</code>	<code>form_fieldtype</code>
<code>set_form_fields()</code>	<code>form_field</code>
<code>set_form_init()</code>	<code>form_hook</code>
<code>set_form_opts()</code>	<code>form_field_opts</code>
<code>set_form_page()</code>	<code>form_page</code>
<code>set_form_sub()</code>	<code>form_win</code>
<code>set_form_term()</code>	<code>form_hook</code>

curses Routine Name	Manual Page Name
<i>set_form_userptr()</i>	<i>form_userptr</i>
<i>set_form_win()</i>	<i>form_win</i>
<i>set_max_field()</i>	<i>form_field_buffer</i>
<i>set_new_page()</i>	<i>form_new_page</i>
<i>unpost_form()</i>	<i>form_post</i>

RETURN VALUE

Routines that return pointers return NULL on error. Routines that return an integer return one of the following error codes:

- E_OK** The routine succeeded.
- E_CONNECTED**
The field is already connected to a form.
- E_SYSTEM_ERROR**
System error occurred (see *errno*).
- E_BAD_ARGUMENT**
Routine detected an incorrect or out-of-range argument.
- E_POSTED**
The form is already posted.
- E_BAD_STATE**
Routine was called from an initialization or termination function.
- E_NO_ROOM**
Form is too large for its window.
- E_NOT_POSTED**
The form has not been posted.
- E_UNKNOWN_COMMAND**
The form driver code saw an unknown request code.
- E_INVALID_FIELD**
Contents of a field are not valid.
- E_NOT_CONNECTED**
No fields are connected to the form.
- E_REQUEST_DENIED**
The form driver could not process the request.

SEE ALSO

curses and pages whose names begin *form_* for detailed descriptions of the entry points.

NOTES

The header file **<form.h>** automatically includes the header files **<curses.h>** and **<eti.h>**.

In your library list, **libform.a** should be before **libncurses.a**; that is, you want to say **-lform -lncurses**, not the other way around (which would give you a link error using many linkers).

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for ncurses by Eric S. Raymond.

NAME

pos_form_cursor() – position a form window cursor

SYNOPSIS

```
#include <form.h>

int pos_form_cursor(FORM *form);
```

DESCRIPTION

The function *pos_form_cursor()* restores the cursor to the position required for the forms driver to continue processing requests. This is useful after **curses** routines have been called to do screen-painting in response to a form operation.

RETURN VALUE

This routine returns one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR System error occurred (see *errno*).
E_BAD_ARGUMENT Routine detected an incorrect or out-of-range argument.
E_NOT_POSTED The form has not been posted.

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`data_ahead()`, `data_behind()` – test for off-screen data

SYNOPSIS

```
#include <form.h>

bool data_ahead(FORM *form);
bool data_behind(FORM *form);
```

DESCRIPTION

The function `data_ahead()` tests whether there is off-screen data ahead in the given form. It returns TRUE (1) or FALSE (0).

The function `data_behind()` tests whether there is off-screen data behind in the given form. It returns TRUE (1) or FALSE (0).

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`form_driver()` – command-processing loop of the form system

SYNOPSIS

```
#include <form.h>

int form_driver(FORM *form, int c);
```

DESCRIPTION

Once a form has been posted (displayed), you should funnel input events to it through `form_driver()`. This routine has two major input cases; either the input is a form navigation request or it's a printable ASCII character. The form driver requests are as follows:

```
REQ_NEXT_PAGE
    Move to the next page.
REQ_PREV_PAGE
    Move to the previous page.
REQ_FIRST_PAGE
    Move to the first page.
REQ_LAST_PAGE
    Move to the last field.
REQ_NEXT_FIELD
    Move to the next field.
REQ_PREV_FIELD
    Move to the previous field.
REQ_FIRST_FIELD
    Move to the first field.
REQ_LAST_FIELD
    Move to the last field.
REQ_SNEXT_FIELD
    Move to the sorted next field.
REQ_SPREV_FIELD
    Move to the sorted previous field.
REQ_SFIRST_FIELD
    Move to the sorted first field.
REQ_SLAST_FIELD
    Move to the sorted last field.
REQ_LEFT_FIELD
    Move left to a field.
REQ_RIGHT_FIELD
    Move right to a field.
REQ_UP_FIELD
    Move up to a field.
REQ_DOWN_FIELD
    Move down to a field.
REQ_NEXT_CHAR
    Move to the next char.
REQ_PREV_CHAR
    Move to the previous char.
REQ_NEXT_LINE
    Move to the next line.
```

REQ_PREV_LINE
Move to the previous line.

REQ_NEXT_WORD
Move to the next word.

REQ_PREV_WORD
Move to the previous word.

REQ_BEG_FIELD
Move to the beginning of the field.

REQ_END_FIELD
Move to the end of the field.

REQ_BEG_LINE
Move to the beginning of the line.

REQ_END_LINE
Move to the end of the line.

REQ_LEFT_CHAR
Move left in the field.

REQ_RIGHT_CHAR
Move right in the field.

REQ_UP_CHAR
Move up in the field.

REQ_DOWN_CHAR
Move down in the field.

REQ_NEW_LINE
Insert or overlay a new line.

REQ_INS_CHAR
Insert a blank at the cursor.

REQ_INS_LINE
Insert a blank line at the cursor.

REQ_DEL_CHAR
Delete character at the cursor.

REQ_DEL_PREV
Delete character before the cursor.

REQ_DEL_LINE
Delete line at the cursor.

REQ_DEL_WORD
Delete blank-delimited word at the cursor.

REQ_CLR_EOL
Clear to end of line from cursor.

REQ_CLR_EOF
Clear to end of field from cursor.

REQ_OVL_MODE
Enter overlay mode.

REQ_INS_MODE
Enter insert mode.

REQ_SCR_FLINE
Scroll the field forward a line.

REQ_SCR_BLINE
Scroll the field backward a line.

REQ_SCR_FPAGE
Scroll the field forward a page.

REQ_SCR_BPAGE

Scroll the field backward a page.
 REQ_SCR_FHPAGE
 Scroll the field forward half a page.
 REQ_SCR_BHPAGE
 Scroll the field backward half a page.
 REQ_SCR_FCHAR
 Scroll the field forward a character.
 REQ_SCR_BCHAR
 Scroll the field backward a character.
 REQ_SCR_HFLINE
 Horizontal scroll the field forward a line.
 REQ_SCR_HBLINE
 Horizontal scroll the field backward a line.
 REQ_SCR_HFHALF
 Horizontal scroll the field forward half a line.
 REQ_SCR_HBHALF
 Horizontal scroll the field backward half a line.
 REQ_VALIDATION
 Validate field.
 REQ_PREV_CHOICE
 Display previous field choice.
 REQ_NEXT_CHOICE
 Display next field choice.

If the second argument is a printable ASCII character, the driver places it in the current position in the current field. If it is one of the forms requests listed above, that request is executed.

If the second argument is neither printable ASCII nor one of the above pre-defined form requests, the driver assumes it is an application-specific command and returns `E_UNKNOWN_COMMAND`. Application-defined commands should be defined relative to **MAX_COMMAND**, the maximum value of these pre-defined requests.

RETURN VALUE

form_driver return one of the following error codes:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see *errno*).
E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
E_BAD_STATE
 Routine was called from an initialization or termination function.
E_NOT_POSTED
 The form has not been posted.
E_UNKNOWN_COMMAND
 The form driver code saw an unknown request code.
E_INVALID_FIELD
 Contents of field is invalid.

E_REQUEST_DENIED

The form driver could not process the request.

SEE ALSO

curses, forms

NOTES

The header file **<form.h>** automatically includes the header files **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`set_form_fields()`, `form_fields`, `field_count()`, `move_field()` – make and break connections between fields and forms

SYNOPSIS

```
#include <form.h>

int set_form_fields(FORM *form, FIELD **fields);
FIELD **form_fields(FORM *form);
int field_count(FORM *form);
int move_field(FIELD *field, int frow, int fcol);
```

DESCRIPTION

The function `set_form_fields()` changes the field pointer array of the given *form*. The array must be terminated by a NULL.

The function `form_fields()` returns the field array of the given form.

The function `field_count()` returns the count of fields in *form*.

The function `move_field()` move the given field (which must be disconnected) to a specified location on the screen.

RETURN VALUES

The function `form_fields()` returns NULL on error.

The function `field_count()` returns ERR (the general **curses** error return value) on error.

The functions `set_form_fields()` and `move_field()` return one of the following codes on error:

E_OK The routine succeeded.
 E_SYSTEM_ERROR
 System error occurred (see *errno*).
 E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
 E_POSTED
 The form is already posted.
 E_CONNECTED
 The field is already connected to a form.

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

The SVr4 forms library documentation specifies the `field_count()` error value as -1 (which is the value of ERR).

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`set_field_fore()`, `field_fore()`, `set_field_back()`, `field_back()`, `set_field_pad()`, `field_pad()` – color and attribute control for form fields

SYNOPSIS

```
#include <form.h>

int set_field_fore(FIELD *field, chtype attr);
chtype field_fore(FIELD *field);
int set_field_back(FIELD *field, chtype attr);
chtype field_back(FIELD *field);
int set_field_pad(FIELD *field, int pad);
chtype field_pad(FIELD *field);
```

DESCRIPTION

The function `set_field_fore()` sets the foreground attribute of *field*. This is the highlight used to display the field contents. The function `field_fore()` returns the foreground attribute. The default is `A_STANDOUT`.

The function `set_field_back()` sets the background attribute of *form*. This is the highlight used to display the extent fields in the form. The function `field_back()` returns the background attribute. The default is `A_NORMAL`.

The function `set_field_pad()` sets the character used to fill the field. The function `field_pad()` returns the given form's pad character. The default is a blank.

RETURN VALUE

These routines return one of the following:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).
`E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.

SEE ALSO

curses and 3X pages whose names begin *form_* for detailed descriptions of the entry points.

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_field_buffer(), field_buffer(), set_field_status(), field_status(), set_max_field() – field buffer control

SYNOPSIS

```
#include <form.h>

int set_field_buffer(FIELD *field, int buf, char *value);
char *field_buffer(FIELD *field, int buffer);
int set_field_status(FIELD *field, int status);
int field_status(FIELD *field);
int set_max_field(FIELD *field, int max);
```

DESCRIPTION

The function *set_field_buffer()* sets the numbered buffer of the given field to contain a given string. Buffer 0 is the displayed value of the field; other numbered buffers may be allocated by applications through the *nbuffers* argument of *new_field()* but are not manipulated by the forms library. The function *field_buffer()* returns the foreground attribute.

The function *set_field_status()* sets the associated status flag of *field*; *field_status()* gets the current value. The status flag is set to a nonzero value whenever the field changes.

The function *set_max_field()* sets the maximum size for a dynamic field. An argument of 0 turns off any maximum size threshold for that field.

RETURN VALUE

The *field_buffer()* function returns NULL on error.

The *field_status()* function returns TRUE or FALSE.

The remaining routines return one of the following:

```
E_OK      The routine succeeded.
E_SYSTEM_ERROR
           errno )
E_BAD_ARGUMENT
           Routine detected an incorrect or out-of-range argument.
```

SEE ALSO

curses, *new_field*, *forms* and 3X pages whose names begin *form_* for detailed descriptions of the entry points.

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

field_info(), dynamic_field_info() – retrieve field characteristics

SYNOPSIS

```
#include <form.h>

int field_info(FIELD *field, int *rows, int *cols,
               int *frow, int *fcol, int *nrow, int *nbuf);
int dynamic_field_info(FIELD *field, int *rows, int *cols,
                       int *max);
```

DESCRIPTION

The function *field_info()* returns the sizes and other attributes passed in to the field at its creation time. The attributes are: height, width, row of upper-left corner, column of upper-left corner, number off-screen rows, and number of working buffers.

The function *dynamic_field_info()* returns the actual size of the field, and its maximum possible size. If the field has no size limit, the location addressed by the third argument will be set to 0. (A field can be made dynamic by turning off the O_STATIC).

RETURN VALUE

These routines return one of the following:

E_OK The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

E_BAD_ARGUMENT

Routine detected an incorrect or out-of-range argument.

SEE ALSO

curses, and 3X pages whose names begin *form_* for detailed descriptions of the entry points.

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_field_just(), field_just() – retrieve field characteristics

SYNOPSIS

```
#include form.h

int set_field_just(FIELD *field, int justification);
int field_just(FIELD *field);
```

DESCRIPTION

The function *set_field_just()* sets the justification attribute of a field; returns a field's justification attribute. The attribute may be one of NO_JUSTIFICATION, JUSTIFY_RIGHT, JUSTIFY_LEFT, or JUSTIFY_CENTER.

RETURN VALUE

The function *field_just()* returns one of: NO_JUSTIFICATION, JUSTIFY_RIGHT, JUSTIFY_LEFT, or JUSTIFY_CENTER.

The function *set_field_just()* return one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR System error occurred (see *errno*).
E_BAD_ARGUMENT Routine detected an incorrect or out-of-range argument.

SEE ALSO

curses and 3X pages whose names begin *form_* for detailed descriptions of the entry points.

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`new_field()`, `dup_field()`, `link_field()`, `free_field()` – create and destroy form fields

SYNOPSIS

```
#include <form.h>

FIELD *new_field(int height, int width, int toprow, int leftcol,
                 int offscreen, int nbuffers);
FIELD *dup_field(FIELD *field, int toprow, int leftcol);
FIELD *link_field(FIELD *field, int toprow, int leftcol);
int free_field(FIELD *field);
```

DESCRIPTION

The function `new_field()` allocates a new field and initializes it from the parameters given: height, width, row of upper-left corner, column of upper-left corner, number off-screen rows, and number of working buffers.

The function `dup_field()` duplicates a field at a new location. Most attributes (including current contents, size, validation type, buffer count, growth threshold, justification, fireground, background, padd character, options, and user pointer) are copied. Field status and the field page bit are not copied.

The function `link_field()` acts like `dup_field()`, but the new field shares buffers with its parent. Attribute data is separate.

The function `free_field()` de-allocates storage associated with a field.

RETURN VALUE

The functions `new_field()`, `dup_field()`, and `link_field()` return NULL on error.

The function `free_field()` returns one of the following:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).
`E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

It may be unwise to count on the set of attributes copied by `dup_field()` being portable; the System V forms library documents are not very explicit on what gets copied and was not.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_field_opts(), field_opts_on(), field_opts_off(), field_opts() – set and get field options

SYNOPSIS

```
#include <form.h>

int set_field_opts(FIELD *field, OPTIONS opts);
int field_opts_on(FIELD *field, OPTIONS opts);
int field_opts_off(FIELD *field, OPTIONS opts);
OPTIONS field_opts(FIELD *field);
```

DESCRIPTION

The function *set_field_opts()* sets all the given field's option bits (field option bits may be logically-ored together).

The function *field_opts_on()* turns on the given option bits, and leaves others alone.

The function *field_opts_off()* turns off the given option bits, and leaves others alone.

The function *field_opts()* returns the field's current option bits.

The following options are defined (all are on by default):

O_VISIBLE

The field is displayed. If this option is off, display of the field is suppressed,

O_ACTIVE

The field is visited during processing. If this option is off, the field will not be reachable by navigation keys.

O_PUBLIC

The field contents are displayed as data is entered.

O_EDIT

The field can be edited.

O_WRAP

Words that don't fit on a line are wrapped to the next line. Words are blank-separated.

O_BLANK

The field is cleared whenever a character is entered at the first position.

O_AUTOSKIP

Skip to the next field when this one fills

O_NULLOK

Allow a blank field.

O_STATIC

Field buffers are fixed to field's original size.

O_PASSOK

Validate field only if modified by user.

RETURN VALUE

Except for *field_opts()*, each routine returns one of the following:

E_OK

The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

E_CURRENT

The field is the current field.

SEE ALSO

curses, forms

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_field_userptr(), field_userptr() – associate application data with a form field

SYNOPSIS

```
#include <form.h>

int set_field_userptr(FIELD *field, char *userptr);
char field_userptr(FIELD *field);
```

DESCRIPTION

Every form field has a field that can be used to hold application-specific data (that is, the form-driver code leaves it alone). These functions get and set that field.

RETURN VALUE

The function *field_userptr()* returns NULL on error. The function *set_field_userptr()* returns

E_OK The routine succeeded.
E_SYSTEM_ERROR System error occurred (see *errno*).

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

The user pointer should be a void pointer. We leave it as a char pointer for SVr4 compatibility.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_field_type(), field_type(), field_arg() – data type validation for fields

SYNOPSIS

```
#include <form.h>

int set_field_type(FIELD *field, FIELDTYPE *type, ...);
FIELDTYPE *field_type(FIELD *field);
char *field_arg(FIELD *field);
```

DESCRIPTION

The function *set_field_type()* declares a data type for a given form field. This is the type checked by validation functions. The types are as follows:

TYPE_ALNUM

Alphanumeric data. Requires a third argument, a minimum field width.

TYPE_ALPHA

Character data. Requires a third argument, a minimum field width.

TYPE_ENUM

Accept one of a specified set of strings. Requires a third (char **) argument pointing to a string list; a fourth flag argument to enable case-sensitivity; and a fifth flag argument specifying whether a partial match must be a unique one (if this flag is off, a prefix matches the first of any set of more than one list elements with that prefix).

TYPE_INTEGER

Integer data, parseable to an integer by *atoi()*. Requires a third integer precision argument that sets zero-padding, a fourth argument constraining minimum value, and a fifth constraining maximum value.

TYPE_NUMERIC

Numeric data (may have a decimal-point part). Requires following arguments as for **TYPE_INTEGER** data.

TYPE_REGEX

Regular expression data. Requires a regular expression third argument; the data is valid if the regular expression matches it. Regular expressions are in the format of *regcomp* and *regex*.

It is possible to set up new programmer-defined field types. See the *form_fieldtype()* manual page.

RETURN VALUE

The functions *field_type()* and *field_arg()* return NULL on error. The function *set_field_type()* returns one of the following:

E_OK

The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

SEE ALSO

atoi(), *curses*, *forms*, *form_fieldtype*, *regcomp()*, *regex*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`new_fieldtype()`, `free_fieldtype()`, `set_fieldtype_arg()`, `set_fieldtype_choice()`
`link_fieldtype form_field` – define validation-field types

SYNOPSIS

```
#include <form.h>

FIELDTYPE *new_fieldtype(int (*field_check)(FIELD *, char *),
                        int (*char_check)(int, char *));
int free_fieldtype(FIELDTYPE *fieldtype);
in set_fieldtype_arg(FIELDTYPE *fieldtype
                    char *(*mak_arg)(va_list *),
                    char *(*copy_arg)(char *),
                    void (*free_arg)(char *));
in set_fieldtype_choice(FIELDTYPE *fieldtype
                       int (*next_choice)(FIELD *, char *),
                       int (*prev_choice)(FIELD *, char *));
FIELDTYPE *link_fieldtype(FIELDTYPE *type1, FIELDTYPE *type2);
```

DESCRIPTION

The function `new_fieldtype()` creates a new field type usable for data validation. You supply it with `field_check`, a predicate to check the validity of an entered data string whenever the user attempt to leave a field. The **(FIELD *)** argument is passed in so the validation predicate can see the field's buffer, sizes and other attributes; the second argument is an argument-block structure, about which more below.

You also supply with `char_check`, a function to validate input characters as they are entered; it will be passed the character to be checked and a pointer to an argument-block structure.

The function `free_fieldtype()` frees the space allocated for a given validation type.

The function `set_fieldtype()` associates three storage-management functions with a field type. The `mak_arg` function is automatically applied to the list of arguments you give `set_field_type()` when attaching validation to a field; its job is to bundle these into an allocated argument-block object which can later be passed to validation predicated. The other two hook arguments should copy and free argument-block structures. They will be used by the forms-driver code.

The form driver requests `REQ_NEXT_CHOICE` and `REQ_PREV_CHOICE` assume that the possible values of a field form an ordered set, and provide the forms user with a way to move through the set. The `set_fieldtype_choice()` function allows forms programmers to define successor and predecessor functions for the field type. These functions take the field pointer and an argument-block structure as arguments.

RETURN VALUE

The character-pointer-valued routines return NULL on error.

The integer-valued routines return one of the following codes on error:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).
`E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.

E_CONNECTED

The field is already connected to a form.

SEE ALSO

curses, forms

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

All of the **(char *)** arguments of these functions should actually be **(void *)**. The type has been left uncorrected for strict compatibility with System V.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_field_init(), set_field_term(), field_term(), form_init(), set_form_init(),
set_form_term(), form_term() – set hooks for automatic invocation by applications

SYNOPSIS

```
#include <form.h>

int set_field_init(FORM *form, void (*func)(FORM *));
void (*)(FORM *) field_init(MWNU *form);
int set_field_term(FORM *form, void (*func)(FORM *));
void (*)(FORM *) field_term(MWNU *form);
int set_form_init(FORM *form, void (*func)(FORM *));
void (*)(FORM *) form_init(MWNU *form);
int set_form_term(FORM *form, void (*func)(FORM *));
void (*)(FORM *) form_term(MWNU *form);
```

DESCRIPTION

These functions make it possible to set hook functions to be called at various points in the automatic processing of input event codes by *form_driver()*.

The function *set_field_init()* sets a hook to be called at form-post time and each time the selected field changes (after the change). *field_init()* returns the current field init hook, if any (NULL if there is no such hook).

The function *set_field_term()* sets a hook to be called at form-unpost time and each time the selected field changes (before the change). *field_term()* returns the current field term hook, if any (NULL if there is no such hook).

The function *set_form_init()* sets a hook to be called at form-post time and just after a page change once it is posted. *form_init()* returns the current form init hook, if any (NULL if there is no such hook).

The function *set_form_term()* sets a hook to be called at form-unpost time and just before a page change change once it is posted. *form_term()* returns the current form term hook, if any (NULL if there is no such hook).

RETURN VALUE

Routines that return pointers return NULL on error. Other routines return one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see *errno*).

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`new_form()`, `free_form()` – create and destroy forms

SYNOPSIS

```
#include <form.h>

FORM *new_form(FIELD **fields);
char *free_form(FORM *form);
```

DESCRIPTION

The function `new_form()` creates a new form connected to a specified field pointer array (which must be NULL-terminated).

The function `free_form()` disconnects *form* from its field array and frees the storage allocated for the form.

RETURN VALUE

The function `new_form()` returns NULL on error.

The function `free_form()` returns one of the following:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).
`E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.
`E_POSTED`
 The form has already been posted.

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_new_page(), new_page() – form pagination functions

SYNOPSIS

```
#include <form.h>

int set_new_page(FIELD *field, int justification);
int new_page(FIELD *field);
```

DESCRIPTION

The function *set_new_page()* marks the given field as the beginning of a new page on its form.

The function *new_page()* is a predicate which tests if a given field marks a page beginning on its form.

RETURN VALUE

The function *new_page()* returns TRUE or FALSE.

The function *set_new_page()* return one of the following:

E_OK The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

E_CONNECTED

The given field is already connected to a form.

SEE ALSO

curses end 3X pages whose names begin *form_* for detailed descriptions of the entry points.

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_form_opts(), form_opts_on(), form_opts_off() – set and get form options

SYNOPSIS

```
#include <form.h>

int set_form_opts(FORM *form, OPTIONS opts);
int form_opts_on(FORM *form, OPTIONS opts);
int form_opts_off(FORM *form, OPTIONS opts);
OPTIONS form_opts(FORM *form);
```

DESCRIPTION

The function *set_form_opts()* sets all the given form's option bits (form option bits may be logically-ored together).

The function *form_opts_on()* turns on the given option bits, and leaves others alone.

The function *form_opts_off()* turns off the given option bits, and leaves others alone.

The function *form_opts()* returns the form's current option bits.

The following options are defined (all are on by default):

O_NL_OVERLOAD

Overload the REQ_NEW_LINE forms driver request so that calling it at the end of a field goes to the next field.

O_BS_OVERLOAD

Overload the REQ_DEL_PREV forms driver request so that calling it at the beginning of a field goes to the previous field.

RETURN VALUE

Except for *form_opts()*, each routine returns one of the following:

E_OK The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_current_field(), current_field(), set_form_page() – set and get form page number

SYNOPSIS

```
#include <form.h>

int set_current_field(FORM *form, int page);
FIELD *current_field(FORM *);
int set_form_page(FORM *form, int n);
int form_page(FORM *form);
```

DESCRIPTION

The function *set_current()*field sets the current field of the given form; *current_field()* returns the current field of the given form.

The function *set_form_page()* sets the form's page number (goes to page *n* of the form).

The function *form_page()* returns the form's current page number.

RETURN VALUE

Except for *form_page()*, each routine returns one of the following:

E_OK The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

E_BAD_ARGUMENT

Routine detected an incorrect or out-of-range argument.

E_BAD_STATE

Routine was called from an initialization or termination function.

E_INVALID_FIELD

Contents of a field are not valid.

E_REQUEST_DENIED

The form driver could not process the request.

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`post_form()`, `unpost_form()` – write or erase forms from associated subwindows

SYNOPSIS

```
#include <form.h>

int post_form(FORM *form);
int unpost_form(FORM *form);
```

DESCRIPTION

The function `post_form()` displays a form to its associated subwindow. To trigger physical display of the subwindow, use `refresh()` or some equivalent **curses** routine (the implicit `doupdate()` triggered by an **curses** input request will do).

The function `unpost_form()` erases form from its associated subwindow.

RETURN VALUE

These routines return one of the following:

- `E_OK` The routine succeeded.
- `E_SYSTEM_ERROR`
 System error occurred (see *errno*).
- `E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.
- `E_POSTED`
 The form has already been posted.
- `E_BAD_STATE`
 Routine was called from an initialization or termination function.
- `E_NO_ROOM`
 Form is too large for its window.
- `E_NOT_POSTED`
 The form has not been posted.
- `E_NOT_CONNECTED`
 No items are connected to the form.

SEE ALSO

curses, *doupdate*, *forms*, *refresh*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`form_userptr()`, `set_form_userptr()` – associate application data with a form item

SYNOPSIS

```
#include <form.h>

int set_form_userptr(FORM *form, char *userptr);
char form_userptr(FORM *form);
```

DESCRIPTION

Every form and every form item has a field that can be used to hold application-specific data (that is, the form-driver code leaves it alone). These functions get and set the form user pointer field.

RETURN VALUE

The function `form_userptr()` returns NULL on error.

The function `set_form_userptr()` returns one of the following:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

The user pointer should be a void pointer. We leave it as a char pointer for SVr4 compatibility.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`form_win()`, `set_form_win()`, `form_sub()`, `set_form_sub()`, `scale_form()` – make and break form window and subwindow associations

SYNOPSIS

```
#include <form.h>

int set_form_win(FORM *form, WINDOW *win);
WINDOW *form_win(FORM *form);
int set_form_sub(FORM *form, WINDOW *sub);
WINDOW *form_sub(FORM *form);
int scale_form(FORM *form, int *rows, int *columns);
```

DESCRIPTION

Every form has an associated pair of **curses** windows. The form window displays any title and border associated with the window; the form subwindow displays the items of the form that are currently available for selection.

The first four functions get and set those windows. It is not necessary to set either window; by default, the driver code uses *stdscr* for both.

In the *set_* functions, window argument of NULL is treated as though it were *stdscr*. A form argument of NULL is treated as a request to change the system default form window or subwindow.

The function *scale_window()* returns the minimum size required for the subwindow of *form*.

RETURN VALUE

Routines that return pointers return NULL on error. Routines that return an integer return one of the following error codes:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see *errno*).
E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
E_POSTED
 The form has already been posted.
E_NOT_CONNECTED
 No items are connected to the form.

SEE ALSO

curses, *forms*

NOTES

The header file **<form.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V forms library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

menus – curses extension for programming menus

SYNOPSIS

```
#include <menu.h>
```

DESCRIPTION

The **menus** library provides terminal-independent facilities for composing menu systems on character-cell terminals. The library includes: item routines, which create and modify menu items; and menu routines, which group items into menus, display menus on the screen, and handle interaction with the user.

The **menus** library uses the **curses** libraries, and a curses initialization routine such as **initscr** must be called before using any of these functions. To use the **menus** library, link with the options **-lmenu -lcurses**.

Current Default Values for Item Attributes

The **menus** library maintains a default value for item attributes. You can get or set this default by calling the appropriate *get_* or *set_* routine with a NULL item pointer. Changing this default with a *set_* function affects future item creations, but does not change the rendering of items already created.

Routine Name Index

The following table lists each menu routine and the name of the manual page on which it is described.

curses Routine Name	Manual Page Name
<i>current_item()</i>	mitem_current
<i>free_item()</i>	mitem_new
<i>free_menu()</i>	menu_new
<i>item_count()</i>	menu_items
<i>item_description()</i>	mitem_name
<i>item_index()</i>	mitem_current
<i>item_init()</i>	menu_hook
<i>item_name()</i>	mitem_name
<i>item_opts()</i>	mitem_opts
<i>item_opts_off()</i>	mitem_opts
<i>item_opts_on()</i>	mitem_opts
<i>item_term()</i>	menu_hook
<i>item_userptr()</i>	mitem_userptr
<i>item_value()</i>	mitem_value
<i>item_visible()</i>	mitem_visible
<i>menu_back()</i>	menu_attrbs
<i>menu_driver()</i>	menu_driver
<i>menu_fore()</i>	menu_fore

curses Routine Name	Manual Page Name
<i>menu_format()</i>	menu_format
<i>menu_grey()</i>	menu_attribs
<i>menu_init()</i>	menu_hook
<i>menu_items()</i>	menu_items
<i>menu_mark()</i>	menu_mark
<i>menu_opts()</i>	menu_opts
<i>menu_opts_off()</i>	menu_opts
<i>menu_opts_on()</i>	menu_opts
<i>menu_pad()</i>	menu_attribs
<i>menu_pattern()</i>	menu_pattern
<i>menu_sub()</i>	menu_win
<i>menu_term()</i>	menu_hook
<i>menu_userptr()</i>	menu_userptr
<i>menu_win()</i>	menu_win
<i>new_item()</i>	mitem_new
<i>new_menu()</i>	menu_new
<i>pos_menu_cursor()</i>	menu_cursor
<i>post_menu()</i>	menu_post
<i>scale_menu()</i>	menu_win
<i>set_current_item()</i>	menu_current_item
<i>set_item_init()</i>	menu_hook
<i>set_item_opts()</i>	mitem_opts
<i>set_item_term()</i>	menu_hook
<i>set_item_userptr()</i>	mitem_userptr
<i>set_item_value()</i>	mitem_value
<i>set_menu_back()</i>	menu_attribs
<i>set_menu_fore()</i>	menu_attribs
<i>set_menu_format()</i>	menu_format
<i>set_menu_grey()</i>	menu_attribs
<i>set_menu_init()</i>	menu_hook
<i>set_menu_items()</i>	menu_items
<i>set_menu_mark()</i>	menu_mark
<i>set_menu_opts()</i>	mitem_opts
<i>set_menu_pad()</i>	menu_attribs
<i>set_menu_pattern()</i>	menu_pattern
<i>set_menu_sub()</i>	menu_win
<i>set_menu_term()</i>	menu_hook
<i>set_menu_userptr()</i>	menu_userptr
<i>set_menu_win()</i>	menu_win
<i>set_top_row()</i>	mitem_current
<i>top_row()</i>	mitem_current
<i>unpost_menu()</i>	menu_post

RETURN VALUE

Routines that return pointers return NULL on error. Routines that return an integer return one of the following error codes:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see errno).
E_BAD_ARGUMENT

Routine detected an incorrect or out-of-range argument.

E_POSTED

The menu is already posted.

E_BAD_STATE

Routine was called from an initialization or termination function.

E_NO_ROOM

Menu is too large for its window.

E_NOT_POSTED

The menu has not been posted.

E_UNKNOWN_COMMAND

The menu driver code saw an unknown request code.

E_NO_MATCH

Character failed to match.

E_NOT_SELECTABLE

The designated item cannot be selected.

E_NOT_CONNECTED

No items are connected to the menu.

E_REQUEST_DENIED

The menu driver could not process the request.

SEE ALSO

curses and pages whose names begin *menu_* for detailed descriptions of the entry points.

NOTES

The header file **<menu.h>** automatically includes the header files **<curses.h>** and **<eti.h>**.

In your library list, libmenu.a should be before libncurses.a; that is, you want to say **-lmenu -lncurses**, not the other way around (which would give you a link error using many linkers).

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for ncurses by Eric S. Raymond.

NAME

menu_fore(), set_menu_fore(), menu_back(), set_menu_back(), menu_grey(),
set_menu_grey(), menu_pad() set_menu_pad() – color and attribute control for menus

SYNOPSIS

```
#include <menu.h>

int set_menu_fore(MENU *menu, chtype attr);
chtype menu_fore(MENU *menu);
int set_menu_back(MENU *menu, chtype attr);
chtype menu_back(MENU *menu);
int set_menu_grey(MENU *menu, chtype attr);
chtype menu_grey(MENU *menu);
int set_menu_pad(MENU *menu, int pad);
chtype menu_pad(MENU *menu);
```

DESCRIPTION

The function *set_menu_fore()* sets the foreground attribute of *menu*. This is the highlight used for selected menu items. *menu_fore()* returns the foreground attribute. The default is A_STANDOUT.

The function *set_menu_back()* sets the background attribute of *menu*. This is the highlight used for selectable (but not currently selected) menu items. The function *menu_back()* returns the background attribute. The default is A_NORMAL.

The function *set_menu_grey()* sets the grey attribute of *menu*. This is the highlight used for un-selectable menu items in menus that permit more than one selection. The function *menu_grey()* returns the grey attribute. The default is A_UNDERLINE.

The function *set_menu_pad()* sets the character used to fill the space between the name and description parts of a menu item. *menu_pad()* returns the given menu's pad character. The default is a blank.

RETURN VALUE

These routines return one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR System error occurred (see *errno*).
E_BAD_ARGUMENT Routine detected an incorrect or out-of-range argument.

SEE ALSO

curses and reference pages whose names begin *menu_* for detailed descriptions of the entry points.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

pos_menu_cursor() – position a menu's cursor

SYNOPSIS

```
#include <menu.h>

int pos_menu_cursor(MENU *menu);
```

DESCRIPTION

The function *pos_menu_cursor()* restores the cursor to the current position associated with the menu's selected item. This is useful after **curses** routines have been called to do screen-painting in response to a menu select.

RETURN VALUE

This routine returns one of the following:

- E_OK The routine succeeded.
- E_SYSTEM_ERROR
 System error occurred (see *errno*).
- E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
- E_NOT_POSTED
 The menu has not been posted.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`menu_driver()` – command-processing loop of the menu system

SYNOPSIS

```
#include <menu.h>

int menu_driver(MENU *menu, int c);
```

DESCRIPTION

Once a menu has been posted (displayed), you should funnel input events to it through `menu_driver()`. This routine has two major input cases; either the input is a menu navigation request or it's a printable ASCII character. The menu driver requests are as follows:

<code>REQ_LEFT_ITEM</code>	Move left to an item.
<code>REQ_RIGHT_ITEM</code>	Move right to an item.
<code>REQ_UP_ITEM</code>	Move up to an item.
<code>REQ_DOWN_ITEM</code>	Move down to an item.
<code>REQ_SCR_ULINE</code>	Scroll up a line.
<code>REQ_SCR_DLINE</code>	Scroll down a line.
<code>REQ_SCR_DPAGE</code>	Scroll down a page.
<code>REQ_SCR_UPAGE</code>	Scroll up a page.
<code>REQ_FIRST_ITEM</code>	Move to the first item.
<code>REQ_LAST_ITEM</code>	Move to the last item.
<code>REQ_NEXT_ITEM</code>	Move to the next item.
<code>REQ_PREV_ITEM</code>	Move to the previous item.
<code>REQ_TOGGLE_ITEM</code>	Select / deselect an item.
<code>REQ_CLEAR_PATTERN</code>	Clear the menu pattern buffer.
<code>REQ_BACK_PATTERN</code>	Delete the previous character from the pattern buffer.
<code>REQ_NEXT_MATCH</code>	Move to the next item matching the pattern match.
<code>REQ_PREV_MATCH</code>	Move to the previous item matching the pattern match.

If the second argument is a printable ASCII character, the code appends it to the pattern buffer and attempts to move to the next item matching the new pattern. If there is no such match, `menu_driver()` returns `E_NO_MATCH` and deletes the appended character from the buffer.

If the second argument is one of the above pre-defined requests, the corresponding action is performed.

If the second argument is neither printable ASCII nor one of the above pre-defined menu requests, the driver assumes it is an application-specific command and returns `E_UNKNOWN_COMMAND`. Application-defined commands should be defined relative to `MAX_COMMAND`, the maximum value of these pre-defined requests.

RETURN VALUE

`menu_driver()` return one of the following error codes:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`

System error occurred (see *errno*).

E_BAD_ARGUMENT

Routine detected an incorrect or out-of-range argument.

E_BAD_STATE

Routine was called from an initialization or termination function.

E_NOT_POSTED

The menu has not been posted.

E_UNKNOWN_COMMAND

The menu driver code saw an unknown request code.

E_NO_MATCH

Character failed to match.

E_REQUEST_DENIED

The menu driver could not process the request.

SEE ALSO

curses, menus.

NOTES

The header file **<menu.h>** automatically includes the header files **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_menu_format(), menu_format() – set and get menu sizes

SYNOPSIS

```
#include <menu.h>
```

```
int set_menu_format(MENU *menu, int rows, int cols);
```

```
int menu_format(MENU *menu, int *rows, int *cols);
```

DESCRIPTION

The function *set_menu_format()* sets the maximum display size of the given window. If this size is too small to display all menu items, the menu will be made scrollable.

The default format is 16 rows, 1 column. Calling *set_menu_format()* with a null menu pointer will change this default. A zero row or column argument to *set_menu_format()* is interpreted as a request not to change the current value.

The function *get_menu_format()* returns the maximum-size constraints for the given menu into the storage addressed by *rows* and *cols*.

RETURN VALUE

These routines returns one of the following:

E_OK The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

E_BAD_ARGUMENT

Routine detected an incorrect or out-of-range argument.

E_POSTED

The menu is already posted.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_item_init(), item_init(), set_item_term(), item_term(), set_menu_init(), menu_init(), set_menu_term(), menu_term() – set hooks for automatic invocation by applications

SYNOPSIS

```
#include <menu.h>

int set_item_init(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) item_init(MWNU *menu);
int set_item_term(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) item_term(MWNU *menu);
int set_menu_init(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) menu_init(MWNU *menu);
int set_menu_term(MENU *menu, void (*func)(MENU *));
void (*)(MENU *) menu_term(MWNU *menu);
```

DESCRIPTION

These functions make it possible to set hook functions to be called at various points in the automatic processing of input event codes by *menu_driver()*.

The function *set_item_init()* sets a hook to be called at menu-post time and each time the selected item changes (after the change). *item_init()* returns the current item init hook, if any (NULL if there is no such hook).

The function *set_item_term()* sets a hook to be called at menu-unpost time and each time the selected item changes (before the change). *item_term()* returns the current item term hook, if any (NULL if there is no such hook).

The function *set_menu_init()* sets a hook to be called at menu-post time and just after the top row on the menu changes once it is posted. *menu_init()* returns the current menu init hook, if any (NULL if there is no such hook).

The function *set_menu_term()* sets a hook to be called at menu-unpost time and just before the top row on the menu changes once it is posted. *menu_term()* returns the current menu term hook, if any (NULL if there is no such hook).

RETURN VALUE

Routines that return pointers return NULL on error. Other routines return one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see *errno*).

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`set_menu_items()`, `menu_items()`, `item_count()` – make and break connections between items and menus

SYNOPSIS

```
#include <menu.h>

int set_menu_items(MENU *menu, ITEM **items);
ITEM **menu_items(MENU *menu);
int item_count(MENU *menu);
```

DESCRIPTION

The function `set_menu_items()` changes the item pointer array of the given *menu*. The array must be terminated by a NULL.

The function `menu_items()` returns the item array of the given menu.

The function `item_count()` returns the count of items in *menu*.

RETURN VALUES

The function `menu_items()` returns NULL on error.

The function `item_count()` returns ERR (the general **curses** error return value) on error.

The function `set_menu_items()` returns one of the following codes on error:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see *errno*).
E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
E_POSTED
 The menu is already posted.
E_NOT_CONNECTED
 No items are connected to the menu.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

The SVr4 menu library documentation specifies the `item_count()` error value as -1 (which is the value of ERR).

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

menu_mark(), set_menu_mark() – get and set the menu mark string

SYNOPSIS

```
#include <menu.h>
int set_menu_mark(MENU *menu, char *mark);
char *menu_mark(MENU *menu);
```

DESCRIPTION

In order to make menu selections visible on older terminals without highlighting or color capability, the menu library marks selected items in a menu with a prefix string.

The function `set_menu_mark()` sets the mark string for the given menu. Calling `set_menu_mark()` with a null menu item will abolish the mark string. Note that changing the length of the mark string for a menu while the menu is posted is likely to produce unhelpful behavior.

The default string is "-" (a dash). Calling `set_menu_mark()` with a NULL menu argument will change this default.

The function `menu_mark()` returns the menu's mark string (or NULL if there is none).

RETURN VALUE

The function `menu_mark()` returns NULL on error. The function `set_menu_mark()` may return the following error codes:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR` System error occurred (see *errno*).
`E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`new_menu()`, `free_menu()` – create and destroy menus

SYNOPSIS

```
#include <menu.h>

MENU *new_menu(ITEM **items);
char *free_menu(MENU *menu);
```

DESCRIPTION

The function `new_menu()` creates a new menu connected to a specified item pointer array (which must be NULL-terminated).

The function `free_menu()` disconnects `menu` from its item array and frees the storage allocated for the menu.

RETURN VALUE

The function `new_menu()` returns NULL on error.

The function `free_menu()` returns one of the following:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).
`E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.
`E_POSTED`
 The menu has already been posted.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`set_menu_opts()`, `menu_opts_on()`, `menu_opts_off()`, `menu_opts()` – set and get menu options

SYNOPSIS

```
#include <menu.h>

int set_menu_opts(MENU *menu, OPTIONS opts);
int menu_opts_on(MENU *menu, OPTIONS opts);
int menu_opts_off(MENU *menu, OPTIONS opts);
OPTIONS menu_opts(MENU *menu);
```

DESCRIPTION

The function `fset_menu_opts()` sets all the given menu's option bits (menu option bits may be logically-ored together).

The function `fmenu_opts_on()` turns on the given option bits, and leaves others alone.

The function `fmenu_opts_off()` turns off the given option bits, and leaves others alone.

The function `fmenu_opts()` returns the menu's current option bits.

The following options are defined (all are on by default):

<code>O_ONEVALUE</code>	Only one item can be selected for this menu.
<code>O_SHOWDESC</code>	Display the item descriptions when the menu is posted.
<code>O_ROWMAJOR</code>	Display the menu in row-major order.
<code>O_IGNORECASE</code>	Ignore the case when pattern-matching.
<code>O_SHOWMATCH</code>	Move the cursor to within the item name while pattern-matching.
<code>O_NONCYCLIC</code>	On next-item and previous-item requests, don't wrap around to the other end of the menu.

RETURN VALUE

Except for `menu_opts()`, each routine returns one of the following:

<code>E_OK</code>	The routine succeeded.
<code>E_SYSTEM_ERROR</code>	System error occurred (see <i>errno</i>).
<code>E_POSTED</code>	The menu is already posted.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_menu_pattern(), menu_pattern() – get and set a menu's pattern buffer

SYNOPSIS

```
#include <menu.h>

int set_menu_pattern(MENU *menu, char *pattern);
char *menu_pattern(MENU *menu);
```

DESCRIPTION

Every menu has an associated pattern match buffer. As input events that are printable ASCII characters come in, they are appended to this match buffer and tested for a match, as described in *menu_driver*.

The function *set_menu_pattern()* sets the pattern buffer for the given menu and tries to find the first matching item. If it succeeds, that item becomes current; if not, the current item does not change.

The function *menu_pattern()* returns the pattern buffer of the given *menu*.

RETURN VALUE

The function *menu_pattern()* returns NULL on error. The function *set_menu_pattern()* may return the following error codes:

E_OK The routine succeeded.
E_SYSTEM_ERROR System error occurred (see *errno*).
E_BAD_ARGUMENT Routine detected an incorrect or out-of-range argument.
E_NO_MATCH Character failed to match.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

post_menu(), unpost_menu() – write or erase menus from associated subwindows

SYNOPSIS

```
#include <menu.h>

int post_menu(MENU *menu);
int unpost_menu(MENU *menu);
```

DESCRIPTION

The function *post_menu()* displays a menu to its associated subwindow. To trigger physical display of the subwindow, use *refresh()* or some equivalent **curses** routine (the implicit *doupdate()* triggered by an **curses** input request will do).

The function *unpost_menu()* erases menu from its associated subwindow.

RETURN VALUE

These routines return one of the following:

- E_OK The routine succeeded.
- E_SYSTEM_ERROR
 System error occurred (see *errno*).
- E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
- E_POSTED
 The menu has already been posted.
- E_BAD_STATE
 Routine was called from an initialization or termination function.
- E_NO_ROOM
 Menu is too large for its window.
- E_NOT_POSTED
 The menu has not been posted.
- E_NOT_CONNECTED
 No items are connected to the menu.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_menu_userptr(), menu_userptr() – associate application data with a menu item

SYNOPSIS

```
#include <menu.h>

int set_menu_userptr(MENU *menu, char *userptr);
char menu_userptr(MENU *menu);
```

DESCRIPTION

Every menu and every menu item has a field that can be used to hold application-specific data (that is, the menu-driver code leaves it alone). These functions get and set the menu user pointer field.

RETURN VALUE

The function `menu_userptrreturns()` NULL

The function `set_menu_userptr()` returns one of the following:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

The user pointer should be a void pointer. We leave it as a char pointer for SVr4 compatibility.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_menu_window(), menu_window(), set_menu_sub(), menu_sub(), scale_window() –
make and break menu window and subwindow associations

SYNOPSIS

```
#include <menu.h>

int set_menu_window(MENU *menu, WINDOW *win);
WINDOW *menu_window(MENU *menu);
int set_menu_sub(MENU *menu, WINDOW *sub);
WINDOW *menu_sub(MENU *menu);
int scale_window(MENU *menu, int *rows,
                 int *columns);
```

DESCRIPTION

Every menu has an associated pair of **curses** windows. The menu window displays any title and border associated with the window; the menu subwindow displays the items of the menu that are currently available for selection.

The first four functions get and set those windows. It is not necessary to set either window; by default, the driver code uses *stdscr* for both.

In the *set_* functions, window argument of NULL is treated as though it were *stdscr*. A menu argument of NULL is treated as a request to change the system default menu window or subwindow.

The function *scale_window()* returns the minimum size required for the subwindow of *menu*.

RETURN VALUE

Routines that return pointers return NULL on error. Routines that return an integer return one of the following error codes:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see *errno*).
E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
E_POSTED
 The menu has already been posted.
E_NOT_CONNECTED
 No items are connected to the menu.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`set_current_item()`, `current_item()`, `set_top_row()`, `top_row()`, `item_index()` – set and get current menu item

SYNOPSIS

```
#include <menu.h>

int set_current_item(MENU *menu, ITEM *item);
ITEM *current_item(MENU *menu);
int set_top_row(MENU *menu, int row);
int top_row(MENU *menu);
int item_index(ITEM *item);
```

DESCRIPTION

The function `set_current_item()` sets the current item (the item on which the menu cursor is positioned). `current_item()` returns a pointer to the current item in the given menu.

The function `set_top_row()` sets the top row of the menu to show the given row (the top row is initially 0, and is reset to this value whenever the `O_ROWMAJOR` option is toggled). The item leftmost on the given row becomes current. The function `top_row()` returns the number of the top menu row being displayed.

The function `item_index()` returns the (zero-origin) index of `item` in the menu's item pointer list.

RETURN VALUE

`current_item()` returns NULL on error.

`top_row()` and `item_index()` return ERR (the general **curses** error value) on error.

`set_current_item()` and `set_top_row()` return one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR
 System error occurred (see *errno*).
E_BAD_ARGUMENT
 Routine detected an incorrect or out-of-range argument.
E_BAD_STATE
 Routine was called from an initialization or termination function.
E_NOT_CONNECTED
 No items are connected to the menu.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

The SVr4 menu library documentation specifies the `top_row()` and `item_index()` error value as -1 (which is the value of ERR).

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`item_name()`, `item_description()` – get menu item name and description fields

SYNOPSIS

```
#include <menu.h>

char *item_name(ITEM *item);
char *item_description(ITEM *item);
```

DESCRIPTION

The function `item_name()` returns the name part of the given item. The function `item_description()` returns the description part of the given item.

RETURN VALUE

These routines returns NULL on error.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

`new_item()`, `free_item()` – create and destroy menu items

SYNOPSIS

```
#include <menu.h>

ITEM *new_item(char *name, char *description);
int free_item(ITEM *item);
```

DESCRIPTION

The function `new_item()` allocates a new item and initializes it from the *name* and *description* pointers. It does not make its own copy of these items.

The function `free_item()` de-allocates an item.

RETURN VALUE

The function `new_item()` returns NULL on error.

The function `free_item()` returns one of the following:

`E_OK` The routine succeeded.
`E_SYSTEM_ERROR`
 System error occurred (see *errno*).
`E_BAD_ARGUMENT`
 Routine detected an incorrect or out-of-range argument.
`E_NOT_CONNECTED`
 No items are connected to the menu.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_item_opts(), item_opts_on(), item_opts_off(), item_opts() – set and get menu item options

SYNOPSIS

```
#include <menu.h>

int set_item_opts(ITEM *item, OPTIONS opts);
int item_opts_on(ITEM *item, OPTIONS opts);
int item_opts_off(ITEM *item, OPTIONS opts);
OPTIONS item_opts(ITEM *item);
```

DESCRIPTION

The function *set_item_opts()* sets all the given item's option bits (menu option bits may be logically-ored together).

The function *item_opts_on()* turns on the given option bits, and leaves others alone.

The function *item_opts_off()* turns off the given option bits, and leaves others alone.

The function *item_opts()* returns the item's current option bits.

There is only one defined option bit mask, O_SELECTABLE. When this is on, the item may be selected during menu processing. This option defaults to on.

RETURN VALUE

Except for *item_opts()*, each routine returns one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR System error occurred (see *errno*).

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_item_userptr(), item_userptr() – associate application data with a menu item

SYNOPSIS

```
#include <menu.h>

int set_item_userptr(ITEM *item, char *userptr);
char item_userptr(ITEM *item);
```

DESCRIPTION

Every menu item has a field that can be used to hold application-specific data (that is, the menu-driver code leaves it alone). These functions get and set that field.

RETURN VALUE

item_userptr() returns NULL on error. *set_item_userptr()* returns one of the following:

E_OK The routine succeeded.
E_SYSTEM_ERROR System error occurred (see *errno*).

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

The user pointer should be a **void** pointer. We leave it as a **char** pointer for SVr4 compatibility.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

set_item_value(), item_value() – set and get menu item values

SYNOPSIS

```
#include <menu.h>

char *set_item_value(ITEM *item, int bool);
char *item_value(ITEM *item);
```

DESCRIPTION

If you turn off the menu option O_ONEVALUE (e.g., with *set_menu_opts()* or *menu_opts_off()*), the menu becomes multi-valued; that is, more than one item may simultaneously be selected.

In a multi_valued menu, you can used to select the given menu item (second argument TRUE) or deselect it (second argument FALSE).

RETURN VALUE

The function *set_item_value()* returns one of the following:

E_OK The routine succeeded.

E_SYSTEM_ERROR

System error occurred (see *errno*).

E_REQUEST_DENIED

The menu driver could not process the request.

SEE ALSO

curses, *menus*, *menu_opts*

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

item_visible() – check visibility of a menu item

SYNOPSIS

```
#include <menu.h>

int item_visible(ITEM *item);
```

DESCRIPTION

A menu item is visible when it is in the portion of a posted menu that is mapped onto the screen (if the menu is scrollable, in particular, this portion will be smaller than the whole menu).

RETURN VALUES

The function *item_visible()* returns TRUE if *item* is visible, and FALSE if it is not.

SEE ALSO

curses, *menus*.

NOTES

The header file **<menu.h>** automatically includes the header file **<curses.h>**.

PORTABILITY

These routines emulate the System V menu library. They were not supported on Version 7 or BSD versions.

AUTHORS

Juergen Pfeifer. Manual pages and adaptation for new curses by Eric S. Raymond.

NAME

bottom_panel(), del_panel(), hide_panel(), move_panel(), new_panel(), panel_above(), panel_below(), panel_hidden(), panel_userptr(), panel_window(), replace_panel(), set_panel_userptr(), show_panel(), top_panel(), update_panels() panel – panel stack extension for curses

SYNOPSIS

```
#include <panel.h>

cc [flags] sourcefiles -lpanel -lcurses

PANEL *new_panel(WINDOW *win)
int bottom_panel(PANEL *pan)
int top_panel(PANEL *pan)
int show_panel(PANEL *pan)
void update_panels(void);
int hide_panel(PANEL *pan)
WINDOW *panel_window(PANEL *pan)
int replace_panel(PANEL *pan, WINDOW *window)
int move_panel(PANEL *pan, int starty, int startx)
int panel_hidden(PANEL *pan)
PANEL *panel_above(PANEL *pan)
PANEL *panel_below(PANEL *pan)
int set_panel_userptr(PANEL *pan, char *ptr)
char *panel_userptr(PANEL *pan)
int del_panel(PANEL *pan)
```

DESCRIPTION

Panels are *curses* windows with the added feature of depth. Panel functions allow the use of stacked windows and ensure the proper portions of each window and the curses *stdscr* window are hidden or displayed when panels are added, moved, modified or removed. The set of currently visible panels is the stack of panels. The *stdscr* window is beneath all panels, and is not considered part of the stack.

A window is associated with every panel. The panel routines enable you to create, move, hides, and show panels, as well as position a panel at any desired location in the stack.

Panel routines are a functional layer added to **curses**, make only high-level curses calls, and work anywhere terminfo curses does.

FUNCTIONS

The following functions are part of the **panel** package.

```
new_panel(WINDOW
    allocates a PANEL structure, associates it with win,
    places the panel on the top of the stack (causes it to
    be displayed above any other panel) and returns a pointer
    to the new panel.

void
    refreshes the virtual screen to reflect the relations
    between the panels in the stack, but does not call
    doupdate() to refresh the physical screen. Use this
    function and not wrefresh() or wnoutrefresh(). update_panels()
```

may be called more than once before a call to *doupdate()*, but *doupdate()* is the function responsible for updating the physical screen.

del_panel(PANEL)
removes the given panel from the stack and deallocates the **PANEL** structure (but not its associated window).

hide_panel(PANEL)
removes the given panel from the panel stack and thus hides it from view. The **PANEL** structure is not lost, merely removed from the stack.

show_panel(PANEL)
makes a hidden panel visible by placing it on top of the panels in the panel stack. See COMPATIBILITY below.

top_panel(PANEL)
puts the given visible panel on top of all panels in the stack. See COMPATIBILITY below.

bottom_panel(PANEL)
puts panel at the bottom of all panels.

move_panel(PANEL)
moves the given panel window so that its upper-left corner is at *starty*, *startx*. It does not change the position of the panel in the stack. Be sure to use this function, not *mvwin()*, to move a panel window.

replace_panel(PANEL)
replaces the current window of panel with *window* (useful, for example if you want to resize a panel; if you're using **ncurses**, you can call *replace_panel()* on the output of *wresize()*). It does not change the position of the panel in the stack.

panel_above(PANEL)
returns a pointer to the panel above *pan*. If the panel argument is **(PANEL *)0**, it returns a pointer to the bottom panel in the stack.

panel_below(PANEL)
returns a pointer to the panel just below *pan*. If the panel argument is **(PANEL *)0**, it returns a pointer to the top panel in the stack.

set_panel_userptr(*pan*,*ptr*)
sets the panel's user pointer.

panel_userptr(PANEL)
returns the user pointer for a given panel.

panel_window(PANEL)
returns a pointer to the window of the given panel.

DIAGNOSTICS

Each routine that returns a pointer returns NULL if an error occurs. Each routine that returns an int value returns OK if it executes successfully and ERR if not.

COMPATIBILITY

Reasonable care has been taken to ensure compatibility with the native panel facility introduced in SVr3.2 (inspection of the SVr4 manual pages suggests the programming interface is unchanged). The **PANEL** data structures are merely similar. The programmer is cautioned not to directly use **PANEL** fields.

The functions *show_panel()* and *top_panel()* are identical in this implementation, and work equally well with displayed or hidden panels. In the native System V implementation, *show_panel()* is intended for making a hidden panel visible (at the top of the stack) and *top_panel()* is intended for making an already-visible panel move to the top of the stack. You are cautioned to use the correct function to ensure compatibility with native panel libraries.

NOTE

In your library list, **libpanel.a** should be before **libncurses.a**; that is, you want to say **-lpanel -lncurses**, not the other way around (which would give you a link error using some linkers).

FILES

The **panel** package makes use of the following files:

panel.h interface for the panels library
libpanel.a
the panels library itself

SEE ALSO

curses

AUTHOR

Originally written by Warren Tucker <wht@n4hgf.mt-park.ga.us>, primarily to assist in porting u386mon to systems without a native panels library. Repackaged for ncurses by Zeyd ben-Halim.