Writing Programs with NCURSES

---

https://invisible-island.net/ncurses/

Writing Programs with NCURSES

by Eric S. Raymond and Zeyd M. Ben-Halim
updates since release 1.9.9e by Thomas Dickey

Contents

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 screen full 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 UNIX desktops increasingly dominated by X, Motif, and Tcl/Tk. Nevertheless, UNIX still supports tty lines and X supports *xterm(1)*; 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.

A Brief History of Curses

Historically, the first ancestor of `curses` was the routines written to provide screen-handling for the `vi` editor; these used the `termcap` database facility (both released in 3BSD) for describing terminal capabilities. These routines were abstracted into a documented library and first released with the early BSD UNIX versions. All of this work was done by students at the University of California (Berkeley campus). The curses library was first published in 4.0BSD, a year after 3BSD (i.e., late 1980).

After graduation, one of those students went to work at AT&T Bell Labs, and made an improved `termcap` library called `terminfo` (i.e., "libterm"), and adapted the curses library to use this. That was subsequently released in System V Release 2 (early 1984). Thereafter, other developers added to the curses and terminfo libraries. For instance, a student at Cornell University wrote an improved terminfo library as well as a tool (`tic`) to compile the terminal descriptions. As a general rule, AT&T did not identify the developers in the source-code or documentation; the `tic` and `infocmp` programs are the exceptions.

System V Release 3 (System III UNIX) from Bell Labs featured a rewritten and much-improved `curses` library, along with the `tic` program (late 1986).

To recap, 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.

Scope of This Document

This document describes `ncurses`, a free 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 original maintainer of this 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. Jürgen Pfeifer wrote all of the menu and forms code as well as the Ada95 binding. Ongoing work is being done by Thomas Dickey (maintainer). Contact the current maintainers at bug-ncurses@gnu.org.

This document also describes the panels 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.

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.

The Curses Library

An Overview of Curses

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 `-lncurses` in your LDFLAGS or on the command line. There is no need for any other libraries.

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 does not bear any necessary relation to what is really on the terminal screen; it is 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.

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.

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 failure (-1).
`OK` error flag returned by routines when things go right.

Using the 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 <stdlib.h>
#include <curses.h>
#include <signal.h>

static void finish(int sig);

int
main(int argc, char *argv[])
{
    int num = 0;

    /* 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) echo();         /* echo input - in color */

    if (has_colors())
    {
        start_color();

        /*
         * Simple color assignment, often all we need.  Color pair 0 cannot
         * be redefined.  This example uses the same value for the color
         * pair as for the foreground color, though of course that is not
         * necessary:
         */
        init_pair(1, COLOR_RED,     COLOR_BLACK);
        init_pair(2, COLOR_GREEN,   COLOR_BLACK);
        init_pair(3, COLOR_YELLOW,  COLOR_BLACK);
        init_pair(4, COLOR_BLUE,    COLOR_BLACK);
        init_pair(5, COLOR_CYAN,    COLOR_BLACK);
        init_pair(6, COLOR_MAGENTA, COLOR_BLACK);
        init_pair(7, COLOR_WHITE,   COLOR_BLACK);
    }

    for (;;)
    {
        int c = getch();     /* refresh, accept single keystroke of input */
        attrset(COLOR_PAIR(num % 8));
        num++;

        /* process the command keystroke */
    }

    finish(0);               /* we are done */
}

static void finish(int sig)
{
    endwin();

    /* do your non-curses wrapup here */

    exit(0);
}
```

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 is not 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.

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 get messed up.

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 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()`(3)-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 `curses.h` The mapping from sequences to `#define` values is determined by `key_` capabilities in the terminal's terminfo entry.

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 `curses.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, `curses.h` will map them to a recognizable (though ugly) set of ASCII defaults.

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 `curses.h` uses to represent the contents of a screen cell. See the `curses.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 have 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.

Mouse Interfacing

The `ncurses` library also provides a mouse interface.

> **NOTE:** this facility is specific to `ncurses`, it is not part of either the XSI Curses standard, nor of System V Release 4, nor BSD curses. System V Release 4 curses contains code with similar interface definitions, however it is not documented. Other than by disassembling the library, we have no way to determine exactly how that mouse code works. 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 in the following environments:

- xterm and similar programs such as rxvt.
- Linux console, when configured with `gpm`(1), Alessandro Rubini's mouse server.
- FreeBSD sysmouse (console)
- OS/2 EMX

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 will 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 touch-screens 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 do not 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(3X)` for full details of the mouse-interface functions.

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.

Function Descriptions

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

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()`, `ripoffline()`, `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. (NOTE: a SCREEN variable is not a *screen* in the sense we are describing in this introduction, but a collection of parameters used to assist in optimizing the display.) 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 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.

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).

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 UNIX 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 will have to save it yourself if need be.

Debugging

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

`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 `curses.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. 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.

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.

Some Notes of Caution

If you find yourself thinking you need to use `noraw()` or `nocbreak()`, think again and move carefully. It is 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)`. Do not 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 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.

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" 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 */
```

Using NCURSES under XTERM

A resize operation in X sends `SIGWINCH` to the application running under xterm. The easiest way to handle `SIGWINCH` is to do an `endwin`, followed by an `refresh` and a screen repaint you code yourself. The `refresh` will pick up the new screen size from the xterm's environment.

That is the standard way, of course (it even works with some vendor's curses implementations). Its drawback is that it clears the screen to reinitialize the display, and does not resize subwindows which must be shrunk. `Ncurses` provides an extension which works better, the `resizeterm` function. That function ensures that all windows are limited to the new screen dimensions, and pads `stdscr` with blanks if the screen is larger.

The `ncurses` library provides a SIGWINCH signal handler, which pushes a `KEY_RESIZE` via the wgetch() calls. When `ncurses` returns that code, it calls `resizeterm` to update the size of the standard screen's window, repainting that (filling with blanks or truncating as needed). It also resizes other windows, but its effect may be less satisfactory because it cannot know how you want the screen re-painted. You will usually have to write special-purpose code to handle `KEY_RESIZE` yourself.

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.

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`.

Tuning for Speed

Use the `addchstr()` family of functions for fast screen-painting of text when you know the text does not contain any control characters. Try to make attribute changes infrequent on your screens. Do not use the `immedok()` option!

Special Features of NCURSES

The `wresize()` function allows you to resize a window in place. The associated `resizeterm()` function simplifies the construction of SIGWINCH handlers, for resizing all windows.

The `define_key()` function allows you to define at runtime function-key control sequences which are not in the terminal description. The `keyok()` function allows you to temporarily enable or disable interpretation of any function-key control sequence.

The `use_default_colors()` function allows you to construct applications which can use the terminal's default foreground and background colors as an additional "default" color. Several terminal emulators support this feature, which is based on ISO 6429.

Ncurses supports up 16 colors, unlike SVr4 curses which defines only 8. While most terminals which provide color allow only 8 colors, about a quarter (including XFree86 xterm) support 16 colors.

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.

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 is 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.

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.

XSI Curses Conformance

The ncurses library is intended to be base-level conformant with the XSI Curses standard from X/Open. Many 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 panel code distributed with ncurses.

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. Many linkers are two-pass and will accept either order, but it is still good practice to put -lpanel first and -lncurses second.

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 will 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 does not 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 is not sufficient because it does not 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 will generate a lot of unnecessary refresh activity and screen flicker.

Panels, Input, and the Standard Screen

You should not mix `wnoutrefresh()` or `wrefresh()` operations with panels code; this will work only if the argument window is either in the top panel or unobscured by any other panels.

The `stsdcr` 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 unobscured.

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

Hiding Panels

It is possible to remove a panel from the deck temporarily; use `hide_panel` for this. Use `show_panel()` to render it visible again. 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.

Miscellaneous Other Facilities

It is 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 `menu` code distributed with `ncurses`.

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`. Many linkers are two-pass and will accept either order, but it is still good practice to put `-lmenu` first and `-lncurses` second.

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 by 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 `post_menu()`.
5. Refresh the screen.
6. Process user requests via an input loop.
7. Unpost the menu using `unpost_menu()`.
8. Free the menu, using `free_menu()`.
9. Free the items using `free_item()`.
10. Terminate `curses`.

Selecting items

Menus may be multi-valued or (the default) single-valued (see the manual page `menu_opts(3x)` 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 unselectable 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.

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(3x)` 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_attribs(3x)` manual page.

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(3x)`.

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

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(3x)`.

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 will 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_PATTERN, 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.

Miscellaneous Other Features

Various menu options can affect the processing and visual appearance and input processing of menus. See menu_opts(3x) 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(3x). 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(3x).

Each item, and each menu, has an associated user pointer on which you can hang application data. See mitem_userptr(3x) and menu_userptr(3x).

The Forms 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 `form` code distributed with `ncurses`.

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`. Many linkers are two-pass and will accept either order, but it is still good practice to put `-lform` first and `-lncurses` second.

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 `post_form()`.
5. Refresh the screen.
6. Process user requests via an input loop.
7. Unpost the form using `unpost_form()`.
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.

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 mist 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 have done an explicit `set_form_win()` 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 `((height + offscreen)*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 `form_field_new(3X)` 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.

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.

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.

Changing the Field Location

It 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()`.

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`.

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 */

chtype field_pad(FIELD *field);

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

chtype new_page(FIELD *field);            /* field to query */
```

The attributes set and returned by the first four functions are normal `curses(3x)` 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.

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 attr);            /* attribute to set */

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

int field_opts_off(FIELD *field,        /* field to alter */
                   int attr);           /* 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.

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 is not 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.

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.

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.

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 does not 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:

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 will want to set this to the field width; if it is greater than the field width, the validation check will always fail. A minimum width of zero makes field completion optional.

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 will want to set this to the field width; if it is greater than the field width, the validation check will always fail. A minimum width of zero makes field completion optional.

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.

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(3)`.

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 */
                   double vmin, double vmax); /* valid range */
```

Valid characters consist of an optional leading minus and digits. possibly including a decimal point. If your system supports locale's, the decimal point character used must be the one defined by your locale. 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(3)`.

TYPE_REGEXP

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_REGEXP,           /* type to associate */
                   char *regexp);         /* expression to match */
```

The syntax for regular expressions is that of `regcomp(3)`. The check for regular-expression match is performed on exit.

Direct Field Buffer Manipulation

The chief 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 is 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 is not 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.

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 principal 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 from. It returns -1 if the form-pointer argument is NULL.

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 will 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 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.

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.

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.

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.

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.

Scrolling Requests

Fields that are dynamic and have grown and fields explicitly created with off-screen 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.

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`. `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.

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.

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.

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.

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.

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.

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.

Union Types

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

```
FIELD *link_fieldtype(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, than 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.

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 is how you do that:

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

FIELDTYPE *new_fieldtype(HOOK f_validate, /* field validator */
                         HOOK c_validate) /* character validator */

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

At least one of the arguments of `new_fieldtype()` 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_fieldtype()` 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.

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 will 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 *type,    /* type to alter */
```

```
                        PTRHOOK make_str,   /* make structure from args */
                        PTRHOOK copy_str,   /* make copy of structure */
                        VOIDHOOK free_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 file pointer and need not check specially for it.

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 is how:

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

int set_fieldtype_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.

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 exempted from the package copyright 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.