

# **ncurses**

**Elaine de Claro  
2002  
Mico Galang  
Fina Mesina  
CS150.3  
Athena Rebong  
Kenneth Talamayan**

**February 22,  
Mr. William Yu**

## INTRODUCTION

### Ncurses

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

The `curses` API may seem something of an archaism on UNIX desktops increasingly dominated by X, Motif, and Tcl/Tk. Nevertheless, UNIX still supports tty lines and X supports `xterm(1)`; the `curses` API has advantages such as back-portability to character-cell terminals, being light weight, and simplicity. For an application that does not require bit-mapped graphics and multiple fonts, an interface implementation using `curses` will typically be a great deal simpler and less expensive than one using an X toolkit.

The **`curses`** library consists of a number of functions which your program can call. Those functions know the various cursor-movement character sequences for a large variety of terminals (this information is in the file `/etc/termcap`). The significance of this is that your program does not have to know that information; it simply calls the **`curses`** functions, and those functions will use your `TERM` value to check the `/etc/termcap` file and then send the proper cursor-movement characters. For example, if your program wanted to clear the screen, it would not (directly) use any character sequences like those above. Instead, it would simply make the call `clear()`; and **`curses`** would do the work on the program's behalf. The `Curses` library forms a wrapper over working with raw terminal codes, and provides highly flexible and efficient API (Application Programming Interface).

`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.

`Ncurses` (new `curses`) is a freely distributable "clone" of System V Release 4.0 (SVr4) `curses`. `Curses` is a pun on the term "cursor optimization". It is a library of functions that manage an application's display on character-cell terminals (e.g., VT100). It uses `terminfo` format, supports pads, colors, multiple highlights, form characters and function key mapping.

### History

Many programs need to make use of a terminal's cursor-movement capabilities. A familiar example is `vi`; most of its commands make use of such capabilities. For example, hitting the ``j` key while in `vi` will make the cursor move up one line. But a potential problem with this is that different terminals have different ways in which to specify a given type of cursor motion. For example, if a program wants to make the cursor move up one line on a VT100 terminal, the program needs to send the characters Escape, `['`, and `'A'` which is, `printf("%c%c%c",27,[', 'A');` while the character code for the Escape key is 27. But for a Televideo 920C terminal, the program would have to send the `ctrl-K` character, which has code 11, `printf("%c",11);`

Clearly, the authors of programs like `vi` would have a very difficult time trying to write different versions for every terminal, and worse yet, anyone else writing a program which needed cursor movement would have to "re-invent the wheel," which is a very big waste of time. This is why the `curses` library was developed.

The first ancestor of `curses` was the routines written to provide screen-handling for the game `rogue`; these used the already-existing `termcap` database facility for describing terminal capabilities. These routines were abstracted into a documented library and first released with the early BSD UNIX versions. System III UNIX from Bell Labs featured a rewritten and much-improved `curses` library. It introduced the `terminfo` format. `Terminfo` is based on Berkeley's `termcap` database, but contains a number of improvements and extensions. Parameterized capabilities strings were introduced, making it possible to describe multiple video attributes, and colors and to handle far more unusual terminals than possible with `termcap`. In the later AT&T System V releases, `curses` evolved to use more facilities and offer more capabilities, going far beyond BSD `curses` in power and flexibility.

`Ncurses` has an evolved history. The package was originated as `pcurses`, written by Pavel Curtis around 1982, maintained by various people through 1986. It was later polished (e.g., ANSI prototypes, reformatted, some bug fixes, but still essentially the same package) and re-issued as `ncurses 1.8.1` in late 1993 by Zeyd Ben-Halim. Subsequent work (through 1.8.8) was driven by Eric Raymond, who eradicated previous signs of authorship with the current copyright notice between 1.8.7 and 1.8.8, early 1995. Later, this extended to incorporating the forms and menus libraries written by Juergen Pfeifer, and a panel library written by Warren Tucker.

## CONCEPTS

### A Window

A window is an internal data representation of an image of what a particular rectangular section of the terminal display may look like. The terminal display as a whole could be said to be a window, its dimensions defined by its outermost extremities, those being the side of the cathode ray tube.

That said, we could then say that the window with the dimensions of one character in length and one character in height is in fact a window of the size of one character. This is the smallest window that `curses` could possibly handle, but a window could also have the dimensions of 128 characters in length and 50 characters in height. Unfortunately, this would be bigger than the size of most terminals, but in theory, anyway, it is a window.

A `curses` window is not a physical entity. It is only a data representation of how you would like a rectangular portion of the physical screen to look. A window is a preallocated resource stored in the machine's memory. As you manipulate it, nothing actually happens to the physical screen until you are ready to update it. When you are ready, you used the `curses` function `wrefresh()` to update the physical screen. This overwrites or superimposes the internal window on the physical screen.

`Curses` provides a default window which represents your terminal screen. Its size is defined by the dimensions of the terminal screen your `curses` program is to work with. `Curses` allows you to manipulate several windows individually, or all at the same time. It also allows windows to contain windows within themselves, known as sub-windows. You can create as many windows as you want; the only limitation is the amount of memory available which your program can use.

The philosophy behind `curses` is really quite simple. You can work with the default window provided by `curses`, or you can create your own window or windows. You may choose to use both methods; you can use the default window provided, as well as create and work with your own. Either way, `curses` does not care.

## The Curses Window

The curses internal representation of a window is defined in the data structure:

```
struct _win_st
```

This structure is defined in the `/usr/include/curses.h` include file and is typedef WINDOW.

The WINDOW structure below is taken from a UNIX system V.2 version of curses.

```
struct _win_st {
    short _cury, _curx;
    short _maxy, _maxx;
    short _begy, _begx;
    short _flags;
    chtype _attrs;
    bool _clear;
    bool _leave;
    bool _scroll;
    bool _use_idl;
    bool _use_keypad;
    bool _use_meta;
    bool _nodelay;
    chtype **_y;
    short *_firstch;
    short *_lastch;
    short _tmarg, _bmarg;
};

typedef struct _win_st WINDOW;
extern WINDOW *stdscr, *curscr;
```

The definition of the above curses data structure is found in the `<curses.h>`. The first line of this code is an `ifndef`, that is an `#ifndef WINDOW`, the corresponding `#endif` of which is at the bottom of the file. This is useful for administering multiple source files.

This structure is curses' internal representation of a window. It contains all the necessary data and information which curses needs to manage the window on the terminal screen. Anything inside or belonging to a window is modifiable. Anything outside is undefined and usually illegal. Even if you want to print some texts on the screen, curses requires you to place the desired texts into a window. It is important to realize that almost all the curses routines totally depend on this window structure.

It is equally important to realize that, although window structure is an internal representation of a curses window, it may not necessarily bear any relation to what is really being displayed on the terminal screen. The window structure is used solely to hold data and information which describes a window, and is used to build a potential image of a portion of the true terminal screen. It is like a buffer area set aside to represent a portion of the screen which you can modify by using the routines provided in the curses library.

## Example: The Hello World!!! Program

```

#include <ncurses.h>

int main()
{
    initscr();          /* Start curses mode          */
    printw("Hello World !!!"); /* Print Hello World      */
    refresh();         /* Refresh it on to the real screen */
    endwin();          /* End curses mode          */

    return 0;
}

```

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.

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 which is 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.

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 in 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 did `printw` actually the data is written to an imaginary window called `stdscr`, 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 bring it to 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 gives good response 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.

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.

## The Terminal Screen

Before curses can manage your terminal screen it needs to know what it looks like. When curses starts up, the first thing it does is clear the screen it then places the cursor in the home position, which is the top left-hand corner of the screen. Curses then knows exactly what your physical screen looks like and where the cursor is situated.

Curses also needs to know how the programmer would like it to look. For this reason, curses provides two WINDOW data structures,  `curscr` and  `stdscr`.

The `curscr` window holds a data representation of what is currently displayed on the real terminal screen, and the `stdscr` window is provided as a default window for the programmer to work with. The `stdscr` window can be manipulated within a program. When making the terminal screen look like the `stdscr` window, a `refresh()` call is made to update the `curscr` window, which in turn will make the terminal screen look like the `stdscr`.

It is important to remember that it is not good practice to access the `curscr` window directly. Changes should be made only to the appropriate window being worked on, and by calling `wrefresh()` on that window, `curses` can be instructed to update the `curscr` window internally. The `curscr` window should be treated as a reserved window for the private use of the `curses` routines only.

## The Window Structure Variables

The `WINDOW` structure contains all the necessary information to enable `curses` to update the terminal screen optimally. By keeping state information about a particular window inside its `WINDOW` data structure, `curses` can obtain the relevant information it needs to carry out its instructions. For example, it needs to know how big the window is, and the whereabouts on the terminal screen it is to be placed. In fact, all the information `curses` requires to maintain a window is contained within the `WINDOW` data structure.

## The VARIABLES

### `_cury` and `_curx`

The variables `_cury` and `_curx` contain the current `y,x` coordinates of the cursor on the window. If characters are added to the window, this is where the next character is to be inserted. Note that the given coordinates are in the `y,x` order and not the conventional `x,y` order.

To specify the location of a window cursor, you need to use a coordinate system. `Curses` uses a coordinate system consisting of horizontal and vertical axes at right angle to each other. The coordinates are numbers that define the location of a point within a given window with reference to an origin. The coordinates themselves are equally spaced units in both `Y` (vertical) and `X` (horizontal) directions. `Y` coordinates are expressed in terms of lines (or rows) down the screen, and `X` coordinates are expressed in terms of columns (or character positions) across it.

The `Y` coordinate is the vertical position within a `curses` window, and its origin is the top-left-hand side of the window. The `X` coordinate is the horizontal position across a `curses` window, its origin also being the top-left-hand side. Both `Y` and `X` origins are 0 (zero)-based. The window coordinated (0,0) is referred to as the home position.

### `_maxy` and `_maxx`

The variables `_maxy` and `_maxx` specify the outermost dimensions of a `curses` window. `_maxy` specifies how many lines span down the window. Similarly, `_maxx` specifies how many columns span across it.

### `_begy` and `_begx`

The variables `_begy` and `_begx` specify the starting `y,x` coordinates of the window. From these variables, `curses` can compute the position of the window in relation to the dimensions of `stdscr`. However, the upper left-hand corners of a window are always 0,0 even if the window is placed in the center of the terminal display.

### `_flags`

The `_flags` variable is used as a bit mask. Many `curses` routines manipulate this bit mask, or at least consult its contents throughout program execution. Generally, the `_flags` variable is used for optimization purposes. The following are descriptions of various flags:

<code>_SUBWIN</code>	This tells curses that the window is a subwindow.
<code>_ENDLINE</code>	This tells curses that the right-most-end of each line in the window is the edge of the screen.
<code>_FULLWIN</code>	This tells curses that the window is the size of the physical screen; that is, the window fills the whole terminal screen.
<code>_SCROLLWIN</code>	This tells curses that the terminal will scroll if a character is placed into the lower right edge of the physical screen. Although present in System V version of curses, it is not used.
<code>_FLUSH</code>	This flag is currently unused.
<code>_ISPAD</code>	This tells curses that the window is a pad; basically this means that the window can be bigger than the physical terminal screen.
<code>_STANDOUT</code>	This tells curses that characters added to the screen are to be displayed in standout mode. This means characters added to the window will be highlighted in some way, normally reverse video. This flag is provided for compatibility reasons only. It has now been replaced by the constant <code>A_STANDOUT</code>
<code>_NOCHANGE</code>	Used for optimization purposes. When refreshing a window, and if a line on the window has not been changed since the last update on the window, curses assumes that there is no need to update it again and so ignores it.
<code>_WINCHANGED</code>	This tells curses that the window has been modified in some way since it was last updated.
<code>_WINMOVED</code>	This tells curses that the cursor has been relocated to another position within a window. When next reading input into another window, the window is <i>refreshed</i> first so that the cursor is in the correct position of the read.

All these flags are manipulated internally by the curses routines. For portability reasons it is bad practice to manipulate the flags within a programmer's curses program. The curses library provides routines that would handle manipulation of the flags.

### `_attrs`

The `_attrs` variable contains various attribute flags relative to a window. Attributes are specified as associated video display capabilities of a terminal. Not all terminals support them.

Various curses routines are used to turn attributes on and off in a window. For example, to display a string of characters on the screen in highlighted video mode, the `standout` video attribute must be turned on resulting to the addition of this attribute to any character added to the screen provided the terminal is capable of displaying characters this way.

Curses can figure out which attributes are set by examining the `_attrs` flag. If attributes are set they are automatically OR'ed into characters as they are added to the window. The package applies these attributes to the terminal screen when `refresh()` is called on the window. For example, if we want to display a string of characters in `standout` mode. As curses updates the physical screen, and finds that the next character is to be displayed in this mode, it sends a special escape sequence to the terminal which, hopefully, renders the terminal in `standout` mode. If curses finds that the next character to display is in normal display mode and `standout` mode is current, it will send the escape sequence to turn the `standout` mode off.

### `_clear`

The `_clear` variable is used within `curses` to specify whether to clear the terminal screen before the next call to `refresh()`. The first time `refresh()` is called, this variable is set to `TRUE`. It is important to realize that if this variable is `TRUE`, then the clear screen sequence is sent to the terminal screen and the screen will be cleared irrespective of the physical size of the window involved. So it can be seen that this variable is significant only for windows that are the full size of the terminal screen. This flag can be cleared with the `clearok()` function.

```
clearok(stdscr, TRUE);
refresh();
```

### **`_leave`**

The `_leave` variable is used to tell `curses` to leave the cursor where it was before the `refresh()` was issued. This flag can be cleared with the `leaveok()` function.

```
leaveok(stdscr, TRUE);
```

This function is useful if you are writing a program which is not affected by where the cursor is left. Leaving the cursor where it is placed after an update reduces the amount of motion needed to move the cursor around the screen, so fewer characters are transmitted to the terminal. However, if `leave` is set, then `refresh()` may place the cursor at an undefined place on the terminal screen after the update is done. `Curses` assumes that leaving the cursor in some undefined place is undesirable, so it tries to make the cursor invisible when it is in this mode.

### **`_tmarg` and `_bmarg`**

The `_tmarg` and `_bmarg` variables contain the top and bottom margins of the scrolling region within a window. If the window is allowed to scroll, the amount scrolled will be the area between and including the lines specified by `_tmarg` and `_bmarg`. These variables are used by the `scroll()` function which scrolls a window up a line.

### **`scroll()`**

The `scroll()` variable tells `curses` if the window is allowed to scroll or not. This is set or cleared in a similar way with the `scrollok()` function. The following example tells `curses` that scrolling is allowed in the `stdscr` window.

```
scrollok(stdscr, TRUE);
```

### **`_use_idl`**

The `_use_idl` variable tells `curses` to use the terminal's insert/delete line feature, if the terminal is capable. You can set or clear this flag within the `idlok()` function. The insert/delete line mode tends to be visually annoying but it is often required for screen scrolling. For this reason, `curses` by default supplies all windows with this flag disabled except for UNIX System V.3 `curses` which manages the insert/delete line feature differently.

### **`_use_keypad`**

The `_use_keypad` variable tells `curses` to use the keypad. If the keyboard does not have a keypad, then `curses` will ignore this variable. The `keypad` is the set of keys on the terminal keyboard which perform some sort of action. By default, `curses` disables the keypad and does not treat these keys specially. If, however, the keypad is enabled, then the `curses` input routine `wgetch()` returns a value which corresponds to a set of defined constants in the `<curses.h>` include file. These constants have symbolic names which relate to these keys. If the keypad is disabled then `wgetch()` will return the next character in the character sequence produced by the key. The function `keypad()` is used to enable or disable keypad. It tells `curses` whether to treat the sequences of characters generated by the keys on the keypad specially.

### **`_use_meta`**

The `_use_meta` variable is used to tell curses if it is to operate in *meta* mode. If `_use_meta` is enabled, characters returned by `wgetch()` are not stripped of the eight bit, which is the normal default mode. This mode is useful for terminals which support a *meta shift key*, typically used to select a non-text character set. On UNIX systems prior to System V this can only be achieved by setting the terminal into raw mode.

Eight bit processing is not necessarily a very good idea. Many applications use only seven bits, especially if used over a network. If anything between the terminal and the application strips the top bit, eight bit processing is impossible. Not all UNIX systems can do eight-bit processing, although this is often due to a deficiency in the system's terminal device driver. The function `meta()` is used to enable or disable *meta* mode.

```
meta(stdscr, TRUE);
```

#### `_nodelay`

The `_nodelay` variable tells the input routine `wgetch()` to return without delay if there are no characters waiting in the input queue. This is useful to prevent the terminal from hanging, waiting for keyboard input. This mode is by default disabled, and the function `nodelay()` is used to set or clear the `_nodelay` flag. This example turns delay mode off in window `mywin`. When `wgetch()` is next called on this window it will return without hanging, also `wgetch()` will return `-1`.

```
nodelay(mywin, TRUE);
```

#### `_y`

The `_y` variable is of most interest to us, since this is the two-dimensional array of pointers to lines containing characters that we manipulate in a window. `_y[n]` is the *n*th line, and `_y[n][j]` is the *j*th character on the *n*th line. The array's size is defined by the curses initialization routines `newwin()` and `newterm()`.

#### `_firstch` and `_lastch`

The `_firstch` and `_lastch` arrays are to help curses optimize the number of characters on a window it is to update. Both these arrays have an element for every line in the window. Certain curses routines such as `wrefresh()` test these arrays to find out if a line has changed in a way since last being updated. Their purpose is to tell curses where to start and finish working on each line within the window when `refresh()` is called, thus helping curses optimize the number of characters on the window which need updating on the terminal screen. For example, the curses routine `touchwin()` is used to touch every character in a window, therefore making curses think the whole window has been changed. In fact none of the characters are changed at the window at all. The routine simply sets every `_firstch` element to 0, which tells curses to start updating from the beginning of each line. Then every `_lastch` element is set to the width of the window, which tells curses to end updating at the end of each line. Basically this forces curses to think that every character position in the window has been changed and needs updating on the physical screen.

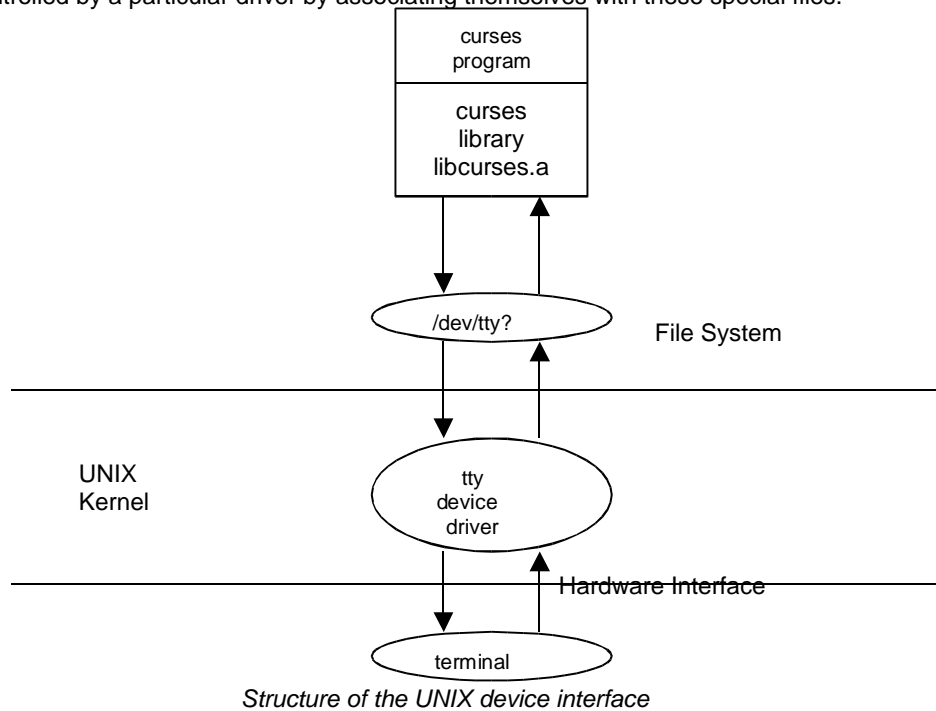
When `refresh()` is called on the *touched* window, it tests the contents of these arrays to find out where on each line it is to begin and end updating.

## The Terminal

All interaction between the user and the UNIX operating system is handled by an asynchronous terminal connected through some port. The normal standard communication method used for these devices is RS-232. This standard provides a method by which data can be transmitted by the terminal's keyboard and received by the computer. Similarly, the computer can transmit data to the terminal which is then displayed on the screen.

The only way UNIX knows about a particular device is through a piece of software linked into the kernel called a device driver. A device driver provides a platform by which a program can read from and write to a device through system entry points provided in the way of C function calls, called system calls.

Basically, these system calls are subroutines built within the UNIX kernel, but UNIX adopts a unique approach to devices. Each device is represented by a special file which exists in the file system along with other ordinary files and directories. User programs can interact with a device controlled by a particular driver by associating themselves with these special files.



The *tty* device driver essentially controls the data path between the computer and the terminal. However, although a program may have some control over a terminal through this device driver, it has no concept of what is involved in controlling the terminal.

The keyboard contains a set of alphanumeric keys which generate codes for the different letters, numbers and symbols as defined by the American Standard Code for Information Interchange (ASCH). If a key is pressed on the keyboard, an ASCH character code is generated and sent to the computer. The device driver controlling the port to which the terminal is connected receives the character and then proceeds to echo it back to the terminal so that it can be displayed. This assumes that the terminal is operating in full-duplex mode. At the same time, the character is placed into the internal system character input buffers so that a process using the `read(2)` system call on this device is able to read it.

On the terminal, as each character is received, it is placed on the screen next to the last character that was received, and so on, until it comes to the edge of the terminal screen, in which case the terminal normally starts a new line below the last one displayed. However, for some codes received by the terminal, the terminal performs some sort of action instead of displaying the character. For example, pressing a valid combination of CONTROL key and another key will produce some action.

Codes which tell the terminal to perform some sort of action are called **control codes**. The problem is that each terminal responds differently to them.

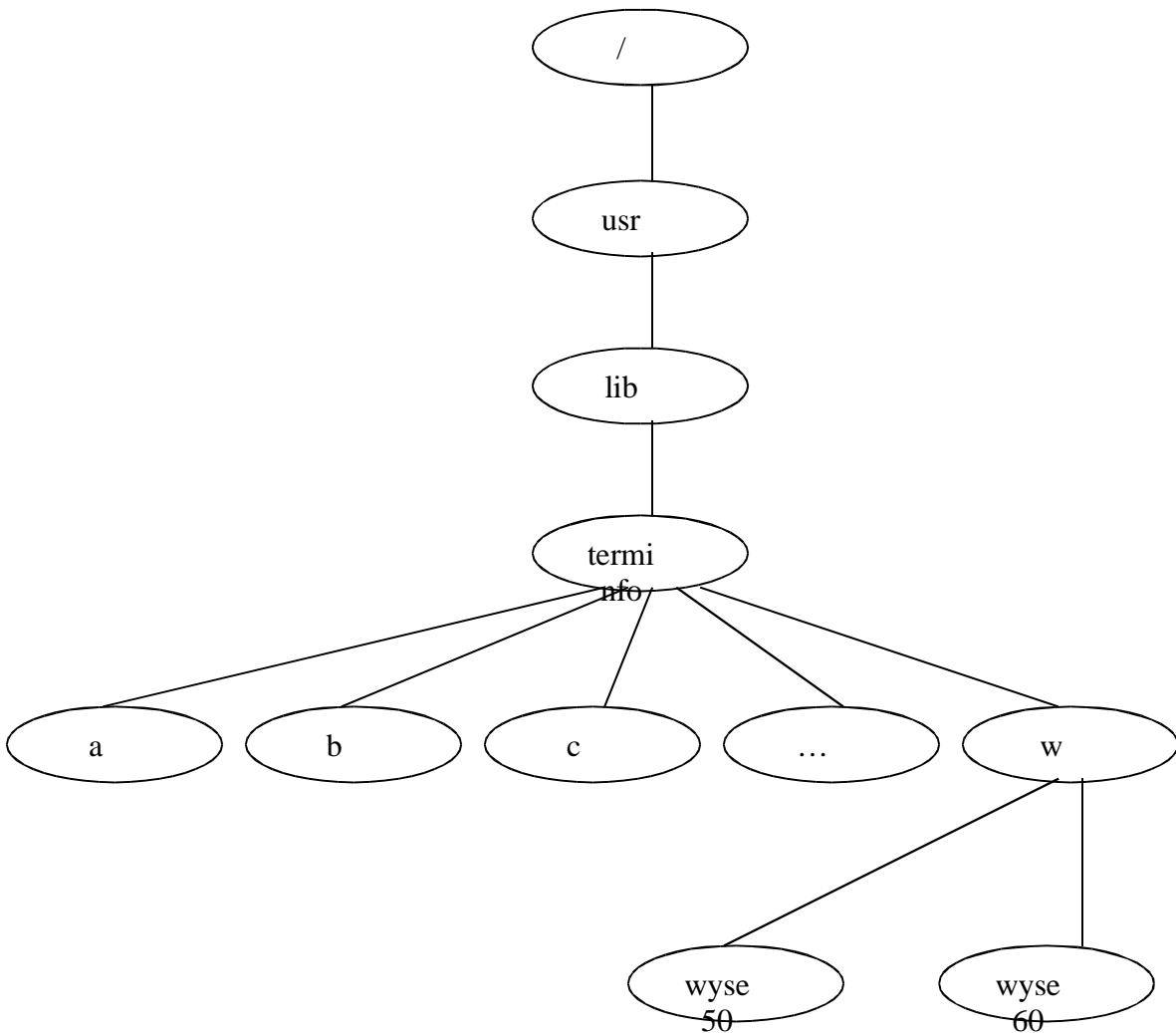
Bill Joy, creator of *vi*, came up with the idea of creating a database of terminal descriptions, outlining each individual terminal's capabilities. He also created a set of functions

which could read this database to figure out how to control the terminal it was working with. Those routines were taken by Ken Arnold, and he created the curses package.

## Terminfo

The term *terminfo* refers to a group of routines which are built into the curses library and use the capabilities of a terminal. Terminfo also refers to a database of binary files, with each file describing an individual terminal's capabilities.

The terminfo database contains descriptions for over 150 different terminals which can be used with curses written applications, and it is growing rapidly with every new release of the operating system.



### *Example of the terminfo database structure*

A terminfo description specifies what a particular terminal can do and what steps are needed to perform certain operations on the physical screen. For example, the description will specify how many lines and columns a terminal has; whether to use X-on/X-off handshaking; if the terminal is capable of displaying in reverse video; what function keys the terminal has; what special control codes are required to move the cursor around the screen; and so on. The terminfo database is a set of compiled files, with each file describing an individual terminal.

These files are referred to as **terminfo description files** and are normally found in a directory under `/usr/lib/terminfo`. They are compiled from text-based files. Each directory under this has a single character name, which is the first character of the terminal's name. For example, the Wyse 50 terminal: its terminfo description file would normally have a full path name of `/usr/lib/terminfo/w/wyse50`. However, it is possible to have your own version of a terminfo database.

As mentioned earlier, terminfo also refers to a set of functions which form part of the curses library. These functions are low-level routines which curses itself is built upon. They are provided for programmers who want to access the terminal directly without using the curses-based screen management functions. For curses to work properly, it has to be able to search the terminfo database to find the description file containing information about the terminal.

Before curses can search the terminfo database for a terminal type, it needs to know what its name is. This is achieved by assigning the terminal's name to the environment variable **TERM**.

This can be done by defining this variable in **.profile** for Bourne or Korn shell, or **.login** for Berkeley C shell. It can also be set on the command line.  
For Bourne shell:

```
TERM=wyse50 ; export TERM
```

For Berkeley csh:

```
setenv TERM wyse50
```

When starting up curses or terminfo-based programs, the terminfo databases is scanned until it finds an entry matching that which was specified in the shell environment variable: **TERM**.

Curses does make use of other environment variable, but unlike the **TERM** environment variable, they are not mandatory. If there is the environment variable **TERMINFO** defined, curses will use a local terminfo description database directory instead of the default `/usr/lib/terminfo` directory.

## **WIDGETS**

### **CDK**

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 directly in programs.

CDK was developed over several years of writing text based applications for clients. Vexus kept building the same widget sets over and over that they created a library which they took from client to client. This library matured and built into what CDK is today. Once they had a library that they were proud to show off, they put it into the open source world in 1996.

For more information on CDK and to download a copy of it, go to: <http://www.vexus.ca/CDK.html>

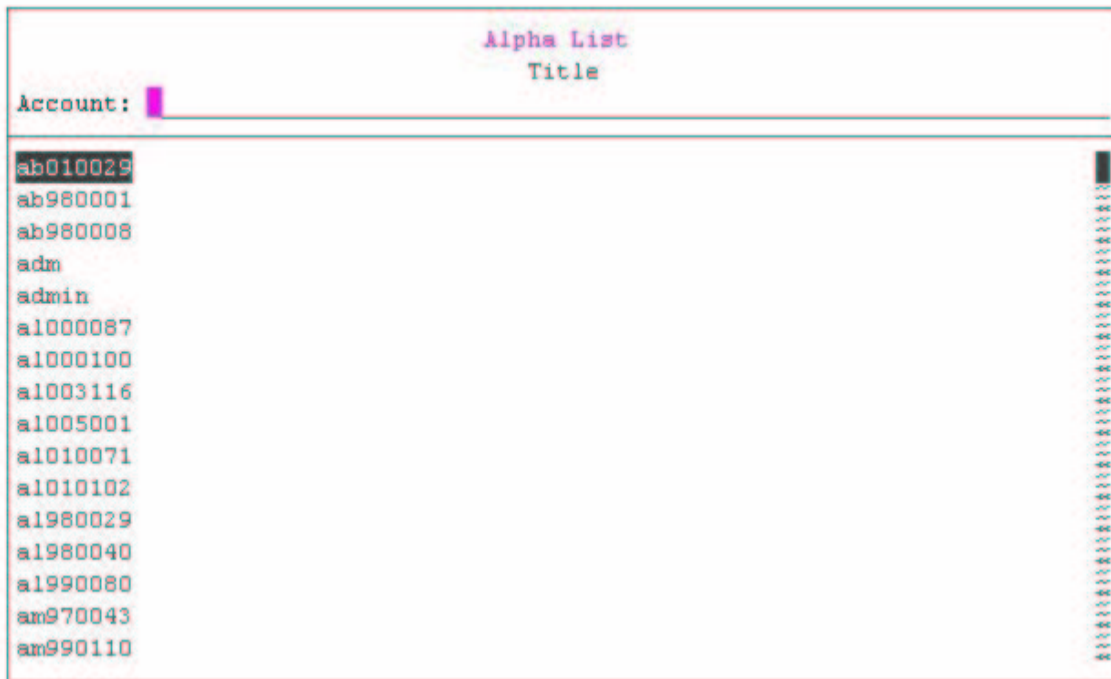
Currently CDK has a complement of over 21 ready to use widgets. The following table lists all of the available widgets and a quick description of what the widget can be used for.

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	Ceates 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.

### Code Samples of Widget Usage

Example # 1  
ALPHALIST



### Code for `alphalist_ex.c`

```
#include "cdk.h"
```

```

#ifdef HAVE_XCURSES
char *XCursesProgramName="alphalist_ex";
#endif

/*
 * This program demonstrates the Cdk alphalist widget.
 */
#define MAXINFOLINES 10000

int getUserList (char **list, int maxItems);

int main (int argc, char **argv)
{
    /* Declare variables. */
    CDKSCREEN *cdkscreen = (CDKSCREEN *)NULL;
    CDKALPHALIST *alphaList = (CDKALPHALIST *)NULL;
    WINDOW *cursesWin = (WINDOW *)NULL;
    char *title = "<C></B/24>Alpha List\n<C>Title";
    char *label = "</B>Account: ";
    char *word = (char *)NULL;
    char *info[MAXINFOLINES], *mesg[10], temp[256];
    int count;

    /* Set up CDK. */
    cursesWin = initscr();
    cdkscreen = initCDKScreen (cursesWin);

    /* Start color. */
    initCDKColor();

    /* Get the user list. */
    count = getUserList (info, MAXINFOLINES);

    /* Create the alpha list widget. */
    alphaList = newCDKAlphalist (cdkscreen, CENTER, CENTER,
                                0, 0, title, label,
                                info, count,
                                '_', A_REVERSE, TRUE, FALSE);

    /* Let them play with the alpha list. */
    word = activateCDKAlphalist (alphaList, NULL);

    /* Determine what the user did. */
    if (alphaList->exitType == vESCAPE_HIT) {
        mesg[0] = "<C>You hit escape. No word was selected.";
        mesg[1] = "";
        mesg[2] = "<C>Press any key to continue.";
        popupLabel (cdkscreen, mesg, 3);
    }

    else if (alphaList->exitType == vNORMAL) {
        mesg[0] = "<C>You selected the following";
        sprintf (temp, "<C>(%s)", word);
        mesg[1] = copyChar (temp);
        mesg[2] = "";
        mesg[3] = "<C>Press any key to continue.";
        popupLabel (cdkscreen, mesg, 4);
    }

    /* Clean up. */
    destroyCDKAlphalist (alphaList);
    delwin (cursesWin);
    endCDK();
    exit (0);
}

/*
 * This reads the passwd file and retrieves user information.
 */
int getUserList (char **list, int maxItems)
{

```

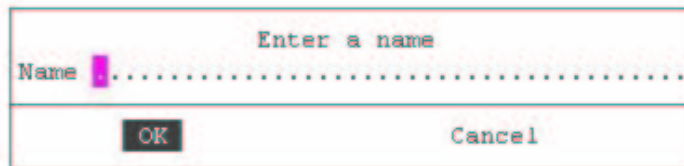
```

struct passwd *ent;
int x = 0;

while ( (ent = getpwent ()) != NULL) {
    list[x++] = copyChar (ent->pw_name);
}
return x;
}

```

## Example #2 BUTTONBOX



### Code for `buttonbox_ex.c`

```

#include "cdk.h"

#ifdef HAVE_XCURSES
char *XCursesProgramName="buttonbox_ex";
#endif

int entryCB (EObjectType cdktype, void *object, void *clientData, chtype key);

/*
 * This program demonstrates the Cdk buttonbox widget.
 */
int main (int argc, char **argv)
{
    /* Declare variables. */
    CDKSCREEN *cdkscreen = (CDKSCREEN *)NULL;
    CDKBUTTONBOX *buttonWidget = (CDKBUTTONBOX *)NULL;
    CDKENTRY *entry = (CDKENTRY *)NULL;
    WINDOW *cursesWin = (WINDOW *)NULL;
}

```

```

char *buttons[]          = {" OK ", " Cancel "};
char *info               = (char *)NULL;
int selection;

/* Set up CDK. */
cursesWin = initscr();
cdkscreen = initCDKScreen (cursesWin);

/* Start color. */
initCDKColor();

/* Create the entry widget. */
entry = newCDKEntry (cdkscreen, CENTER, CENTER,
                    "<C>Enter a name", "Name ", A_NORMAL, '|', vMIXED,
                    40, 0, 256, TRUE, FALSE);

/* Create the button box widget. */
buttonWidget = newCDKButtonbox (cdkscreen,
                                WIN_XPOS (entry->win),
                                WIN_YPOS (entry->win) + entry->boxHeight - 1,
                                1, entry->boxWidth - 1,
                                NULL, 1, 2,
                                buttons, 2, A_REVERSE,
                                TRUE, FALSE);

/* Set the lower left and right characters of the box. */
setCDKEntryLLChar (entry, ACS_LTEE);
setCDKEntryLRChar (entry, ACS_RTEE);
setCDKButtonboxULChar (buttonWidget, ACS_LTEE);
setCDKButtonboxURChar (buttonWidget, ACS_RTEE);

/*
 * Bind the Tab key in the entry field to send a
 * Tab key to the button box widget.
 */
bindCDKObject (vENTRY, entry, KEY_TAB, entryCB, buttonWidget);

/* Activate the entry field. */
drawCDKButtonbox (buttonWidget, TRUE);
info = copyChar (activateCDKEntry (entry, NULL));
selection = buttonWidget->currentButton;

/* Clean up. */
destroyCDKButtonbox (buttonWidget);
destroyCDKEntry (entry);
destroyCDKScreen (cdkscreen);
endCDK();
delwin (cursesWin);

/* Spit out some info. */
printf ("You typed in (%s) and selected button (%s)\n", info, buttons[selection]);
exit (0);
}

int entryCB (EObjectType cdktype, void *object, void *clientData, chtype key)
{
    CDKBUTTONBOX *buttonbox = (CDKBUTTONBOX *)clientData;
    injectCDKButtonbox (buttonbox, key);
    return TRUE;
}

```

Example #3  
CALENDAR



**Code for calendar\_ex.c**

```
#include "cdk.h"

#ifdef HAVE_XCURSES
char *XCursesProgramName="calendar_ex";
#endif

int createCalendarMarkCB (EObjectType objectType, void *object, void *clientData, chtype key);
int removeCalendarMarkCB (EObjectType objectType, void *object, void *clientData, chtype key);

/*
 * This program demonstrates the Cdk calendar widget.
 */
int main (int argc, char **argv)
{
    /* Declare variables. */
    CDKSCREEN *cdkscreen = (CDKSCREEN *)NULL;
    CDKCALENDAR *calendar = (CDKCALENDAR *)NULL;
    WINDOW *cursesWin = (WINDOW *)NULL;
    char *title = "<C></U>CDK Calendar Widget<C>Demo";
    int day, month, year, ret;
    char *mesg[10], temp[256];
    struct tm *dateInfo;
    time_t cclk, retVal;

    /*
     * Get the current dates and set the default values for
    */
}
```

```

* the day/month/year values for the calendar.
*/
time (&clck);
dateInfo      = localtime (&clck);
day           = dateInfo->tm_mday;
month         = dateInfo->tm_mon + 1;
year         = dateInfo->tm_year + 1900;

/* Check the command line for options. */
while (1) {
    ret = getopt (argc, argv, "d:m:y:t");

    /* Are there any more command line options to parse. */
    if (ret == -1) {
        break;
    }

    switch (ret) {
        case 'd':
            day = atoi (optarg);
            break;

        case 'm':
            month = atoi (optarg);
            break;

        case 'y':
            year = atoi (optarg);
            break;

        case 't':
            title = copyChar (optarg);
            break;
    }
}

/* Set up CDK. */
cursesWin = initscr();
cdkscreen = initCDKScreen (cursesWin);

/* Start CDK Colors. */
initCDKColor();

/* Create the calendar widget. */
calendar = newCDKCalendar (cdkscreen, CENTER, CENTER,
                           title, day, month, year,
                           COLOR_PAIR(16)|A_BOLD,
                           COLOR_PAIR(24)|A_BOLD,
                           COLOR_PAIR(32)|A_BOLD,
                           COLOR_PAIR(40)|A_REVERSE,
                           TRUE, FALSE);

/* Is the widget NULL? */
if (calendar == (CDKCALENDAR *)NULL) {
    /* Clean up the memory. */
    destroyCDKScreen (cdkscreen);

    /* End curses... */
    endCDK();

    /* Spit out a message. */
    printf ("Oops. Can't seem to create the calendar. Is the window too small?\n");
    exit (1);
}

/* Create a key binding to mark days on the calendar. */
bindCDKObject (vCALENDAR, calendar, 'm', createCalendarMarkCB, calendar);
bindCDKObject (vCALENDAR, calendar, 'M', createCalendarMarkCB, calendar);
bindCDKObject (vCALENDAR, calendar, 'r', removeCalendarMarkCB, calendar);
bindCDKObject (vCALENDAR, calendar, 'R', removeCalendarMarkCB, calendar);

```

```

/* Draw the calendar widget. */
drawCDKCalendar (calendar, calendar->box);

/* Let the user play with the widget. */
retVal = activateCDKCalendar (calendar, NULL);

/* Check which day they selected. */
if (calendar->exitType == vESCAPE_HIT) {
    msg[0] = "<C>You hit escape. No date selected.";
    msg[1] = "";
    msg[2] = "<C>Press any key to continue.";
    popupLabel (cdkscreen, msg, 3);
}

else if (calendar->exitType == vNORMAL) {
    msg[0] = "You selected the following date";
    sprintf (temp, "<C></B/16>%02d/%02d/%d (dd/mm/yyyy)", calendar->day, calendar->month, calendar->year);
    msg[1] = copyChar (temp);
    msg[2] = "<C>Press any key to continue.";
    popupLabel (cdkscreen, msg, 3);
    freeChar (msg[1]);
}

/* Clean up and exit. */
destroyCDKCalendar (calendar);
destroyCDKScreen (cdkscreen);
delwin (cursesWin);
endCDK();
fflush (stdout);
printf ("Selected Time: %s\n", ctime(&retVal));
exit (0);
}

/*
 * This adds a marker to the calendar.
 */
int createCalendarMarkCB (EObjectType objectType, void *object, void *clientData, chtype key)
{
    CDKCALENDAR *calendar = (CDKCALENDAR *)object;

    setCDKCalendarMarker (calendar,
                          calendar->day,
                          calendar->month,
                          calendar->year,
                          COLOR_PAIR (5) | A_REVERSE);

    drawCDKCalendar (calendar, calendar->box);
    return 0;
}

/*
 * This removes a marker from the calendar.
 */
int removeCalendarMarkCB (EObjectType objectType, void *object, void *clientData, chtype key)
{
    CDKCALENDAR *calendar = (CDKCALENDAR *)object;

    removeCDKCalendarMarker (calendar,
                              calendar->day,
                              calendar->month,
                              calendar->year);

    drawCDKCalendar (calendar, calendar->box);
    return 0;
}

```

Example #4  
MENU

File Edit

Help



```
Title: File
Sub-Title: Save
```

```
This saves the current info.█
```

### Code for menu\_ex.c

```
#include "cdk.h"

#ifdef HAVE_XCURSES
char *XCursesProgramName="menu_ex";
#endif

int displayCallback (EObjectType cdktype, void *object, void *clientData, chtype input);
char *menulist[MAX_MENU_ITEMS][MAX_SUB_ITEMS];
char *menuInfo[3][4] = {{"", "This saves the current info.", "This exits the program.", ""},
                        {"", "This cuts text", "This copies text", "This pastes text"},
                        {"", "Help for editing", "Help for file management", "Info about the program"}};

/*
 * This program demonstratres the Cdk menu widget.
 */
int main (int argc, char **argv)
{
    /* Declare variables. */
    CDKSCREEN *cdkscreen = (CDKSCREEN *)NULL;
    CDKLABEL *infoBox = (CDKLABEL *)NULL;
    CDKMENU*menu = (CDKMENU *)NULL;
    WINDOW*cursesWin = (WINDOW *)NULL;
    int submenuSize[3], menuLoc[4];
    char *mesg[10], temp[256];
    int selection;

    /* Set up CDK. */
    cursesWin = initscr();
    cdkscreen = initCDKScreen (cursesWin);

    /* Start CDK color. */
    initCDKColor();

    /* Set up the menu. */
    menulist[0][0] = "</B>File<!B>" ; menulist[1][0] = "</B>Edit<!B>"; menulist[2][0] = "</B>Help<!B>";
```

```

menulist[0][1] = "</B>Save<!B>"; menulist[1][1] = "</B>Cut<!B> "; menulist[2][1] = "</B>On Edit <!B>";
menulist[0][2] = "</B>Exit<!B>"; menulist[1][2] = "</B>Copy<!B>"; menulist[2][2] = "</B>On File <!B>";
        menulist[1][3] = "</B>Paste<!B>"; menulist[2][3] = "</B>About...<!B>";
submenuSize[0] = 3;      menuLoc[0] = LEFT;
submenuSize[1] = 4;      menuLoc[1] = LEFT;
submenuSize[2] = 4;      menuLoc[2] = RIGHT;

/* Create the label window. */
msg[0] = "                ";
msg[1] = "                ";
msg[2] = "                ";
msg[3] = "                ";
infoBox = newCDKLabel (cdkscreen, CENTER, CENTER, msg, 4, TRUE, TRUE);

/* Create the menu. */
menu = newCDKMenu (cdkscreen, menulist, 3, submenuSize, menuLoc,
                  TOP, A_UNDERLINE, A_REVERSE);

/* Create the post process function. */
setCDKMenuPostProcess (menu, displayCallback, infoBox);

/* Draw the CDK screen. */
refreshCDKScreen (cdkscreen);

/* Activate the menu. */
selection = activateCDKMenu (menu, (chtype *)NULL);

/* Determine how the user exited from the widget. */
if (menu->exitType == vEARLY_EXIT) {
    msg[0] = "<C>You hit escape. No menu item was selected.";
    msg[1] = "";
    msg[2] = "<C>Press any key to continue.";
    popupLabel (cdkscreen, msg, 3);
}

else if (menu->exitType == vNORMAL) {
    sprintf (temp, "<C>You selected menu %d, submenu %d", selection/100, selection%100);
    msg[0] = copyChar (temp);
    msg[1] = "";
    msg[2] = "<C>Press any key to continue.";
    popupLabel (cdkscreen, msg, 3);
    freeChar (msg[0]);
}

/* Clean up. */
destroyCDKMenu (menu);
destroyCDKLabel (infoBox);
destroyCDKScreen (cdkscreen);
delwin (cursesWin);
endCDK();
exit (0);
}

/*
 * This gets called after every movement.
 */
int displayCallback (EObjectType cdktype, void *object, void *clientData, chtype input)
{
    CDKMENU *menu      = (CDKMENU *)object;
    CDKLABEL *infoBox  = (CDKLABEL *)clientData;
    char *msg[10];
    char temp[256];

    /* Recreate the label message. */
    sprintf (temp, "Title: %s", menulist[menu->currentTitle][0]);
    msg[0] = strdup (temp);
    sprintf (temp, "Sub-Title: %s", menulist[menu->currentTitle][menu->currentSubtitle+1]);
    msg[1] = strdup (temp);
    msg[2] = "";
    sprintf (temp, "<C>%s", menuInfo[menu->currentTitle][menu->currentSubtitle+1]);
    msg[3] = strdup (temp);

```

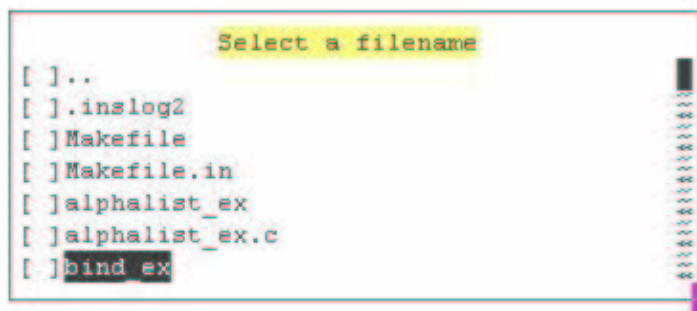
```

/* Set the message of the label. */
setCDKLabel (infoBox, mesg, 4, TRUE);
drawCDKLabel (infoBox, TRUE);

/* Clean up. */
freeChar (mesg[0]);
freeChar (mesg[1]);
freeChar (mesg[3]);
return 0;
}

```

## Example #5 RADIO BUTTONS



### Code for radio\_ex.c

```

#include "cdk.h"

#ifdef HAVE_XCURSES
char *XCursesProgramName="radio_ex";
#endif

/*
 * This program demonstrates the Cdk radio widget.
 */
int main (int argc, char **argv)
{
    /* Declare variables. */
    CDKSCREEN *cdkscreen = (CDKSCREEN *)NULL;
    CDKRADIO *radio = (CDKRADIO *)NULL;
    WINDOW *cursesWin = (WINDOW *)NULL;
    char *title = "<C></5>Select a filename";
    char *item[200], *mesg[10], temp[100];
    int selection, count, x;

    /* Set up CDK. */
    cursesWin = initscr();
    cdkscreen = initCDKScreen (cursesWin);

```

```

/* Set up CDK Colors. */
initCDKColor();

/* Use the current directory list to fill the radio list. */
count = getDirectoryContents(".", item, 200);

/* Create the radio list. */
radio = newCDKRadio (cdkscreen, CENTER, CENTER, RIGHT,
                    10, 40, title, item, count,
                    '#|A_REVERSE, 1,
                    A_REVERSE, TRUE, FALSE);

/* Check if the radio list is NULL. */
if (radio == (CDKRADIO *)NULL) {
    /* Exit CDK. */
    destroyCDKScreen (cdkscreen);
    endCDK();

    /* Print out a message and exit. */
    printf ("Oops. Can't seem to create the radio widget. Is the window too small??\n");
    exit (1);
}

/* Activate the radio list. */
selection = activateCDKRadio (radio, (chtype *)NULL);

/* Check the exit status of the widget. */
if (radio->exitType == vESCAPE_HIT) {
    msg[0] = "<C>You hit escape. No item selected.";
    msg[1] = "";
    msg[2] = "<C>Press any key to continue.";
    popupLabel (cdkscreen, msg, 3);
}

else if (radio->exitType == vNORMAL) {
    msg[0] = "<C>You selected the filename";
    sprintf (temp, "<C>%s", item[selection]);
    msg[1] = copyChar (temp);
    msg[2] = "";
    msg[3] = "<C>Press any key to continue.";
    popupLabel (cdkscreen, msg, 4);
    freeChar (msg[1]);
}

/* Clean up. */
for (x=0; x < count; x++) {
    freeChar (item[x]);
}
destroyCDKRadio (radio);
destroyCDKScreen (cdkscreen);
delwin (cursesWin);
endCDK();
exit (0);
}

```

## End of Code Samples

## BUILDING AN NCURSE PROGRAM

### Compiling With the Ncurses Library

To use ncurses library functions, you have to include ncurses.h and to link the program with ncurses library the flag -lncurses should be added. ncurses.h already includes stdio.h.

### How to Build an Ncurses Program

#### *Basic Program Structure*

To learn how to create an ncurses program, you need to understand the structure of an ncurses-based program. The basic structure of an ncurses program is as follows:

1. command line processing

Before starting up ncurses, it is a good idea to do any command line processing first. Do not enter into ncurses mode if you are not yet ready to do so.

2. initialize ncurses

Many internal variables and structures must be set up before ncurses is ready to use. Therefore, ncurses must be initialized. This is done with `initscr ( )` which sets up the provided windows, `curscr` and `stdscr`, initializes the terminal, and then places the program into an in-curses mode.

3. set up desired terminal I/O modes

If you want to set any special I/O modes in the terminal driver, this should be done after initialization. These modes are normally set up once only, and then remain unmodified throughout program execution.

4. continue processing in ncurses mode

The program now operating in curses mode can continue processing with the curses facilities. For example, create/delete windows, add characters to a window, get keyboard input, and so on.

5. end ncurses mode

When processing is complete and the program is ready to return to the shell (or calling process), the terminal must first be placed back into its original operating modes, the modes that were current before entering into the program. This is done with `endwin ( )`

6. exit program

*Compiling with the ncurses Library*

To use ncurses library functions, you have to include `ncurses.h` and to link the program with ncurses library the flag `-lncurses` should be added. `ncurses.h` already includes `stdio.h`.

For example:

To compile: **`gcc -Wall -Werror -o main main.c -lncurses`**  
To run: **`./main`**

## WINDOW MANAGEMENT

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.

### *Creating Windows*

If you want to use a window other than the default windows supplied by `ncurses` (`stdscr` and `curscr`), we need to create it before we can access it. `ncurses` provides the function `newwin` (`lines, cols, begy, begx`) for this purpose. The `newwin` (`lines, cols, begy, begx`) function requires 4 arguments. These arguments tell `ncurses` the dimensions of your new window, and where you want the new window placed on the terminal screen.

The arguments are specified as follows:

`lines` the maximum vertical dimension of the new window, specified in units of lines

`cols` the maximum horizontal dimension of the new window specified in units of columns

`begy` the line coordinate, specifying where the new window will start in relation to the `stdscr` vertical dimension

`begx` the column coordinate, specifying where the new window will start in relation to the `stdscr` horizontal dimension.

`newwin` (`lines, cols, begy, begx`) computes the dimension of the new window by using the following equations:

```
if ( begy + lines > LINES)
    lines = LINES - begy;
if ( begx + cols > COLS)
    cols = COLS - begx;
if ( lines == 0 )
    lines = LINES - begy;
if ( cols == 0 )
    cols = COLS - begx;
```

The dimensions of the new window must be equal to or within the maximum dimensions of the `stdscr` window, which is specified in the variables `LINES` and `COLS`.

The following example creates a window which is 10 lines high by 40 columns across, and it is placed in relation to `stdscr` at line 5, column 10:

```
#include <ncurses.h>

main( )
{
```

```

WINDOW *new;
initscr();
noecho();
nodelay(stdscr,TRUE);
    if (LINES < 10 || COLS < 40)
        fatal_err("terminal screen too small");

    new = newwin (10,40,5,10)
    if (new == (WINDOW *) NULL)
        fatal_err("memory error");
    /* processing continues */
}

fatal_err (str)
char *str;
{
    mvprintw(LINES - 1, 0, "Fatal Error: %s\n", str);
    refresh ( );
    endwin ( );
    exit (1);
}

```

This is a typical way to start a curses program. After initializing curses with `initscr()`, the terminal operating modes are set. `newwin()` is then called to initialize a new window structure. However if `newwin()` cannot allocate enough memory for it, it will fail and `(WINDOW *) NULL` is returned. The program then calls `fatal_err()` which prints the error message to the lower left-hand side of the terminal screen. This routine `mvprintw()` is used for this.

#### *Adding characters to a window*

The most primitive way to add a character to a window is by using `waddch(win,ch)`. This function is used by all curses functions which add characters to a window in some way. We could say that `waddch()` is the lowest level curses function used to add a character to a window.

In keeping with the philosophy of UNIX, that is, **build on the work of others**, the standard I/O package function `printf()` has been implemented in the curses package as `wprintw()`. This function actually uses the standard `printf()` function to expand the format string argument. It then adds it to the given window with `waddstr()`. This means that you can format a string and pass it to `wprintw()` as an argument, just as you would with `printf()`.

Functions that add characters/prints string to a window:

```

int waddch(win,ch)           puts char in stdscr
int addch(ch)
int mvaddch(y,x,ch)
int mvwaddch(win,y,x,ch)
int waddstr(win,str)       puts string in stdscr
int addstr(str)
int mvaddstr(y,x,str)
int mvwaddstr(win,y,x,str)
int mvprintw(y,x,fmt,args)  move & print string in stdscr
int mvwprintw(win,x,y,fmt,args)  move & print string in a window
int wprintw(win,fmt,args)   print string in a window

```

#### **A Simple printw example**

```

#include <ncurses.h> /* ncurses.h includes stdio.h */
#include <string.h>

int main()
{
    char mesg[]="Hello World"; /* 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-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;
}

```

The code above prints a hello world at the center of the screen. 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.

#### *Moving around a window*

To move to a new position within a window you use the `wmove(win,y,x)` function. The new coordinates must be within the bounds of the window's maximum dimensions of `ERR` is returned and the coordinates are not changed.

This function is rarely used on its own. Usually you would want to move to a particular location within a window and print a message of some sort, or read something from the keyboard and display characters typed at some location other than where the cursor is. For this reason, a set of move macros are provided in the `<ncurses.h>` header file. This example uses `mvwaddstr()`:

<code>#include &lt;ncurses.h&gt;</code>
<code>main( )</code>
<code>{</code>
<code>    WINDOW *mywin;</code>
<code>initscr();</code>
<code>mywin = newwin (10,40,5,10)</code>
<code>    if (mywin == (WINDOW *) NULL) {</code>
<code>        mvaddstr(LINES - 1, 0, "memory error");</code>
<code>        refresh ( );</code>
<code>        endwin ( );</code>
<code>        exit (1);</code>
<code>    }</code>
<code>    mvaddstr(mywin,10/2,40/2, "Hello World");</code>
<code>    wrefresh ( );</code>
<code>    endwin ( );</code>
<code>    exit (0);</code>
<code>}</code>

A function prefixed with `mv` uses the `move()` function and requires at least two arguments: the new `y,x` coordinates to move to. Some routines are prefixed with both `mv` and `w`, and in this case the `mv` prefix is placed before the `w` prefix but the window argument is

always specified first. For example, the functions `mvprintw()` and `mvwprintw()` are effectively the same, in that both functions print a string to a window at a specified location. The only exception is that `mvprintw()` is not window-specific. It is used solely to print to `stdscr`, and so does not require a window argument.

### *Reading from a Window*

The only method of reading a character from a window is to use the pseudo-function `winch(win)` (`inch()` for `stdscr`). This function returns the character under the cursor at the current `y,x` coordinates of a window, although this character may contain attributes if they are set, as well as the character itself.

The main thing to remember here is that all the characters in a window are stored in the two-dimensional array `_y[][]` which is part of the window structure referring to the window you are working with; it's a bit like having a private memory map of the terminal screen. If you want to print the whole screen you simply call `wprtscr(curscr)`.

Some functions:

<code>int winch(win)</code>	get char at window cursor
<code>int inch()</code>	
<code>int mvinch(y,x)</code>	
<code>int mvwinch(win,y,x)</code>	
<code>int wansch(win,ch)</code>	insert character in a window
<code>int insch(ch)</code>	
<code>int mvansch(y,x,ch)</code>	
<code>int mvwansch(win,y,x,ch)</code>	
<code>int winsertln(win)</code>	insert new line in a window
<code>int insertln()</code>	
<code>int mvinsertln(y,x)</code>	
<code>int mvwinertln(win,y,x)</code>	

### *Deleting characters in a Window*

The function `wdelch(win)` is used to delete a character under the cursor on the specified window. Each character along the current line after the cursor in the window is shifted left one position and the last character becomes blank. The current `y,x` coordinates remain unchanged.

Some functions:

<code>int wdelch(win)</code>	delete a char in a window
<code>int delch()</code>	
<code>int mvdelch(y,x)</code>	
<code>int mvwdelch(win,y,x)</code>	
<code>int wdeleteln(win)</code>	delete a line in a window
<code>int deleteln()</code>	
<code>int mvdeleteln(y,x)</code>	
<code>int mvwdeleteln(win,y,x)</code>	

### *Deleting a Window*

The process of deleting a window loses the contents of it and all space associated with it is freed up. The function which does this is `delwin(win)`. If the window has any associated sub-windows they should be deleted first because `delwin()` does not delete sub-windows automatically, although by deleting the parent window it effectively invalidates the sub-window. It is important to do this so that memory is freed up for other uses. Also, if you access a sub-window after the parent is deleted, your program may dump core which will almost definitely leave the terminal in an unpredictable state.

### *Relocating a Window*

The function `mvwin(win,begy,begx)` is used to relocate a window. The given window is moved to the new location as specified by `begy` and `begx`, whose coordinates are the window's new top left-hand corner coordinates. The move must not allow the window or portion of it to overlap the terminal screen; if it does, the window is not moved and the function returns `ERR`. When a window is moved from one location to another, the internal `WINDOW` data structure remains intact except, of course, `_begy` and `_begx`. This means that characters displayed with attributes will retain them throughout the move.

## EVENT HANDLING

### *Setting Terminal Input Modes*

In normal operations, the terminal operates in full-duplex mode – meaning the characters that are typed at the keyboard are echoed back to the terminal via the computer. Characters may be typed at any time, even if output is still occurring. The `tty` device driver buffers input characters as they are received in a system character input buffer (`clist`). If the input buffer overflows, or if the total characters in the buffer exceed the system-specified limit without a process reading them, characters may be lost.

Also, normally, input is processed in units of lines which are delimited by a new line (`NL`), end of line (`EOL`), or end of file (`EOF`) character. This means a process attempting to read from the terminal device is suspended until a delimiting character is received. However, certain characters received by the driver have special meaning, and the driver interprets them depending on how it is configured. They are as follows:

**Interrupt** (ASCII `DEL`) Generates an interrupt signal which is received by all running processes associated with the terminal. A process receiving this signal terminates, unless it has been programmed to ignore it.

**Quit** (Control-`I`) Identical to **Interrupt**, except that a file is created in the current directory containing a core image of the process.

**Stop** (Control-`S`) Temporarily suspends output to the terminal.

**Start** (Control-`Q`) Resumes output currently suspended by **Stop**.

**Erase** ( `#` ) erases the last character typed, but will not erase beyond the start of a line, as delimited by a `NL`, `EOL`, or `EOF` character. This is often set to Control-`H`.

**Kill** ( `@` ) Deletes the entire line, as delimited by a `NL`, `EOF`, or `EOL` character. This is often set to Control-`C` or Control-`U`.

**New-Line** (ASCII `NL`) Line delimiter (satisfies the read). Characters stored in the `tty` driver's internal buffer are passed over to the process which is reading from the terminal and the buffer is flushed.

`Curses` retains these terminal modes by default. Consequently, this makes problems for programs which must know exactly what keys have been depressed, without being interpreted. Also, in most cases, an interactive `curses` program should not be kept waiting for a new-line character to be typed at the keyboard.

To combat these problems, `curses` provides a comprehensive set of routines for setting and reading the terminal driver modes. Some of these functions are summarized below:

`cbreak()` Sets the terminal into `cbreak` mode. This means that characters typed at the terminal are not buffered by the `tty` driver; they become immediately available to the program as they are entered at the keyboard, without having to wait for a line delimiter.

`raw()` The terminal is set into raw mode. This works the same way `cbreak` does, except that all characters are passed directly to the program without being interpreted. This includes interrupt, quit, start and stop characters.

`echo()` used in conjunction with `wgetch()`. If `echo` is enabled, then `wgetch()` automatically adds character typed at the keyboard to the working window. When the window is refreshed the characters read by `wgetch()` are sent back to the terminal.

`noecho()` In this mode, characters are not added to the window by `wgetch()`. It is up to you to add them to it. This is normally done with `waddch()` followed by `wrefresh()`. This mode is often required by programs which have to do their own echoing, possibly in a controlled area of the screen, or if the characters entered are control characters specific to the program, specifying some action. This type of character is often not displayed at all.

`savetty()` This function saves the settings of the current terminal (`tty`) modes in the internal buffer. It is automatically called by `initscr()`.

`resetty()` This function restores the terminal (`tty`) modes to what was originally saved in the internal buffer, set by `savetty()`.

### Key Management

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.

### Reading from the Keyboard

The function `wgetch(win)` is used to get keyboard input from a specified window. Depending on what input modes are set, `wgetch()` returns the key typed at the keyboard. 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 be 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.

Functions that read input:

<code>int wgetch(win)</code>	get char via a window
<code>int getch()</code>	
<code>int mvgetch(y,x)</code>	
<code>int mvwgetch(win,y,x)</code>	
<code>int wgetstr(win, str)</code>	get string via window to a buffer
<code>int getstr(str)</code>	
<code>int mvgetstr(y,x, str)</code>	
<code>int mvwgetstr(win,y,x, str)</code>	
<code>int scanw(fmt, args)</code>	get values via <code>stdscr</code>
<code>int mvscanw(y,x, fmt, args)</code>	move & get values via <code>stdscr</code>
<code>int mvwscanw(win,y,x, fmt, args)</code>	move & get values via a window

### A Simple scanw example

```

#include <curses.h> /* ncurses.h includes stdio.h */
#include <string.h>

int main()
{
    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-strlen(mesg))/2,"%s",mesg);
/* print the message at the center of the
screen */
    getstr(str);
    mvprintw(23, 0, "You Entered: %s", str);
    getch();
    endwin();

    return 0;
}

```

The code above gets the string that the user types in using `getstr ( )` and then prints it using `mvprintw ( )`.

#### *Interfacing with the mouse*

A good user interface should be able to handle input from both keyboard and mouse. To handle mouse events, 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 of the function above 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

Once a class of mouse events have been enabled, the `getch()` class of functions will return `KEY_MOUSE` every time some mouse event happens. Then the mouse event can be retrieved with a function called `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's 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 that 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");
```

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

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

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

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

**Example: Access a menu with mouse !!!**

```

#include < curses.h>

#define WIDTH 30
#define HEIGHT 10

int startx = 0;
int starty = 0;

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

    /* Initialize curses */
    initscr();
    clear();
    noecho();
    cbreak(); //Line buffering disabled. pass on 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");
    refresh();
    attroff(A_REVERSE);

    /* Print the menu for the first time */
    menu_win = newwin(HEIGHT, WIDTH, starty, startx);
    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 +
strlen(choices[choice]))
        {
            if(choice == n_choices - 1)
                *p_choice = -1;
            else
                *p_choice = choice + 1;
            break;
        }
}

```

## REFERENCES

Goodheart, Berny. UNIX Curses Explained. Australia: Prentice Hall, 1991.

Ben-Halim, Zeyd M. and Raymond, Eric S.  
<http://web.cs.mun.ca/~rod/ncurses/ncurses.html#scope>

Dickey, Thomas. <http://dickey.his.com/ncurses/ncurses.faq.html>  
<http://dickey.his.com/ncurses/ncurses.html>

Matloff, Norman.  
<http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.html>

Padala, Pradeep. [http://www.cise.ufl.edu/~ppadala/ncurses/NCURSES\\_HOWTO/](http://www.cise.ufl.edu/~ppadala/ncurses/NCURSES_HOWTO/)  
[http://www.linuxdoc.org/HOWTO/NCURSES-Programming- HOWTO/tools.html](http://www.linuxdoc.org/HOWTO/NCURSES-Programming-HOWTO/tools.html)

<http://www.cs.vu.nl/pub/minix/2.0.0/wwwman/man3/curses.3.html>

<http://www.vexus.ca/CDK.html>