Introduction

This file documents the support for various windowing systems in

NetHack. The support is through a standard interface, separating the

main NetHack code from window-system specific code. The implementation

supports multiple window systems in the same binary. Even if you only

wish to support one window-port on your port, you will need to follow

the instructions in Section IX to get a compilable binary.

Contents:

I. Window Types and Terminology

II. Interface Specification

III. Global variables

IV. WINCAP preferences support

V. New or respecified common, high level routines

VI. Helper routines

VII. Game startup

VIII. Conventions

IX. Implementation and Multi-window support

I. Window Types and Terminology

There are 5 basic window types, used to call create\_nhwindow():

NHW\_MESSAGE (top line)

NHW\_STATUS (bottom lines)

NHW\_MAP (main dungeon)

NHW\_MENU (inventory or other "corner" windows)

NHW\_TEXT (help/text, full screen paged window)

The tty window-port also uses NHW\_BASE (the base display) internally.

NHW\_MENU windows can be used for either menu or text display. Their

basic feature is that for the tty-port, if the window is small enough,

it appears in the corner of the tty display instead of overwriting

the whole screen. The first call to add information to the window

will decide if it is going to be used to display a menu or text.

If start\_menu() is called, then it will be used as a menu. If

putstr() is called, it will be used as text. Once decided, there

is no turning back. For the tty-port, if the data is too large for

a single screen then the data is paged (with --more--) between pages.

Only NHW\_MENU type windows can be used for menus.

NHW\_TEXT windows are used to display a large amount of textual data.

This is the type of window one would use for displaying a help file,

for example. In the tty window-port, windows of type NHW\_TEXT can

page using the DEF\_PAGER, if DEF\_PAGER is defined. There exists an

assumption that the font for text windows is monospaced. The help

files are all formatted accordingly.

"window" is always of type winid. This is currently implemented as an

integer, but doesn't necessarily have to be done that way. There are

a few fixed window names that are known throughout the code:

WIN\_MESSAGE (top line)

WIN\_STATUS (bottom lines)

WIN\_MAP (main dungeon)

WIN\_INVEN (inventory)

Other windows are created and destroyed as needed.

"Port" in this document refers to a CPU/OS/hardware platform (UNIX, MSDOS

TOS, etc.) "window-port" refers to the windowing platform. This is

orthogonal (e.g. UNIX might use either a tty window-port or an X11

window-port).

II. Interface Specification

All functions below are void unless otherwise noted.

A. Low-level routines:

raw\_print(str) -- Print directly to a screen, or otherwise guarantee that

the user sees str. raw\_print() appends a newline to str.

It need not recognize ASCII control characters. This is

used during startup (before windowing system initialization

-- maybe this means only error startup messages are raw),

for error messages, and maybe other "msg" uses. E.g.

updating status for micros (i.e, "saving").

raw\_print\_bold(str)

-- Like raw\_print(), but prints in bold/standout (if possible).

curs(window, x, y)

-- Next output to window will start at (x,y), also moves

displayable cursor to (x,y). For backward compatibility,

1 <= x < cols, 0 <= y < rows, where cols and rows are

the size of window.

-- For variable sized windows, like the status window, the

behavior when curs() is called outside the window's limits

is unspecified. The mac port wraps to 0, with the status

window being 2 lines high and 80 columns wide.

-- Still used by curs\_on\_u(), status updates, screen locating

(identify, teleport).

-- NHW\_MESSAGE, NHW\_MENU and NHW\_TEXT windows do not

currently support curs in the tty window-port.

putstr(window, attr, str)

-- Print str on the window with the given attribute. Only

printable ASCII characters (040-0126) must be supported.

Multiple putstr()s are output on separate lines. Attributes

can be one of

ATR\_NONE (or 0)

ATR\_ULINE

ATR\_BOLD

ATR\_BLINK

ATR\_INVERSE

If a window-port does not support all of these, it may map

unsupported attributes to a supported one (e.g. map them

all to ATR\_INVERSE). putstr() may compress spaces out of

str, break str, or truncate str, if necessary for the

display. Where putstr() breaks a line, it has to clear

to end-of-line.

-- putstr should be implemented such that if two putstr()s

are done consecutively the user will see the first and

then the second. In the tty port, pline() achieves this

by calling more() or displaying both on the same line.

get\_nh\_event() -- Does window event processing (e.g. exposure events).

A noop for the tty and X window-ports.

int nhgetch() -- Returns a single character input from the user.

-- In the tty window-port, nhgetch() assumes that tgetch()

will be the routine the OS provides to read a character.

Returned character \_must\_ be non-zero.

int nh\_poskey(int \*x, int \*y, int \*mod)

-- Returns a single character input from the user or a

a positioning event (perhaps from a mouse). If the

return value is non-zero, a character was typed, else,

a position in the MAP window is returned in x, y and mod.

mod may be one of

CLICK\_1 /\* mouse click type 1 \*/

CLICK\_2 /\* mouse click type 2 \*/

The different click types can map to whatever the

hardware supports. If no mouse is supported, this

routine always returns a non-zero character.

B. High-level routines:

print\_glyph(window, x, y, glyph)

-- Print the glyph at (x,y) on the given window. Glyphs are

integers at the interface, mapped to whatever the window-

port wants (symbol, font, color, attributes, ...there's

a 1-1 map between glyphs and distinct things on the map).

char yn\_function(const char \*ques, const char \*choices, char default)

-- Print a prompt made up of ques, choices and default.

Read a single character response that is contained in

choices or default. If choices is NULL, all possible

inputs are accepted and returned. This overrides

everything else. The choices are expected to be in

lower case. Entering ESC always maps to 'q', or 'n',

in that order, if present in choices, otherwise it maps

to default. Entering any other quit character (SPACE,

RETURN, NEWLINE) maps to default.

-- If the choices string contains ESC, then anything after

it is an acceptable response, but the ESC and whatever

follows is not included in the prompt.

-- If the choices string contains a '#' then accept a count.

Place this value in the global "yn\_number" and return '#'.

-- This uses the top line in the tty window-port, other

ports might use a popup.

getlin(const char \*ques, char \*input)

-- Prints ques as a prompt and reads a single line of text,

up to a newline. The string entered is returned without the

newline. ESC is used to cancel, in which case the string

"\033\000" is returned.

-- getlin() must call flush\_screen(1) before doing anything.

-- This uses the top line in the tty window-port, other

ports might use a popup.

-- getlin() can assume the input buffer is at least BUFSZ

bytes in size and must truncate inputs to fit, including

the nul character.

int get\_ext\_cmd(void)

-- Get an extended command in a window-port specific way.

An index into extcmdlist[] is returned on a successful

selection, -1 otherwise.

player\_selection()

-- Do a window-port specific player type selection. If

player\_selection() offers a Quit option, it is its

responsibility to clean up and terminate the process.

You need to fill in pl\_character[0].

display\_file(str, boolean complain)

-- Display the file named str. Complain about missing files

iff complain is TRUE.

update\_inventory()

-- Indicate to the window port that the inventory has been

changed.

-- Merely calls display\_inventory() for window-ports that

leave the window up, otherwise empty.

doprev\_message()

-- Display previous messages. Used by the ^P command.

-- On the tty-port this scrolls WIN\_MESSAGE back one line.

update\_positionbar(char \*features)

-- Optional, POSITIONBAR must be defined. Provide some

additional information for use in a horizontal

position bar (most useful on clipped displays).

Features is a series of char pairs. The first char

in the pair is a symbol and the second char is the

column where it is currently located.

A '<' is used to mark an upstairs, a '>'

for a downstairs, and an '@' for the current player

location. A zero char marks the end of the list.

C. Window Utility Routines

init\_nhwindows(int\* argcp, char\*\* argv)

-- Initialize the windows used by NetHack. This can also

create the standard windows listed at the top, but does

not display them.

-- Any commandline arguments relevant to the windowport

should be interpreted, and \*argcp and \*argv should

be changed to remove those arguments.

-- When the message window is created, the variable

iflags.window\_inited needs to be set to TRUE. Otherwise

all plines() will be done via raw\_print().

\*\* Why not have init\_nhwindows() create all of the "standard"

\*\* windows? Or at least all but WIN\_INFO? -dean

exit\_nhwindows(str)

-- Exits the window system. This should dismiss all windows,

except the "window" used for raw\_print(). str is printed

if possible.

window = create\_nhwindow(type)

-- Create a window of type "type."

clear\_nhwindow(window)

-- Clear the given window, when appropriate.

display\_nhwindow(window, boolean blocking)

-- Display the window on the screen. If there is data

pending for output in that window, it should be sent.

If blocking is TRUE, display\_nhwindow() will not

return until the data has been displayed on the screen,

and acknowledged by the user where appropriate.

-- All calls are blocking in the tty window-port.

-- Calling display\_nhwindow(WIN\_MESSAGE,???) will do a

--more--, if necessary, in the tty window-port.

destroy\_nhwindow(window)

-- Destroy will dismiss the window if the window has not

already been dismissed.

start\_menu(window)

-- Start using window as a menu. You must call start\_menu()

before add\_menu(). After calling start\_menu() you may not

putstr() to the window. Only windows of type NHW\_MENU may

be used for menus.

add\_menu(windid window, int glyph, const anything identifier,

char accelerator, char groupacc,

int attr, char \*str, boolean preselected)

-- Add a text line str to the given menu window. If identifier

is 0, then the line cannot be selected (e.g. a title).

Otherwise, identifier is the value returned if the line is

selected. Accelerator is a keyboard key that can be used

to select the line. If the accelerator of a selectable

item is 0, the window system is free to select its own

accelerator. It is up to the window-port to make the

accelerator visible to the user (e.g. put "a - " in front

of str). The value attr is the same as in putstr().

Glyph is an optional glyph to accompany the line. If

window port cannot or does not want to display it, this

is OK. If there is no glyph applicable, then this

value will be NO\_GLYPH.

-- All accelerators should be in the range [A-Za-z],

but there are a few exceptions such as the tty player

selection code which uses '\*'.

-- It is expected that callers do not mix accelerator

choices. Either all selectable items have an accelerator

or let the window system pick them. Don't do both.

-- Groupacc is a group accelerator. It may be any character

outside of the standard accelerator (see above) or a

number. If 0, the item is unaffected by any group

accelerator. If this accelerator conflicts with

the menu command (or their user defined alises), it loses.

The menu commands and aliases take care not to interfere

with the default object class symbols.

-- If you want this choice to be preselected when the

menu is displayed, set preselected to TRUE.

end\_menu(window, prompt)

-- Stop adding entries to the menu and flushes the window

to the screen (brings to front?). Prompt is a prompt

to give the user. If prompt is NULL, no prompt will

be printed.

\*\* This probably shouldn't flush the window any more (if

\*\* it ever did). That should be select\_menu's job. -dean

int select\_menu(windid window, int how, menu\_item \*\*selected)

-- Return the number of items selected; 0 if none were chosen,

-1 when explicitly cancelled. If items were selected, then

selected is filled in with an allocated array of menu\_item

structures, one for each selected line. The caller must

free this array when done with it. The "count" field

of selected is a user supplied count. If the user did

not supply a count, then the count field is filled with

-1 (meaning all). A count of zero is equivalent to not

being selected and should not be in the list. If no items

were selected, then selected is NULL'ed out. How is the

mode of the menu. Three valid values are PICK\_NONE,

PICK\_ONE, and PICK\_ANY, meaning: nothing is selectable,

only one thing is selectable, and any number valid items

may selected. If how is PICK\_NONE, this function should

never return anything but 0 or -1.

-- You may call select\_menu() on a window multiple times --

the menu is saved until start\_menu() or destroy\_nhwindow()

is called on the window.

-- Note that NHW\_MENU windows need not have select\_menu()

called for them. There is no way of knowing whether

select\_menu() will be called for the window at

create\_nhwindow() time.

char message\_menu(char let, int how, const char \*mesg)

-- tty-specific hack to allow single line context-sensitive

help to behave compatibly with multi-line help menus.

-- This should only be called when a prompt is active; it

sends `mesg' to the message window. For tty, it forces

a --More-- prompt and enables `let' as a viable keystroke

for dismissing that prompt, so that the original prompt

can be answered from the message line "help menu".

-- Return value is either `let', '\0' (no selection was made),

or '\033' (explicit cancellation was requested).

-- Interfaces which issue prompts and messages to separate

windows typically won't need this functionality, so can

substitute genl\_message\_menu (windows.c) instead.

D. Misc. Routines

make\_sound(???) -- To be determined later. THIS IS CURRENTLY UN-IMPLEMENTED.

nhbell() -- Beep at user. [This will exist at least until sounds are

redone, since sounds aren't attributable to windows anyway.]

mark\_synch() -- Don't go beyond this point in I/O on any channel until

all channels are caught up to here. Can be an empty call

for the moment

wait\_synch() -- Wait until all pending output is complete (\*flush\*() for

streams goes here).

-- May also deal with exposure events etc. so that the

display is OK when return from wait\_synch().

delay\_output() -- Causes a visible delay of 50ms in the output.

Conceptually, this is similar to wait\_synch() followed

by a nap(50ms), but allows asynchronous operation.

askname() -- Ask the user for a player name.

cliparound(x, y)-- Make sure that the user is more-or-less centered on the

screen if the playing area is larger than the screen.

-- This function is only defined if CLIPPING is defined.

number\_pad(state)

-- Initialize the number pad to the given state.

suspend\_nhwindows(str)

-- Prepare the window to be suspended.

resume\_nhwindows()

-- Restore the windows after being suspended.

start\_screen() -- Only used on Unix tty ports, but must be declared for

completeness. Sets up the tty to work in full-screen

graphics mode. Look at win/tty/termcap.c for an

example. If your window-port does not need this function

just declare an empty function.

end\_screen() -- Only used on Unix tty ports, but must be declared for

completeness. The complement of start\_screen().

outrip(winid, int)

-- The tombstone code. If you want the traditional code use

genl\_outrip for the value and check the #if in rip.c.

preference\_update(preference)

-- The player has just changed one of the wincap preference

settings, and the NetHack core is notifying your window

port of that change. If your window-port is capable of

dynamically adjusting to the change then it should do so.

Your window-port will only be notified of a particular

change if it indicated that it wants to be by setting the

corresponding bit in the wincap mask.

III. Global variables

The following global variables are defined in decl.c and must be used by

the window interface to the rest of NetHack.

char toplines[BUFSZ] Contains the last message printed to the WIN\_MESSAGE

window, used by Norep().

winid WIN\_MESSAGE, WIN\_MAP, WIN\_STATUS, WIN\_INVEN

The four standard windows.

char \*AE, \*AS; Checked in options.c to see if we should switch

to DEC\_GRAPHICS. It is #ifdefed VMS and UNIX.

int LI, CO; Set in sys/unix/ioctl.c.

The following appears to be Unix specific. Other ports using the tty

window-port should also declare this variable in one of your sys/\*.c files.

short ospeed; Set and declared in sys/unix/unixtty.c (don't

know about other sys files).

The following global variable is defined in options.c. It equates a

list of wincap option names with their associated bit-mask [see

section IV WINCAP preferences support]. The array is zero-terminated.

struct wc\_Opt wc\_options[];

One entry for each available WINCAP option.

Each entry has a wc\_name field and a wc\_bit

field.

IV. WINCAP preferences support

Starting with NetHack 3.4.0, the window interface was enhanced to provide

a common way of setting window port user preferences from the config file,

and from the command line for some settings.

The wincap preference settings all have their underlying values stored

in iflags fields. The names of the wincap related fields are all pre-

fixed with wc\_ to make it easy to identify them. Your window port can

access the fields directly.

Your window port identifies what options it will react to and support

by setting bits in the window\_procs wincap mask. See section IX for

details of where the wincap mask resides.

Two things control whether any preference setting appears in the

'O' command options menu during the game:

1. The option must be marked as being supported by having its

bit set in the window\_procs wincap mask.

2. The option must have its optflag field set to SET\_IN\_GAME in order

to be able to set the option, or marked DISP\_IN\_GAME if you just

want to reveal what the option is set to.

Both conditions must be true to be able to see or set the option from

within NetHack.

The default values for the optflag field for all the options are

hard-coded into the option in options.c. The default value for

the options can be altered by calling

set\_wc\_option\_mod\_status(optmask, status)

specifying the option modification status to one of SET\_IN\_FILE,

DISP\_IN\_GAME, or SET\_IN\_GAME.

The setting of any wincap option is handled by the NetHack core option

processing code. You do not have to provide a parser in your window

port, nor should you set the values for the iflags.wc\_\* fields

directly within the port code. The port code should honor whatever

values were put there by the core when processing options, either

in the config file, or by the 'O' command.

You may be wondering what values your window port will find in the

iflags.wc\_\* fields for options that the user has not specified

in his/her config file. Put another way, how does you port code

tell if an option has not been set? The next paragraph explains that.

If the core does not set an option, it will still be initialized

to its default value. Those default values for the

iflags.wc\_\* fields are:

o All boolean fields are initialized to the starting

value specified for that option in the boolopt array in

options.c. The window-port should respect that setting

unless it has a very good reason for not doing so.

o All int fields are initialized to zero. Zero is not a valid

setting for any of the int options, so if your port code

encounters a zero there, it can assume that the preference

option was not specified. In that case, the window-port code

should use a default setting that the port is comfortable with.

It should write the default setting back into the iflags.wc\_\*

field. That is the only time that your window-port could should

update those fields.

o All "char \*" fields will be null pointers. Be sure to check for

that in your window-port code before using such a pointer, or

you'll end up triggering a nasty fault.

Here are the wincap preference settings that your port can choose

to support:

+--------------------+--------------------+--------------------+--------+

| | | iflags field | data |

| player option | bit in wincap mask | for value | type |

|--------------------+--------------------+--------------------+--------+

| align\_message | WC\_ALIGN\_MESSAGE | wc\_align\_message |int |

| align\_status | WC\_ALIGN\_STATUS | wc\_align\_status |int |

| ascii\_map | WC\_ASCII\_MAP | wc\_ascii\_map |boolean |

| color | WC\_COLOR | wc\_color |boolean |

| eight\_bit\_tty | WC\_EIGHT\_BIT\_IN | wc\_eight\_bit\_input |boolean |

| font\_map | WC\_FONT\_MAP | wc\_font\_map |char \* |

| font\_menu | WC\_FONT\_MENU | wc\_font\_menu |char \* |

| font\_message | WC\_FONT\_MESSAGE | wc\_font\_message |char \* |

| font\_status | WC\_FONT\_STATUS | wc\_font\_status |char \* |

| font\_text | WC\_FONT\_TEXT | wc\_font\_text |char \* |

| font\_size\_map | WC\_FONTSIZ\_MAP | wc\_fontsiz\_map |int |

| font\_size\_menu | WC\_FONTSIZ\_MENU | wc\_fontsiz\_menu |int |

| font\_size\_message | WC\_FONTSIZ\_MESSAGE | wc\_fontsiz\_message |int |

| font\_size\_status | WC\_FONTSIZ\_STATUS | wc\_fontsiz\_status |int |

| font\_size\_text | WC\_FONTSIZ\_TEXT | wc\_fontsiz\_text |int |

| hilite\_pet | WC\_HILITE\_PET | wc\_hilite\_pet |boolean |

| large\_font | WC\_LARGE\_FONT | wc\_large\_font |boolean |

| map\_mode | WC\_MAP\_MODE | wc\_map\_mode |int |

| player\_selection | WC\_PLAYER\_SELECTION| wc\_player\_selection|int |

| popup\_dialog | WC\_POPUP\_DIALOG | wc\_popup\_dialog |boolean |

| preload\_tiles | WC\_PRELOAD\_TILES | wc\_preload\_tiles |boolean |

| scroll\_margin | WC\_SCROLL\_MARGIN | wc\_scroll\_margin |int |

| splash\_screen | WC\_SPLASH\_SCREEN | wc\_splash\_screen |boolean |

| tiled\_map | WC\_TILED\_MAP | wc\_tiled\_map |boolean |

| tile\_width | WC\_TILE\_WIDTH | wc\_tile\_width |int |

| tile\_height | WC\_TILE\_HEIGHT | wc\_tile\_height |int |

| tile\_file | WC\_TILE\_FILE | wc\_tile\_file |char \* |

| use\_inverse | WC\_INVERSE | wc\_inverse |boolean |

| vary\_msgcount | WC\_VARY\_MSGCOUNT | wc\_vary\_msgcount |int |

| windowcolors | WC\_WINDOWCOLORS | wc\_foregrnd\_menu |char \* |

| | | wc\_backgrnd\_menu |char \* |

| | | wc\_foregrnd\_message|char \* |

| | | wc\_backgrnd\_message|char \* |

| | | wc\_foregrnd\_status |char \* |

| | | wc\_backgrnd\_status |char \* |

| | | wc\_foregrnd\_text |char \* |

| | | wc\_backgrnd\_text |char \* |

+--------------------+--------------------+--------------------+--------+

align\_message -- where to place message window (top, bottom, left, right)

align\_status -- where to place status window (top, bottom, left, right).

ascii\_map -- port should display an ascii map if it can.

color -- port should display color if it can.

eight\_bit\_tty -- port should allow eight bit input.

font\_map -- port should use a font by this name for map window.

font\_menu -- port should use a font by this name for menu windows.

font\_message -- port should use a font by this name for message window.

font\_size\_map -- port should use this size font for the map window.

font\_size\_menu -- port should use this size font for menu windows.

font\_size\_message

-- port should use this size font for the message window.

font\_size\_status-- port should use this size font for the status window.

font\_size\_text -- port should use this size font for text windows.

font\_status -- port should use a font by this name for status window.

font\_text -- port should use a font by this name for text windows.

hilite\_pet -- port should mark pets in some special way on the map.

large\_font -- port should use a large font.

map\_mode -- port should display the map in the manner specified.

player\_selection

-- dialog or prompts for choosing character.

popup\_dialog -- port should pop up dialog boxes for input.

preload\_tiles -- port should preload tiles into memory.

scroll\_margin -- port should scroll the display when the hero or cursor

is this number of cells away from the edge of the window.

splash\_screen -- port should/should not display an opening splashscreen.

tiled\_map -- port should display a tiled map if it can.

tile\_width -- port should display tiles with this width or round to closest

if it can.

tile\_height -- port should display tiles with this height or round to closest

if it can.

tile\_file -- open this alternative tile file. The file name is likely to be

window-port or platform specific.

use\_inverse -- port should display inverse when NetHack asks for it.

vary\_msgcount -- port should display this number of messages at a time in

the message window.

windowcolors

-- port should use these colors for window foreground/background

colors. Syntax:

menu fore/back message fore/back status fore/back text fore/back

Whenever one of these settings is adjusted, the port is notified of a change

to the setting by calling the port's preference\_update() routine. The port

is only notified if it has indicated that it supports that option by setting

the option's bit in the port's wincap mask. The port can choose to adjust

for the change to an option that it receives notification about, or ignore it.

The former approach is recommended. If you don't want to deal with a

user-initiated setting change, then the port should call

set\_wc\_option\_mod\_status(mask, SET\_IN\_FILE) to make the option invisible to

the user.

Functions available for the window port to call:

set\_wc\_option\_mod\_status(optmask, status)

-- Adjust the optflag field for a set of options to

specify whether the port wants the option to appear

in the 'O' command options menu, The second parameter,

"status" can be set to SET\_IN\_FILE, DISP\_IN\_GAME,

or SET\_IN\_GAME (SET\_IN\_FILE implies that the option

is completely hidden during the game).

set\_option\_mod\_status(optnam, status)

-- Adjust the optflag field for one of the core options

that is not part of the wincap suite. A port might use

this to override the default initialization setting for

status specified in options.c. Note that you have to

specify the option by name and that you can only set

one option per call unlike set\_wc\_option\_mod\_status().

Adding a new wincap option:

To add a new wincap option, please follow all these steps:

1. Add the option to the wincap preference settings table above.

2. Add the description to the paragraph below the chart.

3. Add the WC\_ to the bit list in include/winprocs.h (if there is room).

4. Add the wc\_ field(s) to the iflags structure in flag.h.

5. Add an appropriate parser to parseoptions() in options.c.

6. Add code to display current value to get\_compopt\_value() in options.c.

7. Document the option in Guidebook.mn and Guidebook.tex.

8. Add the bit name to the OR'd values in your window port's winprocs struct

wincap mask if your port supports the option.

V. New or respecified common, high level routines

These are not part of the interface, but mentioned here for your information.

char display\_inventory(lets, want\_reply)

-- Calls a start\_menu()/add\_menu()/select\_menu() sequence.

It returns the item selected, or '\0' if none is selected.

Returns '\033' if the menu was canceled.

raw\_printf(str, ...)

-- Like raw\_print(), but accepts arguments like printf(). This

routine processes the arguments and then calls raw\_print().

-- The mac version #defines error raw\_printf. I think this

is a reasonable thing to do for most ports.

pline(str, ...)

-- Prints a string to WIN\_MESSAGE using a printf() interface.

It has the variants You(), Your(), Norep(), and others

in pline.c which all use the same mechanism. pline()

requires the variable "char toplines[]" be defined; Every

putstr() on WIN\_MESSAGE must copy str to toplines[] for use

by Norep() and pline(). If the window system is not active

(!iflags.window\_inited) pline() uses raw\_print().

VI. Helper Routines

These are not part of the interface. They may be called by your

window port routines to perform the desired task, instead of duplicating

the necessary code in each window port.

mapglyph(int glyph, int \*ochar, int \*ocolor, unsigned \*special, int x, int y)

-- Maps glyph at x,y to NetHack ascii character and color.

If it represents something special such as a pet, that

information is returned as set bits in "special."

Usually called from the window port's print\_glyph()

routine.

VII. Game startup

The following is the general order in which calls from main() should be made,

as they relate to the window system. The actual code may differ, but the

order of the calls should be the same.

choose\_windows(DEFAULT\_WINDOW\_SYS) /\* choose a default window system \*/

initoptions() /\* read the resource file \*/

init\_nhwindows() /\* initialize the window system \*/

process\_options(argc, argv) /\* process command line options or equiv \*/

if(save file is present) {

display\_gamewindows() /\* create & display the game windows \*/

dorestore() /\* restore old game; pline()s are OK \*/

} else {

player\_selection() /\* select a player type using a window \*/

display\_gamewindows() /\* create & display the game windows \*/

}

pline("Hello, welcome...");

Choose\_windows() is a common routine, and calling it in main() is necessary

to initialize the function pointer table to \_something\_ so that calls to

raw\_print() will not fail. Choose\_windows() should be called almost

immediately upon entering main(). Look at unixmain.c for an example.

Display\_gamewindows() is a common routine that displays the three standard

game windows (WIN\_MESSAGE, WIN\_MAP, and WIN\_STATUS). It is normally called

just before the "Hello, welcome" message.

Process\_options() is currently still unique to each port. There may be need

in the future to make it possible to replace this on a per window-port basis.

VIII. Conventions

init\_nhwindows() is expected to display a gee-whiz banner window, including

the Copyright message. It is recommended that the COPYRIGHT\_BANNER\_A,

COPYRIGHT\_BANNER\_B, and COPYRIGHT\_BANNER\_C macros from patchlevel.h be used

for constructing the Copyright message. COPYRIGHT\_BANNER\_A is a

quoted string that has the NetHack copyright declaration,

COPYRIGHT\_BANNER\_B is a quoted string that states who the copyright

belongs to, and COPYRIGHT\_BANNER\_C simply says "See License for

details." Be sure to #include "patchlevel.h" to define these macros.

Using the macros will prevent having to update the Copyright information

in each window-port prior to each release.

Ports (MSDOS, TOS, MAC, etc) \_may\_ use window-port specific routines in

their port specific files, \_AT\_THEIR\_OWN\_RISK\_. Since "port" and

"window-port" are orthogonal, you make your "port" code less portable by

using "window-port" specific routines. Every effort should be made to

use window-port interface routines, unless there is something port

specific that is better suited (e.g. msmsg() for MSDOS).

The tty window-port is contained in win/tty, the X window port is contained

in win/X11. The files in these directories contain \_only\_ window port code,

and may be replaced completely by other window ports.

IX. Implementation and Multi-window support

NetHack 3.2 and higher support multiple window systems in the same binary.

When writing a new window-port, you need to follow the following guidelines:

1) Pick a unique prefix to identify your window-port. For example, the tty

window port uses "tty"; the X11 window-port uses "X11".

2) When declaring your interface function, precede the function names with

your unique prefix. E.g:

void tty\_init\_nhwindows()

{

/\* code for initializing windows in the tty port \*/

}

When calling window functions from within your port code, we suggest

calling the prefixed version to avoid unnecessary overhead. However,

you may safely call the non-prefixed version (e.g. putstr() rather than

tty\_putstr()) as long as you #include "hack.h". If you do not

include hack.h and use the non-prefixed names, you will get compile

or link-time errors.

We also suggest declaring all functions and port-specific data with

this prefix to avoid unexpected overlaps with other window-ports.

The tty and X11 ports do not currently follow this suggestion, but do

use separate non-overlapping convention for naming data and internal

functions.

3) Declare a structure, "struct window\_procs prefix\_procs", (with your

prefix instead of "prefix") and fill in names of all of your

interface functions. The first entry in this structure is the name

of your window-port, which should be the prefix. The second entry

is the wincap mask that identifies what window port preference

settings your port will react to and support. The other entries

are the function addresses.

Assuming that you followed the convention in (2), you can safely copy

the structure definition from an existing window-port and just change

the prefixes. That will guarantee that you get the order of your

initializations correct (not all compilers will catch out-of-order

function pointer declarations).

4) Add a #define to config.h identifying your window-port in the

"Windowing systems" section. Follow the "prefix\_GRAPHICS" convention

for your window-port.

5) Add your prefix to the list of valid prefixes listed in the "Known

systems are" comment.

6) Edit makedefs.c and add a string for your windowing system to window\_opts

inside an #ifdef prefix\_GRAPHICS.

7) Edit windows.c and add an external reference to your prefix\_procs inside

an #ifdef prefix\_GRAPHICS. Also add an entry to the win\_choices

structure for your window-port of the form:

#ifdef prefix\_GRAPHICS

{ &prefix\_procs, prefix\_init\_function },

#endif

The init\_function is necessary for some compilers and systems to force

correct linking. If your system does not need such massaging, you

may put a null pointer here.

You should declare prefix\_procs and prefix\_init\_function as extern's

in your win\*.h file, and #include that file at the beginning of

windows.c, also inside an #ifdef prefix\_GRAPHICS. Some win\*.h files

are rather sensitive, and you might have to duplicate your

prefix\_procs and prefix\_init\_function's instead of including win\*.h.

The tty port includes wintty.h, the X11 port duplicates the declarations.

8) If your port uses Makefile.src, add the .c and .o files and an

appropriate comment in the section on "WINSRC" and "WINOBJ". See

Makefile.src for the style to use. If you don't use Makefile.src,

we suggest using a similar convention for the make-equivalent used

on your system. Also add your new source and binaries to WINSRC and

WINOBJ (if you want the NetHack binary to include them, that is).

9) Look at your port's portmain.c (the file containing main()) and make

sure that all of the calls match the the requirements laid out in

Section VII.

Now, proceed with compilation and installation as usual. Don't forget

to edit Makefile.src (or its equivalent) and config.h to set the

window-ports you want in your binary, the default window-port to use,

and the .o's needed to build a valid game.

One caveat. Unfortunately, if you incorrectly specify the

DEFAULT\_WINDOW\_SYS, NetHack will dump core (or whatever) without

printing any message, because raw\_print() cannot function without first

setting the window-port.