Data Types

The following data types are declared:

```
    WINDOW*   pointer to screen representation
    SCREEN*   pointer to terminal descriptor
    bool      boolean data type
    chtype    representation of a character in a window
    cchar_t   the wide-character equivalent of chtype
    attr_t    for WA_-style attributes
```

The actual WINDOW and SCREEN objects used to store information are created by the corresponding routines and a pointer to them is provided. All manipulation is through that pointer.

Variables

The following variables are defined:

```
    LINES         number of lines on terminal screen
    COLS          number of columns on terminal screen
    stdscr        pointer to the default screen window
    curscr        pointer to the current screen image
    SP            pointer to the current SCREEN struct
    Mouse_status  status of the mouse
    COLORS        number of colors available
    COLOR_PAIRS   number of color pairs available
    TABSIZE       size of one TAB block
    acs_map[]     alternate character set map
    ttytype[]     terminal name/description
```

Constants

The following constants are defined:

General

```
    FALSE         boolean false value
    TRUE          boolean true value
    NULL          zero pointer value
    ERR           value returned on error condition
    OK            value returned on successful completion
```

Video Attributes

Normally, attributes are a property of the character.

For chtype:

```
    A_ALTCHARSET  use the alternate character set
    A_BLINK       bright background or blinking
    A_BOLD        bright foreground or bold
    A_DIM         half bright -- no effect in PDCurses
    A_INVIS       invisible -- no effect in PDCurses
```

```
    A_ITALIC      italic
    A_LEFT        line along the left edge
    A_PROTECT     protected -- no effect in PDCurses
    A_REVERSE     reverse video
    A_RIGHT       line along the right edge
    A_STANDOUT    terminal's best highlighting mode
    A_UNDERLINE   underline

    A_ATTRIBUTES  bit-mask to extract attributes
    A_CHARTEXT    bit-mask to extract a character
    A_COLOR       bit-mask to extract a color-pair
```

Not all attributes will work on all terminals. A_ITALIC is not standard, but is shared with ncurses.

For attr_t:

```
    WA_ALTCHARSET same as A_ALTCHARSET
    WA_BLINK      same as A_BLINK
    WA_BOLD       same as A_BOLD
    WA_DIM        same as A_DIM
    WA_INVIS      same as A_INVIS
    WA_ITALIC     same as A_ITALIC
    WA_LEFT       same as A_LEFT
    WA_PROTECT    same as A_PROTECT
    WA_REVERSE    same as A_REVERSE
    WA_RIGHT      same as A_RIGHT
    WA_STANDOUT   same as A_STANDOUT
    WA_UNDERLINE  same as A_UNDERLINE
```

The following are also defined, for compatibility, but currently have no effect in PDCurses: A_HORIZONTAL, A_LOW, A_TOP, A_VERTICAL and their WA_* equivalents.

The Alternate Character Set

For use in chtypes and with related functions. These are a portable way to represent graphics characters on different terminals.

VT100-compatible symbols -- box characters:

```
    ACS_ULCORNER  upper left box corner
    ACS_LLCORNER  lower left box corner
    ACS_URCORNER  upper right box corner
    ACS_LRCORNER  lower right box corner
    ACS_RTEE      right "T"
    ACS_LTEE      left "T"
    ACS_BTEE      bottom "T"
    ACS_TTEE      top "T"
    ACS_HLINE     horizontal line
    ACS_VLINE     vertical line
    ACS_PLUS      plus sign, cross, or four-corner piece
```

VT100-compatible symbols -- other:

```
    ACS_S1        scan line 1
    ACS_S9        scan line 9
    ACS_DIAMOND   diamond
```

```
    ACS_CKBOARD   checkerboard -- 50% grey
    ACS_DEGREE    degree symbol
    ACS_PLMINUS   plus/minus sign
    ACS_BULLET    bullet
```

Teletype 5410v1 symbols -- these are defined in SysV curses, but are not well-supported by most terminals. Stick to VT100 characters for optimum portability:

```
    ACS_LARROW    left arrow
    ACS_RARROW    right arrow
    ACS_DARROW    down arrow
    ACS_UARROW    up arrow
    ACS_BOARD     checkerboard -- lighter (less dense) than
                  ACS_CKBOARD
    ACS_LANTERN   lantern symbol
    ACS_BLOCK     solid block
```

That goes double for these -- undocumented SysV symbols. Don't use them:

```
    ACS_S3        scan line 3
    ACS_S7        scan line 7
    ACS_LEQUAL    less than or equal
    ACS_GEQUAL    greater than or equal
    ACS_PI        pi
    ACS_NEQUAL    not equal
    ACS_STERLING  pounds sterling symbol
```

Box character aliases:

```
    ACS_BSSB          same as ACS_ULCORNER
    ACS_SSBB          same as ACS_LLCORNER
    ACS_BBSS          same as ACS_URCORNER
    ACS_SBBS          same as ACS_LRCORNER
    ACS_SBSS          same as ACS_RTEE
    ACS_SSSB          same as ACS_LTEE
    ACS_SSBS          same as ACS_BTEE
    ACS_BSSS          same as ACS_TTEE
    ACS_BSBS          same as ACS_HLINE
    ACS_SBSB          same as ACS_VLINE
    ACS_SSSS          same as ACS_PLUS
```

For cchar_t and wide-character functions, WACS_ equivalents are also defined.

Colors

For use with init_pair(), color_set(), etc.:

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

Use these instead of numeric values. The definition of the colors depends on the implementation of curses.

Input Values

The following constants might be returned by getch() if keypad() has been enabled. Note that not all of these may be supported on a particular terminal:

```
    KEY_BREAK     break key
    KEY_DOWN      the four arrow keys
    KEY_UP
    KEY_LEFT
    KEY_RIGHT
    KEY_HOME      home key (upward+left arrow)
    KEY_BACKSPACE backspace
    KEY_F0        function keys; space for 64 keys is reserved
    KEY_F(n)      (KEY_F0+(n))
    KEY_DL        delete line
    KEY_IL        insert line
    KEY_DC        delete character
    KEY_IC        insert character
    KEY_EIC       exit insert character mode
    KEY_CLEAR     clear screen
    KEY_EOS       clear to end of screen
    KEY_EOL       clear to end of line
    KEY_SF        scroll 1 line forwards
    KEY_SR        scroll 1 line backwards (reverse)
    KEY_NPAGE     next page
    KEY_PPAGE     previous page
    KEY_STAB      set tab
    KEY_CTAB      clear tab
    KEY_CATAB     clear all tabs
    KEY_ENTER     enter or send
    KEY_SRESET    soft (partial) reset
    KEY_RESET     reset or hard reset
    KEY_PRINT     print or copy
    KEY_LL        home down or bottom (lower left)
    KEY_A1        upper left of virtual keypad
    KEY_A3        upper right of virtual keypad
    KEY_B2        center of virtual keypad
    KEY_C1        lower left of virtual keypad
    KEY_C3        lower right of virtual keypad

    KEY_BTAB      Back tab key
    KEY_BEG       Beginning key
    KEY_CANCEL    Cancel key
    KEY_CLOSE     Close key
    KEY_COMMAND   Cmd (command) key
    KEY_COPY      Copy key
    KEY_CREATE    Create key
    KEY_END       End key
    KEY_EXIT      Exit key
    KEY_FIND      Find key
    KEY_HELP      Help key
    KEY_MARK      Mark key
    KEY_MESSAGE   Message key
```

```
    KEY_MOVE      Move key
    KEY_NEXT      Next object key
    KEY_OPEN      Open key
    KEY_OPTIONS   Options key
    KEY_PREVIOUS  Previous object key
    KEY_REDO      Redo key
    KEY_REFERENCE Reference key
    KEY_REFRESH   Refresh key
    KEY_REPLACE   Replace key
    KEY_RESTART   Restart key
    KEY_RESUME    Resume key
    KEY_SAVE      Save key
    KEY_SBEG      Shifted beginning key
    KEY_SCANCEL   Shifted cancel key
    KEY_SCOMMAND  Shifted command key
    KEY_SCOPY     Shifted copy key
    KEY_SCREATE   Shifted create key
    KEY_SDC       Shifted delete char key
    KEY_SDL       Shifted delete line key
    KEY_SELECT    Select key
    KEY_SEND      Shifted end key
    KEY_SEOL      Shifted clear line key
    KEY_SEXIT     Shifted exit key
    KEY_SFIND     Shifted find key
    KEY_SHELP     Shifted help key
    KEY_SHOME     Shifted home key
    KEY_SIC       Shifted input key
    KEY_SLEFT     Shifted left arrow key
    KEY_SMESSAGE  Shifted message key
    KEY_SMOVE     Shifted move key
    KEY_SNEXT     Shifted next key
    KEY_SOPTIONS  Shifted options key
    KEY_SPREVIOUS Shifted prev key
    KEY_SPRINT    Shifted print key
    KEY_SREDO     Shifted redo key
    KEY_SREPLACE  Shifted replace key
    KEY_SRIGHT    Shifted right arrow
    KEY_SRSUME    Shifted resume key
    KEY_SSAVE     Shifted save key
    KEY_SSUSPEND  Shifted suspend key
    KEY_SUNDO     Shifted undo key
    KEY_SUSPEND   Suspend key
    KEY_UNDO      Undo key

The virtual keypad is arranged like this:

    A1     up     A3
    left   B2  right
    C1    down    C3

This list is incomplete -- see curses.h for the full list, and use the
testcurs demo to see what values are actually returned. The above are
just the keys required by X/Open. In particular, PDCurses defines many
CTL_ and ALT_ combinations; these are not portable.
```

NIEISTOTNE
Definitions and Variables (curses.h)
====================================

Define before inclusion (only those needed):

```
    XCURSES         True if compiling for X11.
    PDC_RGB         True if you want to use RGB color definitions
                    (Red = 1, Green = 2, Blue = 4) instead of BGR.
    PDC_WIDE        True if building wide-character support.
    PDC_DLL_BUILD   True if building a Windows DLL.
    PDC_NCMOUSE     Use the ncurses mouse API instead
                    of PDCurses' traditional mouse API.
```

Defined by this header:

```
    PDCURSES        Enables access to PDCurses-only routines.
    PDC_BUILD       Defines API build version.
    PDC_VER_MAJOR   Major version number
    PDC_VER_MINOR   Minor version number
    PDC_VERDOT      Version string
```

Text Attributes
===============

PDCurses uses a 32-bit integer for its chtype:

```
    +-------------------------------------------------------------------+
    |31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|..| 2| 1| 0|
    +-------------------------------------------------------------------+
          color pair        |     modifiers         |   character eg 'a'
```

There are 256 color pairs (8 bits), 8 bits for modifiers, and 16 bits
for character data. The modifiers are bold, underline, right-line,
left-line, italic, reverse and blink, plus the alternate character set
indicator.

Functions
=========

addch

Synopsis

```
    int addch(const chtype ch);
    int waddch(WINDOW* win, const chtype ch);
```

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

int addrawch(chtype ch);
int waddrawch(WINDOW* win, chtype ch);
int mvaddrawch(int y, int x, chtype ch);
int mvwaddrawch(WINDOW* win, int y, int x, chtype ch);

int add_wch(const cchar_t* wch);
int wadd_wch(WINDOW* win, const cchar_t* wch);
int mvadd_wch(int y, int x, const cchar_t* wch);
int mvwadd_wch(WINDOW* win, int y, int x, const cchar_t* wch);
int echo_wchar(const cchar_t* wch);
int wecho_wchar(WINDOW* win, const cchar_t* wch);
```

Description

addch() adds the chtype ch to the default window (stdscr) at the current cursor position, and advances the cursor. Note that chtypes can convey both text (a single character) and attributes, including a color pair. add_wch() is the wide-character version of this function, taking a pointer to a cchar_t instead of a chtype.

waddch() is like addch(), but also lets you specify the window. (This is in fact the core output routine.) wadd_wch() is the wide version.

mvaddch() moves the cursor to the specified (y, x) position, and adds ch to stdscr. mvadd_wch() is the wide version.

mvwaddch() moves the cursor to the specified position and adds ch to the specified window. mvwadd_wch() is the wide version.

echochar() adds ch to stdscr at the current cursor position and calls refresh(). echo_wchar() is the wide version.

wechochar() adds ch to the specified window and calls wrefresh(). wecho_wchar() is the wide version.

addrawch(), waddrawch(), mvaddrawch() and mvwaddrawch() are PDCurses-specific wrappers for addch() etc. that disable the translation of control characters.

The following applies to all these functions:

If the cursor moves on to the right margin, an automatic newline is performed. If scrollok is enabled, and a character is added to the bottom right corner of the window, the scrolling region will be scrolled up one line. If scrolling is not allowed, ERR will be returned.

If ch is a tab, newline, or backspace, the cursor will be moved appropriately within the window. If ch is a newline, the clrtoeol routine is called before the cursor is moved to the beginning of the next line. If newline mapping is off, the cursor will be moved to

the next line, but the x coordinate will be unchanged. If ch is a tab the cursor is moved to the next tab position within the window. If ch is another control character, it will be drawn in the ^X notation. Calling the inch() routine after adding a control character returns the representation of the control character, not the control character.

Video attributes can be combined with a character by ORing them into the parameter. Text, including attributes, can be copied from one place to another by using inch() and addch().

Note that in PDCurses, for now, a cchar_t and a chtype are the same. The text field is 16 bits wide, and is treated as Unicode (UCS-2) when PDCurses is built with wide-character support (define PDC_WIDE). So, in functions that take a chtype, like addch(), both the wide and narrow versions will handle Unicode. But for portability, you should use the wide functions.

Return Value

All functions return OK on success and ERR on error.

addchstr

Synopsis

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

int add_wchstr(const cchar_t* wch);
int add_wchnstr(const cchar_t* wch, int n);
int wadd_wchstr(WINDOW* win, const cchar_t* wch);
int wadd_wchnstr(WINDOW* win, const cchar_t* wch, int n);
int mvadd_wchstr(int y, int x, const cchar_t* wch);
int mvadd_wchnstr(int y, int x, const cchar_t* wch, int n);
int mvwadd_wchstr(WINDOW* win, int y, int x, const cchar_t* wch);
int mvwadd_wchnstr(WINDOW* win, int y, int x, const cchar_t* wch,
                   int n);
```

Description

These routines write a chtype or cchar_t string directly into the window structure, starting at the current or specified position. The four routines with n as the last argument copy at most n elements, but no more than will fit on the line. If n == -1 then the whole string is copied, up to the maximum number that will fit on the line.

The cursor position is not advanced. These routines do not check for

newline or other special characters, nor does any line wrapping occur.

Return Value

All functions return OK or ERR.

addstr

Synopsis

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

int addwstr(const wchar_t *wstr);
int addnwstr(const wchar_t *wstr, int n);
int waddwstr(WINDOW* win, const wchar_t *wstr);
int waddnwstr(WINDOW* win, const wchar_t *wstr, int n);
int mvaddwstr(int y, int x, const wchar_t *wstr);
int mvaddnwstr(int y, int x, const wchar_t *wstr, int n);
int mvwaddwstr(WINDOW* win, int y, int x, const wchar_t *wstr);
int mvwaddnwstr(WINDOW* win, int y, int x, const wchar_t *wstr, int n);
```

Description

These routines write all the characters of the null-terminated string str or wide-character string wstr to the given window. The functionality is similar to calling waddch() once for each character in the string; except that, when PDCurses is built with wide-character support enabled, the narrow-character functions treat the string as a multibyte string in the current locale, and convert it. The routines with n as the last argument write at most n characters; if n is negative, then the entire string will be added.

Return Value

All functions return OK or ERR.

ARTRYBUTY

attr

Synopsis

```
int attroff(chtype attrs);
int wattroff(WINDOW* win, chtype attrs);
int attron(chtype attrs);
int wattron(WINDOW* win, chtype attrs);
```

```
    int attrset(chtype attrs);
    int wattrset(WINDOW* win, chtype attrs);
    int standend(void);
    int wstandend(WINDOW* win);
    int standout(void);
    int wstandout(WINDOW* win);

    int color_set(short color_pair, void* opts);
    int wcolor_set(WINDOW* win, short color_pair, void* opts);

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

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

    chtype getattrs(WINDOW* win);

    int underend(void);
    int wunderend(WINDOW* win);
    int underscore(void);
    int wunderscore(WINDOW* win);
```

Description

These functions manipulate the current attributes and/or colors of the named window. These attributes can be any combination of A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_BLINK, A_UNDERLINE. These constants are defined in <curses.h> and can be combined with the bitwise-OR operator (|).

The current attributes of a window are applied to all chtypes that are written into the window with waddch(). Attributes are a property of the chtype, and move with the character through any scrolling or insert/delete operations.

wattrset() sets the current attributes of the given window to attrs. attrset() is the stdscr version.

wattroff() turns off the named attributes without affecting any other attributes;
wattron() turns them on.

wcolor_set() sets the window color to the value of color_pair. opts is unused.

standout() is the same as attron(A_STANDOUT). standend() is the same as attrset(A_NORMAL); that is, it turns off all attributes.

The attr_* and wattr_* functions are intended for use with the WA_* attributes. In PDCurses, these are the same as A_*, and there is no difference in bevahior from the chtype-based functions. In all cases, opts is unused.

wattr_get() retrieves the attributes and color pair for the specified window.

wchgat() sets the color pair and attributes for the next n cells on the current line of a given window, without changing the existing text, or alterting the window's attributes. An n of -1 extends the change to the edge of the window. The changes take effect immediately. opts is unused.

wunderscore() turns on the A_UNDERLINE attribute; wunderend() turns it off. underscore() and underend() are the stdscr versions.

Return Value

All functions return OK on success and ERR on error.

---

beep

Synopsis

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

Description

beep() sounds the audible bell on the terminal, if possible; if not, it calls flash().

flash() "flashes" the screen, by inverting the foreground and background of every cell, pausing, and then restoring the original attributes.

Return Value

These functions return ERR if called before initscr(), otherwise OK.

Synopsis

```
int bkgd(chtype ch);
```

```
    void bkgdset(chtype ch);
    chtype getbkgd(WINDOW* win);
    int wbkgd(WINDOW* win, chtype ch);
    void wbkgdset(WINDOW* win, chtype ch);

    int bkgrnd(const cchar_t* wch);
    void bkgrndset(const cchar_t* wch);
    int getbkgrnd(cchar_t* wch);
    int wbkgrnd(WINDOW* win, const cchar_t* wch);
    void wbkgrndset(WINDOW* win, const cchar_t* wch);
    int wgetbkgrnd(WINDOW* win, cchar_t* wch);
```

Description

bkgdset() and wbkgdset() manipulate the background of a window. The background is a chtype consisting of any combination of attributes and a character; it is combined with each chtype added or inserted to the window by waddch() or winsch(). Only the attribute part is used to set the background of non-blank characters, while both character and attributes are used for blank positions.

bkgd() and wbkgd() not only change the background, but apply it immediately to every cell in the window.

wbkgrnd(), wbkgrndset() and wgetbkgrnd() are the "wide-character" versions of these functions, taking a pointer to a cchar_t instead of a chtype. However, in PDCurses, cchar_t and chtype are the same.

The attributes that are defined with the attrset()/attron() set of functions take precedence over the background attributes if there is a conflict (e.g., different color pairs).

Return Value

bkgd() and wbkgd() return OK, unless the window is NULL, in which case they return ERR.

---

border

Synopsis

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

```
    int mvvline(int y, int x, chtype ch, int n);
    int mvwhline(WINDOW* win, int y, int x, chtype ch, int n);
    int mvwvline(WINDOW* win, int y, int x, chtype ch, int n);

    int border_set(const cchar_t* ls, const cchar_t* rs,
                   const cchar_t* ts, const cchar_t* bs,
                   const cchar_t* tl, const cchar_t* tr,
                   const cchar_t* bl, const cchar_t* br);
    int wborder_set(WINDOW* win, const cchar_t* ls, const cchar_t* rs,
                                 const cchar_t* ts, const cchar_t* bs,
                                 const cchar_t* tl, const cchar_t* tr,
                                 const cchar_t* bl, const cchar_t* br);
    int box_set(WINDOW* win, const cchar_t* verch, const cchar_t* horch);
    int hline_set(const cchar_t* wch, int n);
    int vline_set(const cchar_t* wch, int n);
    int whline_set(WINDOW* win, const cchar_t* wch, int n);
    int wvline_set(WINDOW* win, const cchar_t* wch, int n);
    int mvhline_set(int y, int x, const cchar_t* wch, int n);
    int mvvline_set(int y, int x, const cchar_t* wch, int n);
    int mvwhline_set(WINDOW* win, int y, int x, const cchar_t* wch, int n);
    int mvwvline_set(WINDOW* win, int y, int x, const cchar_t* wch, int n);
```

Description

border(), wborder(), and box() draw a border around the edge of the window. If any argument is zero, an appropriate default is used:

ls	left side of border	ACS_VLINE
rs	right side of border	ACS_VLINE
ts	top side of border	ACS_HLINE
bs	bottom side of border	ACS_HLINE
tl	top left corner of border	ACS_ULCORNER
tr	top right corner of border	ACS_URCORNER
bl	bottom left corner of border	ACS_LLCORNER
br	bottom right corner of border	ACS_LRCORNER

hline() and whline() draw a horizontal line, using ch, starting from the current cursor position. The cursor position does not change. The line is at most n characters long, or as many as will fit in the window.

vline() and wvline() draw a vertical line, using ch, starting from the current cursor position. The cursor position does not change. The line is at most n characters long, or as many as will fit in the window.

The *_set functions are the "wide-character" versions, taking pointers to cchar_t instead of chtype. Note that in PDCurses, chtype and cchar_t are the same.

Return Value

These functions return OK on success and ERR on error.

---

clear

Synopsis

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

Description

erase() and werase() copy blanks (i.e. the background chtype) to every cell of the window.

clear() and wclear() are similar to erase() and werase(), but they also call clearok() to ensure that the the window is cleared on the next wrefresh().

clrtobot() and wclrtobot() clear the window from the current cursor position to the end of the window.

clrtoeol() and wclrtoeol() clear the window from the current cursor position to the end of the current line.

Return Value

All functions return OK on success and ERR on error.

---

color

Synopsis

```
bool has_colors(void);
int  start_color(void);
int  init_pair(short pair, short fg, short bg);
int  pair_content(short pair, short* fg, short* bg);
bool can_change_color(void);
int  init_color(short color, short red, short green, short blue);
int  color_content(short color, short* red, short* green, short* blue);

int  assume_default_colors(int f, int b);
```

```
int  use_default_colors(void);

int PDC_set_line_color(short color);
```

Description

To use these routines, first, call start_color(). Colors are always used in pairs, referred to as color-pairs. A color-pair is created by init_pair(), and consists of a foreground color and a background color. After initialization, COLOR_PAIR(n) can be used like any other video attribute.

has_colors() reports whether the terminal supports color.

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables: COLORS and COLOR_PAIRS (respectively defining the maximum number of colors and color-pairs the terminal is capable of displaying).

init_pair() changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be redefined, and the new values of the foreground and background colors. The pair number must be between 0 and COLOR_PAIRS - 1, inclusive. The foreground and background must be between 0 and COLORS - 1, inclusive. If the color pair was previously initialized, the screen is refreshed, and all occurrences of that color-pair are changed to the new definition.

pair_content() is used to determine what the colors of a given color-pair consist of.

can_change_color() indicates if the terminal has the capability to change the definition of its colors.

init_color() is used to redefine a color, if possible. Each of the components -- red, green, and blue -- is specified in a range from 0 to 1000, inclusive.

color_content() reports the current definition of a color in the same format as used by init_color().

assume_default_colors() and use_default_colors() emulate the ncurses extensions of the same names. assume_default_colors(f, b) is essentially the same as init_pair(0, f, b) (which isn't allowed); it redefines the default colors. use_default_colors() allows the use of -1 as a foreground or background color with init_pair(), and calls assume_default_colors(-1, -1); -1 represents the foreground or background color that the terminal had at startup. If the environment variable PDC_ORIGINAL_COLORS is set at the time start_color() is called, that's equivalent to calling use_default_colors().

PDC_set_line_color() is used to set the color, globally, for the color of the lines drawn for the attributes: A_UNDERLINE, A_LEFT and A_RIGHT. A value of -1 (the default) indicates that the current foreground color should be used.

NOTE: COLOR_PAIR() and PAIR_NUMBER() are implemented as macros.

Return Value

   All functions return OK on success and ERR on error, except for
   has_colors() and can_change_colors(), which return TRUE or FALSE

debug

Synopsis

```
    void traceon(void);
    void traceoff(void);
    void PDC_debug(const char*  , ...);
```

Description

   traceon() and traceoff() toggle the recording of debugging
   information to the file "trace". Although not standard, similar
   functions are in some other curses implementations.

   PDC_debug() is the function that writes to the file, based on whether
   traceon() has been called. It's used from the PDC_LOG() macro.

   The environment variable PDC_TRACE_FLUSH controls whether the trace
   file contents are fflushed after each write. The default is not. Set
   it to enable this (may affect performance).

---

delch

Synopsis

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

Description

   The character under the cursor in the window is deleted. All
   characters to the right on the same line are moved to the left one
   position and the last character on the line is filled with a blank.
   The cursor position does not change (after moving to y, x if
   coordinates are specified).

Return Value

   All functions return OK on success and ERR on error.
---

deleteln

Synopsis

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

    int mvdeleteln(int y, int x);
    int mvwdeleteln(WINDOW* win, int y, int x);
    int mvinsertln(int y, int x);
    int mvwinsertln(WINDOW* win, int y, int x);
```

Description

   With the deleteln() and wdeleteln() functions, the line under the
   cursor in the window is deleted. All lines below the current line are
   moved up one line. The bottom line of the window is cleared. The
   cursor position does not change.

   With the insertln() and winsertn() functions, a blank line is
   inserted above the current line and the bottom line is lost.

   mvdeleteln(), mvwdeleteln(), mvinsertln() and mvwinsertln() allow
   moving the cursor and inserting/deleting in one call.

Return Value

   All functions return OK on success and ERR on error.

---

getch

Synopsis

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

    int get_wch(wint_t* wch);
```

```
int wget_wch(WINDOW* win, wint_t* wch);
int mvget_wch(int y, int x, wint_t* wch);
int mvwget_wch(WINDOW* win, int y, int x, wint_t* wch);
int unget_wch(const wchar_t wch);

unsigned long PDC_get_key_modifiers(void);
int PDC_return_key_modifiers(bool flag);
```

Description

With the getch(), wgetch(), mvgetch(), and mvwgetch() functions, a character is read from the terminal associated with the window. In nodelay mode, if there is no input waiting, the value ERR is returned. In delay mode, the program will hang until the system passes text through to the program. Depending on the setting of cbreak(), this will be after one character or after the first newline. Unless noecho() has been set, the character will also be echoed into the designated window.

If keypad() is TRUE, and a function key is pressed, the token for that function key will be returned instead of the raw characters. Possible function keys are defined in <curses.h> with integers beginning with 0401, whose names begin with KEY_.

If nodelay(win, TRUE) has been called on the window and no input is waiting, the value ERR is returned.

ungetch() places ch back onto the input queue to be returned by the next call to wgetch().

flushinp() throws away any type-ahead that has been typed by the user and has not yet been read by the program.

wget_wch() is the wide-character version of wgetch(), available when PDCurses is built with the PDC_WIDE option. It takes a pointer to a wint_t rather than returning the key as an int, and instead returns KEY_CODE_YES if the key is a function key. Otherwise, it returns OK or ERR. It's important to check for KEY_CODE_YES, since regular wide characters can have the same values as function key codes.

unget_wch() puts a wide character on the input queue.

PDC_get_key_modifiers() returns the keyboard modifiers (shift, control, alt, numlock) effective at the time of the last getch() call. Use the macros PDC_KEY_MODIFIER_* to determine which modifier(s) were set. PDC_return_key_modifiers() tells getch() to return modifier keys pressed alone as keystrokes (KEY_ALT_L, etc.). These may not work on all platforms.

NOTE: getch() and ungetch() are implemented as macros, to avoid conflict with many DOS compiler's runtime libraries.

Return Value

These functions return ERR or the value of the character, meta character or function key token.

getstr

Synopsis

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

int get_wstr(wint_t* wstr);
int wget_wstr(WINDOW* win, wint_t* wstr);
int mvget_wstr(int y, int x, wint_t* wstr);
int mvwget_wstr(WINDOW* win, int, int, wint_t* wstr);
int getn_wstr(wint_t* wstr, int n);
int wgetn_wstr(WINDOW* win, wint_t* wstr, int n);
int mvgetn_wstr(int y, int x, wint_t* wstr, int n);
int mvwgetn_wstr(WINDOW* win, int y, int x, wint_t* wstr, int n);
```

Description

   These routines call wgetch() repeatedly to build a string, interpreting erase and kill characters along the way, until a newline or carriage return is received. When PDCurses is built with wide-character support enabled, the narrow-character functions convert the wgetch()'d values into a multibyte string in the current locale before returning it. The resulting string is placed in the area pointed to by *str. The routines with n as the last argument read at most n characters.

   Note that there's no way to know how long the buffer passed to wgetstr() is, so use wgetnstr() to avoid buffer overflows.

Return Value

   These functions return ERR on failure or any other value on success.

getyx

Synopsis

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

    void getsyx(int y, int x);
    void setsyx(int y, int x);

    int getbegy(WINDOW* win);
    int getbegx(WINDOW* win);
    int getcury(WINDOW* win);
    int getcurx(WINDOW* win);
    int getpary(WINDOW* win);
    int getparx(WINDOW* win);
    int getmaxy(WINDOW* win);
    int getmaxx(WINDOW* win);
```

Description

The getyx() macro (defined in curses.h -- the prototypes here are merely illustrative) puts the current cursor position of the specified window **into** y and x. getbegyx() and getmaxyx() return the starting coordinates and size of the specified window, respectively. getparyx() returns the starting coordinates of the parent's window, if the specified window is a subwindow; otherwise it sets y and x to -1. These are all macros.

getsyx() gets the coordinates of the virtual screen cursor, and stores them in y and x. If leaveok() is TRUE, it returns -1, -1. If lines have been removed with ripoffline(), then getsyx() includes these lines in its count; so, the returned y and x values should only be used with setsyx().

setsyx() sets the virtual screen cursor to the y, x coordinates. If either y or x is -1, leaveok() is set TRUE, else it's set FALSE.

getsyx() and setsyx() are meant to be used by a library routine that manipulates curses windows without altering the position of the cursor. Note that getsyx() is defined only as a macro.

getbegy(), getbegx(), getcurx(), getcury(), getmaxy(), getmaxx(), getpary(), and getparx() return the appropriate coordinate or size values, or ERR in the case of a NULL window.

inch

Synopsis

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

int in_wch(cchar_t* wcval);
int win_wch(WINDOW* win, cchar_t* wcval);
int mvin_wch(int y, int x, cchar_t* wcval);
int mvwin_wch(WINDOW* win, int y, int x, cchar_t* wcval);
```

Description

The inch() functions retrieve the character and attribute from the current or specified window position, in the form of a chtype. If a NULL window is specified, (chtype)ERR is returned.

The in_wch() functions are the wide-character versions; instead of returning a chtype, they store a cchar_t at the address specified by wcval, and return OK or ERR. (No value is stored when ERR is returned.) Note that in PDCurses, chtype and cchar_t are the same.

---

inchstr

Synopsis

```
int inchstr(cchtype* ch);
int inchnstr(cchtype* ch, int n);
int winchstr(WINDOW* win, cchtype* ch);
int winchnstr(WINDOW* win, cchtype* ch, int n);
int mvinchstr(int y, int x, cchtype* ch);
int mvinchnstr(int y, int x, cchtype* ch, int n);
int mvwinchstr(WINDOW* , int y, int x, cchtype* ch);
int mvwinchnstr(WINDOW* , int y, int x, cchtype* ch, int n);

int in_wchstr(cchar_t* wch);
int in_wchnstr(cchar_t* wch, int n);
int win_wchstr(WINDOW* win, cchar_t* wch);
int win_wchnstr(WINDOW* win, cchar_t* wch, int n);
int mvin_wchstr(int y, int x, cchar_t* wch);
int mvin_wchnstr(int y, int x, cchar_t* wch, int n);
int mvwin_wchstr(WINDOW* win, int y, int x, cchar_t* wch);
int mvwin_wchnstr(WINDOW* win, int y, int x, cchar_t* wch, int n);
```

Description

   These routines read a chtype or cchar_t string from the window,
   starting at the current or specified position, and ending at the
   right margin, or after n elements, whichever is less.

Return Value

   All functions return the number of elements read, or ERR on error.

---

initscr

Synopsis

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

    int resize_term(int nlines, int ncols);
    bool is_termresized(void);
    const char*  curses_version(void);
    void PDC_get_version(PDC_VERSION* ver);

    int set_tabsize(int tabsize);
```

Description

   initscr() should be the first curses routine called. It will
   initialize all curses data structures, and arrange that the first
   call to refresh() will clear the screen. In case of error, initscr()
   will write a message to standard error and end the program.

   endwin() should be called before exiting or escaping from curses mode
   temporarily. It will restore tty modes, move the cursor to the lower
   left corner of the screen and reset the terminal into the proper
   non-visual mode. To resume curses after a temporary escape, call
   refresh() or doupdate().

   isendwin() returns TRUE if endwin() has been called without a
   subsequent refresh, unless SP is NULL.

   In some implementations of curses, newterm() allows the use of
   multiple terminals. Here, it's just an alternative interface for
   initscr(). It always returns SP, or NULL.

   delscreen() frees the memory allocated by newterm() or initscr(),
   since it's not freed by endwin(). This function is usually not
   needed. In PDCurses, the parameter must be the value of SP, and

delscreen() sets SP to NULL.

set_term() does nothing meaningful in PDCurses, but is included for compatibility with other curses implementations.

resize_term() is effectively two functions: When called with nonzero values for nlines and ncols, it attempts to resize the screen to the given size. When called with (0, 0), it merely adjusts the **int**ernal structures to match the current size after the screen is resized by the user. On the currently supported platforms, SDL, Windows console, and X11 allow user resizing, while DOS, OS/2, SDL and Windows console allow programmatic resizing. If you want to support user resizing, you should check for getch() returning KEY_RESIZE, and/or call is_termresized() at appropriate times; if either condition occurs, call resize_term(0, 0). Then, with either user or programmatic resizing, you'll have to resize any windows you've created, as appropriate; resize_term() only handles stdscr and curscr.

is_termresized() returns TRUE if the curses screen has been resized by the user, and a call to resize_term() is needed. Checking for KEY_RESIZE is generally preferable, unless you're not handling the keyboard.

curses_version() returns a string describing the version of PDCurses.

PDC_get_version() fills a PDC_VERSION structure provided by the user with more detailed version info (see curses.h).

set_tabsize() sets the tab **int**erval, stored in TABSIZE.

Return Value

All functions return NULL on error, except endwin(), which always returns OK, and resize_term(), which returns either OK or ERR.

---

inopts

Synopsis

```
int cbreak(void);
int nocbreak(void);
int echo(void);
int noecho(void);
int halfdelay(int tenths);
int intrflush(WINDOW* win, bool bf);
int keypad(WINDOW* win, bool bf);
int meta(WINDOW* win, bool bf);
int nl(void);
int nonl(void);
int nodelay(WINDOW* win, bool bf);
int notimeout(WINDOW* win, bool bf);
int raw(void);
int noraw(void);
```

```
    void noqiflush(void);
    void noqiflush (void);
    void timeout(int delay);
    void wtimeout(WINDOW* win, int delay);
    int typeahead(int fildes);

    int crmode(void);
    int nocrmode(void);

    bool is_keypad(const WINDOW* win);
```

Description

cbreak() and nocbreak() toggle cbreak mode. In cbreak mode, characters typed by the user are made available immediately, and erase/kill character processing is not performed. In nocbreak mode, typed characters are buffered until a newline or carriage return. Interrupt and flow control characters are unaffected by this mode. PDCurses always starts in cbreak mode.

echo() and noecho() control whether typed characters are echoed by the input routine. Initially, input characters are echoed. Subsequent calls to echo() and noecho() do not flush type-ahead.

halfdelay() is similar to cbreak(), but allows for a time limit to be specified, in tenths of a second. This causes getch() to block for that period before returning ERR if no key has been received. tenths must be between 1 and 255.

keypad() controls whether getch() returns function/special keys as single key codes (e.g., the left arrow key as KEY_LEFT). Per X/Open, the default for keypad mode is OFF. You'll probably want it on. With keypad mode off, if a special key is pressed, getch() does nothing or returns ERR.

nodelay() controls whether wgetch() is a non-blocking call. If the option is enabled, and no input is ready, wgetch() will return ERR. If disabled, wgetch() will hang until input is ready.

nl() enables the translation of a carriage return into a newline on input. nonl() disables this. Initially, the translation does occur.

raw() and noraw() toggle raw mode. Raw mode is similar to cbreak mode, in that characters typed are immediately passed through to the user program. The difference is that in raw mode, the INTR, QUIT, SUSP, and STOP characters are passed through without being interpreted, and without generating a signal.

In PDCurses, the meta() function sets raw mode on or off.

timeout() and wtimeout() set blocking or non-blocking reads for the specified window. If the delay is negative, a blocking read is used; if zero, then non-blocking reads are done -- if no input is waiting, ERR is returned immediately. If the delay is positive, the read blocks for the delay period; if the period expires, ERR is returned. The delay is given in milliseconds, but this is rounded down to 50ms

(1/20th sec) intervals, with a minimum of one interval if a postive delay is given; i.e., 1-99 will wait 50ms, 100-149 will wait 100ms, etc.

intrflush(), notimeout(), noqiflush(), qiflush() and typeahead() do nothing in PDCurses, but are included for compatibility with other curses implementations.

crmode() and nocrmode() are archaic equivalents to cbreak() and nocbreak(), respectively.

is_keypad() reports whether the specified window is in keypad mode.

Return Value

All functions except is_keypad() and the void functions return OK on success and ERR on error.

---

insch

Synopsis

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

int insrawch(chtype ch);
int winsrawch(WINDOW* win, chtype ch);
int mvinsrawch(int y, int x, chtype ch);
int mvwinsrawch(WINDOW* win, int y, int x, chtype ch);

int ins_wch(const cchar_t* wch);
int wins_wch(WINDOW* win, const cchar_t* wch);
int mvins_wch(int y, int x, const cchar_t* wch);
int mvwins_wch(WINDOW* win, int y, int x, const cchar_t* wch);
```

Description

The insch() functions insert a chtype into the window at the current or specified cursor position. The cursor is NOT advanced. A newline is equivalent to clrtoeol(); tabs are expanded; other control characters are converted as with unctrl().

The ins_wch() functions are the wide-character equivalents, taking cchar_t pointers rather than chtypes.

Video attributes can be combined with a character by ORing them into the parameter. Text, including attributes, can be copied from one place to another using inch() and insch().

insrawch() etc. are PDCurses-specific wrappers for insch() etc. that

disable the translation of control characters.

Return Value

All functions return OK on success and ERR on error.

---

insstr

Synopsis

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

int ins_wstr(const wchar_t* wstr);
int ins_nwstr(const wchar_t* wstr, int n);
int wins_wstr(WINDOW* win, const wchar_t* wstr);
int wins_nwstr(WINDOW* win, const wchar_t*wstr, int n);
int mvins_wstr(int y, int x, const wchar_t* wstr);
int mvins_nwstr(int y, int x, const wchar_t* wstr, int n);
int mvwins_wstr(WINDOW* win, int y, int x, const wchar_t* wstr);
int mvwins_nwstr(WINDOW* win, int y, int x, const wchar_t* wstr, int n);
```

Description

The insstr() functions insert a character string into a window at the current cursor position, by repeatedly calling winsch(). When PDCurses is built with wide-character support enabled, the narrow-character functions treat the string as a multibyte string in the current locale, and convert it first. All characters to the right of the cursor are moved to the right, with the possibility of the rightmost characters on the line being lost. The cursor position does not change (after moving to y, x, if specified). The routines with n as the last argument insert at most n characters; if n is negative, then the entire string is inserted.

Return Value

All functions return OK on success and ERR on error.

instr

Synopsis

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

    int inwstr(wchar_t *wstr);
    int innwstr(wchar_t *wstr, int n);
    int winwstr(WINDOW* win, wchar_t *wstr);
    int winnwstr(WINDOW* win, wchar_t *wstr, int n);
    int mvinwstr(int y, int x, wchar_t *wstr);
    int mvinnwstr(int y, int x, wchar_t *wstr, int n);
    int mvwinwstr(WINDOW* win, int y, int x, wchar_t *wstr);
    int mvwinnwstr(WINDOW* win, int y, int x, wchar_t *wstr, int n);
```

Description

   These functions take characters (or wide characters) from the current
   or specified position in the window, and return them as a string in
   str (or wstr). Attributes are ignored. The functions with n as the
   last argument return a string at most n characters long.

Return Value

   Upon successful completion, innstr(), mvinnstr(), mvwinnstr() and
   winnstr() return the number of characters actually read into the
   string; instr(), mvinstr(), mvwinstr() and winstr() return OK.
   Otherwise, all these functions return ERR.

kernel

Synopsis

```
    int def_prog_mode(void);
    int def_shell_mode(void);
    int reset_prog_mode(void);
    int reset_shell_mode(void);
    int resetty(void);
    int savetty(void);
    int ripoffline(int line, int (*init)(WINDOW* , int));
    int curs_set(int visibility);
    int napms(int ms);

    int draino(int ms);
    int resetterm(void);
    int fixterm(void);
```

```
    int saveterm(void);
```

Description

def_prog_mode() and def_shell_mode() save the current terminal modes as the "program" (in curses) or "shell" (not in curses) state for use by the reset_prog_mode() and reset_shell_mode() functions. This is done automatically by initscr().

reset_prog_mode() and reset_shell_mode() restore the terminal to "program" (in curses) or "shell" (not in curses) state. These are done automatically by endwin() and doupdate() after an endwin(), so they would normally not be called before these functions.

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

curs_set() alters the appearance of the cursor. A visibility of 0 makes it disappear; 1 makes it appear "normal" (usually an underline) and 2 makes it "highly visible" (usually a block).

ripoffline() reduces the size of stdscr by one line. If the "line" parameter is positive, the line is removed from the top of the screen; if negative, from the bottom. Up to 5 lines can be ripped off stdscr by calling ripoffline() repeatedly. The function argument, init, is called from within initscr() or newterm(), so ripoffline() must be called before either of these functions. The init function receives a pointer to a one-line WINDOW, and the width of the window. Calling ripoffline() with a NULL init function pointer is an error.

napms() suspends the program for the specified number of milliseconds. draino() is an archaic equivalent. Note that since napms() attempts to give up a time slice and yield control back to the OS, all times are approximate. (In DOS, the delay is actually rounded down to 50ms (1/20th sec) **int**ervals, with a minimum of one **int**erval; i.e., 1-99 will wait 50ms, 100-149 will wait 100ms, etc.) 0 returns immediately.

resetterm(), fixterm() and saveterm() are archaic equivalents for reset_shell_mode(), reset_prog_mode() and def_prog_mode(), respectively.

Return Value

All functions return OK on success and ERR on error, except curs_set(), which returns the previous visibility.

keyname

Synopsis

```
char*  keyname(int key);

char*  key_name(wchar_t c);

bool has_key(int key);
```

Description

keyname() returns a string corresponding to the argument key. key may be any key returned by wgetch().

key_name() is the wide-character version. It takes a wchar_t parameter, but still returns a char* .

has_key() returns TRUE for recognized keys, FALSE otherwise. This function is an ncurses extension.

mouse

Synopsis

```
int mouse_set(mmask_t mbe);
int mouse_on(mmask_t mbe);
int mouse_off(mmask_t mbe);
int request_mouse_pos(void);
void wmouse_position(WINDOW* win, int *y, int *x);
mmask_t getmouse(void);

int mouseinterval(int wait);
bool wenclose(const WINDOW* win, int y, int x);
bool wmouse_trafo(const WINDOW* win, int *y, int *x, bool to_screen);
bool mouse_trafo(int *y, int *x, bool to_screen);
mmask_t mousemask(mmask_t mask, mmask_t* oldmask);
int nc_getmouse(MEVENT* event);
int ungetmouse(MEVENT* event);
bool has_mouse(void);
```

Description

As of PDCurses 3.0, there are two separate mouse interfaces: the classic interface, which is based on the undocumented Sys V mouse functions; and an ncurses-compatible interface. Both are active at all times, and you can mix and match functions from each, though it's not recommended. The ncurses interface is essentially an emulation layer built on top of the classic interface; it's here to allow

easier porting of ncurses apps.

The classic interface: mouse_set(), mouse_on(), mouse_off(), request_mouse_pos(), wmouse_position(), and getmouse(). An application using this interface would start by calling mouse_set() or mouse_on() with a non-zero value, often ALL_MOUSE_EVENTS. Then it would check for a KEY_MOUSE return from getch(). If found, it would call request_mouse_pos() to get the current mouse status.

mouse_set(), mouse_on() and mouse_off() are analagous to attrset(), attron() and attroff(). These functions set the mouse button events to trap. The button masks used in these functions are defined in curses.h and can be or'ed together. They are the group of masks starting with BUTTON1_RELEASED.

request_mouse_pos() requests curses to fill in the Mouse_status structure with the current state of the mouse.

wmouse_position() determines if the current mouse position is within the window passed as an argument. If the mouse is outside the current window, -1 is returned in the y and x arguments; otherwise the y and x coordinates of the mouse (relative to the top left corner of the window) are returned in y and x.

getmouse() returns the current status of the trapped mouse buttons as set by mouse_set() or mouse_on().

The ncurses interface: mouseinterval(), wenclose(), wmouse_trafo(), mouse_trafo(), mousemask(), nc_getmouse(), ungetmouse() and has_mouse(). A typical application using this interface would start by calling mousemask() with a non-zero value, often ALL_MOUSE_EVENTS. Then it would check for a KEY_MOUSE return from getch(). If found, it would call nc_getmouse() to get the current mouse status.

mouseinterval() sets the timeout for a mouse click. On all current platforms, PDCurses receives mouse button press and release events, but must synthesize click events. It does this by checking whether a release event is queued up after a press event. If it gets a press event, and there are no more events waiting, it will wait for the timeout interval, then check again for a release. A press followed by a release is reported as BUTTON_CLICKED; otherwise it's passed through as BUTTON_PRESSED. The default timeout is 150ms; valid values are 0 (no clicks reported) through 1000ms. In x11, the timeout can also be set via the clickPeriod resource. The return value from mouseinterval() is the old timeout. To check the old value without setting a new one, call it with a parameter of -1. Note that although there's no classic equivalent for this function (apart from the clickPeriod resource), the value set applies in both interfaces.

wenclose() reports whether the given screen-relative y, x coordinates fall within the given window.

wmouse_trafo() converts between screen-relative and window-relative coordinates. A to_screen parameter of TRUE means to convert from window to screen; otherwise the reverse. The function returns FALSE if the coordinates aren't within the window, or if any of the

parameters are NULL. The coordinates have been converted when the function returns TRUE.

mouse_trafo() is the stdscr version of wmouse_trafo().

mousemask() is nearly equivalent to mouse_set(), but instead of OK/ERR, it returns the value of the mask after setting it. (This isn't necessarily the same value passed in, since the mask could be altered on some platforms.) And if the second parameter is a non-null pointer, mousemask() stores the previous mask value there. Also, since the ncurses interface doesn't work with PDCurses' BUTTON_MOVED events, mousemask() filters them out.

nc_getmouse() returns the current mouse status in an MEVENT struct. This is equivalent to ncurses' getmouse(), renamed to avoid conflict with PDCurses' getmouse(). But if you define PDC_NCMOUSE before including curses.h, it defines getmouse() to nc_getmouse(), along with a few other redefintions needed for compatibility with ncurses code. nc_getmouse() calls request_mouse_pos(), which (not getmouse()) is the classic equivalent.

ungetmouse() is the mouse equivalent of ungetch(). However, PDCurses doesn't maintain a queue of mouse events; only one can be pushed back, and it can overwrite or be overwritten by real mouse events.

has_mouse() reports whether the mouse is available at all on the current platform.

---

move

Synopsis

```
int move(int y, int x);
int mvcur(int oldrow, int oldcol, int newrow, int newcol);
int wmove(WINDOW* win, int y, int x);
```

Description

move() and wmove() move the cursor associated with the window to the given location. This does not move the physical cursor of the terminal until refresh() is called. The position specified is relative to the upper left corner of the window, which is (0,0).

mvcur() moves the physical cursor without updating any window cursor positions.

Return Value

All functions return OK on success and ERR on error.

---

outopts

Synopsis

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

    int raw_output(bool bf);

    bool is_leaveok(const WINDOW* win);
```

Description

   With clearok(), if bf is TRUE, the next call to wrefresh() with this
   window will clear the screen completely and redraw the entire screen.

   immedok(), called with a second argument of TRUE, causes an automatic
   wrefresh() every time a change is made to the specified window.

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

   wsetscrreg() sets a scrolling region in a window; "top" and "bot" are
   the line numbers for the top and bottom margins. If this option and
   scrollok() are enabled, any attempt to move off the bottom margin
   will cause all lines in the scrolling region to scroll up one line.
   setscrreg() is the stdscr version.

   idlok() and idcok() do nothing in PDCurses, but are provided for
   compatibility with other curses implementations.

   raw_output() enables the output of raw characters using the standard
   add and *ins* curses functions (that is, it disables translation of
   control characters).

   is_leaveok() reports whether the specified window is in leaveok mode.

Return Value

   All functions except is_leaveok() return OK on success and ERR on
   error.

---

overlay

Synopsis

```
    int overlay(const WINDOW* src_w, WINDOW* dst_w)
    int overwrite(const WINDOW* src_w, WINDOW* dst_w)
    int copywin(const WINDOW* src_w, WINDOW* dst_w, int src_tr,
                int src_tc, int dst_tr, int dst_tc, int dst_br,
                int dst_bc, int _overlay)
```

Description

   overlay() and overwrite() copy all the text from src_w into dst_w.
   The windows need not be the same size. Those characters in the source
   window that intersect with the destination window are copied, so that
   the characters appear in the same physical position on the screen.
   The difference between the two functions is that overlay() is non-
   destructive (blanks are not copied) while overwrite() is destructive
   (blanks are copied).

   copywin() is similar, but doesn't require that the two windows
   overlap. The arguments src_tc and src_tr specify the top left corner
   of the region to be copied. dst_tc, dst_tr, dst_br, and dst_bc
   specify the region within the destination window to copy to. The
   argument "overlay", if TRUE, indicates that the copy is done non-
   destructively (as in overlay()); blanks in the source window are not
   copied to the destination window. When overlay is FALSE, blanks are
   copied.

Return Value

   All functions return OK on success and ERR on error.

---

pad

Synopsis

```
    WINDOW* newpad(int nlines, int ncols);
    WINDOW* subpad(WINDOW* orig, int nlines, int ncols,
                   int begy, int begx);
    int prefresh(WINDOW* win, int py, int px, int sy1, int sx1,
                 int sy2, int sx2);
    int pnoutrefresh(WINDOW* w, int py, int px, int sy1, int sx1,
                     int sy2, int sx2);
    int pechochar(WINDOW* pad, chtype ch);
    int pecho_wchar(WINDOW* pad, const cchar_t* wch);

    bool is_pad(const WINDOW* pad);
```

Description

A pad is a special kind of window, which is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. You can use a pad when you need a large window, and only a part of the window will be on the screen at one time. Pads are not refreshed automatically (e.g., from scrolling or echoing of input). You can't call wrefresh() with a pad as an argument; use prefresh() or pnoutrefresh() instead. Note that these routines require additional parameters to specify the part of the pad to be displayed, and the location to use on the screen.

newpad() creates a new pad data structure.

subpad() creates a new sub-pad within a pad, at position (begy, begx), with dimensions of nlines lines and ncols columns. This position is relative to the pad, and not to the screen as with subwin. Changes to either the parent pad or sub-pad will affect both. When using sub-pads, you may need to call touchwin() before calling prefresh().

pnoutrefresh() copies the specified pad to the virtual screen.

prefresh() calls pnoutrefresh(), followed by doupdate().

These routines are analogous to wnoutrefresh() and wrefresh(). (py, px) specifies the upper left corner of the part of the pad to be displayed; (sy1, sx1) and (sy2, sx2) describe the screen rectangle that will contain the selected part of the pad.

pechochar() is functionally equivalent to addch() followed by a call to prefresh(), with the last-used coordinates and dimensions. pecho_wchar() is the wide-character version.

is_pad() reports whether the specified window is a pad.

Return Value

All functions except is_pad() return OK on success and ERR on error.

---

panel

Synopsis

```
int bottom_panel(PANEL* pan);
int del_panel(PANEL* pan);
int hide_panel(PANEL* pan);
int move_panel(PANEL* pan, int starty, int startx);
PANEL* new_panel(WINDOW* win);
PANEL* panel_above(const PANEL* pan);
PANEL* panel_below(const PANEL* pan);
int panel_hidden(const PANEL* pan);
```

```
 const void* panel_userptr(const PANEL* pan);
 WINDOW* panel_window(const PANEL* pan);
 int replace_panel(PANEL* pan, WINDOW* win);
 int set_panel_userptr(PANEL* pan, const void* uptr);
 int show_panel(PANEL* pan);
 int top_panel(PANEL* pan);
 void update_panels(void);
```

Description

For historic reasons, and for compatibility with other versions of curses, the panel functions are prototyped in a separate header, panel.h. In many implementations, they're also in a separate library, but PDCurses incorporates them.

The panel functions provide a way to have depth relationships between curses windows. Panels can overlap without making visible the overlapped portions of underlying windows. The initial curses window, stdscr, lies beneath all panels. The set of currently visible panels is the 'deck' of panels.

You can create panels, fetch and set their associated windows, shuffle panels in the deck, and manipulate them in other ways.

bottom_panel() places pan at the bottom of the deck. The size, location and contents of the panel are unchanged.

del_panel() deletes pan, but not its associated winwow.

hide_panel() removes a panel from the deck and thus hides it from view.

move_panel() moves the curses window associated with pan, so that its upper lefthand corner is at the supplied coordinates. (Don't use mvwin() on the window.)

new_panel() creates a new panel associated with win and returns the panel pointer. The new panel is placed at the top of the deck.

panel_above() returns a pointer to the panel in the deck above pan, or NULL if pan is the top panel. If the value of pan passed is NULL, this function returns a pointer to the bottom panel in the deck.

panel_below() returns a pointer to the panel in the deck below pan, or NULL if pan is the bottom panel. If the value of pan passed is NULL, this function returns a pointer to the top panel in the deck.

panel_hidden() returns OK if pan is hidden and ERR if it is not.

panel_userptr() - Each panel has a user pointer available for maintaining relevant information. This function returns a pointer to that information previously set up by set_panel_userptr().

panel_window() returns a pointer to the curses window associated with the panel.

replace_panel() replaces the current window of pan with win.

set_panel_userptr() - Each panel has a user pointer available for maintaining relevant information. This function sets the value of that information.

show_panel() makes a previously hidden panel visible and places it back in the deck on top.

top_panel() places pan on the top of the deck. The size, location and contents of the panel are unchanged.

update_panels() refreshes the virtual screen to reflect the depth relationships between the panels in the deck. The user must use doupdate() to refresh the physical screen.

Return Value

Each routine that returns a pointer to an object returns NULL if an error occurs. Each panel routine that returns an integer, returns OK if it executes successfully and ERR if it does not.

---

printw

Synopsis

```
int printw(const char*  fmt, ...);
int wprintw(WINDOW* win, const char*  fmt, ...);
int mvprintw(int y, int x, const char*  fmt, ...);
int mvwprintw(WINDOW* win, int y, int x, const char*  fmt,...);
int vwprintw(WINDOW* win, const char*  fmt, va_list varglist);
int vw_printw(WINDOW* win, const char*  fmt, va_list varglist);
```

Description

The printw() functions add a formatted string to the window at the current or specified cursor position. The format strings are the same as used in the standard C library's printf(). (printw() can be used as a drop-in replacement for printf().)

The duplication between vwprintw() and vw_printw() is for historic reasons. In PDCurses, they're the same.

Return Value

All functions return the number of characters printed, or ERR on error.

refresh

Synopsis

```
    int refresh(void);
    int wrefresh(WINDOW* win);
    int wnoutrefresh(WINDOW* win);
    int doupdate(void);
    int redrawwin(WINDOW* win);
    int wredrawln(WINDOW* win, int beg_line, int num_lines);
```

Description

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

   wnoutrefresh() and doupdate() allow multiple updates with more efficiency than wrefresh() alone. wrefresh() works by first calling wnoutrefresh(), which copies the named window to the virtual screen. It then calls doupdate(), which compares the virtual screen to the physical screen and does the actual update. A series of calls to wrefresh() will result in alternating calls to wnoutrefresh() and doupdate(), causing several bursts of output to the screen. By first calling wnoutrefresh() for each window, it is then possible to call doupdate() only once.

   In PDCurses, redrawwin() is equivalent to touchwin(), and wredrawln() is the same as touchline(). In some other curses implementations, there's a subtle distinction, but it has no meaning in PDCurses.

Return Value

   All functions return OK on success and ERR on error.

---

scanw

Synopsis

```
    int scanw(const char*  fmt, ...);
    int wscanw(WINDOW* win, const char*  fmt, ...);
    int mvscanw(int y, int x, const char*  fmt, ...);
    int mvwscanw(WINDOW* win, int y, int x, const char*  fmt, ...);
    int vwscanw(WINDOW* win, const char*  fmt, va_list varglist);
    int vw_scanw(WINDOW* win, const char*  fmt, va_list varglist);
```

Description

These routines correspond to the standard C library's scanf() family. Each gets a string from the window via wgetnstr(), and uses the resulting line as input for the scan.

The duplication between vwscanw() and vw_scanw() is for historic reasons. In PDCurses, they're the same.

Return Value

On successful completion, these functions return the number of items successfully matched. Otherwise they return ERR.

Zapisywanie do pliku
scr_dump

Synopsis

```
    int putwin(WINDOW* win, FILE* filep);
    WINDOW* getwin(FILE* filep);
    int scr_dump(const char*  filename);
    int scr_init(const char*  filename);
    int scr_restore(const char*  filename);
    int scr_set(const char*  filename);
```

Description

   getwin() reads window-related data previously stored in a file by
   putwin(). It then creates and initialises a new window using that
   data.

   putwin() writes all data associated with a window **int**o a file, using
   an unspecified format. This information can be retrieved later using
   getwin().

   scr_dump() writes the current contents of the virtual screen to the
   file named by filename in an unspecified format.

   scr_restore() function sets the virtual screen to the contents of the
   file named by filename, which must have been written using
   scr_dump(). The next refresh operation restores the screen to the way
   it looked in the dump file.

   In PDCurses, scr_init() does nothing, and scr_set() is a synonym for
   scr_restore(). Also, scr_dump() and scr_restore() save and load from
   curscr. This differs from some other implementations, where
   scr_init() works with curscr, and scr_restore() works with newscr;
   but the effect should be the same. (PDCurses has no newscr.)

Return Value

   On successful completion, getwin() returns a pointer to the window it
   created. Otherwise, it returns a null pointer. Other functions return
   OK or ERR.

scroll

Synopsis

```
int scroll(WINDOW* win);
int scrl(int n);
int wscrl(WINDOW* win, int n);
```

Description

scroll() causes the window to scroll up one line. This involves moving the lines in the window data strcture.

With a positive n, scrl() and wscrl() scroll the window up n lines (line i + n becomes i); otherwise they scroll the window down n lines.

For these functions to work, scrolling must be enabled via scrollok(). Note also that scrolling is not allowed if the supplied window is a pad.

Return Value

All functions return OK on success and ERR on error.

---

slk

Synopsis

```
int slk_init(int fmt);
int slk_set(int labnum, const char*  label, int justify);
int slk_refresh(void);
int slk_noutrefresh(void);
char*  slk_label(int labnum);
int slk_clear(void);
int slk_restore(void);
int slk_touch(void);
int slk_attron(const chtype attrs);
int slk_attr_on(const attr_t attrs, void* opts);
int slk_attrset(const chtype attrs);
int slk_attr_set(const attr_t attrs, short color_pair, void* opts);
int slk_attroff(const chtype attrs);
int slk_attr_off(const attr_t attrs, void* opts);
int slk_color(short color_pair);

int slk_wset(int labnum, const wchar_t *label, int justify);

int PDC_mouse_in_slk(int y, int x);
void PDC_slk_free(void);
void PDC_slk_initialize(void);
```

```
    wchar_t *slk_wlabel(int labnum)
```

Description

   These functions manipulate a window that contain Soft Label Keys
   (SLK). To use the SLK functions, a call to slk_init() must be made
   BEFORE initscr() or newterm(). slk_init() removes 1 or 2 lines from
   the useable screen, depending on the format selected.

   The line(s) removed from the screen are used as a separate window, in
   which SLKs are displayed.

   slk_init() requires a single parameter which describes the format of
   the SLKs as follows:

   0   3-2-3 format
   1   4-4 format
   2   4-4-4 format (ncurses extension)
   3   4-4-4 format with index line (ncurses extension)
   2 lines used
   55   5-5 format (pdcurses format)

   slk_refresh(), slk_noutrefresh() and slk_touch() are analogous to
   refresh(), noutrefresh() and touch().

Return Value

   All functions return OK on success and ERR on error.

---

termattr

Synopsis

```
    int baudrate(void);
    char erasechar(void);
    bool has_ic(void);
    bool has_il(void);
    char killchar(void);
    char*  longname(void);
    chtype termattrs(void);
    attr_t term_attrs(void);
    char*  termname(void);

    int erasewchar(wchar_t* ch);
    int killwchar(wchar_t* ch);

    char wordchar(void);
```

Description

   baudrate() is supposed to return the output speed of the terminal. In
   PDCurses, it simply returns INT_MAX.

has_ic and has_il() return TRUE. These functions have meaning in some other implementations of curses.

erasechar() and killchar() return ^H and ^U, respectively -- the ERASE and KILL characters. In other curses implementations, these may vary by terminal type. erasewchar() and killwchar() are the wide-character versions; they take a pointer to a location in which to store the character, and return OK or ERR.

longname() returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of the string is 128 characters. It is defined only after the call to initscr() or newterm().

termname() returns a pointer to a static area containing a short description of the current terminal (14 characters).

termattrs() returns a logical OR of all video attributes supported by the terminal.

wordchar() is a PDCurses extension of the concept behind the functions erasechar() and killchar(), returning the "delete word" character, ^W.

---

touch

Synopsis

```
int touchwin(WINDOW* win);
int touchline(WINDOW* win, int start, int count);
int untouchwin(WINDOW* win);
int wtouchln(WINDOW* win, int y, int n, int changed);
bool is_linetouched(WINDOW* win, int line);
bool is_wintouched(WINDOW* win);

int touchoverlap(const WINDOW* win1, WINDOW* win2);
```

Description

touchwin() and touchline() throw away all information about which parts of the window have been touched, pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change.

untouchwin() marks all lines in the window as unchanged since the last call to wrefresh().

wtouchln() makes n lines in the window, starting at line y, look as if they have (changed == 1) or have not (changed == 0) been changed since the last call to wrefresh().

is_linetouched() returns TRUE if the specified line in the specified

window has been changed since the last call to wrefresh().

is_wintouched() returns TRUE if the specified window has been changed since the last call to wrefresh().

touchoverlap(win1, win2) marks the portion of win2 which overlaps with win1 as modified.

Return Value

All functions return OK on success and ERR on error except is_wintouched() and is_linetouched().

util

Synopsis

```
char*  unctrl(chtype c);
void filter(void);
void use_env(bool x);
int delay_output(int ms);

int getcchar(const cchar_t* wcval, wchar_t *wch, attr_t* attrs,
             short* color_pair, void* opts);
int setcchar(cchar_t* wcval, const wchar_t *wch, const attr_t attrs,
             short color_pair, const void* opts);
wchar_t *wunctrl(cchar_t* wc);

int PDC_mbtowc(wchar_t *pwc, const char*  s, size_t n);
size_t PDC_mbstowcs(wchar_t *dest, const char*  src, size_t n);
size_t PDC_wcstombs(char*  dest, const wchar_t *src, size_t n);
```

Description

unctrl() expands the text portion of the chtype c into a printable string. Control characters are changed to the "^X" notation; others are passed through. wunctrl() is the wide-character version of the function.

filter() and use_env() are no-ops in PDCurses.

delay_output() inserts an ms millisecond pause in output.

getcchar() works in two modes: When wch is not NULL, it reads the cchar_t pointed to by wcval and stores the attributes in attrs, the color pair in color_pair, and the text in the wide-character string wch. When wch is NULL, getcchar() merely returns the number of wide characters in wcval. In either mode, the opts argument is unused.

setcchar constructs a cchar_t at wcval from the wide-character text at wch, the attributes in attr and the color pair in color_pair. The opts argument is unused.

Currently, the length returned by getcchar() is always 1 or 0. Similarly, setcchar() will only take the first wide character from wch, and ignore any others that it "should" take (i.e., combining characters). Nor will it correctly handle any character outside the basic multilingual plane (UCS-2).

Return Value

wunctrl() returns NULL on failure. delay_output() always returns OK.

getcchar() returns the number of wide characters wcval points to when wch is NULL; when it's not, getcchar() returns OK or ERR.

setcchar() returns OK or ERR.

WINDOW

Składnia

```
WINDOW* newwin(int nlines, int ncols, int begy, int begx);
WINDOW* derwin(WINDOW* orig, int nlines, int ncols,
               int begy, int begx);
WINDOW* subwin(WINDOW* orig, int nlines, int ncols,
               int begy, int begx);
WINDOW* dupwin(WINDOW* win);
int     delwin(WINDOW* win);
int     mvwin(WINDOW* win, int y, int x);
int     mvderwin(WINDOW* win, int pary, int parx);
int     syncok(WINDOW* win, bool bf);
void    wsyncup(WINDOW* win);
void    wcursyncup(WINDOW* win);
void    wsyncdown(WINDOW* win);

WINDOW* resize_window(WINDOW* win, int nlines, int ncols);
int     wresize(WINDOW* win, int nlines, int ncols);
WINDOW* PDC_makelines(WINDOW* win);
WINDOW* PDC_makenew(int nlines, int ncols, int begy, int begx);
void    PDC_sync(WINDOW* win);
```

Opis

newwin() creates a new window with the given number of lines, nlines and columns, ncols. The upper left corner of the window is at line begy, column begx. If nlines is zero, it defaults to LINES - begy; ncols to COLS - begx. Create a new full-screen window by calling newwin(0, 0, 0, 0).

delwin() deletes the named window, freeing all associated memory. In the case of overlapping windows, subwindows should be deleted before the main window.

mvwin() moves the window so that the upper left-hand corner is at position (y,x). If the move would cause the window to be off the screen, it is an error and the window is not moved. Moving subwindows

is allowed.

subwin() creates a new subwindow within a window. The dimensions of the subwindow are nlines lines and ncols columns. The subwindow is at position (begy, begx) on the screen. This position is relative to the screen, and not to the window orig. Changes made to either window will affect both. When using this routine, you will often need to call touchwin() before calling wrefresh().

derwin() is the same as subwin(), except that begy and begx are relative to the origin of the window orig rather than the screen. There is no difference between subwindows and derived windows.

mvderwin() moves a derived window (or subwindow) inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

dupwin() creates an exact duplicate of the window win.

wsyncup() causes a touchwin() of all of the window's parents.

If wsyncok() is called with a second argument of TRUE, this causes a wsyncup() to be called every time the window is changed.

wcursyncup() causes the current cursor position of all of a window's ancestors to reflect the current cursor position of the current window.

wsyncdown() causes a touchwin() of the current window if any of its parent's windows have been touched.

resize_window() allows the user to resize an existing window. It returns the pointer to the new window, or NULL on failure.

wresize() is an ncurses-compatible wrapper for resize_window(). Note that, unlike ncurses, it will NOT process any subwindows of the window. (However, you still can call it _on_ subwindows.) It returns OK or ERR.

PDC_makenew() allocates all data for a new WINDOW* except the actual lines themselves. If it's unable to allocate memory for the window structure, it will free all allocated memory and return a NULL pointer.

PDC_makelines() allocates the memory for the lines.

PDC_sync() handles wrefresh() and wsyncup() calls when a window is changed.

Return Value

newwin(), subwin(), derwin() and dupwin() return a pointer to the new window, or NULL on failure. delwin(), mvwin(), mvderwin() and syncok() return OK or ERR. wsyncup(), wcursyncup() and wsyncdown() return nothing.

Errors

   It is an error to call resize_window() before calling initscr().
   Also, an error will be generated if we fail to create a newly sized
   replacement window for curscr, or stdscr. This could happen when
   increasing the window size. NOTE: If this happens, the previously
   successfully allocated windows are left alone; i.e., the resize is
   NOT cancelled for those windows.

Nieportable , sprawdzić

clipboard

Synopsis

```
    int PDC_getclipboard(char* contents, long* length);
    int PDC_setclipboard(const char*  contents, long length);
    int PDC_freeclipboard(char*  contents);
    int PDC_clearclipboard(void);
```

Description

   PDC_getclipboard() gets the textual contents of the system's
   clipboard. This function returns the contents of the clipboard in the
   contents argument. It is the responsibility of the caller to free the
   memory returned, via PDC_freeclipboard(). The length of the clipboard
   contents is returned in the length argument.

   PDC_setclipboard copies the supplied text into the system's
   clipboard, emptying the clipboard prior to the copy.

   PDC_clearclipboard() clears the internal clipboard.

Return Values

   indicator of success/failure of call.
   PDC_CLIP_SUCCESS   the call was successful
   PDC_CLIP_MEMORY_ERROR   unable to allocate sufficient memory for
                           the clipboard contents
   PDC_CLIP_EMPTY   the clipboard contains no text
   PDC_CLIP_ACCESS_ERROR   no clipboard support

---

pdcsetsc

Synopsis

```
    int PDC_set_blink(bool blinkon);
    int PDC_set_bold(bool boldon);
    void PDC_set_title(const char*  title);
```

Description

   PDC_set_blink() toggles whether the A_BLINK attribute sets an actual
   blink mode (TRUE), or sets the background color to high intensity
   (FALSE). The default is platform-dependent (FALSE in most cases). It
   returns OK if it could set the state to match the given parameter,
   ERR otherwise.

   PDC_set_bold() toggles whether the A_BOLD attribute selects an actual
   bold font (TRUE), or sets the foreground color to high intensity
   (FALSE). It returns OK if it could set the state to match the given
   parameter, ERR otherwise.

   PDC_set_title() sets the title of the window in which the curses
   program is running. This function may not do anything on some
   platforms.

sb

--

Synopsis

```
    int sb_init(void)
    int sb_set_horz(int total, int viewport, int cur)
    int sb_set_vert(int total, int viewport, int cur)
    int sb_get_horz(int* total, int* viewport, int* cur)
    int sb_get_vert(int* total, int* viewport, int* cur)
    int sb_refresh(void);
```

Description

   These functions manipulate the scrollbar.

Return Value

   All functions return OK on success and ERR on error.
