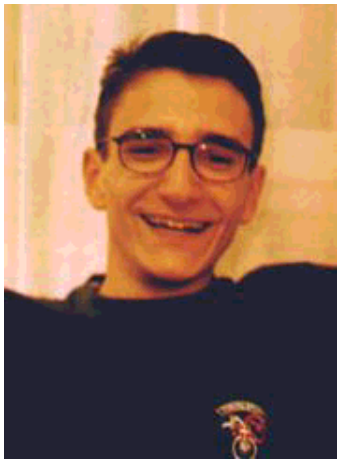
LinuxFocus article number 233
http://linuxfocus.org



by Reha K. Gerçeker
<gerceker/at/itu.edu.tr>

About the author:

Reha is a student of computer engineering in Istanbul, Turkey. He loves the freedom Linux provides as a software development platform. He spends much of his time in front of his computer, writing programms. He wishes to become a smart programmer someday.

Translated to English by:
Reha K. Gerçeker
<gerceker/at/itu.edu.tr>

Introduction to Ncurses

Abstract:

Ncurses is a library that provides function-key mapping, screen painting functions and the ability to use multiple non-overlapping windows on text-based terminals.

What is Ncurses?

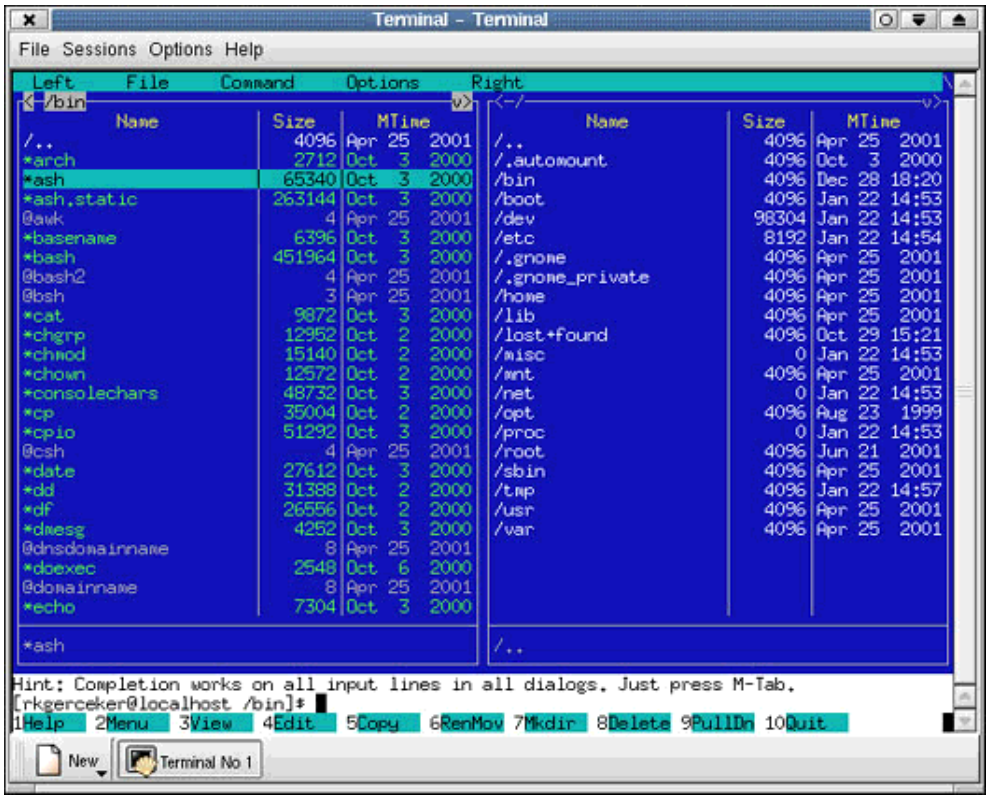
Do you want your programs to have a colorful terminal based interface? Ncurses is a library that provides window functionallity for text-based terminals. Things that ncurses is capable of:

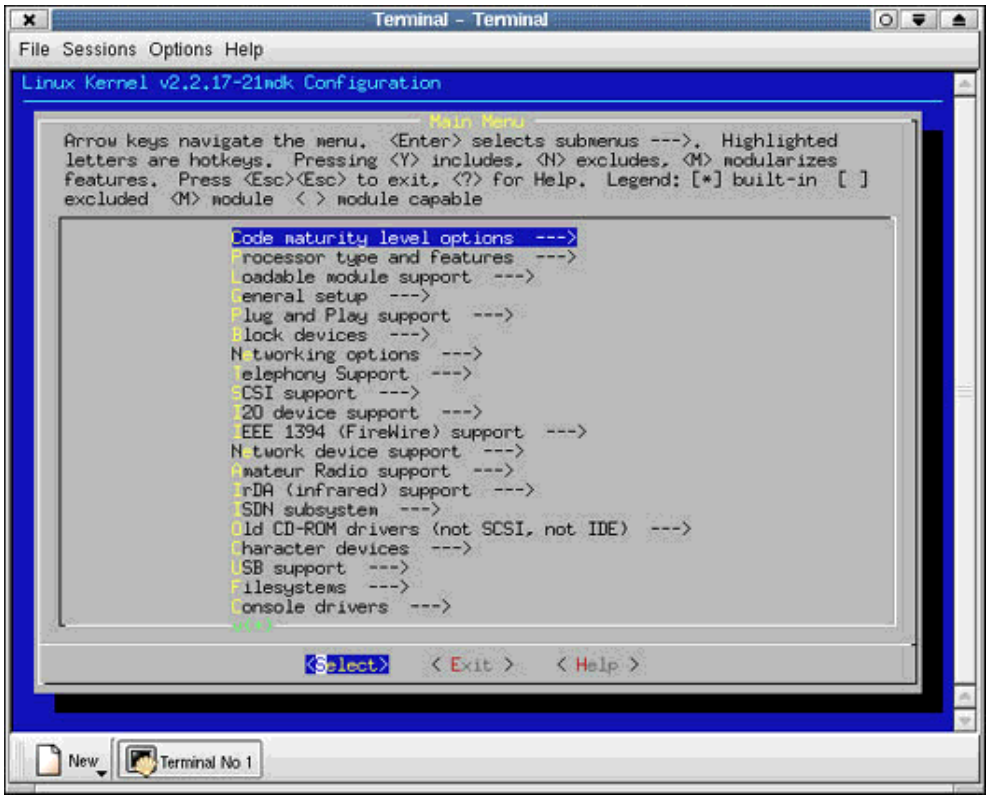
- Use whole screen as you like.

- Create and manage windows.
- Use 8 different colors.
- Give your program mouse support.
- Use the function keys from the keyboard.

It is possible to use ncurses on any ANSI/POSIX conforming UNIX system. Apart from that, the library is able to detect terminal properties from the system database and act accordingly, providing a terminal-independent interface. Therefore, ncurses could be used and trusted for designs that are supposed to work across different platforms and various terminals.

Midnight Commander is one of the examples that is written using ncurses. Also the interface used for kernel configuration on the console is written with ncurses. You see their snapshots below.





Where to Download?

Ncurses is developed under GNU/Linux. To download the latest release, get detailed information and find other related links, visit www.gnu.org/software/ncurses/.

Basics

In order to use the library, you should include curses.h in your source code and be sure to link your code with the curses library. This is done by giving the parameter -lcurses to gcc.

It is necessary to know about a fundamental data structure while working under ncurses. That is the WINDOW structure and, as understood easily from the name, it is used to represent the windows that you create. Almost all functions of the library take a WINDOW pointer as a parameter.

Most commonly used components of ncurses are windows. Even if you don’t create your own windows, the screen is considered as an own window. As the FILE descriptor stdout of the standard I/O library represents the screen (when there are no redirections), ncurses has the WINDOW pointer stdscr that does the same job. In addition to stdscr, another WINDOW pointer named curscr is defined in the library. As stdscr represents the screen, curscr represents the current screen known to the library. You may ask "What is the difference?" Keep on reading.

In order to use ncurses functions and variables in your programs, you have to call the function initscr.

This function allocates memory for variables like stdscr, curscr and makes the library ready for use. In other words, all ncurses functions must follow initscr. In the same manner, you should call endwin when you are done with ncurses. This frees the memory used by ncurses. After calling endwin you can't use ncurses functions unless you call initscr again.

Between the calls to initscr and endwin, be sure not to send output to the screen using the functions of the standard I/O library. Otherwise, you may get an unlikely and usually corrupted output on screen. When ncurses is active, use its functions to send output to the screen. Before calling initscr or after calling endwin, you can do what you want.

Updating the Screen: refresh

The WINDOW structure not only keeps height, width and position of the window but also holds the contents of the window. When you write to a window, the contents of the window is changed but this doesn't mean that it appears on the screen immediately. In order to have the screen updated, either refresh or wrefresh has to be called.

Here lies the difference between stdscr and curscr. While curscr keeps contents of the current screen, stdscr might have different information after calls to the ncurses output functions. If you want the latest changes of the stdscr dumped on curscr, you need to call refresh. In other words, refresh is the only function that deals with curscr. It is recommended that you don't mess with curscr and leave it to the refresh function to update curscr.

refresh has a mechanism to update the screen as fast as possible. When the function is called, it updates only the altered lines of the window. This saves CPU time as it prevents the program from writing the same information again on the screen. This mechanism is the reason why ncurses functions and standard I/O functions can produce bad results when used together; when ncurses functions are called they set a flag which tells refresh that the line has changed whereas nothing like this happens when you call a standard I/O function.

refresh and wrefresh basically do the same thing. wrefresh takes a WINDOW pointer as parameter and refreshes the contents of that window only. refresh() is equivalent to wrefresh(stdscr). As I will talk about later, like wrefresh, most ncurses functions have macros that apply these functions for stdscr.

Creating New Windows

Let's now talk about subwin and newwin, the functions that create new windows. Both of these functions take the height, width, and coordinates of the upper left corner of the new windows as parameters. In turn, they return a WINDOW pointer that represents your new window. You can use this new pointer with wrefresh and other functions that I will talk about later.

"If they do the same thing, why double the functions?" you might be asking. You are right, they are slightly different. subwin creates the new window as the subwindow of another one. A window created

in this way inherits properties of the parent window. These properties could later be changed without affecting the parent window.

Apart from this, there is one thing that ties the parent and child windows together. The character array that keeps the contents of a window is shared between the parent and child windows. In other words, characters at the intersection of the two windows, could be altered by any one of them. If the parent writes to such a square, then the child's content is also changed. The other way round is also true.

Unlike subwin, newwin creates a brand new window. Such a window, unless it has its own subwindows, does not share its character array with another window. The advantage of using subwin is that the usage of a shared character array uses less memory. However, when windows get to write over each other, using newwin brings its own advantages.

You can create your subwindows at any depth. Every subwindow might have subwindows of its own, but then keep in mind that the same character array is shared by more than two windows.

When you are done with a window you have created, you can delete that window using the function delwin. I suggest you consult the man pages for the parameter lists of these functions.

Write to Windows, Read from Windows

We have talked about stdscr, curscr, refreshing the screen and creating new windows. But then how do we write to a window? Or how do we read data from a window?

Functions used for these purposes resemble their counterparts of the standard I/O library. Among these functions are printw instead of printf, scanw instead of scanf, addch instead of putc or putchar, getch instead of getc or getchar. They are used as usual, only their names are different. Similarly, addstr could be used to write a string to a window and getstr could be used to read a string from a window. All these functions with a 'w' letter added in front of their name and a WINDOW pointer as the first parameter, do their job on a different window from stdscr. For example, printw(...) and wprintw(stdscr, ...) are equivalents, just like refresh() and wrefresh(stdscr).

It would be a long story to go into the details of these functions. Man pages are the best source to learn their descriptions, prototypes, return values and other notes. I suggest you check the man pages for every function that you use. They offer detailed and valuable information. The last section of this article where I present an example program may also serve as a tutorial on how to use the functions.

Physical and Logical Cursors

It is necessary to explain physical and logical cursors after speaking about writing to and reading from windows. What is meant by physical cursor is the usual blinking cursor on the screen and there is only one physical cursor. On the other hand, the logical cursors belongs to ncurses windows and every windows has one of them. Therefore there may be several logical cursors.

The logical cursor is at the square of the window where the writing or reading process will begin. Therefore, being able to move the logical cursor around means that you can write to any spot of the screen or window at any time. This is an advantage of ncurses over the standard I/O library.

The function that moves the logical cursor is either move or, as you might easily guess, wmove. move is a macro of wmove, written for the stdscr.

Another issue is the coordination of physical and logical cursors. The position the physical cursor will end up after a writing process is determined by the _leave flag which exists in the WINDOW structure. If _leave is set, the logical cursor is moved to the position of the physical cursor (where the last character is written) after writing is done. If _leave is not set, the physical cursor is taken back to the position of the logical cursor (where the first character is written) after the writing is done. The _leave flag is controlled by the leaveok function.

The function that moves the physical cursor is mvcur. Unlike others, mvcur takes effect immediately rather than at the next refresh. If you want the physical cursor to be invisible, then use the function curs_set. Check the man pages for details.

There are also macros that combine the moving and writing functions described above into one simple call. These are explained nicely on the same man pages that are about addch, addstr, printw, getch, getstr, scanw etc.

Clearing Windows

Writing to windows is done. But how do we clear windows, lines or characters?

Clearing, in ncurses, means to fill the square, line or the contents of the window with white spaces. Functions I have explained below fill the necessary squares with white spaces and clear therefore the screen.

First let’s talk about functions that deal with the clearing of a character or a line. Functions delch and wdelch delete the character that is under the logical cursor of the window and shift the following characters on the same line left. deleteln and wdeleteln delete the line of the logical cursor and shifts up all the lines below.

The functions clrtoeol and wclrtoeol delete all characters on the same line to the right of the logical cursor. clrtobot and wclrtobot first call wclrtoeol to delete all characters to the right of the logical cursor and then delete all the following lines.

Other than these, there are functions that clear the whole screen or a window. There are two methods for clearing a whole screen. The first is to fill all squares with white spaces and call refresh and the other is to use the built-in terminal control code. The first method is slower than the second as it requires all squares on the screen get rewritten whereas the second clears the whole screen immediately.

erase and werase fill the character array of a window with white spaces. At the next refresh, the window

will be cleared. However, if the window to be cleared fills the whole screen, it is not very clever to use these functions. They use the first method described above. When the window to be cleared is screen wide, it is advantageous to use the functions below.

Before going into other functions, it is time to mention the _clear flag. It exists in the WINDOW structure and if it is set, it asks refresh to send the control code to the terminal when it is called. When called, refresh checks if the window is screen wide (using the _FULLWIN flag) and if so, it clears the screen with the built-in terminal method. It then only writes characters other than white spaces on the screen. This makes clearing the whole screen quicker. The reason why the terminal method is used only for windows that fill whole screen is that the terminal control code clears the whole screen not only the window itself. The _clear flag is controlled by the function clearok.

The functions clear and wclear are used to clear windows that are screen wide. In fact, these function are equivalent to calling a werase and a clearok. First, they fill the window's character array with white spaces. Then, by setting the _clear flag, they clear the screen using the built-in terminal method if the window is screen wide or else they refresh all squares of the window filling them with white spaces.

As a result, if you know that the window to be cleared is full screen then use clear or wclear. It produces faster the result. However, it is no difference to use wclear or werase when the window is not full screen.

Using Colors

The colors you see on the screen should be thought of as color pairs. That is because each square has a background and a foreground color. To write in color with ncurses means to create your own color pairs and use those pairs to write to a window.

Just like initscr needs to be called to start ncurses, start_color needs to be called to initiate colors. The function you need to use to create your color pairs is init_pair. When you create a color pair with init_pair, this pair is associated with the number you gave the function as the first parameter. Then, whenever you want to use this pair, you refer to it by calling COLOR_PAIR with that associated number.

Apart from creating color pairs, you need to have the necessary functions that write with a different color pair. This is done by the functions attron and wattron. These functions, until attroff or wattroff is called, cause everything to be written to the corresponding window in the color pair you selected.

There are also the functions bkgd and wbkgd that change the color pair associated with a whole window. When called, they change the background and foreground colors of all squares of the window. That means, at the next refresh, every square of the window is rewritten with the new color pair.

See the man pages for the colors available and details of the functions mentioned here.

Boxes Around Windows

You can create boxes around your windows to create a good look for your program. There is a macro in the library called box that does this for you. Unlike others, no wbox exists; box takes a WINDOW pointer as an argument.

You can find out the easy details of box on the man pages. There is something else that should be mentioned. Putting a window into a box means simply to write the necessary characters into the character array of the window that correspond to the boundary squares. If you write to those squares later somehow, the box would get corrupted. To prevent this, you create an inner window inside the original window with subwin, put the original window in a box and use the inner window to write to the window when necessary.

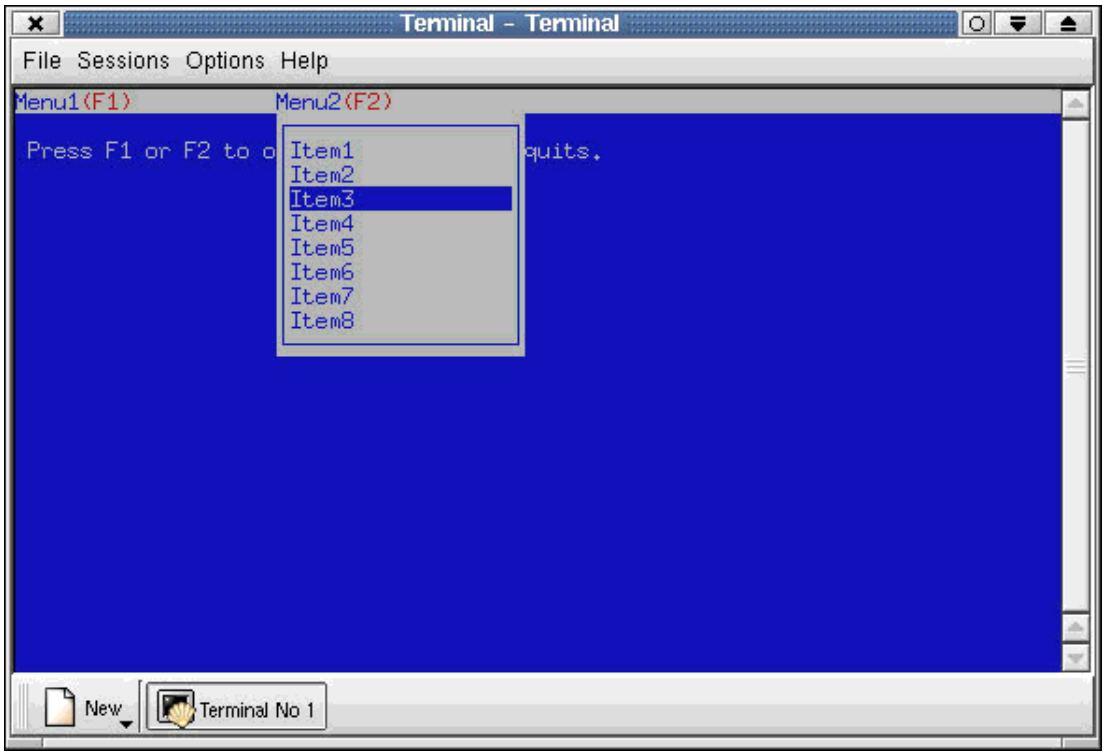
Function Keys

In order to be able to use the function keys, the _use_keypad flag must be set in the window you are getting input from. keypad is the function that sets the value of _use_keypad. When you set _use_keypad, you can get input from the keyboard as usual with the input functions.

In this case, if you use getch to get data for example, you should be careful to hold that data in an int variable rather than a char variable. This is because numerical values of function keys are greater than the values a char variable can hold. You do not need to know these numerical values of the function keys but instead use their defined names from the library. These names are listed in the man page of getch.

An Example

We are now going to analyse a nice and simple program. In this program, menus are created using ncurses and the selection of an option from a menu is demonstrated. An interesting aspect of this program is the use of ncurses windows to generate a menu effect. You see a snapshot below:



The program starts with the included header files as usual. Then we define the constants that are the ASCII values of the enter and escape keys.

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

#define ENTER 10
#define ESCAPE 27
```

The function below is called first when the program runs. It first calls initscr to initialise curses and then start_color to make use of colors possible. Color pairs that are going to be used throughout the program are defined later. The call curs_set(0) makes the physical cursor invisible. noecho stops the input from the keyboard being displayed on the screen. You can also use the noecho function to control input coming from the keyboard and display only the parts that you want to display. The echo function should be called when necessary to remove the effect of noecho. The function below finally calls keypad to enable the function keys when getting input from stdscr. This is necessary as we will use F1, F2 and cursor keys later in the program.

```
void init_curses()
{
    initscr();
    start_color();
    init_pair(1,COLOR_WHITE,COLOR_BLUE);
    init_pair(2,COLOR_BLUE,COLOR_WHITE);
    init_pair(3,COLOR_RED,COLOR_WHITE);
    curs_set(0);
    noecho();
    keypad(stdscr,TRUE);
}
```

The next function creates the menubar that appears on top of the screen. You may check the main function below and see that the menubar that appears as a single line at the top of the screen is in fact defined as a single line subwindow of stdscr. The function below takes the pointer to that window as a parameter, first changes its background color and then writes the menu names. We use waddstr to write the menu names, another function could have been used. Pay attention to wattron calls that is used to write with a different color pair (number 3) rather than using the default color pair (number 2). Remember pair number 2 was set as default on the first line by wbkgd. wattroff is called when we want to switch to the default color pair.

```
void draw_menubar(WINDOW *menubar)
{
    wbkgd(menubar,COLOR_PAIR(2));
    waddstr(menubar,"Menu1");
    wattron(menubar,COLOR_PAIR(3));
    waddstr(menubar,"(F1)");
    wattroff(menubar,COLOR_PAIR(3));
    wmove(menubar,0,20);
    waddstr(menubar,"Menu2");
    wattron(menubar,COLOR_PAIR(3));
    waddstr(menubar,"(F2)");
    wattroff(menubar,COLOR_PAIR(3));
}
```

The next function draws the menus when F1 or F2 is pressed. To create the menu effect a new window with the same white color as the menubar is created over the blue window that makes up the background. We don't want this new window to overwrite previously written characters on the background. They should stay there after the menu closes. This is why the menu window can't be created as a subwindow of stdscr. As you see below, the window items[0] is created with the function newwin and the other 8 items windows are created as subwindows of items[0]. Here items[0] is used to draw a box around the menu and the other items windows are used to show the selected item in the menu and also not to overwrite the characters of the box around the menu. To make an item look like selected, it is sufficient to make its background color different then the rest of the items. That is what is done in the third line from bottom; the background color of the first item is made different than the others and so when the menu pops up, it is the first item that is selected.

```
WINDOW **draw_menu(int start_col)
{
    int i;
    WINDOW **items;
    items=(WINDOW **)malloc(9*sizeof(WINDOW *));

    items[0]=newwin(10,19,1,start_col);
    wbkgd(items[0],COLOR_PAIR(2));
    box(items[0],ACS_VLINE,ACS_HLINE);
    items[1]=subwin(items[0],1,17,2,start_col+1);
    items[2]=subwin(items[0],1,17,3,start_col+1);
    items[3]=subwin(items[0],1,17,4,start_col+1);
    items[4]=subwin(items[0],1,17,5,start_col+1);
    items[5]=subwin(items[0],1,17,6,start_col+1);
    items[6]=subwin(items[0],1,17,7,start_col+1);
    items[7]=subwin(items[0],1,17,8,start_col+1);
    items[8]=subwin(items[0],1,17,9,start_col+1);
    for (i=1;i<9;i++)
        wprintw(items[i],"Item%d",i);
    wbkgd(items[1],COLOR_PAIR(1));
    wrefresh(items[0]);
```

```
    return items;
}
```

The next function simply deletes the menu window created by the function above. It first deletes the items windows with delwin and then frees the memory allocated for the items pointer.

```
void delete_menu(WINDOW **items,int count)
{
    int i;
    for (i=0;i<count;i++)
        delwin(items[i]);
    free(items);
}
```

The scroll_menu function lets us scroll between and inside menus. It reads the keys pressed on the keyboard with getch. If up or down cursor keys are pressed, then the item above or below is selected. This is, as you will recall, done by making the background color of the selected item different than the rest. If right or left cursor keys are pressed, the open menu is closed and the other is opened. If the enter key is pressed, then the selected item is returned. If ESC is pressed, the menus are closed without selecting an item. The function ignores other input keys. In this function getch is able to read the cursor keys from the keyboard. Let me remind you that this is possible as the first function init_curses called keypad(stdscr,TRUE) and the return value of getch is kept in an int variable rather than a char variable since values of the function keys are larger than a char variable can hold.

```
int scroll_menu(WINDOW **items,int count,int menu_start_col)
{
    int key;
    int selected=0;
    while (1) {
        key=getch();
        if (key==KEY_DOWN || key==KEY_UP) {
            wbkgd(items[selected+1],COLOR_PAIR(2));
            wnoutrefresh(items[selected+1]);
            if (key==KEY_DOWN) {
                selected=(selected+1) % count;
            } else {
                selected=(selected+count-1) % count;
            }
            wbkgd(items[selected+1],COLOR_PAIR(1));
            wnoutrefresh(items[selected+1]);
            doupdate();
        } else if (key==KEY_LEFT || key==KEY_RIGHT) {
            delete_menu(items,count+1);
            touchwin(stdscr);
            refresh();
            items=draw_menu(20-menu_start_col);
            return scroll_menu(items,8,20-menu_start_col);
        } else if (key==ESCAPE) {
            return -1;
        } else if (key==ENTER) {
            return selected;
        }
    }
}
```

Lastly there is the main function. It uses all the functions I have written and described above to make the program work properly. It also reads keys pressed with getch and if F1 or F2 is pressed, it draws the

coresponding menu window with draw_menu. After that it calls scroll_menu and lets the user make a selection from the menus. After scroll_menu returns, it deletes the menu windows and prints the selected item on the messagebar.

I should mention the function touchwin. If refresh was directly called without touchwin after the menus were closed, the last open menu would have stayed on the screen. This is because the menu functions do not change stdscr at all and when refresh is called, it doesn't rewrite any character of stdscr since it assumes the window is not changed. touchwin sets all the flags in the WINDOW structure that tell refresh all lines of the window has changed and so at the next refresh whole window has to be rewritten even if the contents of window hasn't changed. The information written on the stdscr stays there after the menus close since the menus do not write over stdscr but instead are created as new windows.

```
int main()
{
    int key;
    WINDOW *menubar,*messagebar;

    init_curses();

    bkgd(COLOR_PAIR(1));
    menubar=subwin(stdscr,1,80,0,0);
    messagebar=subwin(stdscr,1,79,23,1);
    draw_menubar(menubar);
    move(2,1);
    printw("Press F1 or F2 to open the menus. ");
    printw("ESC quits.");
    refresh();

    do {
        int selected_item;
        WINDOW **menu_items;
        key=getch();
        werase(messagebar);
        wrefresh(messagebar);
        if (key==KEY_F(1)) {
            menu_items=draw_menu(0);
            selected_item=scroll_menu(menu_items,8,0);
            delete_menu(menu_items,9);
            if (selected_item<0)
                wprintw(messagebar,"You haven't selected any item.");
            else
                wprintw(messagebar,
                  "You have selected menu item %d.",selected_item+1);
            touchwin(stdscr);
            refresh();
        } else if (key==KEY_F(2)) {
            menu_items=draw_menu(20);
            selected_item=scroll_menu(menu_items,8,20);
            delete_menu(menu_items,9);
            if (selected_item<0)
                wprintw(messagebar,"You haven't selected any item.");
            else
                wprintw(messagebar,
                  "You have selected menu item %d.",selected_item+1);
            touchwin(stdscr);
            refresh();
        }
    } while (key!=ESCAPE);
```

```
    delwin(menubar);
    delwin(messagebar);
    endwin();
    return 0;
}
```

If you copy the code to a file named example.c and remove my explanations, you can compile the code with

```
gcc -Wall example.c -o example -lcurses
```

and test the program. You can as well download the code below in the refernces chapter.

Conclusion

I've talked about the basics of ncurses that are sufficient to create a good interface for your program. However, capabilities of the library are not limited to what is explained here. You will discover many other things in the man pages I often asked you to read and you will understand that the information presented here is only an introducton.

References

- The exmple program: example.c
- The ncurses website:www.gnu.org/software/ncurses/

2005-01-14, generated by lfparser_pdf version 2.51