NCURSES Programming HOWTO

Pradeep Padala

ppadala@gmail.com

Thomas E. Dickey

dickey@invisible-island.net

v2.1, 2024-09-08

Revision History

Revision 2.1 2024-09-08 Revised by: dickey
Fixes for the sample programs. Remove obsolete mailing addresses. Update publication date.
Revision 2.0 2022-12-03 Revised by: dickey
Fixes for the sample programs, Correct documentation errata.
Revision 1.9 2005-06-20 Revised by: ppadala
The license has been changed to the MIT-style license used by NCURSES. Note that the programs are also re-licensed under this.
Revision 1.8 2005-06-17 Revised by: ppadala
Lots of updates. Added references and perl examples. Changes to examples. Many grammatical and stylistic changes to the content. Changes to NCURSES history.
Revision 1.7.1 2002-06-25 Revised by: ppadala
Added a README file for building and instructions for building from source.
Revision 1.7 2002-06-25 Revised by: ppadala
Added "Other formats" section and made a lot of fancy changes to the programs. Inlining of programs is gone.
Revision 1.6.1 2002-02-24 Revised by: ppadala
Removed the old Changelog section, cleaned the makefiles
Revision 1.6 2002-02-16 Revised by: ppadala
Corrected a lot of spelling mistakes, added ACS variables section
Revision 1.5 2002-01-05 Revised by: ppadala
Changed structure to present proper TOC
Revision 1.3.1 2001-07-26 Revised by: ppadala
Corrected maintainers paragraph, Corrected stable release number
Revision 1.3 2001-07-24 Revised by: ppadala
Added copyright notices to main document (LDP license) and programs (GPL), Corrected printw_example.
Revision 1.2 2001-06-05 Revised by: ppadala
Incorporated ravi's changes. Mainly to introduction, menu, form, justforfun sections
Revision 1.1 2001-05-22 Revised by: ppadala
Added "a word about window" section, Added scanw_example.

This document is intended to be an "All in One" guide for programming with ncurses and its sister libraries. We graduate from a simple "Hello World" program to more complex form manipulation. No prior experience in ncurses is assumed. Send comments to this address (mailto:bug-ncurses@gnu.org)

1. Introduction

In the olden days of teletype terminals, terminals were away from computers and were connected to them through serial cables. The terminals could be configured by sending a series of bytes. All the capabilities (such as moving the cursor to a new location, erasing part of the screen, scrolling the screen, changing modes, etc.) of terminals could be accessed through these series of bytes. These control sequences are usually called escape sequences, because they start with an escape(0x1B) character. Even today, with proper emulation, we can send escape sequences to the emulator and achieve the same effect on a terminal window.

Suppose you wanted to print a line in color. Try typing this on your console.

```
echo "^[[0;31;40mIn Color"
```

The first character is an escape character, which looks like two characters ^ and [. To be able to print it, you have to press CTRL+V and then the ESC key. All the others are normal printable characters. You should be able to see the string "In Color" in red. It stays that way and to revert back to the original mode type this.

```
echo "^[[0;37;40m"
```

Now, what do these magic characters mean? Difficult to comprehend? They might even be different for different terminals. So the designers of UNIX invented a mechanism named `termcap`. It is a file that lists all the capabilities of a particular terminal, along with the escape sequences needed to achieve a particular effect. In the later years, this was replaced by `terminfo`. Without delving too much into details, this mechanism allows application programs to query the terminfo database and obtain the control characters to be sent to a terminal or terminal emulator.

1.1. What is NCURSES?

You might be wondering, what the import of all this technical gibberish is. In the above scenario, every application program is supposed to query the terminfo and perform the necessary stuff (sending control characters, etc.). It soon became difficult to manage this complexity and this gave birth to 'CURSES'. Curses is a pun on the name "cursor optimization". The Curses library forms a wrapper over working with raw terminal codes, and provides highly flexible and efficient API (Application Programming Interface). It provides functions to move the cursor, create windows, produce colors, play with mouse, etc. The application programs need not worry about the underlying terminal capabilities.

So what is NCURSES? NCURSES is a clone of the original System V Release 4.0 (SVr4) curses. It is a freely distributable library, fully compatible with older version of curses. In short, it is a library of functions that manages an application's display on character-cell terminals. In the remainder of the document, the terms curses and ncurses are used interchangeably.

A detailed history of NCURSES can be found in the NEWS file from the source distribution. The current package is maintained by Thomas Dickey (mailto:dickey@invisible-island.net). You can contact the maintainers at bug-ncurses@gnu.org (mailto:bug-ncurses@gnu.org).

1.2. What we can do with NCURSES

NCURSES not only creates a wrapper over terminal capabilities, but also gives a robust framework to create nice looking UI (User Interface)s in text mode. It provides functions to create windows, etc. Its sister libraries panel, menu and form provide an extension to the basic curses library. These libraries usually come along with curses. One can create applications that contain multiple windows, menus, panels and forms. Windows can be managed independently, can provide 'scrollability' and even can be hidden.

Menus provide the user with an easy command selection option. Forms allow the creation of easy-to-use data entry and display windows. Panels extend the capabilities of ncurses to deal with overlapping and stacked windows.

These are just some of the basic things we can do with ncurses. As we move along, We will see all the capabilities of these libraries.

1.3. Where to get it

All right, now that you know what you can do with ncurses, you must be rearing to get started. NCURSES is usually shipped with your installation. In case you don't have the library or want to compile it on your own, read on.

Compiling the package

NCURSES can be obtained from

- the home page at https://invisible-island.net (https://invisible-island.net/), as well as
- https://ftp.gnu.org/pub/gnu/ncurses/ or
- any of the mirror sites mentioned in https://www.gnu.org/order/ftp.html.

Read the README and INSTALL (https://invisible-island.net/ncurses/INSTALL.html) files for details on to how to install it. It usually involves the following operations.

```
tar zxvf ncurses<version>.tar.gz  # unzip and untar the archive
cd ncurses<version>               # cd to the directory
./configure                       # configure the build according to your
                                  # environment
make                              # make it
```

```
    su root                           # become root
    make install                      # install it
```

1.4. Purpose/Scope of the document

This document is intended to be a "All in One" guide for programming with ncurses and its sister libraries. We graduate from a simple "Hello World" program to more complex form manipulation. No prior experience in ncurses is assumed. The writing is informal, but a lot of detail is provided for each of the examples.

1.5. About the Programs

All the programs in the document are available in gzipped form here (https://invisible-island.net/ncurses/howto/ncurses_programs.tar.gz). Ungzip and untar it. The directory structure looks like this.

```
ncurses
   |
   |----> JustForFun     -- just for fun programs
   |----> basics         -- basic programs
   |----> demo           -- output files go into this directory after make
   |          |
   |          |----> exe -- exe files of all example programs
   |----> forms          -- programs related to form library
   |----> menus          -- programs related to menus library
   |----> panels         -- programs related to panels library
   |----> perl           -- perl equivalents of the examples (contributed
   |                            by Anuradha Ratnaweera)
   |----> Makefile       -- the top level Makefile
   |----> README         -- the top level README file. contains instructions
   |----> COPYING        -- copyright notice
```

The individual directories contain the following files.

```
Description of files in each directory
--------------------------------------
JustForFun
    |
    |----> hanoi.c   -- The Towers of Hanoi Solver
    |----> life.c    -- The Game of Life demo
    |----> magic.c   -- An Odd Order Magic Square builder
    |----> queens.c  -- The famous N-Queens Solver
    |----> shuffle.c -- A fun game, if you have time to kill
    |----> tt.c      -- A very trivial typing tutor

  basics
    |
```

```
    |----> acs_vars.c            -- ACS_ variables example
    |----> hello_world.c         -- Simple "Hello World" Program
    |----> init_func_example.c   -- Initialization functions example
    |----> key_code.c            -- Shows the scan code of the key pressed
    |----> mouse_menu.c          -- A menu accessible by mouse
    |----> other_border.c        -- Shows usage of other border functions apa
    |                               -- rt from box()
    |----> printw_example.c      -- A very simple printw() example
    |----> scanw_example.c       -- A very simple getstr() example
    |----> simple_attr.c         -- A program that can print a c file with
    |                               -- comments in attribute
    |----> simple_color.c        -- A simple example demonstrating colors
    |----> simple_key.c          -- A menu accessible with keyboard UP, DOWN
    |                               -- arrows
    |----> temp_leave.c          -- Demonstrates temporarily leaving curses mode
    |----> win_border.c          -- Shows Creation of windows and borders
    |----> with_chgat.c          -- chgat() usage example

  forms
    |
    |----> form_attrib.c     -- Usage of field attributes
    |----> form_options.c    -- Usage of field options
    |----> form_simple.c     -- A simple form example
    |----> form_win.c        -- Demo of windows associated with forms

  menus
    |
    |----> menu_attrib.c     -- Usage of menu attributes
    |----> menu_item_data.c  -- Usage of item_name(), etc. functions
    |----> menu_multi_column.c    -- Creates multi columnar menus
    |----> menu_scroll.c     -- Demonstrates scrolling capability of menus
    |----> menu_simple.c     -- A simple menu accessed by arrow keys
    |----> menu_toggle.c     -- Creates multi valued menus and explains
    |                           -- REQ_TOGGLE_ITEM
    |----> menu_userptr.c    -- Usage of user pointer
    |----> menu_win.c        -- Demo of windows associated with menus

  panels
    |
    |----> panel_browse.c    -- Panel browsing through tab. Usage of user
    |                           -- pointer
    |----> panel_hide.c      -- Hiding and Un hiding of panels
    |----> panel_resize.c    -- Moving and resizing of panels
    |----> panel_simple.c    -- A simple panel example

  perl
    |----> 01-10.pl          -- Perl equivalents of first ten example programs
```

There is a top level Makefile included in the main directory. It builds all the files and puts the ready-to-use exes in demo/exe directory. You can also do selective make by going into the corresponding directory. Each directory contains a README file explaining the purpose of each c file in the directory.

For every example, I have included path name for the file relative to the examples directory.

All the programs are released under the same license that is used by ncurses (MIT-style). This gives you the ability to do pretty much anything other than claiming them as yours. Feel free to use them in your programs as appropriate.

1.6. Other Formats of the document

This howto is also available in other formats. Here are the links to other formats of this document.

1.6.1. Alternative formats

- Acrobat PDF Format (https://invisible-island.net/ncurses/howto/NCURSES-Programming-HOWTO.pdf)
- PostScript Format (https://invisible-island.net/ncurses/howto/NCURSES-Programming-HOWTO.ps)
- In Multiple HTML pages (https://invisible-island.net/ncurses/howto/NCURSES-Programming-HOWTO-html.tar.gz)
- A single HTML file. (https://invisible-island.net/ncurses/howto/NCURSES-Programming-HOWTO.html)

1.6.2. Building from source

The sources for this HOWTO can be retrieved from

```
https://github.com/ThomasDickey/ncurses-howto-snapshots
```

These tools were used to format the HOWTO and build the examples:

- docbook-utils (a Debian package)
- gcc

1.7. Credits

I thank *Sharath* and *Emre Akbas* for helping me with few sections. The introduction was initially written by Sharath. I rewrote it with few excerpts taken from his initial work. Emre helped in writing printw and scanw sections.

Perl equivalents of the example programs were contributed by *Anuradha Ratnaweera*.

Then comes *Ravi Parimi*, my dearest friend, who has been on this project before even one line was written. He constantly bombarded me with suggestions and patiently reviewed the whole text. He also checked each program on Linux and Solaris.

1.8. Wish List

This is the wish list, in the order of priority. If you have a wish or you want to work on completing the wish, mail me (mailto:ppadala@gmail.com).

- Add examples to last parts of forms section.
- Prepare a Demo showing all the programs and allow the user to browse through description of each program. Let the user compile and see the program in action. A dialog based interface is preferred.
- Add debug info. _tracef, _tracemouse stuff.
- Accessing termcap, terminfo using functions provided by ncurses package.
- Working on two terminals simultaneously.
- Add more stuff to miscellaneous section.

1.9. Copyright

authorization.

2. Hello World !!!

Welcome to the world of curses. Before we plunge into the library and look into its various features, let's write a simple program and say hello to the world.

2.1. Compiling With the NCURSES Library

To use ncurses library functions, you have to include ncurses.h in your programs. To link the program with ncurses the flag -lncurses should be added.

```
    #include <ncurses.h>
    .
    .
    .

    compile and link: gcc <program file> -lncurses
```

Example 1. The Hello World !!! Program

```
#include <curses.h>

int
main(void)
{
    initscr();                  /* Start curses mode              */
    printw("Hello World !!!");  /* Print Hello World              */
    refresh();                  /* Print it on to the real screen */
    getch();                    /* Wait for user input */
    endwin();                   /* End curses mode                */

    return 0;
}
```

2.2. Dissection

The above program prints "Hello World !!!" to the screen and exits. This program shows how to initialize curses and do screen manipulation and end curses mode. Let's dissect it line by line.

2.2.1. About initscr()

The function initscr() initializes the terminal in curses mode. In some implementations, it clears the screen and presents a blank screen. To do any screen manipulation using curses package this has to be called first. This function initializes the curses system and allocates memory for our present window (called `stdscr`) and some other data-structures. Under extreme cases this function might fail due to insufficient memory to allocate memory for curses library's data structures.

After this is done, we can do a variety of initializations to customize our curses settings. These details will be explained later .

2.2.2. The mysterious refresh()

The next line printw prints the string "Hello World !!!" on to the screen. This function is analogous to normal printf in all respects except that it prints the data on a window called stdscr at the current (y,x) co-ordinates. Since our present co-ordinates are at 0,0 the string is printed at the left hand corner of the window.

This brings us to that mysterious refresh(). Well, when we called printw the data is actually written to an imaginary window, which is not updated on the screen yet. The job of printw is to update a few flags and data structures and write the data to a buffer corresponding to stdscr. In order to show it on the screen, we need to call refresh() and tell the curses system to dump the contents on the screen.

The philosophy behind all this is to allow the programmer to do multiple updates on the imaginary screen or windows and do a refresh once all his screen update is done. refresh() checks the window and updates only the portion which has been changed. This improves performance and offers greater flexibility too. But, it is sometimes frustrating to beginners. A common mistake committed by beginners is to forget to call refresh() after they did some update through printw() class of functions. I still forget to add it sometimes :-)

2.2.3. About endwin()

And finally don't forget to end the curses mode. Otherwise your terminal might behave strangely after the program quits. endwin() frees the memory taken by curses sub-system and its data structures and puts the terminal in normal mode. This function must be called after you are done with the curses mode.

3. The Gory Details

Now that we have seen how to write a simple curses program let's get into the details. There are many functions that help customize what you see on screen and many features which can be put to full use.

Here we go...

4. Initialization

We now know that to initialize curses system the function initscr() has to be called. There are functions which can be called after this initialization to customize our curses session. We may ask the curses system to set the terminal in raw mode or initialize color or initialize the mouse, etc. Let's discuss some of the functions that are normally called immediately after initscr();

4.1. Initialization functions

4.2. raw() and cbreak()

Normally the terminal driver buffers the characters a user types until a new line or carriage return is encountered. But most programs require that the characters be available as soon as the user types them. The above two functions are used to disable line buffering. The difference between these two functions is in the way control characters like suspend (CTRL-Z), interrupt and quit (CTRL-C) are passed to the program. In the raw() mode these characters are directly passed to the program without generating a signal. In the `cbreak()` mode these control characters are interpreted as any other character by the terminal driver. I personally prefer to use raw() as I can exercise greater control over what the user does.

4.3. echo() and noecho()

These functions control the echoing of characters typed by the user to the terminal. `noecho()` switches off echoing. The reason you might want to do this is to gain more control over echoing or to suppress unnecessary echoing while taking input from the user through the getch(), etc. functions. Most of the interactive programs call `noecho()` at initialization and do the echoing of characters in a controlled manner. It gives the programmer the flexibility of echoing characters at any place in the window without updating current (y,x) co-ordinates.

4.4. keypad()

This is my favorite initialization function. It enables the reading of function keys like F1, F2, arrow keys, etc. Almost every interactive program enables this, as arrow keys are a major part of any User Interface.

Do `keypad(stdscr, TRUE)` to enable this feature for the regular screen (stdscr). You will learn more about key management in later sections of this document.

4.5. halfdelay()

This function, though not used very often, is a useful one at times. halfdelay()is called to enable the half-delay mode, which is similar to the cbreak() mode in that characters typed are immediately available to program. However, it waits for 'X' tenths of a second for input and then returns ERR, if no input is available. 'X' is the timeout value passed to the function halfdelay(). This function is useful when you want to ask the user for input, and if he doesn't respond with in certain time, we can do some thing else. One possible example is a timeout at the password prompt.

4.6. Miscellaneous Initialization functions

There are few more functions which are called at initialization to customize curses behavior. They are not used as extensively as those mentioned above. Some of them are explained where appropriate.

4.7. An Example

Let's write a program which will clarify the usage of these functions.

Example 2. Initialization Function Usage example

```
#include <curses.h>

int
main(void)
{
    int ch;

    initscr();                  /* Start curses mode            */
    raw();                      /* Line buffering disabled      */
    keypad(stdscr, TRUE);       /* We get F1, F2 etc..          */
    noecho();                   /* Don't echo() while we do getch */

    printw("Type any character to see it in bold\n");
    ch = getch();               /* If raw() hadn't been called
                                 * we have to press enter before it
                                 * gets to the program          */
    if (ch == KEY_F(1))         /* Without keypad enabled this will */
        printw("F1 Key pressed"); /*  not get to us either       */
                                /* Without noecho() some ugly escape
                                 * characters might have been printed
                                 * on screen                    */
    else {
```

```
        printw("The pressed key is ");
        attron(A_BOLD);
        printw("%c", ch);
        attroff(A_BOLD);
    }
    refresh();                  /* Print it on to the real screen */
    getch();                    /* Wait for user input */
    endwin();                   /* End curses mode                */

    return 0;
}
```

This program is self-explanatory. But I used functions which aren't explained yet. The function `getch()` is used to get a character from user. It is equivalent to normal `getchar()` except that we can disable the line buffering to avoid <enter> after input. Look for more about `getch()` and reading keys in the key management section . The functions attron and attroff are used to switch some attributes on and off respectively. In the example I used them to print the character in bold. These functions are explained in detail later.

5. A Word about Windows

Before we plunge into the myriad ncurses functions, let me clear few things about windows. Windows are explained in detail in following sections

A Window is an imaginary screen defined by curses system. A window does not mean a bordered window which you usually see on Win9X platforms. When curses is initialized, it creates a default window named `stdscr` which represents your 80x25 (or the size of window in which you are running) screen. If you are doing simple tasks like printing few strings, reading input, etc., you can safely use this single window for all of your purposes. You can also create windows and call functions which explicitly work on the specified window.

For example, if you call

```
    printw("Hi There !!!");
    refresh();
```

It prints the string on stdscr at the present cursor position. Similarly the call to refresh(), works on stdscr only.

Say you have created windows then you have to call a function with a 'w' added to the usual function.

```
    wprintw(win, "Hi There !!!");
    wrefresh(win);
```

As you will see in the rest of the document, naming of functions follow the same convention. For each function there usually are three more functions.

```
printw(string);        /* Print on stdscr at present cursor position */
mvprintw(y, x, string);/* Move to (y, x) then print string     */
wprintw(win, string);  /* Print on window win at present cursor position */
                       /* in the window */
mvwprintw(win, y, x, string);   /* Move to (y, x) relative to window */
                                /* co-ordinates and then print         */
```

Usually the w-less functions are macros which expand to corresponding w-function with stdscr as the window parameter.

6. Output functions

I guess you can't wait any more to see some action. Back to our odyssey of curses functions. Now that curses is initialized, let's interact with world.

There are three classes of functions which you can use to do output on screen.

1. addch() class: Print single character with attributes
2. printw() class: Print formatted output similar to printf()
3. addstr() class: Print strings

These functions can be used interchangeably and it is a matter of style as to which class is used. Let's see each one in detail.

6.1. addch() class of functions

These functions put a single character into the current cursor location and advance the position of the cursor. You can give the character to be printed but they usually are used to print a character with some attributes. Attributes are explained in detail in later sections of the document. If a character is associated with an attribute(bold, reverse video etc.), when curses prints the character, it is printed in that attribute.

In order to combine a character with some attributes, you have two options:

- By OR'ing a single character with the desired attribute macros. These attribute macros could be found in the header file `ncurses.h`. For example, you want to print a character ch(of type char) bold and underlined, you would call addch() as below.

  ```
  addch(ch | A_BOLD | A_UNDERLINE);
  ```

- By using functions like `attrset(),attron(),attroff()`. These functions are explained in the Attributes section. Briefly, they manipulate the current attributes of the given window. Once set, the character printed in the window are associated with the attributes until it is turned off.

Additionally, `curses` provides some special characters for character-based graphics. You can draw tables, horizontal or vertical lines, etc. You can find all available characters in the header file `ncurses.h`. Try looking for macros beginning with `ACS_` in this file.

6.2. mvaddch(), waddch() and mvwaddch()

`mvaddch()` is used to move the cursor to a given point, and then print. Thus, the calls:

```
move(row,col);    /* moves the cursor to rowth row and colth column */
addch(ch);
```

can be replaced by

```
mvaddch(row,col,ch);
```

`waddch()` is similar to `addch()`, except that it adds a character into the given window. (Note that `addch()` adds a character into the window `stdscr`.)

In a similar fashion `mvwaddch()` function is used to add a character into the given window at the given coordinates.

Now, we are familiar with the basic output function `addch()`. But, if we want to print a string, it would be very annoying to print it character by character. Fortunately, `ncurses` provides `printf`*-like* or `puts`*-like* functions.

6.3. printw() class of functions

These functions are similar to `printf()` with the added capability of printing at any position on the screen.

6.3.1. printw() and mvprintw

These two functions work much like `printf()`. `mvprintw()` can be used to move the cursor to a position and then print. If you want to move the cursor first and then print using `printw()` function, use `move()` first and then use `printw()` though I see no point why one should avoid using `mvprintw()`, you have the flexibility to manipulate.

6.3.2. wprintw() and mvwprintw

These two functions are similar to above two except that they print in the corresponding window given as argument.

6.3.3. vw_printw()

This function is similar to `vprintf()`. This can be used when variable number of arguments are to be printed.

6.3.4. A Simple printw example

Example 3. A Simple printw example

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

int
main(void)
{
    char mesg[] = "Just a string";      /* message to be appeared on the screen */
    int row, col;               /* to store the number of rows and *
                                 * the number of columns of the screen */
    initscr();                  /* start the curses mode */
    getmaxyx(stdscr, row, col); /* get the number of rows and columns */
    mvprintw(row / 2, (col - (int) strlen(mesg)) / 2, "%s", mesg);
    /* print the message at the center of the screen */
    mvprintw(row - 2, 0, "This screen has %d rows and %d columns\n", row, col);
    printw("Try resizing your window(if possible) and then run this program again");
    refresh();
    getch();
    endwin();

    return 0;
}
```

Above program demonstrates how easy it is to use `printw`. You just feed the coordinates and the message to be appeared on the screen, then it does what you want.

The above program introduces us to a new function `getmaxyx()`, a macro defined in `ncurses.h`. It gives the number of columns and the number of rows in a given window. `getmaxyx()` does this by updating the variables given to it. Since `getmaxyx()` is not a function we don't pass pointers to it, we just give two integer variables.

6.4. addstr() class of functions

`addstr()` is used to put a character string into a given window. This function is similar to calling `addch()` once for each character in a given string. This is true for all output functions. There are other functions from this family such as `mvaddstr()`,`mvwaddstr()` and `waddstr()`, which obey the naming convention of curses.(e.g. mvaddstr() is similar to the respective calls move() and then addstr().) Another function of this family is addnstr(), which takes an integer parameter(say n) additionally. This function puts at most n characters into the screen. If n is negative, then the entire string will be added.

6.5. A word of caution

All these functions take y co-ordinate first and then x in their arguments. A common mistake by beginners is to pass x,y in that order. If you are doing too many manipulations of (y,x) co-ordinates, think of dividing the screen into windows and manipulate each one separately. Windows are explained in the windows section.

7. Input functions

Well, printing without taking input, is boring. Let's see functions which allow us to get input from user. These functions also can be divided into three categories.

1. getch() class: Get a character
2. scanw() class: Get formatted input
3. getstr() class: Get strings

7.1. getch() class of functions

These functions read a single character from the terminal. But there are several subtle facts to consider. For example if you don't use the function cbreak(), curses will not read your input characters contiguously but will begin read them only after a new line or an EOF is encountered. In order to avoid this, the cbreak() function must used so that characters are immediately available to your program. Another widely used function is noecho(). As the name suggests, when this function is set (used), the characters that are keyed in by the user will not show up on the screen. The two functions cbreak() and noecho() are typical examples of key management. Functions of this genre are explained in the key management section .

7.2. scanw() class of functions

These functions are similar to `scanf()` with the added capability of getting the input from any location on the screen.

7.2.1. scanw() and mvscanw

The usage of these functions is similar to that of `sscanf()`, where the line to be scanned is provided by `wgetstr()` function. That is, these functions call to `wgetstr()` function(explained below) and uses the resulting line for a scan.

7.2.2. wscanw() and mvwscanw()

These are similar to above two functions except that they read from a window, which is supplied as one of the arguments to these functions.

7.2.3. vw_scanw()

This function is similar to `vscanf()`. This can be used when a variable number of arguments are to be scanned.

7.3. getstr() class of functions

These functions are used to get strings from the terminal. In essence, this function performs the same task as would be achieved by a series of calls to `getch()` until a newline, carriage return, or end-of-file is received. The resulting string of characters are pointed to by `str`, which is a character pointer provided by the user.

7.4. Some examples

Example 4. A Simple scanw example

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

int
main(void)
{
    char mesg[] = "Enter a string: ";   /* message to be appeared on the screen */
    char str[80];
    int row, col;               /* to store the number of rows and *
```

```
                                 * the number of columns of the screen */
    initscr();                  /* start the curses mode */
    getmaxyx(stdscr, row, col); /* get the number of rows and columns */
    mvprintw(row / 2, (col - (int) strlen(mesg)) / 2, "%s", mesg);
    /* print the message at the center of the screen */
    getstr(str);
    mvprintw(LINES - 2, 0, "You Entered: %s", str);
    getch();
    endwin();

    return 0;
}
```

8. Attributes

We have seen an example of how attributes can be used to print characters with some special effects. Attributes, when set prudently, can present information in an easy, understandable manner. The following program takes a C file as input and prints the file with comments in bold. Scan through the code.

Example 5. A Simple Attributes example

```
/* pager functionality by Joseph Spainhour" <spainhou@bellsouth.net> */
#include <curses.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int ch, row, col;
    int prev = EOF;
    FILE *fp;
    int y, x;

    if (argc != 2) {
        printf("Usage: %s <a c file name>\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        perror("Cannot open input file");
        exit(1);
    }
    initscr();                          /* Start curses mode */
    getmaxyx(stdscr, row, col);         /* find the boundaries of the screeen */
    (void) col;
    while ((ch = fgetc(fp)) != EOF)     /* read the file till we reach the end */
    {
        getyx(stdscr, y, x);            /* get the current cursor position */
```

```
        if (y == (row - 1))             /* are we are at the end of the screen */
        {
            printw("<-Press Any Key->"); /* tell the user to press a key */
            getch();
            clear();                    /* clear the screen */
            move(0, 0);                 /* start at the beginning of the screen */
        }
        if (prev == '/' && ch == '*')   /* If it is / and * then only
                                         * switch bold on */
        {
            attron(A_BOLD);             /* cut bold on */
            getyx(stdscr, y, x);        /* get the current cursor position */
            move(y, x - 1);             /* back up one space */
            printw("%c%c", '/', ch);    /* The actual printing is done here */
        } else
            printw("%c", ch);
        refresh();
        if (prev == '*' && ch == '/')
            attroff(A_BOLD);    /* Switch it off once we got *
                                 * and then / */
        prev = ch;
    }
    endwin();                   /* End curses mode */
    fclose(fp);
    return 0;
}
```

Don't worry about all those initialization and other crap. Concentrate on the while loop. It reads each character in the file and searches for the pattern /*. Once it spots the pattern, it switches the BOLD attribute on with `attron()` . When we get the pattern */ it is switched off by `attroff()` .

The above program also introduces us to two useful functions `getyx()` and `move()`. The first function gets the co-ordinates of the present cursor into the variables y, x. Since getyx() is a macro we don't have to pass pointers to variables. The function `move()` moves the cursor to the co-ordinates given to it.

The above program is really a simple one which doesn't do much. On these lines one could write a more useful program which reads a C file, parses it and prints it in different colors. One could even extend it to other languages as well.

8.1. The details

Let's get into more details of attributes. The functions `attron()`, `attroff()`, `attrset()` , and their sister functions `attr_get()`, etc. can be used to switch attributes on/off , get attributes and produce a colorful display.

The functions attron and attroff take a bit-mask of attributes and switch them on or off, respectively. The following video attributes, which are defined in <curses.h> can be passed to these functions.

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

The last one is the most colorful one :-) Colors are explained in the next sections (#COLOR).

We can OR(|) any number of above attributes to get a combined effect. If you wanted reverse video with blinking characters you can use

```
attron(A_REVERSE | A_BLINK);
```

8.2. attron() vs attrset()

Then what is the difference between attron() and attrset()? attrset sets the attributes of window whereas attron just switches on the attribute given to it. So attrset() fully overrides whatever attributes the window previously had and sets it to the new attribute(s). Similarly attroff() just switches off the attribute(s) given to it as an argument. This gives us the flexibility of managing attributes easily.But if you use them carelessly you may loose track of what attributes the window has and garble the display. This is especially true while managing menus with colors and highlighting. So decide on a consistent policy and stick to it. You can always use `standend()` which is equivalent to `attrset(A_NORMAL)` which turns off all attributes and brings you to normal mode.

8.3. attr_get()

The function attr_get() gets the current attributes and color pair of the window. Though we might not use this as often as the above functions, this is useful in scanning areas of screen. Say we wanted to do some complex update on screen and we are not sure what attribute each character is associated with. Then this function can be used with either attrset or attron to produce the desired effect.

8.4. attr_ functions

There are series of functions like attr_set(), attr_on, etc. These are similar to above functions except that they take parameters of type `attr_t`.

8.5. wattr functions

For each of the above functions we have a corresponding function with 'w' which operates on a particular window. The above functions operate on stdscr.

8.6. chgat() functions

The function chgat() is listed in the end of the man page curs_attr. It actually is a useful one. This function can be used to set attributes for a group of characters without moving. I mean it !!! without moving the cursor :-) It changes the attributes of a given number of characters starting at the current cursor location.

We can give -1 as the character count to update till end of line. If you want to change attributes of characters from current position to end of line, just use this.

```
    chgat(-1, A_REVERSE, 0, NULL);
```

This function is useful when changing attributes for characters that are already on the screen. Move to the character from which you want to change and change the attribute.

Other functions wchgat(), mvchgat(), wchgat() behave similarly except that the w functions operate on the particular window. The mv functions first move the cursor then perform the work given to them. Actually chgat is a macro which is replaced by a wchgat() with stdscr as the window. Most of the "w-less" functions are macros.

Example 6. Chgat() Usage example

```
#include <curses.h>

int
main(void)
{
    initscr();                  /* Start curses mode            */
    start_color();              /* Start color functionality    */

    init_pair(1, COLOR_CYAN, COLOR_BLACK);
    printw("A Big string which i didn't care to type fully ");
    mvchgat(0, 0, -1, A_BLINK, 1, NULL);
    /*
```

```
     * First two parameters specify the position at which to start
     * Third parameter number of characters to update. -1 means till
     * end of line
     * Forth parameter is the normal attribute you wanted to give
     * to the character
     * Fifth is the color index. It is the index given during init_pair()
     * use 0 if you didn't want color
     * Sixth one is always NULL
     */
    refresh();
    getch();
    endwin();                   /* End curses mode                */
    return 0;
}
```

This example also introduces us to the color world of curses. Colors will be explained in detail later. Use 0 for no color.

9. Windows

Windows form the most important concept in curses. You have seen the standard window stdscr above where all the functions implicitly operated on this window. Now to make design even a simplest GUI, you need to resort to windows. The main reason you may want to use windows is to manipulate parts of the screen separately, for better efficiency, by updating only the windows that need to be changed and for a better design. I would say the last reason is the most important in going for windows. You should always strive for a better and easy-to-manage design in your programs. If you are writing big, complex GUIs this is of pivotal importance before you start doing anything.

9.1. The basics

A Window can be created by calling the function `newwin()`. It doesn't create any thing on the screen actually. It allocates memory for a structure to manipulate the window and updates the structure with data regarding the window such as its size, beginy, beginx, etc. Hence in curses, a window is just an abstraction of an imaginary window, which can be manipulated independent of other parts of screen. The function newwin() returns a pointer to structure WINDOW, which can be passed to window related functions such as wprintw(), etc. Finally the window can be destroyed with delwin(). It will deallocate the memory associated with the window structure.

9.2. Let there be a Window !!!

What fun is it, if a window is created and we can't see it. So the fun part begins by displaying the window. The function `box()` can be used to draw a border around the window. Let's explore these functions in more detail in this example.

Example 7. Window Border example

```
#include <curses.h>

WINDOW *create_newwin(int height, int width, int starty, int startx);
void destroy_win(WINDOW *local_win);

int
main(void)
{
    WINDOW *my_win;
    int startx, starty, width, height;
    int ch;

    initscr();                  /* Start curses mode            */
    cbreak();                   /* Line buffering disabled, Pass on
                                 * every thing to me            */
    keypad(stdscr, TRUE);       /* I need that nifty F1         */

    height = 3;
    width = 10;
    starty = (LINES - height) / 2;      /* Calculating for a center placement */
    startx = (COLS - width) / 2;        /* of the window                */
    printw("Press F1 to exit");
    refresh();
    my_win = create_newwin(height, width, starty, startx);

    while ((ch = getch()) != KEY_F(1)) {
        switch (ch) {
        case KEY_LEFT:
            destroy_win(my_win);
            my_win = create_newwin(height, width, starty, --startx);
            break;
        case KEY_RIGHT:
            destroy_win(my_win);
            my_win = create_newwin(height, width, starty, ++startx);
            break;
        case KEY_UP:
            destroy_win(my_win);
            my_win = create_newwin(height, width, --starty, startx);
            break;
        case KEY_DOWN:
            destroy_win(my_win);
            my_win = create_newwin(height, width, ++starty, startx);
            break;
        }
    }

    endwin();                   /* End curses mode                */
    return 0;
}
```

```
WINDOW *
create_newwin(int height, int width, int starty, int startx)
{
    WINDOW *local_win;

    local_win = newwin(height, width, starty, startx);
    box(local_win, 0, 0);       /* 0, 0 gives default characters
                                 * for the vertical and horizontal
                                 * lines                        */
    wrefresh(local_win);        /* Show that box                */

    return local_win;
}

void
destroy_win(WINDOW *local_win)
{
    /* box(local_win, ' ', ' '); : This won't produce the desired
     * result of erasing the window. It will leave its four corners
     * and so an ugly remnant of window.
     */
    wborder(local_win, ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ');
    /* The parameters taken are
     * 1. win: the window on which to operate
     * 2. ls: character to be used for the left side of the window
     * 3. rs: character to be used for the right side of the window
     * 4. ts: character to be used for the top side of the window
     * 5. bs: character to be used for the bottom side of the window
     * 6. tl: character to be used for the top left corner of the window
     * 7. tr: character to be used for the top right corner of the window
     * 8. bl: character to be used for the bottom left corner of the window
     * 9. br: character to be used for the bottom right corner of the window
     */
    wrefresh(local_win);
    delwin(local_win);
}
```

9.3. Explanation

Don't scream. I know it is a big example. But I have to explain some important things here :-). This program creates a rectangular window that can be moved with left, right, up, down arrow keys. It repeatedly creates and destroys windows as user press a key. Don't go beyond the screen limits. Checking for those limits is left as an exercise for the reader. Let's dissect it by line by line.

The `create_newwin()` function creates a window with `newwin()` and displays a border around it with box. The function `destroy_win()` first erases the window from screen by painting a border with

' ' character and then calling `delwin()` to deallocate memory related to it. Depending on the key the user presses, starty or startx is changed and a new window is created.

In the destroy_win, as you can see, I used wborder instead of box. The reason is written in the comments (You missed it. I know. Read the code :-)). wborder draws a border around the window with the characters given to it as the 4 corner points and the 4 lines. To put it clearly, if you have called wborder as below:

```
    wborder(win, '|', '|', '-', '-', '+', '+', '+', '+');
```

it produces something like

```
    +------------+
    |            |
    |            |
    |            |
    |            |
    |            |
    |            |
    +------------+
```

9.4. The other stuff in the example

You can also see in the above examples, that I have used the variables COLS, LINES which are initialized to the screen sizes after initscr(). They can be useful in finding screen dimensions and finding the center co-ordinate of the screen as above. The function `getch()` as usual gets the key from keyboard and according to the key it does the corresponding work. This type of switch- case is very common in any GUI based programs.

9.5. Other Border functions

Above program is grossly inefficient in that with each press of a key, a window is destroyed and another is created. So let's write a more efficient program which uses other border related functions.

The following program uses `mvhline()` and `mvvline()` to achieve similar effect. These two functions are simple. They create a horizontal or vertical line of the specified length at the specified position.

Example 8. More border functions

```
#include <curses.h>

typedef struct _win_border_struct {
    chtype ls, rs, ts, bs, tl, tr, bl, br;
} WIN_BORDER;
```

```
typedef struct _WIN_struct {

    int startx, starty;
    int height, width;
    WIN_BORDER border;
} WIN;

void init_win_params(WIN * p_win);
void print_win_params(WIN * p_win);
void create_box(WIN * win, bool flag);

int
main(void)
{
    WIN win;
    int ch;

    initscr();                  /* Start curses mode            */
    start_color();              /* Start the color functionality */
    cbreak();                   /* Line buffering disabled, Pass on
                                 * everty thing to me                 */
    keypad(stdscr, TRUE);       /* I need that nifty F1         */
    noecho();
    init_pair(1, COLOR_CYAN, COLOR_BLACK);

    /* Initialize the window parameters */
    init_win_params(&win);
    print_win_params(&win);

    attron(COLOR_PAIR(1));
    printw("Press F1 to exit");
    refresh();
    attroff(COLOR_PAIR(1));

    create_box(&win, TRUE);
    while ((ch = getch()) != KEY_F(1)) {
        switch (ch) {
        case KEY_LEFT:
            create_box(&win, FALSE);
            --win.startx;
            create_box(&win, TRUE);
            break;
        case KEY_RIGHT:
            create_box(&win, FALSE);
            ++win.startx;
            create_box(&win, TRUE);
            break;
        case KEY_UP:
            create_box(&win, FALSE);
            --win.starty;
            create_box(&win, TRUE);
            break;
```

```
        case KEY_DOWN:
            create_box(&win, FALSE);
            ++win.starty;
            create_box(&win, TRUE);
            break;
        }
    }
    endwin();                   /* End curses mode                */
    return 0;
}
void
init_win_params(WIN * p_win)
{
    p_win->height = 3;
    p_win->width = 10;
    p_win->starty = (LINES - p_win->height) / 2;
    p_win->startx = (COLS - p_win->width) / 2;

    p_win->border.ls = '|';
    p_win->border.rs = '|';
    p_win->border.ts = '-';
    p_win->border.bs = '-';
    p_win->border.tl = '+';
    p_win->border.tr = '+';
    p_win->border.bl = '+';
    p_win->border.br = '+';

}

void
print_win_params(WIN * p_win)
{
#ifdef _DEBUG
    mvprintw(25, 0, "%d %d %d %d", p_win->startx, p_win->starty,
             p_win->width, p_win->height);
    refresh();
#else
    (void) p_win;
#endif
}

void
create_box(WIN * p_win, bool flag)
{
    int i, j;
    int x, y, w, h;

    x = p_win->startx;
    y = p_win->starty;
    w = p_win->width;
    h = p_win->height;

    if (flag == TRUE) {
```

```
        mvaddch(y, x, p_win->border.tl);
        mvaddch(y, x + w, p_win->border.tr);
        mvaddch(y + h, x, p_win->border.bl);
        mvaddch(y + h, x + w, p_win->border.br);
        mvhline(y, x + 1, p_win->border.ts, w - 1);
        mvhline(y + h, x + 1, p_win->border.bs, w - 1);
        mvvline(y + 1, x, p_win->border.ls, h - 1);
        mvvline(y + 1, x + w, p_win->border.rs, h - 1);

    } else
        for (j = y; j <= y + h; ++j)
            for (i = x; i <= x + w; ++i)
                mvaddch(j, i, ' ');

    refresh();
}
```

10. Colors

10.1. The basics

Life seems dull with no colors. Curses has a nice mechanism to handle colors. Let's get into the thick of the things with a small program.

Example 9. A Simple Color example

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

void print_in_middle(WINDOW *win, int starty, int startx, int width, const char *string);

int
main(void)
{
    initscr();                  /* Start curses mode            */
    if (has_colors() == FALSE) {
        endwin();
        printf("Your terminal does not support color\n");
        exit(1);
    }
    start_color();              /* Start color                  */
    init_pair(1, COLOR_RED, COLOR_BLACK);

    attron(COLOR_PAIR(1));
    print_in_middle(stdscr, LINES / 2, 0, 0, "Viola !!! In color ...");
    attroff(COLOR_PAIR(1));
```

```
    getch();
    endwin();
}
void
print_in_middle(WINDOW *win, int starty, int startx, int width, const char *string)
{
    int length, x, y;
    float temp;

    if (win == NULL)
        win = stdscr;
    getyx(win, y, x);
    if (startx != 0)
        x = startx;
    if (starty != 0)
        y = starty;
    if (width == 0)
        width = 80;

    length = (int) strlen(string);
    temp = (float) (width - length) / 2;
    x = startx + (int) temp;
    mvwprintw(win, y, x, "%s", string);
    refresh();
}
```

As you can see, to start using color, you should first call the function `start_color()`. After that, you can use color capabilities of your terminals using various functions. To find out whether a terminal has color capabilities or not, you can use `has_colors()` function, which returns FALSE if the terminal does not support color.

Curses initializes all the colors supported by terminal when start_color() is called. These can be accessed by the define constants like `COLOR_BLACK` , etc. Now to actually start using colors, you have to define pairs. Colors are always used in pairs. That means you have to use the function `init_pair()` to define the foreground and background for the pair number you give. After that that pair number can be used as a normal attribute with `COLOR_PAIR()` function. This may seem to be cumbersome at first. But this elegant solution allows us to manage color pairs very easily. To appreciate it, you have to look into the the source code of "dialog", a utility for displaying dialog boxes from shell scripts. The developers have defined foreground and background combinations for all the colors they might need and initialized at the beginning. This makes it very easy to set attributes just by accessing a pair which we already have defined as a constant.

The following colors are defined in `curses.h`. You can use these as parameters for various color functions.

```
        COLOR_BLACK   0
        COLOR_RED     1
        COLOR_GREEN   2
        COLOR_YELLOW  3
```

```
COLOR_BLUE    4
COLOR_MAGENTA 5
COLOR_CYAN    6
COLOR_WHITE   7
```

10.2. Changing Color Definitions

The function `init_color()`can be used to change the rgb values for the colors defined by curses initially. Say you wanted to lighten the intensity of red color by a minuscule. Then you can use this function as

```
init_color(COLOR_RED, 700, 0, 0);
/* param 1     : color name
 * param 2, 3, 4 : rgb content min = 0, max = 1000 */
```

If your terminal cannot change the color definitions, the function returns ERR. The function `can_change_color()` can be used to find out whether the terminal has the capability of changing color content or not. The rgb content is scaled from 0 to 1000. Initially RED color is defined with content 1000(r), 0(g), 0(b).

10.3. Color Content

The functions `color_content()` and `pair_content()` can be used to find the color content and foreground, background combination for the pair.

11. Interfacing with the key board

11.1. The Basics

No GUI is complete without a strong user interface and to interact with the user, a curses program should be sensitive to key presses or the mouse actions done by the user. Let's deal with the keys first.

As you have seen in almost all of the above examples, it is very easy to get key input from the user. A simple way of getting key presses is to use `getch()` function. The cbreak mode should be enabled to read keys when you are interested in reading individual key hits rather than complete lines of text (which usually end with a carriage return). keypad should be enabled to get the Functions keys, arrow keys, etc. See the initialization section for details.

`getch()` returns an integer corresponding to the key pressed. If it is a normal character, the integer value will be equivalent to the character. Otherwise it returns a number which can be matched with the constants defined in `curses.h`. For example if the user presses F1, the integer returned is 265. This can be checked using the macro KEY_F() defined in curses.h. This makes reading keys portable and easy to manage.

For example, if you call getch() like this

```
    int ch;

    ch = getch();
```

getch() will wait for the user to press a key, (unless you specified a timeout) and when user presses a key, the corresponding integer is returned. Then you can check the value returned with the constants defined in curses.h to match against the keys you want.

The following code piece will do that job.

```
    if(ch == KEY_LEFT)
        printw("Left arrow is pressed\n");
```

Let’s write a small program which creates a menu which can be navigated by up and down arrows.

11.2. A Simple Key Usage example

Example 10. A Simple Key Usage example

```
#include <curses.h>

#define WIDTH 30
#define HEIGHT 10

int startx = 0;
int starty = 0;

const char *choices[] =
{
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Exit",
};
int n_choices = sizeof(choices) / sizeof(char *);
void print_menu(WINDOW *menu_win, int highlight);

int
main(void)
```

```
{
    WINDOW *menu_win;
    int highlight = 1;
    int choice = 0;
    int c;

    initscr();
    clear();
    noecho();
    cbreak();                   /* Line buffering disabled. pass on everything */
    startx = (80 - WIDTH) / 2;
    starty = (24 - HEIGHT) / 2;

    menu_win = newwin(HEIGHT, WIDTH, starty, startx);
    keypad(menu_win, TRUE);
    mvprintw(0, 0,
             "Use arrow keys to go up and down, Press enter to select a choice");
    refresh();
    print_menu(menu_win, highlight);
    while (1) {
        c = wgetch(menu_win);
        switch (c) {
        case KEY_UP:
            if (highlight == 1)
                highlight = n_choices;
            else
                --highlight;
            break;
        case KEY_DOWN:
            if (highlight == n_choices)
                highlight = 1;
            else
                ++highlight;
            break;
        case 10:
            choice = highlight;
            break;
        default:
            mvprintw(24, 0,
                     "Character pressed is = %3d Hopefully it can be printed as '%c'",
                     c, c);
            refresh();
            break;
        }
        print_menu(menu_win, highlight);
        if (choice != 0)        /* User did a choice come out of the infinite loop */
            break;
    }
    mvprintw(23, 0, "You chose choice %d with choice string %s\n", choice,
             choices[choice - 1]);
    clrtoeol();
    refresh();
    endwin();
```

```
    return 0;
}

void
print_menu(WINDOW *menu_win, int highlight)
{
    int x, y, i;

    x = 2;
    y = 2;
    box(menu_win, 0, 0);
    for (i = 0; i < n_choices; ++i) {
        if (highlight == i + 1) /* Highlight the present choice */
        {
            wattron(menu_win, A_REVERSE);
            mvwprintw(menu_win, y, x, "%s", choices[i]);
            wattroff(menu_win, A_REVERSE);
        } else
            mvwprintw(menu_win, y, x, "%s", choices[i]);
        ++y;
    }
    wrefresh(menu_win);
}
```

12. Interfacing with the mouse

Now that you have seen how to get keys, lets do the same thing from mouse. Usually each UI allows the user to interact with both keyboard and mouse.

12.1. The Basics

Before you do any thing else, the events you want to receive have to be enabled with `mousemask()`.

```
    mousemask(  mmask_t newmask,    /* The events you want to listen to */
                mmask_t *oldmask)    /* The old events mask                */
```

The first parameter to above function is a bit mask of events you would like to listen. By default, all the events are turned off. The bit mask `ALL_MOUSE_EVENTS` can be used to get all the events.

The following are all the event masks:

```
    Name            Description
       ---------------------------------------------------------------------
       BUTTON1_PRESSED          mouse button 1 down
       BUTTON1_RELEASED         mouse button 1 up
```

```
BUTTON1_CLICKED          mouse button 1 clicked
BUTTON1_DOUBLE_CLICKED   mouse button 1 double clicked
BUTTON1_TRIPLE_CLICKED   mouse button 1 triple clicked
BUTTON2_PRESSED          mouse button 2 down
BUTTON2_RELEASED         mouse button 2 up
BUTTON2_CLICKED          mouse button 2 clicked
BUTTON2_DOUBLE_CLICKED   mouse button 2 double clicked
BUTTON2_TRIPLE_CLICKED   mouse button 2 triple clicked
BUTTON3_PRESSED          mouse button 3 down
BUTTON3_RELEASED         mouse button 3 up
BUTTON3_CLICKED          mouse button 3 clicked
BUTTON3_DOUBLE_CLICKED   mouse button 3 double clicked
BUTTON3_TRIPLE_CLICKED   mouse button 3 triple clicked
BUTTON4_PRESSED          mouse button 4 down
BUTTON4_RELEASED         mouse button 4 up
BUTTON4_CLICKED          mouse button 4 clicked
BUTTON4_DOUBLE_CLICKED   mouse button 4 double clicked
BUTTON4_TRIPLE_CLICKED   mouse button 4 triple clicked
BUTTON_SHIFT             shift was down during button state change
BUTTON_CTRL              control was down during button state change
BUTTON_ALT               alt was down during button state change
ALL_MOUSE_EVENTS         report all button state changes
REPORT_MOUSE_POSITION    report mouse movement
```

12.2. Getting the events

Once a class of mouse events have been enabled, getch() class of functions return KEY_MOUSE every time some mouse event happens. Then the mouse event can be retrieved with `getmouse()`.

The code approximately looks like this:

```
MEVENT event;

ch = getch();
if(ch == KEY_MOUSE)
    if(getmouse(&event) == OK)
        .    /* Do some thing with the event */
        .
        .
```

getmouse() returns the event into the pointer given to it. It is a structure which contains

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

The `bstate` is the main variable we are interested in. It tells the button state of the mouse.

Then with a code snippet like the following, we can find out what happened.

```
    if(event.bstate & BUTTON1_PRESSED)
        printw("Left Button Pressed");
```

12.3. Putting it all Together

That's pretty much interfacing with mouse. Let's create the same menu and enable mouse interaction. To make things simpler, key handling is removed.

Example 11. Access the menu with mouse !!!

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

#define WIDTH 30
#define HEIGHT 10

int startx = 0;
int starty = 0;

const char *choices[] =
{"Choice 1",
 "Choice 2",
 "Choice 3",
 "Choice 4",
 "Exit",
};

int n_choices = sizeof(choices) / sizeof(char *);

void print_menu(WINDOW *menu_win, int highlight);
void report_choice(int mouse_x, int mouse_y, int *p_choice);

int
main(void)
{
    int c, choice = 0;
    WINDOW *menu_win;
    MEVENT event;

    /* Initialize curses */
    initscr();
    clear();
    noecho();
    cbreak();                   /* Line buffering disabled. pass everything */
```

```
    /* Try to put the window in the middle of screen */
    startx = (80 - WIDTH) / 2;
    starty = (24 - HEIGHT) / 2;

    attron(A_REVERSE);
    mvprintw(23, 1,
             "Click on Exit to quit (Works best in a virtual console)");
    refresh();
    attroff(A_REVERSE);

    /* Print the menu for the first time */
    menu_win = newwin(HEIGHT, WIDTH, starty, startx);
    keypad(menu_win, TRUE);
    print_menu(menu_win, 1);
    /* Get all the mouse events */
    mousemask(ALL_MOUSE_EVENTS, NULL);

    while (1) {
        c = wgetch(menu_win);
        switch (c) {
        case KEY_MOUSE:
            if (getmouse(&event) == OK) {       /* When the user clicks left mouse button */
                if (event.bstate & BUTTON1_PRESSED) {
                    report_choice(event.x + 1, event.y + 1, &choice);
                    if (choice == -1)   /* Exit chosen */
                        goto end;
                    mvprintw(22, 1,
                             "Choice made is : %d String Chosen is \"%10s\"",
                             choice, choices[choice - 1]);
                    refresh();
                }
            }
            print_menu(menu_win, choice);
            break;
        }
    }
  end:
    endwin();
    return 0;
}

void
print_menu(WINDOW *menu_win, int highlight)
{
    int x, y, i;

    x = 2;
    y = 2;
    box(menu_win, 0, 0);
    for (i = 0; i < n_choices; ++i) {
        if (highlight == i + 1) {
            wattron(menu_win, A_REVERSE);
            mvwprintw(menu_win, y, x, "%s", choices[i]);
```

```
            wattroff(menu_win, A_REVERSE);
        } else
            mvwprintw(menu_win, y, x, "%s", choices[i]);
        ++y;
    }
    wrefresh(menu_win);
}

/* Report the choice according to mouse position */
void
report_choice(int mouse_x, int mouse_y, int *p_choice)
{
    int i, j, choice;

    i = startx + 2;
    j = starty + 3;

    for (choice = 0; choice < n_choices; ++choice)
        if (mouse_y == j + choice
            && mouse_x >= i
            && mouse_x <= i + (int) strlen(choices[choice])) {
            if (choice == n_choices - 1)
                *p_choice = -1;
            else
                *p_choice = choice + 1;
            break;
        }
}
```

12.4. Miscellaneous Functions

The functions mouse_trafo() and wmouse_trafo() can be used to convert to mouse co-ordinates to screen relative co-ordinates. See curs_mouse(3X) man page for details.

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

13. Screen Manipulation

In this section, we will look into some functions, which allow us to manage the screen efficiently and to write some fancy programs. This is especially important in writing games.

13.1. getyx() functions

The function `getyx()` can be used to find out the present cursor co-ordinates. It will fill the values of x and y co-ordinates in the arguments given to it. Since getyx() is a macro you don't have to pass the address of the variables. It can be called as

```
getyx(win, y, x);
/* win: window pointer
 *   y, x: y, x co-ordinates will be put into this variables
 */
```

The function getparyx() gets the beginning co-ordinates of the sub window relative to the main window. This is some times useful to update a sub window. When designing fancy stuff like writing multiple menus, it becomes difficult to store the menu positions, their first option co-ordinates, etc. A simple solution to this problem, is to create menus in sub windows and later find the starting co-ordinates of the menus by using getparyx().

The functions getbegyx() and getmaxyx() store current window's beginning and maximum co-ordinates. These functions are useful in the same way as above in managing the windows and sub windows effectively.

13.2. Screen Dumping

While writing games, some times it becomes necessary to store the state of the screen and restore it back to the same state. The function scr_dump() can be used to dump the screen contents to a file given as an argument. Later it can be restored by scr_restore function. These two simple functions can be used effectively to maintain a fast moving game with changing scenarios.

13.3. Window Dumping

To store and restore windows, the functions `putwin()` and `getwin()` can be used. `putwin()` puts the present window state into a file, which can be later restored by `getwin()`.

The function `copywin()` can be used to copy a window completely onto another window. It takes the source and destination windows as parameters and according to the rectangle specified, it copies the rectangular region from source to destination window. Its last parameter specifies whether to overwrite or just overlay the contents on to the destination window. If this argument is true, then the copying is non-destructive.

14. Miscellaneous features

Now you know enough features to write a good curses program, with all bells and whistles. There are some miscellaneous functions which are useful in various cases. Let's go headlong into some of those.

14.1. curs_set()

This function can be used to make the cursor invisible. The parameter to this function should be

```
    0 : invisible      or
    1 : normal    or
    2 : very visible.
```

14.2. Temporarily Leaving Curses mode

Some times you may want to get back to cooked mode (normal line buffering mode) temporarily. In such a case you will first need to save the tty modes with a call to `def_prog_mode()` and then call `endwin()` to end the curses mode. This will leave you in the original tty mode. To get back to curses once you are done, call `reset_prog_mode()` . This function returns the tty to the state stored by `def_prog_mode()`. Then do refresh(), and you are back to the curses mode. Here is an example showing the sequence of things to be done.

Example 12. Temporarily Leaving Curses Mode

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

int
main(void)
{
    initscr();                  /* Start curses mode              */
    printw("Hello World !!!\n");        /* Print Hello World              */
    refresh();                  /* Print it on to the real screen */
    def_prog_mode();            /* Save the tty modes             */
    endwin();                   /* End curses mode temporarily    */
    system("/bin/sh");          /* Do whatever you like in cooked mode */
    reset_prog_mode();          /* Return to the previous tty mode */
    /* stored by def_prog_mode()      */
    refresh();                  /* Do refresh() to restore the    */
    /* Screen contents                */
    printw("Another String\n"); /* Back to curses use the full    */
    refresh();                  /* capabilities of curses         */
    endwin();                   /* End curses mode                */

    return 0;
}
```

14.3. ACS_ variables

If you have ever programmed in DOS, you know about those nifty characters in extended character set. They are printable only on some terminals. NCURSES functions like `box()` use these characters. All these variables start with ACS meaning alternative character set. You might have noticed me using these characters in some of the programs above. Here is an example showing all the characters.

Example 13. ACS Variables Example

```
#include <curses.h>

int main(void)
{
    initscr();

    printw("Upper left corner           "); addch(ACS_ULCORNER); printw("\n");
    printw("Lower left corner           "); addch(ACS_LLCORNER); printw("\n");
    printw("Lower right corner          "); addch(ACS_LRCORNER); printw("\n");
    printw("Tee pointing right          "); addch(ACS_LTEE); printw("\n");
    printw("Tee pointing left           "); addch(ACS_RTEE); printw("\n");
    printw("Tee pointing up             "); addch(ACS_BTEE); printw("\n");
    printw("Tee pointing down           "); addch(ACS_TTEE); printw("\n");
    printw("Horizontal line             "); addch(ACS_HLINE); printw("\n");
    printw("Vertical line               "); addch(ACS_VLINE); printw("\n");
    printw("Large Plus or cross over    "); addch(ACS_PLUS); printw("\n");
    printw("Scan Line 1                 "); addch(ACS_S1); printw("\n");
    printw("Scan Line 3                 "); addch(ACS_S3); printw("\n");
    printw("Scan Line 7                 "); addch(ACS_S7); printw("\n");
    printw("Scan Line 9                 "); addch(ACS_S9); printw("\n");
    printw("Diamond                     "); addch(ACS_DIAMOND); printw("\n");
    printw("Checker board (stipple)     "); addch(ACS_CKBOARD); printw("\n");
    printw("Degree Symbol               "); addch(ACS_DEGREE); printw("\n");
    printw("Plus/Minus Symbol           "); addch(ACS_PLMINUS); printw("\n");
    printw("Bullet                      "); addch(ACS_BULLET); printw("\n");
    printw("Arrow Pointing Left         "); addch(ACS_LARROW); printw("\n");
    printw("Arrow Pointing Right        "); addch(ACS_RARROW); printw("\n");
    printw("Arrow Pointing Down         "); addch(ACS_DARROW); printw("\n");
    printw("Arrow Pointing Up           "); addch(ACS_UARROW); printw("\n");
    printw("Board of squares            "); addch(ACS_BOARD); printw("\n");
    printw("Lantern Symbol              "); addch(ACS_LANTERN); printw("\n");
    printw("Solid Square Block          "); addch(ACS_BLOCK); printw("\n");
    printw("Less/Equal sign             "); addch(ACS_LEQUAL); printw("\n");
    printw("Greater/Equal sign          "); addch(ACS_GEQUAL); printw("\n");
    printw("Pi                          "); addch(ACS_PI); printw("\n");
    printw("Not equal                   "); addch(ACS_NEQUAL); printw("\n");
    printw("UK pound sign               "); addch(ACS_STERLING); printw("\n");

    refresh();
    getch();
    endwin();

    return 0;
```

```
}
```

15. Other libraries

Apart from the curses library, there are few text mode libraries, which provide more functionality and a lot of features. The following sections explain three standard libraries which are usually distributed along with curses.

16. Panel Library

Now that you are proficient in curses, you wanted to do some thing big. You created a lot of overlapping windows to give a professional windows-type look. Unfortunately, it soon becomes difficult to manage these. The multiple refreshes, updates plunge you into a nightmare. The overlapping windows create blotches, whenever you forget to refresh the windows in the proper order.

Don’t despair. There is an elegant solution provided in panels library. In the words of developers of ncurses

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.

If you have lot of overlapping windows, then panels library is the way to go. It obviates the need of doing series of wnoutrefresh(), doupdate() and relieves the burden of doing it correctly(bottom up). The library maintains information about the order of windows, their overlapping and update the screen properly. So why wait? Let’s take a close peek into panels.

16.1. The Basics

Panel object is a window that is implicitly treated as part of a deck including all other panel objects. The deck is treated as a stack with the top panel being completely visible and the other panels may or may not be obscured according to their positions. So the basic idea is to create a stack of overlapping panels and use panels library to display them correctly. There is a function similar to refresh() which, when called , displays panels in the correct order. Functions are provided to hide or show panels, move panels, change its size, etc. The overlapping problem is managed by the panels library during all the calls to these functions.

The general flow of a panel program goes like this:

1. Create the windows (with newwin()) to be attached to the panels.
2. Create panels with the chosen visibility order. Stack them up according to the desired visibility. The function new_panel() is used to created panels.
3. Call update_panels() to write the panels to the virtual screen in correct visibility order. Do a doupdate() to show it on the screen.
4. Mainpulate the panels with show_panel(), hide_panel(), move_panel(), etc. Make use of helper functions like panel_hidden() and panel_window(). Make use of user pointer to store custom data for a panel. Use the functions set_panel_userptr() and panel_userptr() to set and get the user pointer for a panel.
5. When you are done with the panel use del_panel() to delete the panel.

Let's make the concepts clear, with some programs. The following is a simple program which creates 3 overlapping panels and shows them on the screen.

16.2. Compiling With the Panels Library

To use panels library functions, you have to include panel.h and to link the program with panels library the flag -lpanel should be added along with -lncurses in that order.

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

compile and link: gcc <program file> -lpanel -lncurses
```

Example 14. Panel basics

```
#include <panel.h>

int
main(void)
{
    WINDOW *my_wins[3];
    PANEL *my_panels[3];
    int lines = 10, cols = 40, y = 2, x = 4, i;

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

    /* Create windows for the panels */
    my_wins[0] = newwin(lines, cols, y, x);
    my_wins[1] = newwin(lines, cols, y + 1, x + 5);
    my_wins[2] = newwin(lines, cols, y + 2, x + 10);
```

```
    /*
     * Create borders around the windows so that you can see the effect
     * of panels
     */
    for (i = 0; i < 3; ++i)
        box(my_wins[i], 0, 0);

    /* Attach a panel to each window */
    /* Order is bottom up */
    /* Push 0, order: stdscr-0 */
    /* Push 1, order: stdscr-0-1 */
    /* Push 2, order: stdscr-0-1-2 */
    for (i = 0; i < 3; ++i)
 my_panels[i] = new_panel(my_wins[i]);

    /* Update the stacking order. Last-created panel will be on top */
    update_panels();

    /* Show it on the screen */
    doupdate();

    getch();

    /* Deleting panels does not erase their window */
    for (i = 0; i < 3; ++i)
 del_panel(my_panels[i]);

    endwin();
    return 0;
}
```

As you can see, above program follows a simple flow as explained. The windows are created with newwin() and then they are attached to panels with new_panel(). As we attach one panel after another, the stack of panels gets updated. To put them on screen update_panels() and doupdate() are called.

16.3. Panel Window Browsing

A slightly complicated example is given below. This program creates 3 windows which can be cycled through using tab. Have a look at the code.

Example 15. Panel Window Browsing Example

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

#define NLINES 10
#define NCOLS 40

void init_wins(WINDOW **wins, int n);
void win_show(WINDOW *win, const char *label, int label_color);
```

```
void print_in_middle(WINDOW *win, int starty, int startx,
                     int width, const char *string, chtype color);

int
main(void)
{
    WINDOW *my_wins[3];
    PANEL *my_panels[3];
    PANEL *top;
    int ch;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    /* Initialize all the colors */
    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_GREEN, COLOR_BLACK);
    init_pair(3, COLOR_BLUE, COLOR_BLACK);
    init_pair(4, COLOR_CYAN, COLOR_BLACK);

    init_wins(my_wins, 3);

    /* Attach a panel to each window */
    /* Order is bottom up */
    my_panels[0] = new_panel(my_wins[0]);       /* Push 0, order: stdscr-0 */
    my_panels[1] = new_panel(my_wins[1]);       /* Push 1, order: stdscr-0-1 */
    my_panels[2] = new_panel(my_wins[2]);       /* Push 2, order: stdscr-0-1-2 */

    /* Set up the user pointers to the next panel */
    set_panel_userptr(my_panels[0], my_panels[1]);
    set_panel_userptr(my_panels[1], my_panels[2]);
    set_panel_userptr(my_panels[2], my_panels[0]);

    /* Update the stacking order. 2nd panel will be on top */
    update_panels();

    /* Show it on the screen */
    attron(COLOR_PAIR(4));
    mvprintw(LINES - 2, 0,
             "Use tab to browse through the windows (F1 to Exit)");
    attroff(COLOR_PAIR(4));
    doupdate();

    top = my_panels[2];
    while ((ch = getch()) != KEY_F(1)) {
        switch (ch) {
        case 9:
            top = (PANEL *) panel_userptr(top);
            top_panel(top);
```

```
            break;
        }
        update_panels();
        doupdate();
    }
    endwin();
    return 0;
}

/* Put all the windows */
void
init_wins(WINDOW **wins, int n)
{
    int x, y, i;
    char label[80];

    y = 2;
    x = 10;
    for (i = 0; i < n; ++i) {
        wins[i] = newwin(NLINES, NCOLS, y, x);
        sprintf(label, "Window Number %d", i + 1);
        win_show(wins[i], label, i + 1);
        y += 3;
        x += 7;
    }
}

/* Show the window with a border and a label */
void
win_show(WINDOW *win, const char *label, int label_color)
{
    int height, width;

    getmaxyx(win, height, width);
    (void) height;

    box(win, 0, 0);
    mvwaddch(win, 2, 0, ACS_LTEE);
    mvwhline(win, 2, 1, ACS_HLINE, width - 2);
    mvwaddch(win, 2, width - 1, ACS_RTEE);

    print_in_middle(win, 1, 0, width, label, COLOR_PAIR(label_color));
}

void
print_in_middle(WINDOW *win, int starty, int startx,
                int width, const char *string, chtype color)
{
    int length, x, y;
    float temp;

    if (win == NULL)
        win = stdscr;
```

```
    getyx(win, y, x);
    if (startx != 0)
        x = startx;
    if (starty != 0)
        y = starty;
    if (width == 0)
        width = 80;

    length = (int) strlen(string);
    temp = (float) (width - length) / 2;
    x = startx + (int) temp;
    wattron(win, color);
    mvwprintw(win, y, x, "%s", string);
    wattroff(win, color);
    refresh();
}
```

16.4. Using User Pointers

In the above example I used user pointers to find out the next window in the cycle. We can attach custom information to the panel by specifying a user pointer, which can point to any information you want to store. In this case I stored the pointer to the next panel in the cycle. User pointer for a panel can be set with the function `set_panel_userptr()`. It can be accessed using the function `panel_userptr()` which will return the user pointer for the panel given as argument. After finding the next panel in the cycle, it is brought to the top by the function top_panel(). This function brings the panel given as argument to the top of the panel stack.

16.5. Moving and Resizing Panels

The function `move_panel()` can be used to move a panel to the desired location. It does not change the position of the panel in the stack. Make sure that you use move_panel() instead mvwin() on the window associated with the panel.

Resizing a panel is slightly complex. There is no straight forward function just to resize the window associated with a panel. A solution to resize a panel is to create a new window with the desired sizes, change the window associated with the panel using replace_panel(). Don't forget to delete the old window. The window associated with a panel can be found by using the function panel_window().

The following program shows these concepts, in supposedly simple program. You can cycle through the window with <TAB> as usual. To resize or move the active panel press 'r' for resize 'm' for moving. Then use arrow keys to resize or move it to the desired way and press enter to end your resizing or moving. This example makes use of user data to get the required data to do the operations.

Example 16. Panel Moving and Resizing example

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

typedef struct _PANEL_DATA {
    int x, y, w, h;
    char label[80];
    int label_color;
    PANEL *next;
} PANEL_DATA;

#define NLINES 10
#define NCOLS 40

void init_wins(WINDOW **wins, int n);
void win_show(WINDOW *win, const char *label, int label_color);
void print_in_middle(WINDOW *win, int starty, int startx,
                     int width, const char *string, chtype color);
void set_user_ptrs(PANEL **panels, int n);

int
main(void)
{
    WINDOW *my_wins[3];
    PANEL *my_panels[3];
    PANEL_DATA *top;
    PANEL *stack_top;
    WINDOW *temp_win, *old_win;
    int ch;
    int newx, newy, neww, newh;
    int size = FALSE, move = FALSE;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    /* Initialize all the colors */
    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_GREEN, COLOR_BLACK);
    init_pair(3, COLOR_BLUE, COLOR_BLACK);
    init_pair(4, COLOR_CYAN, COLOR_BLACK);

    init_wins(my_wins, 3);

    /* Attach a panel to each window */
    /* Order is bottom up */
    my_panels[0] = new_panel(my_wins[0]);       /* Push 0, order: stdscr-0 */
    my_panels[1] = new_panel(my_wins[1]);       /* Push 1, order: stdscr-0-1 */
```

```
my_panels[2] = new_panel(my_wins[2]);       /* Push 2, order: stdscr-0-1-2 */

set_user_ptrs(my_panels, 3);
/* Update the stacking order. 2nd panel will be on top */
update_panels();

/* Show it on the screen */
attron(COLOR_PAIR(4));
mvprintw(LINES - 3, 0, "Use 'm' for moving, 'r' for resizing");
mvprintw(LINES - 2, 0,
         "Use tab to browse through the windows (F1 to Exit)");
attroff(COLOR_PAIR(4));
doupdate();

stack_top = my_panels[2];
top = (PANEL_DATA *) panel_userptr(stack_top);
newx = top->x;
newy = top->y;
neww = top->w;
newh = top->h;
while ((ch = getch()) != KEY_F(1)) {
    switch (ch) {
    case 9:         /* Tab */
        top = (PANEL_DATA *) panel_userptr(stack_top);
        top_panel(top->next);
        stack_top = top->next;
        top = (PANEL_DATA *) panel_userptr(stack_top);
        newx = top->x;
        newy = top->y;
        neww = top->w;
        newh = top->h;
        break;
    case 'r':               /* Re-Size */
        size = TRUE;
        attron(COLOR_PAIR(4));
        mvprintw(LINES - 4, 0,
                 "Entered Resizing :Use Arrow Keys to resize and press <ENTER> to end resizing");
        refresh();
        attroff(COLOR_PAIR(4));
        break;
    case 'm':               /* Move */
        attron(COLOR_PAIR(4));
        mvprintw(LINES - 4, 0,
                 "Entered Moving: Use Arrow Keys to Move and press <ENTER> to end moving");
        refresh();
        attroff(COLOR_PAIR(4));
        move = TRUE;
        break;
    case KEY_LEFT:
        if (size == TRUE) {
            --newx;
            ++neww;
        }
```

```
    if (move == TRUE)
        --newx;
    break;
case KEY_RIGHT:
    if (size == TRUE) {
        ++newx;
        --neww;
    }
    if (move == TRUE)
        ++newx;
    break;
case KEY_UP:
    if (size == TRUE) {
        --newy;
        ++newh;
    }
    if (move == TRUE)
        --newy;
    break;
case KEY_DOWN:
    if (size == TRUE) {
        ++newy;
        --newh;
    }
    if (move == TRUE)
        ++newy;
    break;
case 10:                /* Enter */
    move(LINES - 4, 0);
    clrtoeol();
    refresh();
    if (size == TRUE) {
        old_win = panel_window(stack_top);
        temp_win = newwin(newh, neww, newy, newx);
        replace_panel(stack_top, temp_win);
        win_show(temp_win, top->label, top->label_color);
        delwin(old_win);
        size = FALSE;
    }
    if (move == TRUE) {
        move_panel(stack_top, newy, newx);
        move = FALSE;
    }
    break;

}
attron(COLOR_PAIR(4));
mvprintw(LINES - 3, 0, "Use 'm' for moving, 'r' for resizing");
mvprintw(LINES - 2, 0,
         "Use tab to browse through the windows (F1 to Exit)");
attroff(COLOR_PAIR(4));
refresh();
update_panels();
```

```
        doupdate();
    }
    endwin();
    return 0;
}

/* Put all the windows */
void
init_wins(WINDOW **wins, int n)
{
    int x, y, i;
    char label[80];

    y = 2;
    x = 10;
    for (i = 0; i < n; ++i) {
        wins[i] = newwin(NLINES, NCOLS, y, x);
        sprintf(label, "Window Number %d", i + 1);
        win_show(wins[i], label, i + 1);
        y += 3;
        x += 7;
    }
}

/* Set the PANEL_DATA structures for individual panels */
void
set_user_ptrs(PANEL **panels, int n)
{
    PANEL_DATA *ptrs;
    WINDOW *win;
    int x, y, w, h, i;
    char temp[80];

    ptrs = (PANEL_DATA *) calloc((size_t) n, sizeof(PANEL_DATA));

    for (i = 0; i < n; ++i) {
        win = panel_window(panels[i]);
        getbegyx(win, y, x);
        getmaxyx(win, h, w);
        ptrs[i].x = x;
        ptrs[i].y = y;
        ptrs[i].w = w;
        ptrs[i].h = h;
        sprintf(temp, "Window Number %d", i + 1);
        strcpy(ptrs[i].label, temp);
        ptrs[i].label_color = i + 1;
        if (i + 1 == n)
            ptrs[i].next = panels[0];
        else
            ptrs[i].next = panels[i + 1];
        set_panel_userptr(panels[i], &ptrs[i]);
    }
}
```

```
/* Show the window with a border and a label */
void
win_show(WINDOW *win, const char *label, int label_color)
{
    int height, width;

    getmaxyx(win, height, width);
    (void) height;

    box(win, 0, 0);
    mvwaddch(win, 2, 0, ACS_LTEE);
    mvwhline(win, 2, 1, ACS_HLINE, width - 2);
    mvwaddch(win, 2, width - 1, ACS_RTEE);

    print_in_middle(win, 1, 0, width, label, COLOR_PAIR(label_color));
}

void
print_in_middle(WINDOW *win, int starty, int startx,
                int width, const char *string, chtype color)
{
    int length, x, y;
    float temp;

    if (win == NULL)
        win = stdscr;
    getyx(win, y, x);
    if (startx != 0)
        x = startx;
    if (starty != 0)
        y = starty;
    if (width == 0)
        width = 80;

    length = (int) strlen(string);
    temp = (float) (width - length) / 2;
    x = startx + (int) temp;
    wattron(win, color);
    mvwprintw(win, y, x, "%s", string);
    wattroff(win, color);
    refresh();
}
```

Concentrate on the main while loop. Once it finds out the type of key pressed, it takes appropriate action. If ’r’ is pressed resizing mode is started. After this the new sizes are updated as the user presses the arrow keys. When the user presses <ENTER> present selection ends and panel is resized by using the concept explained. While in resizing mode the program doesn’t show how the window is getting resized. It is left as an exercise to the reader to print a dotted border while it gets resized to a new position.

When the user presses 'm' the move mode starts. This is a bit simpler than resizing. As the arrow keys are pressed the new position is updated and pressing of <ENTER> causes the panel to be moved by calling the function move_panel().

In this program the user data which is represented as PANEL_DATA, plays very important role in finding the associated information with a panel. As written in the comments, the PANEL_DATA stores the panel sizes, label, label color and a pointer to the next panel in the cycle.

16.6. Hiding and Showing Panels

A Panel can be hidden by using the function hide_panel(). This function merely removes it form the stack of panels, thus hiding it on the screen once you do update_panels() and doupdate(). It doesn't destroy the PANEL structure associated with the hidden panel. It can be shown again by using the show_panel() function.

The following program shows the hiding of panels. Press 'a' or 'b' or 'c' to show or hide first, second and third windows respectively. It uses a user data with a small variable hide, which keeps track of whether the window is hidden or not. For some reason the function `panel_hidden()` which tells whether a panel is hidden or not is not working. A bug report was also presented by Michael Andres here (http://www.geocrawler.com/archives/3/344/1999/9/0/2643549/)

Example 17. Panel Hiding and Showing example

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

typedef struct _PANEL_DATA {
    int hide;                   /* TRUE if panel is hidden */
} PANEL_DATA;

#define NLINES 10
#define NCOLS 40

void init_wins(WINDOW **wins, int n);
void win_show(WINDOW *win, const char *label, int label_color);
void print_in_middle(WINDOW *win, int starty, int startx,
                     int width, const char *string, chtype color);

int
main(void)
{
    WINDOW *my_wins[3];
    PANEL *my_panels[3];
    PANEL_DATA panel_datas[3];
    PANEL_DATA *temp;
    int ch;

    /* Initialize curses */
```

```
initscr();
start_color();
cbreak();
noecho();
keypad(stdscr, TRUE);

/* Initialize all the colors */
init_pair(1, COLOR_RED, COLOR_BLACK);
init_pair(2, COLOR_GREEN, COLOR_BLACK);
init_pair(3, COLOR_BLUE, COLOR_BLACK);
init_pair(4, COLOR_CYAN, COLOR_BLACK);

init_wins(my_wins, 3);

/* Attach a panel to each window */
/* Order is bottom up */
my_panels[0] = new_panel(my_wins[0]);       /* Push 0, order: stdscr-0 */
my_panels[1] = new_panel(my_wins[1]);       /* Push 1, order: stdscr-0-1 */
my_panels[2] = new_panel(my_wins[2]);       /* Push 2, order: stdscr-0-1-2 */

/* Initialize panel data saying that nothing is hidden */
panel_datas[0].hide = FALSE;
panel_datas[1].hide = FALSE;
panel_datas[2].hide = FALSE;

set_panel_userptr(my_panels[0], &panel_datas[0]);
set_panel_userptr(my_panels[1], &panel_datas[1]);
set_panel_userptr(my_panels[2], &panel_datas[2]);

/* Update the stacking order. 2nd panel will be on top */
update_panels();

/* Show it on the screen */
attron(COLOR_PAIR(4));
mvprintw(LINES - 3, 0,
         "Show or Hide a window with 'a'(first window)  'b'(Second Window)  'c'(Third Window)");
mvprintw(LINES - 2, 0, "F1 to Exit");

attroff(COLOR_PAIR(4));
doupdate();

while ((ch = getch()) != KEY_F(1)) {
    switch (ch) {
    case 'a':
        temp = (PANEL_DATA *) panel_userptr(my_panels[0]);
        if (temp->hide == FALSE) {
            hide_panel(my_panels[0]);
            temp->hide = TRUE;
        } else {
            show_panel(my_panels[0]);
            temp->hide = FALSE;
        }
        break;
```

```
        case 'b':
            temp = (PANEL_DATA *) panel_userptr(my_panels[1]);
            if (temp->hide == FALSE) {
                hide_panel(my_panels[1]);
                temp->hide = TRUE;
            } else {
                show_panel(my_panels[1]);
                temp->hide = FALSE;
            }
            break;
        case 'c':
            temp = (PANEL_DATA *) panel_userptr(my_panels[2]);
            if (temp->hide == FALSE) {
                hide_panel(my_panels[2]);
                temp->hide = TRUE;
            } else {
                show_panel(my_panels[2]);
                temp->hide = FALSE;
            }
            break;
        }
        update_panels();
        doupdate();
    }
    endwin();
    return 0;
}

/* Put all the windows */
void
init_wins(WINDOW **wins, int n)
{
    int x, y, i;
    char label[80];

    y = 2;
    x = 10;
    for (i = 0; i < n; ++i) {
        wins[i] = newwin(NLINES, NCOLS, y, x);
        sprintf(label, "Window Number %d", i + 1);
        win_show(wins[i], label, i + 1);
        y += 3;
        x += 7;
    }
}

/* Show the window with a border and a label */
void
win_show(WINDOW *win, const char *label, int label_color)
{
    int height, width;

    getmaxyx(win, height, width);
```

```
    (void) height;

    box(win, 0, 0);
    mvwaddch(win, 2, 0, ACS_LTEE);
    mvwhline(win, 2, 1, ACS_HLINE, width - 2);
    mvwaddch(win, 2, width - 1, ACS_RTEE);

    print_in_middle(win, 1, 0, width, label, COLOR_PAIR(label_color));
}

void
print_in_middle(WINDOW *win, int starty, int startx,
                int width, const char *string, chtype color)
{
    int length, x, y;
    float temp;

    if (win == NULL)
        win = stdscr;
    getyx(win, y, x);
    if (startx != 0)
        x = startx;
    if (starty != 0)
        y = starty;
    if (width == 0)
        width = 80;

    length = (int) strlen(string);
    temp = (float) (width - length) / 2;
    x = startx + (int) temp;
    wattron(win, color);
    mvwprintw(win, y, x, "%s", string);
    wattroff(win, color);
    refresh();
}
```

16.7. panel_above() and panel_below() Functions

The functions `panel_above()` and `panel_below()` can be used to find out the panel above and below a panel. If the argument to these functions is NULL, then they return a pointer to bottom panel and top panel respectively.

17. Menus Library

The menus library provides a nice extension to basic curses, through which you can create menus. It provides a set of functions to create menus. But they have to be customized to give a nicer look, with colors, etc. Let's get into the details.

A menu is a screen display that assists the user to choose some subset of a given set of items. To put it simple, a menu is a collection of items from which one or more items can be chosen. Some readers might not be aware of multiple item selection capability. Menu library provides functionality to write menus from which the user can chose more than one item as the preferred choice. This is dealt with in a later section. Now it is time for some rudiments.

17.1. The Basics

To create menus, you first create items, and then post the menu to the display. After that, all the processing of user responses is done in an elegant function menu_driver() which is the work horse of any menu program.

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

1. Initialize curses
2. Create items using new_item(). You can specify a name and description for the items.
3. Create the menu with new_menu() by specifying the items to be attached with.
4. Post the menu with menu_post() and refresh the screen.
5. Process the user requests with a loop and do necessary updates to menu with menu_driver.
6. Unpost the menu with menu_unpost()
7. Free the memory allocated to menu by free_menu()
8. Free the memory allocated to the items with free_item()
9. End curses

Let's see a program which prints a simple menu and updates the current selection with up, down arrows.

17.2. Compiling With the Menu Library

To use menu library functions, you have to include menu.h and to link the program with menu library the flag -lmenu should be added along with -lncurses in that order.

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

```
.

compile and link: gcc <program file> -lmenu -lncurses
```

Example 18. Menu Basics

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

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define CTRLD   4

const char *choices[] =
{
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Exit",
};

int
main(void)
{
    ITEM **my_items;
    int c;
    MENU *my_menu;
    int n_choices, i;

    initscr();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    n_choices = ARRAY_SIZE(choices);
    my_items = (ITEM **) calloc((size_t) (n_choices + 1), sizeof(ITEM *));

    for (i = 0; i < n_choices; ++i)
        my_items[i] = new_item(choices[i], choices[i]);
    my_items[n_choices] = (ITEM *) NULL;

    my_menu = new_menu((ITEM **) my_items);
    mvprintw(LINES - 2, 0, "F1 to Exit");
    post_menu(my_menu);
    refresh();

    while ((c = getch()) != KEY_F(1)) {
        switch (c) {
        case KEY_DOWN:
            menu_driver(my_menu, REQ_DOWN_ITEM);
            break;
```

```
        case KEY_UP:
            menu_driver(my_menu, REQ_UP_ITEM);
            break;
        }
    }

    free_item(my_items[0]);
    free_item(my_items[1]);
    free_menu(my_menu);
    endwin();
}
```

This program demonstrates the basic concepts involved in creating a menu using menus library. First we create the items using new_item() and then attach them to the menu with new_menu() function. After posting the menu and refreshing the screen, the main processing loop starts. It reads user input and takes corresponding action. The function menu_driver() is the main work horse of the menu system. The second parameter to this function tells what's to be done with the menu. According to the parameter, menu_driver() does the corresponding task. The value can be either a menu navigational request, an ascii character, or a KEY_MOUSE special key associated with a mouse event.

The menu_driver accepts following navigational requests.

```
     REQ_LEFT_ITEM      Move left to an item.
     REQ_RIGHT_ITEM     Move right to an item.
     REQ_UP_ITEM        Move up to an item.
     REQ_DOWN_ITEM      Move down to an item.
     REQ_SCR_ULINE      Scroll up a line.
     REQ_SCR_DLINE      Scroll down a line.
     REQ_SCR_DPAGE      Scroll down a page.
     REQ_SCR_UPAGE      Scroll up a page.
     REQ_FIRST_ITEM     Move to the first item.
     REQ_LAST_ITEM      Move to the last item.
     REQ_NEXT_ITEM      Move to the next item.
     REQ_PREV_ITEM      Move to the previous item.
     REQ_TOGGLE_ITEM    Select/deselect an item.
     REQ_CLEAR_PATTERN  Clear the menu pattern buffer.
     REQ_BACK_PATTERN   Delete the previous character from the pattern buffer.
     REQ_NEXT_MATCH     Move to the next item matching the pattern match.
     REQ_PREV_MATCH     Move to the previous item matching the pattern match.
```

Don't get overwhelmed by the number of options. We will see them slowly one after another. The options of interest in this example are REQ_UP_ITEM and REQ_DOWN_ITEM. These two options when passed to menu_driver, menu driver updates the current item to one item up or down respectively.

17.3. Menu Driver: The work horse of the menu system

As you have seen in the above example, menu_driver plays an important role in updating the menu. It is very important to understand various options it takes and what they do. As explained above, the second parameter to menu_driver() can be either a navigational request, a printable character or a KEY_MOUSE key. Let's dissect the different navigational requests.

- *REQ_LEFT_ITEM and REQ_RIGHT_ITEM*

  A Menu can be displayed with multiple columns for more than one item. This can be done by using the `menu_format()` function. When a multi columnar menu is displayed these requests cause the menu driver to move the current selection to left or right.

- *REQ_UP_ITEM and REQ_DOWN_ITEM*

  These two options you have seen in the above example. These options when given, makes the menu_driver to move the current selection to an item up or down.

- *REQ_SCR_* options*

  The four options REQ_SCR_ULINE, REQ_SCR_DLINE, REQ_SCR_DPAGE, REQ_SCR_UPAGE are related to scrolling. If all the items in the menu cannot be displayed in the menu sub window, then the menu is scrollable. These requests can be given to the menu_driver to do the scrolling either one line up, down or one page down or up respectively.

- *REQ_FIRST_ITEM, REQ_LAST_ITEM, REQ_NEXT_ITEM and REQ_PREV_ITEM*

  These requests are self explanatory.

- *REQ_TOGGLE_ITEM*

  This request when given, toggles the present selection. This option is to be used only in a multi valued menu. So to use this request the option O_ONEVALUE must be off. This option can be made off or on with set_menu_opts().

- *Pattern Requests*

  Every menu has an associated pattern buffer, which is used to find the nearest match to the ascii characters entered by the user. Whenever ascii characters are given to menu_driver, it puts in to the pattern buffer. It also tries to find the nearest match to the pattern in the items list and moves current selection to that item. The request REQ_CLEAR_PATTERN clears the pattern buffer. The request REQ_BACK_PATTERN deletes the previous character in the pattern buffer. In case the pattern matches more than one item then the matched items can be cycled through REQ_NEXT_MATCH and REQ_PREV_MATCH which move the current selection to the next and previous matches respectively.

- *Mouse Requests*

  In case of KEY_MOUSE requests, according to the mouse position an action is taken accordingly. The action to be taken is explained in the man page as,

```
If  the  second argument is the KEY_MOUSE special key, the
associated mouse event is translated into one of the above
pre-defined  requests.   Currently only clicks in the user
window (e.g. inside the menu display area or  the  decora
tion  window)  are handled. If you click above the display
region of the menu, a REQ_SCR_ULINE is generated,  if  you
doubleclick  a  REQ_SCR_UPAGE  is  generated  and  if  you
tripleclick a REQ_FIRST_ITEM is generated.  If  you  click
below  the  display region of the menu, a REQ_SCR_DLINE is
generated, if you doubleclick a REQ_SCR_DPAGE is generated
and  if  you  tripleclick a REQ_LAST_ITEM is generated. If
you click at an item inside the display area of the  menu,
the menu cursor is positioned to that item.
```

Each of the above requests will be explained in the following lines with several examples whenever appropriate.

17.4. Menu Windows

Every menu created is associated with a window and a sub window. The menu window displays any title or border associated with the menu. The menu sub window displays the menu items currently available for selection. But we didn't specify any window or sub window in the simple example. When a window is not specified, stdscr is taken as the main window, and then menu system calculates the sub window size required for the display of items. Then items are displayed in the calculated sub window. So let's play with these windows and display a menu with a border and a title.

Example 19. Menu Windows Usage example

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

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define CTRLD   4

const char *choices[] =
{
    "Choice 1",
    "Choice 2",
    "Choice 3",
```

```
    "Choice 4",
    "Exit",
    (char *) NULL,
};

void print_in_middle(WINDOW *win, int starty, int startx,
                     int width, const char *string, chtype color);

int
main(void)
{
    ITEM **my_items;
    int c;
    MENU *my_menu;
    WINDOW *my_menu_win;
    int n_choices, i;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);
    init_pair(1, COLOR_RED, COLOR_BLACK);

    /* Create items */
    n_choices = ARRAY_SIZE(choices);
    my_items = (ITEM **) calloc((size_t) n_choices, sizeof(ITEM *));
    for (i = 0; i < n_choices; ++i)
        my_items[i] = new_item(choices[i], choices[i]);

    /* Create menu */
    my_menu = new_menu((ITEM **) my_items);

    /* Create the window to be associated with the menu */
    my_menu_win = newwin(10, 40, 4, 4);
    keypad(my_menu_win, TRUE);

    /* Set main window and sub window */
    set_menu_win(my_menu, my_menu_win);
    set_menu_sub(my_menu, derwin(my_menu_win, 6, 38, 3, 1));

    /* Set menu mark to the string " * " */
    set_menu_mark(my_menu, " * ");

    /* Print a border around the main window and print a title */
    box(my_menu_win, 0, 0);
    print_in_middle(my_menu_win, 1, 0, 40, "My Menu", COLOR_PAIR(1));
    mvwaddch(my_menu_win, 2, 0, ACS_LTEE);
    mvwhline(my_menu_win, 2, 1, ACS_HLINE, 38);
    mvwaddch(my_menu_win, 2, 39, ACS_RTEE);
    mvprintw(LINES - 2, 0, "F1 to exit");
    refresh();
```

```
    /* Post the menu */
    post_menu(my_menu);
    wrefresh(my_menu_win);

    while ((c = wgetch(my_menu_win)) != KEY_F(1)) {
        switch (c) {
        case KEY_DOWN:
            menu_driver(my_menu, REQ_DOWN_ITEM);
            break;
        case KEY_UP:
            menu_driver(my_menu, REQ_UP_ITEM);
            break;
        }
        wrefresh(my_menu_win);
    }

    /* Unpost and free all the memory taken up */
    unpost_menu(my_menu);
    free_menu(my_menu);
    for (i = 0; i < n_choices; ++i)
        free_item(my_items[i]);
    endwin();
}

void
print_in_middle(WINDOW *win, int starty, int startx,
                int width, const char *string, chtype color)
{
    int length, x, y;
    float temp;

    if (win == NULL)
        win = stdscr;
    getyx(win, y, x);
    if (startx != 0)
        x = startx;
    if (starty != 0)
        y = starty;
    if (width == 0)
        width = 80;

    length = (int) strlen(string);
    temp = (float) (width - length) / 2;
    x = startx + (int) temp;
    wattron(win, color);
    mvwprintw(win, y, x, "%s", string);
    wattroff(win, color);
    refresh();
}
```

This example creates a menu with a title, border, a fancy line separating title and the items. As you can see, in order to attach a window to a menu the function set_menu_win() has to be used. Then we attach the sub window also. This displays the items in the sub window. You can also set the mark string which gets displayed to the left of the selected item with set_menu_mark().

17.5. Scrolling Menus

If the sub window given for a window is not big enough to show all the items, then the menu will be scrollable. When you are on the last item in the present list, if you send REQ_DOWN_ITEM, it gets translated into REQ_SCR_DLINE and the menu scrolls by one item. You can manually give REQ_SCR_ operations to do scrolling. Let's see how it can be done.

Example 20. Scrolling Menus example

```
#include <stdlib.h>
#include <string.h>
#include <curses.h>
#include <menu.h>

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define CTRLD   4

const char *choices[] =
{
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Choice 5",
    "Choice 6",
    "Choice 7",
    "Choice 8",
    "Choice 9",
    "Choice 10",
    "Exit",
    (char *) NULL,
};

void print_in_middle(WINDOW *win, int starty, int startx,
                     int width, const char *string, chtype color);

int
main(void)
{
    ITEM **my_items;
    int c;
    MENU *my_menu;
    WINDOW *my_menu_win;
    int n_choices, i;
```

```
/* Initialize curses */
initscr();
start_color();
cbreak();
noecho();
keypad(stdscr, TRUE);
init_pair(1, COLOR_RED, COLOR_BLACK);
init_pair(2, COLOR_CYAN, COLOR_BLACK);

/* Create items */
n_choices = ARRAY_SIZE(choices);
my_items = (ITEM **) calloc((size_t) n_choices, sizeof(ITEM *));
for (i = 0; i < n_choices; ++i)
    my_items[i] = new_item(choices[i], choices[i]);

/* Create menu */
my_menu = new_menu((ITEM **) my_items);

/* Create the window to be associated with the menu */
my_menu_win = newwin(10, 40, 4, 4);
keypad(my_menu_win, TRUE);

/* Set main window and sub window */
set_menu_win(my_menu, my_menu_win);
set_menu_sub(my_menu, derwin(my_menu_win, 6, 38, 3, 1));
set_menu_format(my_menu, 5, 1);

/* Set menu mark to the string " * " */
set_menu_mark(my_menu, " * ");

/* Print a border around the main window and print a title */
box(my_menu_win, 0, 0);
print_in_middle(my_menu_win, 1, 0, 40, "My Menu", COLOR_PAIR(1));
mvwaddch(my_menu_win, 2, 0, ACS_LTEE);
mvwhline(my_menu_win, 2, 1, ACS_HLINE, 38);
mvwaddch(my_menu_win, 2, 39, ACS_RTEE);

/* Post the menu */
post_menu(my_menu);
wrefresh(my_menu_win);

attron(COLOR_PAIR(2));
mvprintw(LINES - 2, 0,
         "Use PageUp and PageDown to scroll down or up a page of items");
mvprintw(LINES - 1, 0, "Arrow Keys to navigate (F1 to Exit)");
attroff(COLOR_PAIR(2));
refresh();

while ((c = wgetch(my_menu_win)) != KEY_F(1)) {
    switch (c) {
    case KEY_DOWN:
        menu_driver(my_menu, REQ_DOWN_ITEM);
        break;
```

```
        case KEY_UP:
            menu_driver(my_menu, REQ_UP_ITEM);
            break;
        case KEY_NPAGE:
            menu_driver(my_menu, REQ_SCR_DPAGE);
            break;
        case KEY_PPAGE:
            menu_driver(my_menu, REQ_SCR_UPAGE);
            break;
        }
        wrefresh(my_menu_win);
    }

    /* Unpost and free all the memory taken up */
    unpost_menu(my_menu);
    free_menu(my_menu);
    for (i = 0; i < n_choices; ++i)
        free_item(my_items[i]);
    endwin();
}

void
print_in_middle(WINDOW *win, int starty, int startx,
                int width, const char *string, chtype color)
{
    int length, x, y;
    float temp;

    if (win == NULL)
        win = stdscr;
    getyx(win, y, x);
    if (startx != 0)
        x = startx;
    if (starty != 0)
        y = starty;
    if (width == 0)
        width = 80;

    length = (int) strlen(string);
    temp = (float) (width - length) / 2;
    x = startx + (int) temp;
    wattron(win, color);
    mvwprintw(win, y, x, "%s", string);
    wattroff(win, color);
    refresh();
}
```

This program is self-explanatory. In this example the number of choices has been increased to ten, which is larger than our sub window size which can hold 6 items. This message has to be explicitly conveyed to the menu system with the function set_menu_format(). In here we specify the number of rows and columns we want to be displayed for a single page. We can specify any number of items to be shown, in the rows variables, if it is less than the height of the sub window. If the key pressed by the user is a PAGE

UP or PAGE DOWN, the menu is scrolled a page due to the requests (REQ_SCR_DPAGE and REQ_SCR_UPAGE) given to menu_driver().

17.6. Multi Columnar Menus

In the above example you have seen how to use the function set_menu_format(). I didn't mention what the cols variable (third parameter) does. Well, If your sub window is wide enough, you can opt to display more than one item per row. This can be specified in the cols variable. To make things simpler, the following example doesn't show descriptions for the items.

Example 21. Milt Columnar Menus Example

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

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define CTRLD   4

const char *choices[] =
{
    "Choice 1", "Choice 2", "Choice 3", "Choice 4", "Choice 5",
    "Choice 6", "Choice 7", "Choice 8", "Choice 9", "Choice 10",
    "Choice 11", "Choice 12", "Choice 13", "Choice 14", "Choice 15",
    "Choice 16", "Choice 17", "Choice 18", "Choice 19", "Choice 20",
    "Exit",
    (char *) NULL,
};

int
main(void)
{
    ITEM **my_items;
    int c;
    MENU *my_menu;
    WINDOW *my_menu_win;
    int n_choices, i;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);
    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_CYAN, COLOR_BLACK);

    /* Create items */
    n_choices = ARRAY_SIZE(choices);
    my_items = (ITEM **) calloc((size_t) n_choices, sizeof(ITEM *));
```

```
for (i = 0; i < n_choices; ++i)
    my_items[i] = new_item(choices[i], choices[i]);

/* Create menu */
my_menu = new_menu((ITEM **) my_items);

/* Set menu option not to show the description */
menu_opts_off(my_menu, O_SHOWDESC);

/* Create the window to be associated with the menu */
my_menu_win = newwin(10, 70, 4, 4);
keypad(my_menu_win, TRUE);

/* Set main window and sub window */
set_menu_win(my_menu, my_menu_win);
set_menu_sub(my_menu, derwin(my_menu_win, 6, 68, 3, 1));
set_menu_format(my_menu, 5, 3);
set_menu_mark(my_menu, " * ");

/* Print a border around the main window and print a title */
box(my_menu_win, 0, 0);

attron(COLOR_PAIR(2));
mvprintw(LINES - 3, 0, "Use PageUp and PageDown to scroll");
mvprintw(LINES - 2, 0, "Use Arrow Keys to navigate (F1 to Exit)");
attroff(COLOR_PAIR(2));
refresh();

/* Post the menu */
post_menu(my_menu);
wrefresh(my_menu_win);

while ((c = wgetch(my_menu_win)) != KEY_F(1)) {
    switch (c) {
    case KEY_DOWN:
        menu_driver(my_menu, REQ_DOWN_ITEM);
        break;
    case KEY_UP:
        menu_driver(my_menu, REQ_UP_ITEM);
        break;
    case KEY_LEFT:
        menu_driver(my_menu, REQ_LEFT_ITEM);
        break;
    case KEY_RIGHT:
        menu_driver(my_menu, REQ_RIGHT_ITEM);
        break;
    case KEY_NPAGE:
        menu_driver(my_menu, REQ_SCR_DPAGE);
        break;
    case KEY_PPAGE:
        menu_driver(my_menu, REQ_SCR_UPAGE);
        break;
    }
```

```
        wrefresh(my_menu_win);
    }

    /* Unpost and free all the memory taken up */
    unpost_menu(my_menu);
    free_menu(my_menu);
    for (i = 0; i < n_choices; ++i)
        free_item(my_items[i]);
    endwin();
}
```

Watch the function call to set_menu_format(). It specifies the number of columns to be 3, thus displaying 3 items per row. We have also switched off the showing descriptions with the function menu_opts_off(). There are couple of functions set_menu_opts(), menu_opts_on() and menu_opts() which can be used to manipulate menu options. The following menu options can be specified.

```
       O_ONEVALUE
            Only one item can be selected for this menu.

       O_SHOWDESC
            Display  the  item  descriptions  when  the  menu  is
            posted.

       O_ROWMAJOR
            Display the menu in row-major order.

       O_IGNORECASE
            Ignore the case when pattern-matching.

       O_SHOWMATCH
            Move the cursor to within the item  name  while  pat
            tern-matching.

       O_NONCYCLIC
            Don't   wrap   around  next-item  and  previous-item,
            requests to the other end of the menu.
```

All options are on by default. You can switch specific attributes on or off with menu_opts_on() and menu_opts_off() functions. You can also use set_menu_opts() to directly specify the options. The argument to this function should be a OR ed value of some of those above constants. The function menu_opts() can be used to find out a menu's present options.

17.7. Multi Valued Menus

You might be wondering what if you switch off the option O_ONEVALUE. Then the menu becomes multi-valued. That means you can select more than one item. This brings us to the request REQ_TOGGLE_ITEM. Let's see it in action.

Example 22. Multi Valued Menus example

```
#include <stdlib.h>
#include <string.h>
#include <curses.h>
#include <menu.h>

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define CTRLD   4

const char *choices[] =
{
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Choice 5",
    "Choice 6",
    "Choice 7",
    "Exit",
};

int
main(void)
{
    ITEM **my_items;
    int c;
    MENU *my_menu;
    int n_choices, i;

    /* Initialize curses */
    initscr();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    /* Initialize items */
    n_choices = ARRAY_SIZE(choices);
    my_items = (ITEM **) calloc((size_t) (n_choices + 1), sizeof(ITEM *));
    for (i = 0; i < n_choices; ++i)
        my_items[i] = new_item(choices[i], choices[i]);
    my_items[n_choices] = (ITEM *) NULL;

    my_menu = new_menu((ITEM **) my_items);

    /* Make the menu multi valued */
    menu_opts_off(my_menu, O_ONEVALUE);

    mvprintw(LINES - 3, 0, "Use <SPACE> to select or unselect an item.");
    mvprintw(LINES - 2, 0,
             "<ENTER> to see presently selected items(F1 to Exit)");
    post_menu(my_menu);
    refresh();
```

```
    while ((c = getch()) != KEY_F(1)) {
        switch (c) {
        case KEY_DOWN:
            menu_driver(my_menu, REQ_DOWN_ITEM);
            break;
        case KEY_UP:
            menu_driver(my_menu, REQ_UP_ITEM);
            break;
        case ' ':
            menu_driver(my_menu, REQ_TOGGLE_ITEM);
            break;
        case 10:                /* Enter */
            {
                char temp[200];
                ITEM **items;

                items = menu_items(my_menu);
                temp[0] = '\0';
                for (i = 0; i < item_count(my_menu); ++i)
                    if (item_value(items[i]) == TRUE) {
                        strcat(temp, item_name(items[i]));
                        strcat(temp, " ");
                    }
                move(20, 0);
                clrtoeol();
                mvaddstr(20, 0, temp);
                refresh();
            }
            break;
        }
    }

    free_item(my_items[0]);
    free_item(my_items[1]);
    free_menu(my_menu);
    endwin();
}
```

Whew, A lot of new functions. Let's take them one after another. Firstly, the REQ_TOGGLE_ITEM. In a multi-valued menu, the user should be allowed to select or un select more than one item. The request REQ_TOGGLE_ITEM toggles the present selection. In this case when space is pressed REQ_TOGGLE_ITEM request is sent to menu_driver to achieve the result.

Now when the user presses <ENTER> we show the items he presently selected. First we find out the items associated with the menu using the function menu_items(). Then we loop through the items to find out if the item is selected or not. The function item_value() returns TRUE if an item is selected. The function item_count() returns the number of items in the menu. The item name can be found with item_name(). You can also find the description associated with an item using item_description().

17.8. Menu Options

Well, by this time you must be itching for some difference in your menu, with lots of functionality. I know. You want Colors !!!. You want to create nice menus similar to those text mode dos games (http://www.jersey.net/~debinjoe/games/). The functions set_menu_fore() and set_menu_back() can be used to change the attribute of the selected item and unselected item. The names are misleading. They don't change menu's foreground or background which would have been useless.

The function set_menu_grey() can be used to set the display attribute for the non-selectable items in the menu. This brings us to the interesting option for an item the one and only O_SELECTABLE. We can turn it off by the function item_opts_off() and after that that item is not selectable. It is like a grayed item in those fancy windows menus. Let's put these concepts in practice with this example

Example 23. Menu Options example

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

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define CTRLD   4

const char *choices[] =
{
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Choice 5",
    "Choice 6",
    "Choice 7",
    "Exit",
};

int
main(void)
{
    ITEM **my_items;
    int c;
    MENU *my_menu;
    int n_choices, i;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);
    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_GREEN, COLOR_BLACK);
    init_pair(3, COLOR_MAGENTA, COLOR_BLACK);
```

```
    /* Initialize items */
    n_choices = ARRAY_SIZE(choices);
    my_items = (ITEM **) calloc((size_t) (n_choices + 1), sizeof(ITEM *));
    for (i = 0; i < n_choices; ++i)
        my_items[i] = new_item(choices[i], choices[i]);
    my_items[n_choices] = (ITEM *) NULL;
    item_opts_off(my_items[3], O_SELECTABLE);
    item_opts_off(my_items[6], O_SELECTABLE);

    /* Create menu */
    my_menu = new_menu((ITEM **) my_items);

    /* Set fore ground and back ground of the menu */
    set_menu_fore(my_menu, COLOR_PAIR(1) | A_REVERSE);
    set_menu_back(my_menu, COLOR_PAIR(2));
    set_menu_grey(my_menu, COLOR_PAIR(3));

    /* Post the menu */
    mvprintw(LINES - 3, 0, "Press <ENTER> to see the option selected");
    mvprintw(LINES - 2, 0, "Up and Down arrow keys to navigate (F1 to Exit)");
    post_menu(my_menu);
    refresh();

    while ((c = getch()) != KEY_F(1)) {
        switch (c) {
        case KEY_DOWN:
            menu_driver(my_menu, REQ_DOWN_ITEM);
            break;
        case KEY_UP:
            menu_driver(my_menu, REQ_UP_ITEM);
            break;
        case 10:                /* Enter */
            move(20, 0);
            clrtoeol();
            mvprintw(20, 0, "Item selected is : %s",
                     item_name(current_item(my_menu)));
            pos_menu_cursor(my_menu);
            break;
        }
    }
    unpost_menu(my_menu);
    for (i = 0; i < n_choices; ++i)
        free_item(my_items[i]);
    free_menu(my_menu);
    endwin();
}
```

17.9. The useful User Pointer

We can associate a user pointer with each item in the menu. It works the same way as user pointer in panels. It is not touched by menu system. You can store any thing you like in that. I usually use it to store the function to be executed when the menu option is chosen (It is selected and may be the user pressed <ENTER>);

Example 24. Menu User Pointer Usage

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

#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define CTRLD   4

const char *choices[] =
{
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Choice 5",
    "Choice 6",
    "Choice 7",
    "Exit",
};

typedef union {
    void (*my_func) (const char *);
    void *data;
} MY_DATA;

void func(const char *name);

int
main(void)
{
    ITEM **my_items;
    int c;
    MENU *my_menu;
    int n_choices, i;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);
    init_pair(1, COLOR_RED, COLOR_BLACK);
    init_pair(2, COLOR_GREEN, COLOR_BLACK);
    init_pair(3, COLOR_MAGENTA, COLOR_BLACK);
```

```
    /* Initialize items */
    n_choices = ARRAY_SIZE(choices);
    my_items = (ITEM **) calloc((size_t) (n_choices + 1), sizeof(ITEM *));
    for (i = 0; i < n_choices; ++i) {
        MY_DATA data = { func };
        my_items[i] = new_item(choices[i], choices[i]);
        /* Set the user pointer */
        set_item_userptr(my_items[i], (void *) &data);
    }
    my_items[n_choices] = (ITEM *) NULL;

    /* Create menu */
    my_menu = new_menu((ITEM **) my_items);

    /* Post the menu */
    mvprintw(LINES - 3, 0, "Press <ENTER> to see the option selected");
    mvprintw(LINES - 2, 0, "Up and Down arrow keys to navigate (F1 to Exit)");
    post_menu(my_menu);
    refresh();

    while ((c = getch()) != KEY_F(1)) {
        switch (c) {
        case KEY_DOWN:
            menu_driver(my_menu, REQ_DOWN_ITEM);
            break;
        case KEY_UP:
            menu_driver(my_menu, REQ_UP_ITEM);
            break;
        case 10:                /* Enter */
            {
                ITEM *cur;
                const MY_DATA *data;

                cur = current_item(my_menu);
                data = item_userptr(cur);
                data->my_func(item_name(cur));
                pos_menu_cursor(my_menu);
                break;
            }
            break;
        }
    }
    unpost_menu(my_menu);
    for (i = 0; i < n_choices; ++i)
        free_item(my_items[i]);
    free_menu(my_menu);
    endwin();
}

void
func(const char *name)
{
```

```
    move(20, 0);
    clrtoeol();
    mvprintw(20, 0, "Item selected is : %s", name);
}
```

18. Forms Library

Well. If you have seen those forms on web pages which take input from users and do various kinds of things, you might be wondering how would any one create such forms in text mode display. It is quite difficult to write those nifty forms in plain ncurses. Forms library tries to provide a basic frame work to build and maintain forms with ease. It has lot of features(functions) which manage validation, dynamic expansion of fields, etc. Let's see it in full flow.

A form is a collection of fields; each field can be either a label(static text) or a data-entry location. The forms also library provides functions to divide forms into multiple pages.

18.1. The Basics

Forms are created in much the same way as menus. First the fields related to the form are created with new_field(). You can set options for the fields, so that they can be displayed with some fancy attributes, validated before the field looses focus, etc. Then the fields are attached to form. After this, the form can be posted to display and is ready to receive inputs. On the similar lines to menu_driver(), the form is manipulated with form_driver(). We can send requests to form_driver to move focus to a certain field, move cursor to end of the field etc. After the user enters values in the fields and validation done, form can be unposted and memory allocated can be freed.

The general flow of control of a forms program looks like this.

1. Initialize curses
2. Create fields using new_field(). You can specify the height and width of the field, and its position on the form.
3. Create the forms with new_form() by specifying the fields to be attached with.
4. Post the form with form_post() and refresh the screen.
5. Process the user requests with a loop and do necessary updates to form with form_driver.
6. Unpost the menu with form_unpost()
7. Free the memory allocated to menu by free_form()
8. Free the memory allocated to the items with free_field()
9. End curses

As you can see, working with forms library is much similar to handling menu library. The following examples will explore various aspects of form processing. Let's start the journey with a simple example. first.

18.2. Compiling With the Forms Library

To use forms library functions, you have to include form.h and to link the program with forms library the flag -lform should be added along with -lncurses in that order.

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

compile and link: gcc <program file> -lform -lncurses
```

Example 25. Forms Basics

```
#include <form.h>

int
main(void)
{
    FIELD *field[3];
    FORM *my_form;
    int ch;

    /* Initialize curses */
    initscr();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    /* Initialize the fields */
    field[0] = new_field(1, 10, 4, 18, 0, 0);
    field[1] = new_field(1, 10, 6, 18, 0, 0);
    field[2] = NULL;

    /* Set field options */
    set_field_back(field[0], A_UNDERLINE);      /* Print a line for the option  */
    field_opts_off(field[0], O_AUTOSKIP);       /* Don't go to next field when this */
                                                /* Field is filled up           */
    set_field_back(field[1], A_UNDERLINE);
    field_opts_off(field[1], O_AUTOSKIP);

    /* Create the form and post it */
    my_form = new_form(field);
    post_form(my_form);
    refresh();
```

```
    mvprintw(4, 10, "Value 1:");
    mvprintw(6, 10, "Value 2:");
    refresh();

    /* Loop through to get user requests */
    while ((ch = getch()) != KEY_F(1)) {
        switch (ch) {
        case KEY_DOWN:
            /* Go to next field */
            form_driver(my_form, REQ_NEXT_FIELD);
            /* Go to the end of the present buffer */
            /* Leaves nicely at the last character */
            form_driver(my_form, REQ_END_LINE);
            break;
        case KEY_UP:
            /* Go to previous field */
            form_driver(my_form, REQ_PREV_FIELD);
            form_driver(my_form, REQ_END_LINE);
            break;
        default:
            /* If this is a normal character, it gets */
            /* Printed                                */
            form_driver(my_form, ch);
            break;
        }
    }

    /* Un post form and free the memory */
    unpost_form(my_form);
    free_form(my_form);
    free_field(field[0]);
    free_field(field[1]);

    endwin();
    return 0;
}
```

Above example is pretty straight forward. It creates two fields with `new_field()`. new_field() takes height, width, starty, startx, number of offscreen rows and number of additional working buffers. The fifth argument number of offscreen rows specifies how much of the field to be shown. If it is zero, the entire field is always displayed otherwise the form will be scrollable when the user accesses not displayed parts of the field. The forms library allocates one buffer per field to store the data user enters. Using the last parameter to new_field() we can specify it to allocate some additional buffers. These can be used for any purpose you like.

After creating the fields, back ground attribute of both of them is set to an underscore with set_field_back(). The AUTOSKIP option is turned off using field_opts_off(). If this option is turned on, focus will move to the next field in the form once the active field is filled up completely.

After attaching the fields to the form, it is posted. Here on, user inputs are processed in the while loop, by making corresponding requests to form_driver. The details of all the requests to the form_driver() are explained later.

18.3. Playing with Fields

Each form field is associated with a lot of attributes. They can be manipulated to get the required effect and to have fun !!!. So why wait?

18.3.1. Fetching Size and Location of Field

The parameters we have given at the time of creation of a field can be retrieved with field_info(). It returns height, width, starty, startx, number of offscreen rows, and number of additional buffers into the parameters given to it. It is a sort of inverse of new_field().

```
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 */
```

18.3.2. Moving the field

The location of the field can be moved to a different position with move_field().

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

As usual, the changed position can be queried with field_infor().

18.3.3. Field Justification

The justification to be done for the field can be fixed using the function set_field_just().

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

The justification mode valued accepted and returned by these functions are NO_JUSTIFICATION, JUSTIFY_RIGHT, JUSTIFY_LEFT, or JUSTIFY_CENTER.

18.3.4. Field Display Attributes

As you have seen, in the above example, display attribute for the fields can be set with set_field_fore() and setfield_back(). These functions set foreground and background attribute of the fields. You can also specify a pad character which will be filled in the unfilled portion of the field. The pad character is set with a call to set_field_pad(). Default pad value is a space. The functions field_fore(), field_back, field_pad() can be used to query the present foreground, background attributes and pad character for the field. The following list gives the usage of functions.

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

chtype field_fore(FIELD *field);        /* field to query */
                                        /* returns foreground attribute */

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

chtype field_back(FIELD *field);        /* field to query */
                                        /* returns background attribute */

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

chtype field_pad(FIELD *field);         /* field to query */
                                        /* returns present pad character */
```

Though above functions seem quite simple, using colors with set_field_fore() may be frustrating in the beginning. Let me first explain about foreground and background attributes of a field. The foreground attribute is associated with the character. That means a character in the field is printed with the attribute you have set with set_field_fore(). Background attribute is the attribute used to fill background of field, whether any character is there or not. So what about colors? Since colors are always defined in pairs, what is the right way to display colored fields? Here's an example clarifying color attributes.

Example 26. Form Attributes example

```
#include <form.h>

int
main(void)
{
    FIELD *field[3];
    FORM *my_form;
    int ch;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
```

```
keypad(stdscr, TRUE);

/* Initialize few color pairs */
init_pair(1, COLOR_WHITE, COLOR_BLUE);
init_pair(2, COLOR_WHITE, COLOR_BLUE);

/* Initialize the fields */
field[0] = new_field(1, 10, 4, 18, 0, 0);
field[1] = new_field(1, 10, 6, 18, 0, 0);
field[2] = NULL;

/* Set field options */
set_field_fore(field[0], COLOR_PAIR(1));    /* Put the field with blue background */
set_field_back(field[0], COLOR_PAIR(2));    /* and white foreground (characters */
                                            /* are printed in white         */
field_opts_off(field[0], O_AUTOSKIP);       /* Don't go to next field when this */
                                            /* Field is filled up           */
set_field_back(field[1], A_UNDERLINE);
field_opts_off(field[1], O_AUTOSKIP);

/* Create the form and post it */
my_form = new_form(field);
post_form(my_form);
refresh();

set_current_field(my_form, field[0]);       /* Set focus to the colored field */
mvprintw(4, 10, "Value 1:");
mvprintw(6, 10, "Value 2:");
mvprintw(LINES - 2, 0,
         "Use UP, DOWN arrow keys to switch between fields");
refresh();

/* Loop through to get user requests */
while ((ch = getch()) != KEY_F(1)) {
    switch (ch) {
    case KEY_DOWN:
        /* Go to next field */
        form_driver(my_form, REQ_NEXT_FIELD);
        /* Go to the end of the present buffer */
        /* Leaves nicely at the last character */
        form_driver(my_form, REQ_END_LINE);
        break;
    case KEY_UP:
        /* Go to previous field */
        form_driver(my_form, REQ_PREV_FIELD);
        form_driver(my_form, REQ_END_LINE);
        break;
    default:
        /* If this is a normal character, it gets */
        /* Printed                                */
        form_driver(my_form, ch);
        break;
    }
```

```
    }

    /* Un post form and free the memory */
    unpost_form(my_form);
    free_form(my_form);
    free_field(field[0]);
    free_field(field[1]);

    endwin();
    return 0;
}
```

Play with the color pairs and try to understand the foreground and background attributes. In my programs using color attributes, I usually set only the background with set_field_back(). Curses simply doesn't allow defining individual color attributes.

18.3.5. Field Option Bits

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

```
int set_field_opts(FIELD *field,          /* field to alter */
                   int 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 */
```

The function set_field_opts() can be used to directly set attributes of a field or you can choose to switch a few attributes on and off with field_opts_on() and field_opts_off() selectively. Anytime you can query the attributes of a field with field_opts(). The following is the list of available options. By default, all options are on.

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. You have seen the usage of switching off O_AUTOSKIP option. The following example clarifies usage of some more options. Other options are explained where appropriate.

Example 27. Field Options Usage example

```
#include <form.h>

#define STARTX 15
#define STARTY 4
#define WIDTH 25

#define N_FIELDS 3

int
main(void)
{
    FIELD *field[N_FIELDS];
    FORM *my_form;
    int ch, i;

    /* Initialize curses */
    initscr();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    /* Initialize the fields */
    for (i = 0; i < N_FIELDS - 1; ++i)
        field[i] = new_field(1, WIDTH, STARTY + i * 2, STARTX, 0, 0);
    field[N_FIELDS - 1] = NULL;

    /* Set field options */
    set_field_back(field[1], A_UNDERLINE);      /* Print a line for the option  */

    field_opts_off(field[0], O_ACTIVE);         /* This field is a static label */
    field_opts_off(field[1], O_PUBLIC);         /* This filed is like a password field */
    field_opts_off(field[1], O_AUTOSKIP);       /* To avoid entering the same field */
                                                /* after last character is entered */

    /* Create the form and post it */
    my_form = new_form(field);
    post_form(my_form);
    refresh();

    set_field_just(field[0], JUSTIFY_CENTER);   /* Center Justification */
    set_field_buffer(field[0], 0, "This is a static Field");
    /* Initialize the field  */
    mvprintw(STARTY, STARTX - 10, "Field 1:");
    mvprintw(STARTY + 2, STARTX - 10, "Field 2:");
    refresh();

    /* Loop through to get user requests */
    while ((ch = getch()) != KEY_F(1)) {
        switch (ch) {
        case KEY_DOWN:
            /* Go to next field */
```

```
            form_driver(my_form, REQ_NEXT_FIELD);
            /* Go to the end of the present buffer */
            /* Leaves nicely at the last character */
            form_driver(my_form, REQ_END_LINE);
            break;
        case KEY_UP:
            /* Go to previous field */
            form_driver(my_form, REQ_PREV_FIELD);
            form_driver(my_form, REQ_END_LINE);
            break;
        default:
            /* If this is a normal character, it gets */
            /* Printed                                */
            form_driver(my_form, ch);
            break;
        }
    }

    /* Un post form and free the memory */
    unpost_form(my_form);
    free_form(my_form);
    free_field(field[0]);
    free_field(field[1]);

    endwin();
    return 0;
}
```

This example, though useless, shows the usage of options. If used properly, they can present information very effectively in a form. The second field being not O_PUBLIC, does not show the characters you are typing.

18.3.6. Field Status

The field status specifies whether the field has got edited or not. It is initially set to FALSE and when user enters something and the data buffer gets modified it becomes TRUE. So a field's status can be queried to find out whether it has been modified or not. The following functions can assist in those operations.

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

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

It is better to check the field's status only after after leaving the field, as data buffer might not have been updated yet as the validation is still due. To guarantee that right status is returned, 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

18.3.7. Field User Pointer

Every field structure contains one pointer that can be used by the user for various purposes. It is not touched by forms library and can be used for any purpose by the user. The following functions set and fetch user pointer.

```
int set_field_userptr(FIELD *field,
           char *userptr);      /* the user pointer you wish to associate */
                                /* with the field    */

char *field_userptr(FIELD *field);      /* fetch user pointer of the field */
```

18.3.8. Variable-Sized Fields

If you want a dynamically changing field with variable width, this is the feature you want to put to full use. This will allow the user to enter more data than the original size of the field and let the field grow. According to the field orientation it will scroll horizontally or vertically to incorporate the new data.

To make a field dynamically growable, the option O_STATIC should be turned off. This can be done with a

```
    field_opts_off(field_pointer, O_STATIC);
```

But it is usually not advisable to allow a field to grow infinitely. You can set a maximum limit to the growth of the field with

```
int set_max_field(FIELD *field,    /* Field on which to operate */
                  int max_growth); /* maximum growth allowed for the field */
```

The field info for a dynamically growable field can be retrieved by

```
int dynamic_field_info( FIELD *field,     /* Field on which to operate */
            int   *prows,     /* number of rows will be filled in this */
            int   *pcols,     /* number of columns will be filled in this*/
            int   *pmax)      /* maximum allowable growth will be filled */
                              /* in this */
```

Though field_info work as usual, it is advisable to use this function to get the proper attributes of a dynamically growable field.

Recall the library routine new_field; a new field created with height set to one will be defined to be a one line field. A new field created with height greater than one will be defined to be a multi line field.

A one line field with O_STATIC turned off (dynamically growable field) will contain a single fixed row, but the number of columns can increase if the user enters more data than the initial field will hold. The number of columns displayed will remain fixed and the additional data will scroll horizontally.

A multi line field with O_STATIC turned off (dynamically growable field) will contain a fixed number of columns, but the number of rows can increase if the user enters more data than the initial field will hold. The number of rows displayed will remain fixed and the additional data will scroll vertically.

The above two paragraphs pretty much describe a dynamically growable field's behavior. The way other parts of forms library behaves is described below:

1. The field option O_AUTOSKIP will be ignored if the option O_STATIC is off and there is no maximum growth specified for the field. Currently, O_AUTOSKIP generates an automatic REQ_NEXT_FIELD form driver request when the user types in the last character position of a field. On a growable field with no maximum growth specified, there is no last character position. If a maximum growth is specified, the O_AUTOSKIP option will work as normal if the field has grown to its maximum size.
2. The field justification will be ignored if the option O_STATIC is off. Currently, set_field_just can be used to JUSTIFY_LEFT, JUSTIFY_RIGHT, JUSTIFY_CENTER the contents of a one line field. A growable one line field will, by definition, grow and scroll horizontally and may contain more data than can be justified. The return from field_just will be unchanged.
3. The overloaded form driver request REQ_NEW_LINE will operate the same way regardless of the O_NL_OVERLOAD form option if the field option O_STATIC is off and there is no maximum growth specified for the field. Currently, if the form option O_NL_OVERLOAD is on, REQ_NEW_LINE implicitly generates a REQ_NEXT_FIELD if called from the last line of a field. If a field can grow without bound, there is no last line, so REQ_NEW_LINE will never implicitly generate a REQ_NEXT_FIELD. If a maximum growth limit is specified and the O_NL_OVERLOAD form option is on, REQ_NEW_LINE will only implicitly generate REQ_NEXT_FIELD if the field has grown to its maximum size and the user is on the last line.
4. The library call dup_field will work as usual; it will duplicate the field, including the current buffer size and contents of the field being duplicated. Any specified maximum growth will also be duplicated.
5. The library call link_field will work as usual; it will duplicate all field attributes and share buffers with the field being linked. If the O_STATIC field option is subsequently changed by a field sharing buffers, how the system reacts to an attempt to enter more data into the field than the buffer will currently hold will depend on the setting of the option in the current field.
6. The library call field_info will work as usual; the variable nrow will contain the value of the original call to new_field. The user should use dynamic_field_info, described above, to query the current size of the buffer.

Some of the above points make sense only after explaining form driver. We will be looking into that in next few sections.

18.4. Form Windows

The form windows concept is pretty much similar to menu windows. Every form is associated with a main window and a sub window. The form main window displays any title or border associated or whatever the user wishes. Then the sub window contains all the fields and displays them according to their position. This gives the flexibility of manipulating fancy form displaying very easily.

Since this is pretty much similar to menu windows, I am providing an example with out much explanation. The functions are similar and they work the same way.

Example 28. Form Windows Example

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

void print_in_middle(WINDOW *win, int starty, int startx,
                     int width, const char *string, chtype color);

int
main(void)
{
    FIELD *field[3];
    FORM *my_form;
    WINDOW *my_form_win;
    int ch, rows, cols;

    /* Initialize curses */
    initscr();
    start_color();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);

    /* Initialize few color pairs */
    init_pair(1, COLOR_RED, COLOR_BLACK);

    /* Initialize the fields */
    field[0] = new_field(1, 10, 6, 1, 0, 0);
    field[1] = new_field(1, 10, 8, 1, 0, 0);
    field[2] = NULL;

    /* Set field options */
    set_field_back(field[0], A_UNDERLINE);
    field_opts_off(field[0], O_AUTOSKIP);
    /* Don't go to next field when this */
    /* Field is filled up             */
    set_field_back(field[1], A_UNDERLINE);
    field_opts_off(field[1], O_AUTOSKIP);

    /* Create the form and post it */
    my_form = new_form(field);
```

```
/* Calculate the area required for the form */
scale_form(my_form, &rows, &cols);

/* Create the window to be associated with the form */
my_form_win = newwin(rows + 4, cols + 4, 4, 4);
keypad(my_form_win, TRUE);

/* Set main window and sub window */
set_form_win(my_form, my_form_win);
set_form_sub(my_form, derwin(my_form_win, rows, cols, 2, 2));

/* Print a border around the main window and print a title */
box(my_form_win, 0, 0);
print_in_middle(my_form_win, 1, 0, cols + 4, "My Form", COLOR_PAIR(1));

post_form(my_form);
wrefresh(my_form_win);

mvprintw(LINES - 2, 0,
         "Use UP, DOWN arrow keys to switch between fields");
refresh();

/* Loop through to get user requests */
while ((ch = wgetch(my_form_win)) != KEY_F(1)) {
    switch (ch) {
    case KEY_DOWN:
        /* Go to next field */
        form_driver(my_form, REQ_NEXT_FIELD);
        /* Go to the end of the present buffer */
        /* Leaves nicely at the last character */
        form_driver(my_form, REQ_END_LINE);
        break;
    case KEY_UP:
        /* Go to previous field */
        form_driver(my_form, REQ_PREV_FIELD);
        form_driver(my_form, REQ_END_LINE);
        break;
    default:
        /* If this is a normal character, it gets */
        /* Printed                                */
        form_driver(my_form, ch);
        break;
    }
}

/* Un post form and free the memory */
unpost_form(my_form);
free_form(my_form);
free_field(field[0]);
free_field(field[1]);

endwin();
```

```
    return 0;
}

void
print_in_middle(WINDOW *win, int starty, int startx,
                int width, const char *string, chtype color)
{
    int length, x, y;
    float temp;

    if (win == NULL)
        win = stdscr;
    getyx(win, y, x);
    if (startx != 0)
        x = startx;
    if (starty != 0)
        y = starty;
    if (width == 0)
        width = 80;

    length = (int) strlen(string);
    temp = (float) (width - length) / 2;
    x = startx + (int) temp;
    wattron(win, color);
    mvwprintw(win, y, x, "%s", string);
    wattroff(win, color);
    refresh();
}
```

18.5. Field Validation

By default, a field will accept any data input by the user. It is possible to attach validation to the field. Then any attempt by the user to leave the field, while it contains data that doesn't match the validation type will fail. Some validation types also have a character-validity check for each time a character is entered in the field.

Validation can be attached to a field with the following function.

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

Once set, the validation type for a field can be queried with

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

The form driver validates the data in a field only when data is entered by the end-user. Validation does not occur when

- the application program changes the field value by calling set_field_buffer.
- linked field values are changed indirectly -- by changing the field to which they are linked

The following are the pre-defined validation types. You can also specify custom validation, though it is a bit tricky and cumbersome.

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);            /* minimum width of field */
```

The width argument sets a minimum width of data. The user has to enter at-least width number of characters before he can leave the field. Typically you'll 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);            /* minimum width of field */
```

The width argument sets a minimum width of data. As with TYPE_ALPHA, typically you'll want to set this to the field width; if it 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 */
                   int vmin, int vmax);   /* valid range */
```

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

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

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

18.6. Form Driver: The work horse of the forms system

As in the menu system, form_driver() plays a very important role in forms system. All types of requests to forms system should be funneled through form_driver().

```
int form_driver(FORM *form,     /* form on which to operate     */
                int request)    /* form request code         */
```

As you have seen some of the examples above, you have to be in a loop looking for user input and then decide whether it is a field data or a form request. The form requests are then passed to form_driver() to do the work.

The requests roughly can be divided into following categories. Different requests and their usage is explained below:

18.6.1. Page Navigation Requests

These requests cause page-level moves through the form, triggering display of a new form screen. A form can be made of multiple pages. If you have a big form with lot of fields and logical sections, then you can divide the form into pages. The function set_new_page() to set a new page at the field specified.

```
int set_new_page(FIELD *field,/* Field at which page break to be set or unset */
         bool new_page_flag); /* should be TRUE to put a break */
```

The following requests allow you to move to different pages

- *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.

18.6.2. Inter-Field Navigation Requests

These requests handle navigation between fields on the same page.

- *REQ_NEXT_FIELD* Move to next field.
- *REQ_PREV_FIELD* Move to previous field.
- *REQ_FIRST_FIELD* Move to the first field.
- *REQ_LAST_FIELD* Move to the last field.
- *REQ_SNEXT_FIELD* Move to sorted next field.
- *REQ_SPREV_FIELD* Move to sorted previous field.
- *REQ_SFIRST_FIELD* Move to the sorted first field.
- *REQ_SLAST_FIELD* Move to the sorted last field.
- *REQ_LEFT_FIELD* Move left to field.
- *REQ_RIGHT_FIELD* Move right to field.
- *REQ_UP_FIELD* Move up to field.
- *REQ_DOWN_FIELD* Move down to field.

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

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

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

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

18.6.3. Intra-Field Navigation Requests

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

- *REQ_NEXT_CHAR* Move to next character.
- *REQ_PREV_CHAR* Move to previous character.
- *REQ_NEXT_LINE* Move to next line.
- *REQ_PREV_LINE* Move to previous line.
- *REQ_NEXT_WORD* Move to next word.
- *REQ_PREV_WORD* Move to previous word.
- *REQ_BEG_FIELD* Move to beginning of field.
- *REQ_END_FIELD* Move to end of field.
- *REQ_BEG_LINE* Move to beginning of line.
- *REQ_END_LINE* Move to end of line.
- *REQ_LEFT_CHAR* Move left in field.
- *REQ_RIGHT_CHAR* Move right in field.
- *REQ_UP_CHAR* Move up in field.
- *REQ_DOWN_CHAR* Move down in field.

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

18.6.4. Scrolling Requests

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

- *REQ_SCR_FLINE* Scroll vertically forward a line.
- *REQ_SCR_BLINE* Scroll vertically backward a line.
- *REQ_SCR_FPAGE* Scroll vertically forward a page.
- *REQ_SCR_BPAGE* Scroll vertically backward a page.
- *REQ_SCR_FHPAGE* Scroll vertically forward half a page.
- *REQ_SCR_BHPAGE* Scroll vertically backward half a page.
- *REQ_SCR_FCHAR* Scroll horizontally forward a character.
- *REQ_SCR_BCHAR* Scroll horizontally backward a character.
- *REQ_SCR_HFLINE* Scroll horizontally one field width forward.
- *REQ_SCR_HBLINE* Scroll horizontally one field width backward.
- *REQ_SCR_HFHALF* Scroll horizontally one half field width forward.
- *REQ_SCR_HBHALF* Scroll horizontally one half field width backward.

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

18.6.5. Editing Requests

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

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

- *REQ_INS_MODE* Set insertion mode.
- *REQ_OVL_MODE* Set overlay mode.
- *REQ_NEW_LINE* New line request (see below for explanation).
- *REQ_INS_CHAR* Insert space at character location.
- *REQ_INS_LINE* Insert blank line at character location.
- *REQ_DEL_CHAR* Delete character at cursor.
- *REQ_DEL_PREV* Delete previous word at cursor.
- *REQ_DEL_LINE* Delete line at cursor.
- *REQ_DEL_WORD* Delete word at cursor.
- *REQ_CLR_EOL* Clear to end of line.
- *REQ_CLR_EOF* Clear to end of field.
- *REQ_CLR_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.

18.6.6. Order Requests

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

- *REQ_NEXT_CHOICE* Place the successor value of the current value in the buffer.
- *REQ_PREV_CHOICE* Place the predecessor value of the current value in the buffer.

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

18.6.7. Application Commands

Form requests are represented as integers above the curses value greater than KEY_MAX and less than or equal to the constant MAX_COMMAND. A value within this range gets ignored by form_driver(). So this can be used for any purpose by the application. It can be treated as an application specific action and take corresponding action.

19. Tools and Widget Libraries

Now that you have seen the capabilities of ncurses and its sister libraries, you are rolling your sleeves up and gearing for a project that heavily manipulates screen. But wait.. It can be pretty difficult to write and maintain complex GUI widgets in plain ncurses or even with the additional libraries. There are some ready-to-use tools and widget libraries that can be used instead of writing your own widgets. You can use some of them, get ideas from the code, or even extend them.

19.1. CDK (Curses Development Kit)

In the author's words

CDK stands for 'Curses Development Kit' and it currently contains 21 ready to use widgets which facilitate the speedy development of full screen curses programs.

The kit provides some useful widgets, which can be used in your programs directly. It is pretty well written and the documentation is very good. The examples in the examples directory can be a good place to start for beginners. The CDK can be downloaded from https://invisible-island.net/cdk/ . Follow the instructions in README file to install it.

19.1.1. Widget List

The following is the list of widgets provided with cdk and their description.

```
Widget Type           Quick Description
===========================================================================
Alphalist             Allows a user to select from a list of words, with
                      the ability to narrow the search list by typing in a
                      few characters of the desired word.
Buttonbox             This creates a multiple button widget.
Calendar              Creates a little simple calendar widget.
Dialog                Prompts the user with a message, and the user
                      can pick an answer from the buttons provided.
Entry                 Allows the user to enter various types of information.
```

```
File Selector        A file selector built from Cdk base widgets. This
                     example shows how to create more complicated widgets
                     using the Cdk widget library.
Graph                Draws a graph.
Histogram            Draws a histogram.
Item List            Creates a pop up field which allows the user to select
                     one of several choices in a small field. Very useful
                     for things like days of the week or month names.
Label                Displays messages in a pop up box, or the label can be
                     considered part of the screen.
Marquee              Displays a message in a scrolling marquee.
Matrix               Creates a complex matrix with lots of options.
Menu                 Creates a pull-down menu interface.
Multiple Line Entry  A multiple line entry field. Very useful
                     for long fields. (like a description
                     field)
Radio List           Creates a radio button list.
Scale                Creates a numeric scale. Used for allowing a user to
                     pick a numeric value and restrict them to a range of
                     values.
Scrolling List       Creates a scrolling list/menu list.
Scrolling Window     Creates a scrolling log file viewer. Can add
                     information into the window while its running.
                     A good widget for displaying the progress of
                     something. (akin to a console window)
Selection List       Creates a multiple option selection list.
Slider               Akin to the scale widget, this widget provides a
                     visual slide bar to represent the numeric value.
Template             Creates a entry field with character sensitive
                     positions. Used for pre-formatted fields like
                     dates and phone numbers.
Viewer               This is a file/information viewer. Very useful
                     when you need to display loads of information.
===========================================================================
```

A few of the widgets are modified by Thomas Dickey in recent versions.

19.1.2. Some Attractive Features

Apart from making our life easier with readily usable widgets, cdk solves one frustrating problem with printing multi colored strings, justified strings elegantly. Special formatting tags can be embedded in the strings which are passed to CDK functions. For Example

If the string

```
"</B/1>This line should have a yellow foreground and a blue
background.<!1>"
```

given as a parameter to newCDKLabel(), it prints the line with yellow foreground and blue background. There are other tags available for justifying string, embedding special drawing characters, etc. Please refer to the man page cdk_display(3X) for details. The man page explains the usage with nice examples.

19.1.3. Conclusion

All in all, CDK is a well-written package of widgets, which if used properly can form a strong frame work for developing complex GUI.

19.2. The dialog

Long long ago, in September 1994, when few people knew linux, Jeff Tranter wrote an article (http://www2.linuxjournal.com/lj-issues/issue5/2807.html) on dialog in Linux Journal. He starts the article with these words..

Linux is based on the Unix operating system, but also features a number of unique and useful kernel features and application programs that often go beyond what is available under Unix. One little-known gem is "dialog", a utility for creating professional-looking dialog boxes from within shell scripts. This article presents a tutorial introduction to the dialog utility, and shows examples of how and where it can be used

As he explains, dialog is a real gem in making professional-looking dialog boxes with ease. It creates a variety of dialog boxes, menus, check lists, etc. It is usually installed by default. If not, you can download it from Thomas Dickey (https://invisible-island.net/dialog/)'s site.

The above-mentioned article gives a very good overview of its uses and capabilities. The man page has more details. It can be used in variety of situations. One good example is building of linux kernel in text mode. Linux kernel uses a modified version of dialog tailored for its needs.

dialog was initially designed to be used with shell scripts. If you want to use its functionality in a c program, then you can use libdialog. The documentation regarding this is sparse. Definitive reference is the dialog.h header file which comes with the library. You may need to hack here and there to get the required output. The source is easily customizable. I have used it on a number of occasions by modifying the code.

19.3. Perl Curses Modules CURSES::FORM and CURSES::WIDGETS

The perl module Curses, Curses::Form and Curses::Widgets give access to curses from perl. If you have curses and basic perl is installed, you can get these modules from CPAN All Modules page (http://www.cpan.org/modules/01modules.index.html). Get the three zipped modules in the Curses category. Once installed you can use these modules from perl scripts like any other module. For more information on perl modules see perlmod man page. The above modules come with good documentation and they have some demo scripts to test the functionality. Though the widgets provided are very rudimentary, these modules provide good access to curses library from perl.

Some of my code examples are converted to perl by Anuradha Ratnaweera and they are available in the `perl` directory.

For more information see man pages Curses(3) , Curses::Form(3) and Curses::Widgets(3). These pages are installed only when the above modules are acquired and installed.

20. Just For Fun !!!

This section contains few programs written by me just for fun. They don't signify a better programming practice or the best way of using ncurses. They are provided here so as to allow beginners to get ideas and add more programs to this section. If you have written a couple of nice, simple programs in curses and want them to included here, contact me (mailto:ppadala@gmail.com).

20.1. The Game of Life

Game of life is a wonder of math. In Paul Callahan (http://www.math.com/students/wonders/life/life.html)'s words

The Game of Life (or simply Life) is not a game in the conventional sense. There are no players, and no winning or losing. Once the "pieces" are placed in the starting position, the rules determine everything that happens later. Nevertheless, Life is full of surprises! In most cases, it is impossible to look at a starting position (or pattern) and see what will happen in the future. The only way to find out is to follow the rules of the game.

This program starts with a simple inverted U pattern and shows how wonderful life works. There is a lot of room for improvement in the program. You can let the user enter pattern of his choice or even take input from a file. You can also change rules and play with a lot of variations. Search on google (https://www.google.com) for interesting information on game of life.

File Path: JustForFun/life.c

20.2. Magic Square

Magic Square, another wonder of math, is very simple to understand but very difficult to make. In a magic square sum of the numbers in each row, each column is equal. Even diagonal sum can be equal. There are many variations which have special properties.

This program creates a simple magic square of odd order.

File Path: JustForFun/magic.c

20.3. Towers of Hanoi

The famous towers of hanoi solver. The aim of the game is to move the disks on the first peg to last peg, using middle peg as a temporary stay. The catch is not to place a larger disk over a small disk at any time.

File Path: JustForFun/hanoi.c

20.4. Queens Puzzle

The objective of the famous N-Queen puzzle is to put N queens on a N X N chess board without attacking each other.

This program solves it with a simple backtracking technique.

File Path: JustForFun/queens.c

20.5. Shuffle

A fun game, if you have time to kill.

File Path: JustForFun/shuffle.c

20.6. Typing Tutor

A simple typing tutor, I created more out of need than for ease of use. If you know how to put your fingers correctly on the keyboard, but lack practice, this can be helpful.

File Path: JustForFun/tt.c

21. References

- NCURSES man pages
- NCURSES FAQ at https://invisible-island.net/ncurses/ncurses.faq.html
- Writing programs with NCURSES by Eric Raymond and Zeyd M. Ben-Halim at https://invisible-island.net/ncurses/ncurses-intro.html - somewhat obsolete. I was inspired by this document and the structure of this HOWTO follows from the original document