HP Windows/9000 Reference

HP 9000 Series 300 Computers

HP Part Number 97069-90022

HEWLETT PACKARD

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

April 1988...Edition 1

COMMAND SUMMARY

NAME

wborder – control window border style, color, label

SYNOPSIS

wborder [**-ntT**] [**-c** fcolor,bcolor] [**-l** label] [window_spec...]

DESCRIPTION

This command lets you change the border style (normal, thin, or none), colors, and/or label for the specified windows (by default, if you give no *window_specs*, the window connected to standard input). See *windows*(1) for an explanation of *window_spec*.

Options are:

-n Set the borders to normal form (label and manipulation areas displayable). This is the default if you give no options.

-t Set the borders to thin form. Label and manipulation areas are concealed and inaccessible, except that you can get a pop-up menu using the thin border.

-T Do not display a border. This option is supported only by the graphics window type.

You can only give one of **-n**, **-t**, or **-T**.

-c *fcolor,bcolor*
Set the border foreground (*fcolor*) and background (*bcolor*) colors (index numbers). Each number specifies an actual color in the color map (there is only one per physical display). The colors are not changed unless you give this option.

Color names or abbreviations (substrings) may be given instead of numbers, in any mixture of upper and lower case. They are mapped to color indices as follows:

black	0 ("b" and "bl" mean "black")
white	1
red	2
yellow	3
green	4
cyan	5
blue	6 ("blu" or "blue" is required)
magenta	7

This mapping is only valid as long as the default color map for the device is not changed.

-l *label* Set the windows' labels to the given string, which should be enclosed in quotes if necessary to have the shell treat it as a single item. The label is not changed unless you give this option.

Up to 128 characters of the label are displayed in the border of a graphics window, and up to 12 characters of the label are displayed in the border of a term0 window. Only the first 12 characters of the label are displayed in the icon label and the default pop-up menu title. Only the first 14 characters are displayed in the title in the SFK area. When a window is sized such that not all of the label would fit in the border, the display length of the label is truncated, down to a minimum of one character. (A window cannot be sized any smaller than this if the border is normal).

Note that the label is not necessarily the true name of the window. Only the true name can be given to other commands as *window_spec*. You can use *wlist*(1) with no arguments to get the true name.

None of the changes made using this command are immediately visible if a window is concealed or occluded. The border label is only displayed when the border is normal.

You can't make the border normal on a window whose current size is less than the minimum size allowed with a normal border. If a window does not have a big enough raster/buffer to allow it to grow to this minimum size, it can never have a normal border.

EXAMPLES

wborder

Set the border to normal width on the window connected to standard input.

wborder -t

Set the border to thin style on the window connected to standard input.

wborder -n -c 23,104 -l'A Window' win4 win5

Set the border to normal, the border foreground color to 23 and background color to 104, and the label to "A Window" for the windows named "win4" and "win5".

wborder -cb,cy

Set the border foreground color to black (0) and the background color to cyan (6) for the window connected to standard input.

wborder -c Red,Green

Set the border foreground color to red (5) and the background color to green (3) for the window connected to standard input.

HARDWARE DEPENDENCIES

Series 500:

The **−T** option is not supported on Series 500. The Series 500 does not support the *null* border type; each window must have either a *thin* or *normal* border.

SEE ALSO

windows(1), wcreate(1), wlist(1),wsh(1),wsize(1),wbanner(3W),wsetbcolor(3W), wsetlabel(3W).

DIAGNOSTICS

The following values are returned by this routine:

0 If no errors are detected.

1 If it encounters an error which prevents changing border attributes for a window, including trouble while expanding a *window_spec* pattern. Note that an error message is also displayed to standard error in this case.

2 If it encounters any error while trying to set attributes for one window. An error message is also displayed to standard error.

NAME

wcreate – create one or more new windows

SYNOPSIS

wcreate [**–w** windowtype] [**–kboiTtMmNnv**] [**–l** x,y] [**–s** w,h] [**–r** w,h] [window_spec...]

DESCRIPTION

This command creates one or more new windows on the display, of the selected *windowtype*, named *window_spec...* (if you give no *window_spec*, it creates one window with a name selected by the window manager). See *windows*(1) for explanations of *windowtype* and *window_spec*.

New windows have default icon positions set, have default viewing (pan) positions, have default border labels and colors and have null typing aids. Other attributes of new windows are controlled by the options:

–w *windowtype*

Type of windows to create (see *windows*(1)). The default window type is the **term0** window. The default base and alternate fonts used for this window type are dependent on the resolution of the display screen and the *$LANG* environment variable. Default fonts are found under the directory **/usr/lib/raster/dflt**. Under that directory are two directories: **b** for the base fonts and **a** for the alternate fonts. Under those directories are three more directories: **v** for the very high resolution displays, **h** for the high resolution displays, and **l** for the low resolution displays. Under those directories is a file named after the current language name as defined by the *$LANG* environment variable; this file is linked to the default font. For example, if *$LANG* is set to **japanese**, the default base font on the high resolution display would be **/usr/lib/raster/dflt/b/h/japanese**, which would be a link to **/usr/lib/raster/8x18/kanji.16K**. The default alternate font on the high resolution display would be found in **/usr/lib/raster/dflt/a/h/japanese**, which would be a link to **/usr/lib/raster/8x18/kana.8K**. If the *$LANG* environment variable is not set or an entry for it does not exist in the default directories, then the following default base and alternate fonts are used:

Very High Resolution:

/usr/lib/raster/10x20/lp.8U
/usr/lib/raster/10x20/lp.b.8U

High Resolution:

/usr/lib/raster/8x16/lp.8U
/usr/lib/raster/8x16/lp.b.8U

Low Resolution:

/usr/lib/raster/6x8/lp.8U
/usr/lib/raster/6x8/lp.b.8I

A Very High Resolution display is one having 1280 pixel columns. This includes the HP98720, HP98548, HP98550 and HP98730. High Resolution displays have a pixel width of 1024. Some High Resolution graphics displays are the HP98700, HP98549, and HP98547. Low Resolution displays have a width of 512 pixels. The HP98543 is a Low Resolution display.

If you want a different base and/or alternate font assigned at the time of window creation, set the environment variables *$WMBASEFONT* and *$WMALTFONT* to the pathnames (or *$WMFONTDIR*-relative basenames) of the desired font files.

–k Attach a new window to the keyboard (i.e. select the last window in the list). The default is to leave the currently selected window attached to the keyboard.

–b Put new windows on the bottom of the display stack (in the order created). By default they are created as topmost windows (if not concealed).

–o Conceal new windows "off screen" (make them non-displayable). By default they are displayable, thus visible if not occluded by other windows.

You can only give one of the **–b** and **–o** options.

–i Make the new window iconic.

–T Do not display a border for the new window.

–t Make new window borders thin, not normal in width. This also allows very small windows and raster/buffer sizes.

You can only give one of the **–T** and **–t** options.

–M Make window rasters retained as byte/pixel. This is the default. This option is ignored for window types like **term0** or **see_thru**, which never retain their rasters.

–m Make window rasters retained as bit/pixel. This option is ignored for window types like **term0** or **see_thru**, which never retain their rasters.

–N Make the window an IMAGE graphics window. This means that the user area of the window is mapped into the image planes while the border is displayed in the overlay planes. This option only applies on the HP98730 display system and is ignored for window types other than **graphics**.

–n Make window rasters non-retained (see *windows*(1)). By default they are retained as byte/pixel, so they consume memory, and the windows may be repainted. This option is ignored for window types like **term0** or **see_thru**, which never retain their rasters.

You can only give one of the **–M**, **–m** , **–N** and **–n** options.

–v Verbose mode: Print the full pathname of each new window (including *$WMDIR* if it is declared in the environment and is used) to standard output as each window is created. By default *wcreate* is silent unless there is an error. Verbose mode is useful when you don't give *window_spec*, but allow the window manager to create a name for you. You can capture new window names in a shell variable, for example (using *sh*(1)):

```
win=` wcreate -vw graphics ` # save window name.
win=` basename "$win" `      # get basename only.
```

–l *x,y* Set the upper left corner location (anchor point) for the contents area of all new windows (the border area is outside this point). The *x,y* coordinates are in screen units (pixels). The default location for the anchor point, set by the window manager (and varied for each window), is near the upper left corner of the screen.

–s *w,h* Set the width and height of the contents area in rows and columns, for **term0** windows, or in pixels, for other window types. Default sizes are 80,24 for **term0** windows and 200,200 for other window types. If an 80,24 window with the specified font will not fit on the display the size of the window is reduced to a size that will fit. For example, if a **term0** window is created with an 5x18 font on a low resolution display and the size of the window is not specified, the default size of the window will be 80,18. (400/18 = 22 - 4(adjustment to allow room for border) = 18)

–r *w,h* Set raster/buffer sizes associated with new windows. The width and height of the rasters/buffers are specified in units appropriate to the window type. Use rows and columns for **term0** windows; use pixel width and pixel height for other window types.

Default values are:

- 80,48 for **term0** scroll buffers (two full pages of text),
- the resolution of the display for **see_thru** window types,
- and the same as the initial window size for other window types.

If you give **−r** but not **−s**, the default window width (height) is decreased, if necessary, to be as small as the raster/buffer width (height). Likewise, if you give **−s** but not **−r**, the default raster/buffer width (height) is increased, if necessary, to be as large as the window width (height).

If new window borders are normal, the windows and rasters/buffers have minimum allowable sizes greater than one pixel (or one character cell). A window's contents area can never grow to be larger than its raster/buffer size, or, for **term0** windows, its screen size (see below), whichever is smaller. Do not confuse the raster/buffer memory with the amount of graphics/text visible at one time, which depends on a window's size and pan position.

For **term0** windows, the screen size, distinct from window and buffer sizes, is set the same as the initial window size. The screen size is the same as given by the *termcap*(5) **co** and **li** values. If the default window size is used, screen size is always set to 80,24 even if the window size is adjusted to allow the window to fit on the screen. It is only useful with library calls and escape sequences (such as those provided by the *curses*(3X) library). It remains set to the initial row and column size, so it may change pixel size with font changes, but cannot be otherwise changed using *wsize(1)* or any other command. See the *Programmer's Manual* sections on *term0* for details.

Normally the owner of the window's special file is set to the process id of the *wcreate* process by the window manager.

EXAMPLES

wcreate Joe John
: Create **term0** windows named "Joe" and "John" near the upper left corner of the screen, whose special files reside "in the usual place" (e.g. the directory specified by $*WMDIR*).

wcreate /tmp/windows/w1
: Create a **term0** window named "w1" near the upper left corner of the screen, whose special file is located in the directory specified in the pathname.

wcreate -w graphics -kotn -l100,200 -s300,300 graphwin
: Create an assigned (selected), concealed, thin-border Starbase window named *graphwin* in the directory $*WMDIR*, whose raster is non-retained. The upper left corner is at x = 100, y = 200 (pixels). The window is 300 pixels on a side. Its corresponding raster is also 300 pixels on a side.

wcreate -kts80,66
: Create a selected, thin-bordered, top-of-display-stack **term0** window with a default name, whose window, screen, and scroll buffer sizes are all 80 columns by 66 rows.

wcreate -w see_thru
: Create a **see_thru** widow with default name, location, and size.

wcreate -w see_thru -ib -l 0,0 -s 1280,1024 image_planes
: Create a **see_thru** window named *image_planes*, which will appear on the screen as an icon on the bottom of the window stack. When the window is made normal, it will be located at the upper-left corner of the display and will have a size of 1280 by 1024 pixels, i.e., it will cover the entire screen. (Use [SHIFT]-[SELECT] to bring another window to the top.)

wcreate -w graphics -Nt -l 100,100 -s 640,512 accelwin
  Create an IMAGE **graphics** window named *accelwin*, which will be approximately one fourth the screen size of the HP98730 display and which will have a thin border.

SEE ALSO

windows(1),wborder(1),wlist(1), wsh(1),wsize(1), wcreate_graphics(3W), wcreate_term0(3W).

DIAGNOSTICS

The following values are returned by *wcreate*:

0 If no errors are detected.

1 If any of the following occurs:

  bad invocation
  invalid window type
  anchor point out of bounds
  location point out of bounds
  negative size (possibly due to bad location point)
  raster size out of bounds
  any other error which prevents creation of any windows

  An appropriate error message is also written to standard output. Also prints a message and returns 1 if it fails to create any one window (but still tries to create the others, unless it has trouble while expanding a *window_spec* pattern).

2 After each window is created, some window attributes must be set separately (i.e. keyboard selection and top or bottom of stack). If an error is detected in attempting to set them, the new window name is still printed if the **-v** option was used. Later a 2 is returned, meaning a window was created incorrectly, unless some other error caused a 1 to be returned instead.

NAME

wdestroy – destroy one or more windows or set autodestroy attributes

SYNOPSIS

wdestroy window_spec...
wdestroy [**-adn**] [window_spec...]

DESCRIPTION

When none of the **-a**, **-d**, or **-n** options are specified, this command destroys (deletes) one or more specified windows. See *windows*(1) for an explanation of *window_spec*. For safety, you must use "–" to refer to the window affiliated with standard input. Note that "wdestroy 'wlist'" works equally well.

Any processes in the process group whose leader opened the deleted window are sent the SIGHUP signal. This typically terminates them, unless they are background processes started using *nohup*(1), or they do something special with that signal.

If a destroyed window was the selected one, then after it is gone the keyboard is attached to the top window, if one exists; otherwise it is not attached to any window. In the latter case, you can select a new window using a pop-up menu. Likewise, if the last window is destroyed, you can create a new one using a pop-up menu.

When called with the **-a**, **-d**, or **-n** options, the specified window is **not** deleted; rather, the window's autodestroy and recovered attributes are affected as follows:

-a Causes the window to be automatically deleted when the window's device interface (special file) is closed by every process that has opened the interface. Note that this is the default state for windows created through the system pop-up menu, and is also the default state for the *wconsole* window created by *wmstart*(1).

-d Causes the window to be deleted when the window's device interface is closed by every process **and** a new window is created. Therefore, to remove windows that are in this state, you must either create a new window (after its device interface is closed by all processes that opened it), or you must explicitly destroy the window via *wdestroy* (with no options specified) or by using the *Destroy* option of the pop-up menu.

-n When this option is used, the specified window won't be deleted when its device interface is closed by every process; you must **explicitly** destroy the window via *wdestroy* (with no options), or you must use the *Destroy* option of the pop-up menu. Note that this is the default state for windows created by *wcreate*(1) or *wsh*(1) with no options specified. In addition, specifying this option is equivalent in effect to using the *Save* item of the pop-up menu on a window.

EXAMPLES

wdestroy –
Delete the window affiliated with standard input.

wdestroy oldwin1 oldwin2
Delete the windows named "oldwin1" and "oldwin2".

wdestroy -a window12
Causes the window named "window12" to be automatically destroyed when its device interface is closed by every process associated with it. For example, if "window12" contains a shell, then terminating the window's shell via *exit*(1) will cause the window to be automatically destroyed.

wdestroy -d pam_win
Automatically delete the window named "pam_win" only when every process that has opened the window's interface has also closed it; additionally, the window won't be

destroyed until a new window is created. For example, if the window contains a shell, then terminating the shell via **ctl–D** or *exit*(1) won't cause the window to be deleted, **until** a new window is created, either by the pop-up menu or the *wsh*(1) or *wcreate*(1) commands.

wdestroy -n wconsole

By default the *wconsole* window, i.e., the first window normally created by *wmstart*(1), is set to be automatically destroyed (**-a**) when you exit the shell in that window. The command shown here reverses the auto-termination status so that *wconsole* won't be automatically destroyed.

SEE ALSO

windows(1),wcreate(1),wlist(1),wsh(1),wautodestroy(3W),wdestroy(3W), wrecover(3W).

DIAGNOSTICS

This routine returns the following values:

0 If no errors are detected.

1 Prints a message to standard error and returns 1, with no windows destroyed, in case of an error which prevents destroying any windows. Also aborts and returns 1 in case of trouble while expanding a *window_spec* pattern.

2 If any one window cannot be destroyed for some reason, prints a message to standard error, continues with the next *window_spec* (if any), and eventually returns 2.

NAME

wdisp – change displayability of windows

SYNOPSIS

wdisp [**-tbo**] [**-ni**] [window_spec...]
wdisp -d
wdisp -u

DESCRIPTION

This command changes one or more windows' positions in the display stack, their concealment, and/or their representation (normal or icon). By default, if you give no *window_specs*, the window connected to standard input is made displayable as the top window on the stack and put in normal (not icon) form. See *windows*(1) for an explanation of *window_spec*.

Options are:

-t Make the windows displayable and on top of the stack (in the order their names appear). This (along with **-n**) is the default if you give no options.

-b Make the windows displayable and on the bottom of the stack (in the order their names appear). They may be partially or totally occluded by other windows, thus not totally visible.

-o Make the windows concealed off-screen. Concealing windows removes them from the stack.

Only one of **-t**, **-b**, and **-o** is allowed. If you give one of them, but neither **-n** nor **-i**, then only the windows' displayability is changed, not their representations.

-n Make the windows go to normal form (if they were iconic). This (along with **-t**) is the default if you give no options.

-i Make the windows go to iconic form (if they were normal).

Only one of **-n** and **-i** is allowed. If you give one of them, but none of **-t**, **-b**, or **-o**, then only the windows' representations are changed, not their displayability.

-d Shuffle the top window down, i.e. the old top window becomes the bottom window and the new top window becomes selected.

-u Shuffle the bottom window up, i.e. the old bottom window becomes the top and selected window.

The **-d** and **-u** options each cannot be combined with any other, nor applied to any *window_specs*. Note also that with these options, the resulting topmost window automatically becomes the selected window.

EXAMPLES

wdisp Make the window connected to standard input displayable, on top of the stack, and normal form.

wdisp -bi
Make the window connected to standard input displayable, at the bottom of the stack, and convert it to icon form.

wdisp -on minnow tadpole
Make the windows "minnow" and "tadpole" concealed and convert them to normal form.

wdisp -u
Shuffle the bottom window to the top of the display stack and make it selected.

SEE ALSO

windows(1),wcreate(1),wlist(1),wsh(1),wautotop(3W),wbottom(3W),wconceal(3W), wiconic(3W),wshuffle(3W),wtop(3W).

DIAGNOSTICS

The following values are returned by this routine:

0	If no errors are detected.
1	Prints a message to standard error and returns 1 if it encounters an error which prevents changing a window or doing a shuffle, including trouble while expanding a *window_spec* pattern.
2	Prints a message to standard error, continues, and later returns 2 if it encounters any error while trying to change an attribute of a window.

NAME

wfont – load and activate fonts

SYNOPSIS

wfont [**-ar**] font_path [window_spec...]
wfont **-f** font_path [window_spec...]
wfont [**-F** base_font_path alt_font_path [window_spec...]]

DESCRIPTION

This command loads and activates the font file(s) specified by *font_path*, for the windows specified by *window_spec...* (by default, if you give no *window_spec*s, the window connected to standard input). See *windows*(1) for an explanation of *window_spec*.

Invoking the command with no arguments is a quick way to reset fonts to defaults for the standard input window. It behaves exactly as if you had typed:

```
wfont -F $WMBASEFONT $WMALTFONT
```

except that, if either variable is null or not defined, the same default is used as in *wcreate*(1).

The affected windows must be of type **term0** (or you get an error). If *font_path* does not begin with "/", "./", or "../", it is assumed to be relative to the directory specified by environment variable *$WMFONTDIR*, or the current directory if that variable is null or undefined.

By default, only the base font is changed for future writing of characters. Currently-displayed characters don't change appearance (there is no immediately visible effect).

Options are:

-a Load and activate *font_path* as the alternate font, not the base font, for the specified windows.

-r Replace the existing base (or with **-a,** alternate) font with *font_path* and repaint the windows. If any visible characters were written in the base (or alternate) font, they immediately change to the new font.

-f Flush all existing fonts for the specified windows, load *font_path* as the new base and alternate font, and repaint the windows. All currently displayed characters switch to the new font. You can't mix this option with any other.

-F Flush all existing fonts for the specified windows, load *base_font_path* as the new base font and *alt_font_path* as the new alternate font, and repaint the windows. If the current base and alternate are the same font, all currently displayed characters switch to the new base font. If the current base and alternate are different, all characters which were written using the current alternate font switch to the new alternate, and all others switch to the new base. You can't mix this option with any other.

If the *font_path*s given with the **-F** option are identical (after *$WMFONTDIR* is possibly prepended to each), the effect is the same as if you had used **-f** with only one *font_path*.

Normally, every font loaded must be the same cell size as those already loaded. The **-f** and **-F** options let you change the cell size. When this happens, window sizes may also be affected, so that the number of rows and columns displayable (and the window's logical screen size) remains unchanged.

Fonts are loaded into memory as needed (no more than eight at once). To avoid confusion, no font is loaded more than once (by font filename). All characters which use a given font use the same instance of it, so all change together when the font is replaced using the **-r** option. Therefore, you can't replace a font with any font which is already loaded (this would lead to it being loaded twice). You can make the already-loaded font the base or alternate font, but you can't use the **-r** option on it.

NOTE: If you need to simulate multiple loading of a font without doing library calls, you can link the font file to another filename, then load the two "different" fonts at the same time.

EXAMPLES

wfont
: Flush all fonts for the standard input window, set the base and alternate fonts to defaults, map all displayed characters to one of these fonts, and repaint the window.

wfont /usr/lib/raster/8x16/cour.8U
: Make the "cour.8U" font the base font for the window connected to standard input. It is loaded if not already in memory.

wfont -a cour.8U win5
wfont -ra lp.b.8U win5
: Make the font "cour.8U" the alternate font for the window "win5" (assume it is already loaded and there are characters which were written using it). Then make the font "lp.b.8U" the alternate font for the window, and repaint it. This has the effect of changing all the "cour.8U" characters to "lp.b.8U".

wfont -f 7x10/helv.8U
: Flush all fonts for the window connected to standard input and repaint it with all characters displayed using the font "7x10/helv.8U".

wfont -F lp.8U lp.b.8U win2
: Flush all fonts for the window "win2", make the base and alternate fonts "lp.8U" and "lp.b.8U" respectively, map all displayed characters to one of these fonts, and repaint the window.

HARDWARE DEPENDENCIES

Series 500:
: HP-15 (2-byte) fonts are not supported on Series 500. Two-byte fonts work only on Series 300.

SEE ALSO

windows(1),wcreate(1),wlist(1),wsh(1),altfont_term0(3W),basefont_term0(3W).

DIAGNOSTICS

The following values are returned by this command:

0 If no errors are detected.

1 Prints a message to standard error and returns 1 if invoked improperly, or if it has trouble while expanding a *window_spec* pattern.

2 Prints a message to standard error, continues, and later returns 2 if it encounters any error while trying to perform font operations for any one window.

WARNINGS

If any application program which calls window library routines previously ran in a window, it is possible that one or more fonts might be loaded multiple times. Thus, different characters displayed in the same font might not be changed together to a new font. You can avoid confusion by using **wfont -f** or **wfont -F** to flush all existing fonts before starting new operations.

wfont fails if you try to use **-f** or **-F** to switch to a smaller font in a window with a normal border and this would make the window size less than the minimum allowed with a normal border.

HP-15 fonts (2-byte fonts) cannot be loaded as the alternate font. Thus they cannot be used with the **-a** or **-f** options or as the second font of the **-F** option.

NAME

windows – window system concepts and commands

DESCRIPTION

There are a number of user-level commands which help you manage windows. Each is described separately. This manual entry covers concepts they share in common.

All numeric variables given to commands are limited to 16 bits, e.g. the (x,y) pair for *wmove(1)* must be in the range −16384 through +16383.

Window Types

The following window types are supported:

term0 Terminal level-0 emulation window. This type acts like an HP 2622 Terminal without block or format mode. It also supports the HP 2627 color escape sequences.

graphics

Starbase graphics window. This type implements the procedural interface defined in the *Starbase Device Drivers Library* and supports the Font Manager and Fast Alpha libraries.

The IMAGE graphics window is a special case for the HP98730 display system. This allows a user to create a graphics window which has the user area mapped to the image planes while the border is displayed in the overlay planes. This window type is useful for using the accelerated driver (HP98731) in windows where the user will have access to the 3D graphics functionality that is available to the raw (non-window) device with the obvious exception of managing shared resources like the color map and the hardware cursor. The IMAGE graphics window can also be used with the non-accelerated driver (HP98730). Use of the IMAGE graphics window type only requires the specification of the proper option or parameter at creation time. Thereafter, it behaves exactly the same as any normal graphics window. The number of IMAGE windows allowed at one time is 31.

see_thru

See_thru window used to view the image planes. This window type is not really a separate window type. It is a special case of the graphics window. Some Series 300 frame buffer devices support two sets of frame buffer memory. One set is called the image planes and is the area where most graphics output is sent; the other set is called the overlay planes and is where the human interface operations take place. The number of planes present in the overlay planes is device-dependent. If an overlay colormap is supported, the colors that can be set include combinations of red, green, blue, and *see_thru*. If a pixel in the overlay planes is *see_thru*, then the pixel color will be the color of the image planes "behind" the overlay planes; otherwise, the pixel color is forced to the color specified by the overlay planes. On this type of a display, the window manager runs in the overlay planes. A **see_thru** window is a non-retained graphics window whose background color is set to *see_thru*. A server, *stserver*, is started for the **see_thru** window. This server catches the SIGWINDOW signal for the repaint event. When a repaint event occurs, *stserver* repaints the window with the *see_thru* index value, causing the window to create a "peep-hole" through the overlay planes to the image planes. The *see_thru* color index value defaults to three (usually yellow), but can be changed either via the SB_OV_SEE_THRU_INDEX environment variable by setting this variable before starting the window manager, or by calling wset_see_thru(3W). The see_thru window type is used only by *wcreate*(1). For all other purposes, the see_thru window is considered a graphics window.

Window Specification

A particular window can be specified to the commands in several ways.

1. Give no *window_spec*s at all.
   In this case, some commands create a window with a default name obtained from the window manager, and others operate on the window connected to standard input, as appropriate. For example:

   wdisp -i
   : Put into iconic form the window connected to standard input.

2. Give "–" as a *window_spec*.
   This also means "default window name from the window manager" or "window connected to standard input", as appropriate. For example:

   wcreate win1 - win2
   : Create three windows named "win1", some default name, and "win2".

3. Give a basename or indefinite pathname.
   If *window_spec* does not begin with "/", "./", or "../", and it is not a pattern (see below), the window's special file resides in the directory specified by environment variable $*WMDIR*. If this variable is null or not defined, or if *window_spec* already begins with "$WMDIR/", its value is not prepended, and the window's location is relative to the window manager's current working directory. For example:

   wcreate wconsole
   : Create the (**term0**) window "$WMDIR/wconsole".

4. Give a definite pathname.
   If *window_spec* begins with "/", "./", or "../", $*WMDIR* is never prepended. For example:

   wcreate -wgraphics /dev/HP 9837A/sales_graph
   : Create the graphics window "sales_graph" whose special file lives in the directory "/dev/HP 9837A".

5. Give a shell-style pattern.
   Window names may be specified using meta-characters, similar to *sh*(1) and *find*(1) (but different than *ed*(1)). Be sure to quote patterns to hide them from the shell. Unlike *find*, which always compares patterns with basenames only, the appearance of '/' requires a pattern to match against full window pathnames (as known to the window manager) rather than basenames. For example:

wsize '*'	Set default sizes for all windows, wherever they are located.
wlist '/*3'	List information on all windows whose full pathnames start with slash and end with '3'.

   Pattern meta-characters are '*', '?', and '['. Following '[', special meanings are also assigned to '!', '–', and ']'. Any character, including '\', may be quoted (made to stand for itself) by preceding it with '\'.

   Patterns are compared against the names of all existing windows known to the window manager. If a pattern contains a '/' anywhere, it must match a window's full pathname. Otherwise, it is only compared against each window's basename. Either way, the result is the alphabetically sorted list of full pathnames of selected windows, or the pattern itself if no window matches.

   Meanings of pattern meta-characters:

*	Matches any string, including the null string.
?	Matches any single character.

[...]	Matches any one of the enclosed characters. A pair of characters separated by '–' matches any character lexically between the pair, inclusive.
[!...]	A NOT operator can be specified immediately following the left bracket to invert the sense of the comparison, i.e. match any single character not enclosed (explicitly and/or in a range) in the brackets. (Note, this is different syntax than the '^' used in regular expressions.)

Like the shell, "[]" matches no character. Unlike the shell, "[!]" matches any character, and '–' in a list is not taken as part of a range unless used properly.

Also note:

Window basenames are truncated to 12 characters (not 14).

Window pathnames are limited to 37 characters after possibly prepending $*WMDIR*, and/or expanding "./" or "../" to the current working directory or its parent, respectively. If you try to refer to a window with a final pathname longer than 37, you get an error from the commands.

Font file pathnames are similarly truncated (to 39), but no check is performed. Attempts to use longer final pathnames normally fail (no such file).

Summary of Command Options

This table tersely summarizes the options recognized by the window commands. Note that commands which do not have any options are not shown in the table (i.e., *wmstart*(1), *wselect*(1), *wmstop*(1)).

	Options and Commands									
option	bordr	creat	destr	wdisp	wfont	wlist	wmove	wsh	wsize	wmrdy
−a	-	-	adest	-	alt	-	-	adest	-	-
−b	-	bottm	-	bottm	-	-	-	bottm	-	-
−cf,b	color	-	-	-	-	-	-	-	-	-
−cxxx	-	-	-	-	-	-	-	cmdln	-	-
−d	-	-	wdest	down	-	-	-	wdest	-	-
−e	-	-	-	-	-	-	-	exist	-	-
−f	-	-	-	-	flush	fonts	-	-	-	-
−F	-	-	-	-	flush	-	-	-	-	-
−g	-	-	-	-	-	-	-	login	-	-
−i	-	icon	-	icon	-	-	icon	icon	-	-
−k	-	keybd	-	-	-	-	-	keybd	-	-
−l	-	-	-	-	-	long	-	-	-	-
−lx,y	-	locat	-	-	-	-	locat	locat	-	-
−lxxx	label	-	-	-	-	-	-	-	-	-
−M	-	byte	-	-	-	-	-	byte	-	-
−m	-	bit	-	-	-	-	-	bit	-	-
−N	-	IMAGE	-	-	-	-	-	IMAGE	-	-
−n	norml	noret	nodst	norml	-	-	-	noret	-	-
−o	-	concl	-	concl	-	-	-	concl	-	-
−r	-	-	-	-	replc	-	-	-	-	-
−rw,h	-	rsize	-	-	-	-	-	rsize	-	-
−sw,h	-	size	-	-	-	-	-	size	size	-
−T	bdrls	bdrls	-	-	-	-	-	bdrls	-	-
−t	thin	thin	-	top	-	-	-	thin	-	-
−u	-	-	-	up	-	-	-	-	-	-
−v	-	verbo	-	-	-	-	-	verbo	-	verbo
−wxxx	-	type	-	-	-	-	-	type	-	-

HARDWARE DEPENDENCIES

Series 500:

IMAGE graphics windows (–**N** option) are not supported on Series 500, and only on HP98730 for Series 300.

Byte/pixel is specified by the absence of –**n**, not the presence of –**M**. Bit/pixel (–**m**) is not supported.

Borderless windows (–**T** option) are not supported on Series 500.

The **see_thru** window type is supported only on the HP 98720 and HP 98730 for Series 300.

HP-15 (two-byte) fonts are not supported on Series 500; they can be used only on Series 300.

SEE ALSO

wmstart(1), wmready(1), wmstop(1), wborder(1), wcreate(1), wdestroy(1), wdisp(1), wfont(1), wlist(1), wmove(1), wselect(1), wsh(1), wsize(1), wset_see_thru(3W).

DIAGNOSTICS

All of the commands try to give helpful error messages. When an error is detected in a call to a system or Windows routine, the commands also print and interpret the value of *errno* (see *errno*(2)). They tell you the name of the routine which returned the error so you can look in the appropriate manual entry for more information.

NAME

wlist – list status of windows or fonts

SYNOPSIS

wlist [–**fl**] [window_spec...]

DESCRIPTION

This command lists information about currently existing windows or their loaded fonts. It only knows about the windows and fonts associated with the one invocation of the window manager (*wm*) which supports the same physical display.

By default, *wlist* prints to standard output the full names of the given windows, or the one connected to standard input if you give no *window_spec*s, one name per line. See *windows*(1) for an explanation of *window_spec*. (Note in particular that you can use '–' to list the window connected to standard input along with others, and '*' to list all windows.)

Options are:

–**f**	List information about fonts currently loaded for the specified windows, which should be of type **term0** (other types appear to have no fonts loaded). Prints the full name of each window, followed by a colon, followed by the full pathnames of all fonts loaded for that window, one name per line.
–**l**	List (long-form) all available and useful information about each window or font. The output formats are described below.

To get information on font files, whether or not they are currently loaded, use *file*(1).

Long-form Output Format

For fonts (–**f** option), each font pathname is preceded by a cell size (pixel width x height) and an activation indicator:

b/a	active as both base and alternate font
base	active as base font
alt	active as alternate font
–	font loaded but not active

For windows, **wlist** prints a title line followed by one line of data for each window. Field titles and values are:

WT Window type:

t0	term0
gr	graphics
gi	IMAGE graphics

K Keyboard attached (window selected):

–	not selected
k	selected

D Display status:

t	top
b	bottom
–	displayable, but not top or bottom
c	concealed offscreen

T Border style:

–	normal
t	thin
T	no border

I Iconic:

–	normal
i	iconic

A Auto Destroy:

–	normal
a	autodestroyable and recoverable
d	recoverable but not autodestroyable

LOCX LOCY
X,Y locations of window's normal state (pixels).

WIDE HIGH
Window's width, height (type-dependent units).

PANX PANY
Window's pan X,Y offsets (type-dependent units). They appear as "**?**" for **term0** windows because that window type does not support panning.

RASW RASH
Window's raster/buffer height, width (type-dependent units).

ILCX ILCY
X,Y location of window's iconic state (pixels).

FGC BGC
Window's foreground, background border colors (indices).

WINDOW
The window's basename.

The output format is carefully arranged so that, even with the longest window basename (12 characters), each output line just fits on one 80-character display line without wrapping. If any numeric value is larger than five digits (with possible sign), the line may wrap around.

EXAMPLES

wlist List the full name of the window connected to standard input.

wlist –f win5
List information on fonts loaded for window "win5".

wlist –l List all available information about the window connected to standard input.

The final example checks whether standard input is a window and prints a message accordingly:

```
if wlist > /dev/null 2>&1
then
   echo "stdin is a window"
else
   echo "stdin is NOT a window"
fi
```

SEE ALSO

windows(1), file(1), wborder(1), wcreate(1), wdestroy(1), wdisp(1), wmove(1), wselect(1), wsh(1), wsize(1).

DIAGNOSTICS

This command returns the following values:

0 If no errors are detected.

1 Prints a message to standard error and returns 1 if it encounters an error which prevents listing any information, including trouble while expanding a *window_spec* pattern.

2 Prints a message to standard error, continues, and later returns 2 if it encounters any error while trying to list information for one window. No information is printed for the affected window (or its fonts).

NAME

wmove – move one or more windows or their icons

SYNOPSIS

wmove [**-i**] [**-l** x,y] [window_spec...]

DESCRIPTION

This command changes the location on the display of the specified windows or their iconic representations, while keeping their sizes and other attributes intact. See *windows*(1) for an explanation of *window_spec*. By default, if you give no *window_spec*s, the window connected to standard input (not its icon) is moved to a default location, with its upper left corner near the upper left corner of the screen. The default location for icons is near the lower left or upper right corner of the screen, depending on your configuration.

Options are:

-i Move windows' iconic representations (icons), not their normal forms. See *windows*(1) for an explanation of icons.

-l *x,y* Move windows' upper left corners to the pixel location given by *x,y*. If you don't give a location, each window or icon is moved to a slightly different location.

The results of *wmove* may not be immediately visible if a window is:
* concealed
* located off-screen
* occluded by others
* normal and its icon is moved
* iconic and its normal form is moved.

EXAMPLES

wmove Move the window connected to standard input to a default location, near the upper left corner of the screen.

wmove -il100,200 win2 win3
Move the icon form of windows "win2" and "win3" to pixel location x = 100, y = 200.

SEE ALSO

windows(1), wborder(1), wcreate(1), wdestroy(1), wdisp(1), wfont(1), wlist(1), wselect(1), wsh(1), wsize(1), wmove(3W).

DIAGNOSTICS

The following values are returned by this command:

0 If no errors are detected.

1 Prints a message to standard error and returns 1 if it encounters an error which prevents moving a window, including trouble while expanding a *window_spec* pattern.

2 Prints a message to standard error, continues, and later returns 2 if it encounters any error while trying to move one window.

NAME

wmready – tell if window manager is awake and ready

SYNOPSIS

wmready [–v]

DESCRIPTION

This command reports on whether the window manager (*wm*) is currently awake, running, and ready to accept requests. It uses the *wm* special file in the directory specified by the environment variable $*WMDIR*. It can be used to wait for *wm* startup, or to check if the window system is already running.

The only option is:

-v Verbose operation: Print a message to standard output in addition to returning a value.

wmready says *wm* is not ready if the *wm* special file can't be opened for read/write within two seconds (using *alarm*(2) for timeout), or if a simple *wm* request fails. If *wm* is "ready" but merely busy (e.g. with a pop-up menu), the command blocks on the request for an indefinite time, but may eventually indicate *wm* is ready.

RETURN VALUE

0	the window manager is ready,
1	the window manager is **not** ready.

SEE ALSO

windows(1), wmstart(1), wmstop(1), alarm(2).

NAME

wmstart – start the window system on one display

SYNOPSIS

wmstart [**optional_args**]

DESCRIPTION

This command initiates the window system for one display, normally the one from which it is invoked. First it sets environment variables as needed (see below). It does not change your $*PATH* variable or your current working directory. It checks that no window manager is currently running using the directory specified by environment variable $*WMDIR*. If the variable is not null, i.e. the window special files directory is not the current directory, and it is also not set to "/" or **/dev**, it attempts to remove any character special files found in or below the specified directory.

wmstart starts the window system, which blocks input to (but not output from) the internal terminal emulator (ITE). The window manager (*wm*) process is started and, if no argument is specified to *wmstart*, it calls *wsh*(1) to create an initial window (named *wconsole*) with a shell attached.

wmstart , is a shell script that normally resides in **/usr/bin**. You can study and modify *wmstart* if so desired. Personal copies of it can be tailored to the specific hardware being used. Desired applications or window commands can be initiated after the window system is up.

Note: If you customize *wmstart*, change a copy, not the original script, so your changes are not lost at the time of system software update. Then execute your copy to start the window system.

The *wmstart* process does not terminate until operation of the window system is terminated. Thus the caller (e.g., *init*(1M) or *sh*(1)) can wait for it to terminate. See below for examples.

Environment Variables

wmstart sets necessary environment variables to default values if undefined or null. Only those are set which must be defined for *wm* or the window commands to work properly. Those already in the environment are not altered (except for $*TERM*), so you can pass in values from outside the window system. Note that you can (in *sh*(1)) set them on the command line which calls *wmstart*, for example:

```
WMDIR=/dev/screen1 wmstart
```

In particular, the $*WMDRIVER* variable tells the window system what type of display you have. If it is undefined or null, *wm* (not *wmstart*) sets it automatically to the right type. If you set it explicitly to the wrong type, you may get strange results.

Here is the complete list of variables. For each variable, the default value (if any) set up by *wmstart* is indicated along with the default value that the window system uses if the variable is undefined or null.

WMDIR

Directory where window device files are put by the window manager. Default: set to **/dev/screen**. Typical values might be: **/dev/screen1**, **/dev/screen2**, etc. If *wm* is called with it undefined or null, "$WMDIR/" is never prepended to a *window_spec*. See *windows(1)* for more on *window_spec*s.

WMSCRN

Device file of the physical display where windows appear. Default: not set by *wmstart*; if undefined, *wm* uses **/dev/crt**, unless the device is HP98730 in which case **/dev/ocrt** is used. Typical values might be: **/dev/crt9837**, **/dev/crt98700**, etc.

WMDRIVER
Starbase driver used to write to the display when windows is running. Default: not set by *wmstart*. If this variable is undefined, then *wm* uses the appropriate driver for the window system running on your display. For example, on Series 300 low- or medium-resolution displays, *wm* uses "hp300l" for the driver.

SB_DISPLAY_ADDR
Memory address in the user address space of the Starbase display device. Default: not set by *wmstart*; if undefined, *wm* uses "0xb00000". Since this is normally a global entity, it should be set and exported from **/etc/profile and /etc/csh.login**. Should different instances of the variable have different values, unpredictable errors may occur.

WMKBD
Device file of the keyboard. Default: set to **/dev/hilkbd**. If *wm* is called with it undefined or null, keyboard input is disabled.

WMINPUTCTLR
Device file of the input controller. Default: set to **/dev/rhil**. If *wm* is called with it undefined or null, keyboard input is disabled.

WMLOCATOR
Device file of the locator. Default: set to **/dev/locator**. If *wm* is called with it undefined or null, locator input is disabled.

WMLOCSCALE
Normally, when the graphics tablet is used as a locator device, the entire graphics tablet maps to the entire screen. By setting this variable, you can map a rectangular area on the graphics tablet to the entire screen, i.e., you can specify that only part of the graphics tablet map to the display screen. This variable requires four coordinates and is set as follows:

WMLOCSCALE="x1 y1 x2 y2"

where *x1,y1* are coordinates on the graphics tablet that correspond to the lower-left corner of the display screen; *x2,y2* correspond to the upper-right corner of the screen.

Coordinates can be specified as either absolute or percentage. Absolute coordinates require knowledge of the graphics tablet resolution; percentage coordinates do not.

Absolute coordinates give an absolute address on the graphics tablet; percentage coordinates give a location with respect to the entire tablet and have a trailing percent sign.

Absolute and percentage coordinates can be mixed. For example, the following causes the display screen to map to the lower-left quadrant of the graphics tablet:

```
WMLOCSCALE="0 0 50% 50%"
```

WMPTYMDIR
Directory where master pseudo-tty (pty) special files are located. Default: not set by *wmstart*; if undefined, *wm* uses **/dev/ptym**.

WMPTYSDIR
Directory where slave pseudo-tty (pty) special files (generic window names) are located. Default: not set by *wmstart*; if undefined, *wm* uses **/dev/pty**.

WMPTYNAME
Starting name of the set of pseudo-ttys (ptys) used for windows. Must follow the pty naming convention: tty[p–v][0–f]. Default: not set by *wmstart*; if undefined, *wm* uses "ttyp8".

WMPTYCNT

Number of contiguous pseudo-ttys (ptys) used by the window manager. Default: not set by *wmstart*; if undefined, *wm* uses "31". Note: The number of open files allowed per process sets an upper limit on this value. Each graphics window type requires one pty and each Term0 window type requires three ptys.

WMPTYCACHECNT

Maximum number of pseudo-ttys (ptys) that the window manager is permitted to keep pre-opened for performance purposes, and thus unusable by other applications. Default: not set by *wmstart*; if undefined, *wm* uses "10". Note: This pty cache applies only for graphics window types or two of the three ptys used by Term0 window types.

WMSHMSPC

Maximum size of shared memory used by window manager. Increments of 0x1000 are strongly recommended. Default: set to "0x200000". Minimum size is 0x20000 even if set to a smaller value.

WMIATIMEOUT

The purpose of this variable is two-fold:

1. It specifies the timeout period (in seconds) for interactive operations, and
2. It determines the number of milliseconds the window manager will not process locator changes. This allows other processes to run when the window manager is tracking.

Specifying Timeout

The two least-significant bytes of this variable (0xFFFF) specify the number of seconds of absolute inactivity in the locator that the window manager will allow during an interactive operation.

If this value is not set, or is less than or equal to zero or null, then it defaults to 60 seconds.

Tracking

The window manager reads information from the locator whenever its position changes; this is known as *tracking*. If the position changes a lot, the window manager (because it is a high-priority process) may nearly consume the CPU, thereby preventing other processes from running. If locator movement tracking is also being done by one or more other processes, it becomes jerky or stops. To more fairly allow other processes to run, the window manager temporarily ignores the locator for a short period of time, thus allowing the other processes to run.

You can control the length of time that the window manager ignores locator information during tracking. Valid values range anywhere from 0 to 255 milliseconds. Values are specified in the third byte of this environment variable (0xFF0000).

If no value is specified or 0 is specified, it defaults to 30 milliseconds.

Values from 1 to 254 can be used. Keep in mind: (1) as this number becomes lower, the echo tracks better on the screen, but user processes can't track as well; (2) as this number becomes higher, the echo tracks worse on the screen, but user processes track the locator as least as well as the window manager.

Note that although time can be specified in one-millisecond increments, the Series 200/300 clock "clicks" every twenty milliseconds. Therefore, you might want to specify time in 20-millisecond increments on Series 200/300.

If the value is 255, then locator information is not ignored.

WMCONFIG

Window manager configuration, the OR of:

0x07 Enable window manager process locking: 0x0 default is for no locking, 0x1 is text only, 0x2 is data only, and 0x4 is shared memory locking. Any combination of the three bits is allowed. This can be used when lots of physical memory is present, the window system is competing with very large processes, and snappy window system performance is still desired. If shared memory locking is enabled, setting WMSHMSPC to the minimum value needed is advisable.

0x08 Enable clear of graphics retained raster unconditionally upon create of a graphics window. The default of 0 is much faster because the clear is delayed until the window is made visible or graphics is done in the window. This enable is for compatibility purposes in the case of programs that meet all of the following: linked prior to 5.2 HP-UX, drawing in a concealed window that the same program did not create, and depend on the window being cleared prior to any gopen.

0x10 Enable double buffering color mode. This causes all colors used in window borders, window icons, softkeys, desktop, popup menus, and term0 window text to be modified so that the visible color for these will not change whenever the display-enabled planes are modified by a Starbase program using double buffering. All colors written to the display are first converted to (C << (N/2) + C), where C is the color being written and N is the number of planes on the display. The window manager will modify the color map so that color indices (C << (N/2)) and (C << (N/2) + C) have the same RGB values as color index C.

Enabling double buffering color mode reduces the effective number of colors from 2^N to 2^(N/2). The window system will force all colors set via environment variables to be within the allowed range.

Double buffering color mode may be enabled only on color displays with 6 or more planes.

0x20 Force the use of software sprites (pointers) which implies that, on displays which provide hardware support for sprites, hardware will not be used for window system sprites. So, when this bit is set, all sprites will be rendered using software. The default of 0 will provide better performance, but at the cost of not being able to represent the full range of raster echo sprites. This enable is for compatibility purposes in the case of programs that use sprites that have more than two color index values, and require that those sprites be displayed exactly as defined. It is also appropriate for an application that wants exclusive access to the hardware sprite.

On the HP 98730 display, the hardware is capable of displaying only two colors for sprites. However, raster echos may contain color index values of up to eight bits in depth (0-255). When the window system is using the hardware for sprites (the default), a conversion must take place to convert the raster echo from n colors to 2 colors. This conversion will make all bgcolor index values in the raster echo be displayed as the bgcolor, and all non-bgcolor index values be displayed as the fgcolor. When the window system is using the software, the raster echo will be displayed as specified. The window's border sprite colors are set respectively to the border's foreground and background color. For all other window sprites, if no call has been made to **wset_hw_sprite_color**, the defaults are fgcolor = 1 and bgcolor = 0.

Default: not set by *wmstart*; if undefined, *wm* uses "0x00", i.e. don't lock the window manager process or shared memory, allow higher performance delayed clear of graphics windows upon create, don't enable double buffering color mode and use the hardware cursor.

WMIUICONFIG

Interactive user interface configuration, the OR of:

0x00007f	Enabled buttons; button 1 is least significant bit.
0x000080	Enable Select key.
0x000100	Top if select over window.
0x000200	Top if going icon/normal.
0x000400	Top if unobscured move/size.
0x000800	Top if obscured move/size.
0x001000	Disable move manipulation symbol.
0x002000	Disable icon/normal manipulation symbol.
0x004000	Disable size manipulation symbol.
0x008000	Disable pause/resume manipulation symbol.
0x010000	Disable pan manipulation symbols.
0x020000	Disable popup menu in border.
0x040000	Disable popup menu over desk top.
0x080000	De-select if going to icon.
0x100000	Icon default position base: 0 == lower left; 1 == upper right.
0x200000	Disable tiler alignment by interactive operations.
0x400000	If set, it lets the sprite (echo) move outside the menu without aborting the pop-up menu operation.
0x800000	If set, interactively changing a window from an icon to normal representation will automatically select the window.
0x1000000	If set, disables the cache of a pop-up window create.
0x2000000	If set, serializes the pop-up window creates so that you cannot do two or more creates simultaneously.
0x4000000	If set, disables audio feedback that an interactive operation was somehow aborted.
0x8000000	If set, reverses the sense of the border arrows for graphics windows whenever the scroll bars are set to pan mode. The sense of the border arrows for term0 windows is also reversed.

Default: not set by *wmstart*; if undefined, *wm* uses "0x80781", i.e. button 1 (left mouse button), Select key, top if select over window, top if going icon/normal, top if unobscured move/size, and de-select if going to icon.

WMRTPRIORITY

Real time priority for window manager and servers. First byte is real time priority for window manager, second byte for servers. Range for each is 0 (highest) to 127 (lowest). If out of range, real time priority is disabled. Default: not set by *wmstart*; if undefined, *wm* uses "0x787c", i.e. 120 for window manager, 124 for servers.

WMDESKPTRN

Dither pattern for desktop; value ranges from 0 to 100. Significant values are 0, 25, 50, 75 and 100; given value is rounded to nearest significant value. Zero means desktop is solid in the desk background color; 100 means solid in the desk foreground color. Default: not set by *wmstart*; if undefined, *wm* uses "50".

WMDESKFGCLR
: Foreground color for the desk top. Default: not set by *wmstart*; if undefined, *wm* uses "0" (black, unless the color map is changed). If out of range for device, forced within range.

WMDESKBGCLR
: Background color for the desk top. Default: not set by *wmstart*; if undefined, *wm* uses "1" (white, unless the color map is changed). If out of range for device, forced within range. Note: If $*WMDESKBGCLR* == $*WMDESKFGCLR*, $*WMDESKBGCLR* is treated if set to the complement of the least significant bit of the foreground color, i.e. black or white unless the color map is changed.

WMBDRFGCLR
: Initial foreground color for window borders. Default: not set by *wmstart*; if undefined, *wm* uses $*WMDESKFGCLR*. If out of range for device, forced within range.

WMBDRBGCLR
: Initial foreground color for window borders. Default: not set by *wmstart*; if undefined, *wm* uses $*WMDESKBGCLR*. If out of range for device, forced within range. Note: If $*WMBDRBGCLR* == $*WMBDRFGCLR*, $*WMBDRBGCLR* is treated if set to the complement of the least significant bit of the foreground color, i.e. black or white unless the color map is changed.

WMMENUFONT
: Font used for pop-up menus. Default: not set by *wmstart*; if undefined, *wm* uses **/usr/lib/raster/dflt/b/v/***language* for the very high resolution display, **/usr/lib/raster/dflt/b/h/***language* for the high resolution display, or **/usr/lib/raster/dflt/b/l/***language* for the low resolution display where, *language* is the value of the $*LANG* environment variable. If $*LANG* is not defined or an entry for the language does not exist, it uses **/usr/lib/raster/10x20/lp.b.8U** for the very high resolution display, **/usr/lib/raster/8x16/lp.b.8U** for the high resolution display, or **/usr/lib/raster/6x8/lp.8U** for the low resolution display.

WMSFKFONT
: Font used for softkey labels. Default: not set by *wmstart*. If this variable is undefined, *wm* uses **/usr/lib/raster/dflt/b/v/***language* for the very high resolution display, **/usr/lib/raster/dflt/b/h/***language* for the high resolution display, and **/usr/lib/raster/dflt/b/l/***language* for the low resolution display, where *language* is the value of the $*LANG* environment variable. If $*LANG* is not defined or an entry for the language does not exist, it uses **/usr/lib/raster/10x20/lp.8U** for the very high resolution display, **/usr/lib/raster/8x16/lp.8U** for the high resolution display, or **/usr/lib/raster/6x8/lp.8U** for the low resolution display.

ICONFONT
: Font used for icons. Default: not set by *wmstart*; if undefined, *wm* uses **/usr/lib/raster/dflt/b/v/***language* for the very high resolution display, **/usr/lib/raster/dflt/b/h/***language* for the high resolution display, or **/usr/lib/raster/dflt/b/l/***language* for the low resolution display where *language* is the value of the $*LANG* environment variable. If $*LANG* is not defined or an entry for the language does not exist, it uses **/usr/lib/raster/6x8/lp.8U** for all displays.

BANNERFONT
: Font used in window borders. Default: not set by *wmstart*; if undefined, *wm* uses **/usr/lib/raster/dflt/b/v/***language* for the very high resolution display, **/usr/lib/raster/dflt/b/h/***language* for the high resolution display, or **/usr/lib/raster/dflt/b/l/***language* for the low resolution display, where *language* is the

value of the *$LANG* environment variable. If *$LANG* is not defined or an entry for the language does not exist, it uses *$WMMENUFONT*.

WMFONTDIR
: Directory under which font files are located. Default: set to **/usr/lib/raster**.

WMBASEFONT
: Default font to load as base font for newly-created **term0** windows. Default: set to **/usr/lib/raster/dflt/b/v/***language* for the very high resolution display, **/usr/lib/raster/dflt/b/h/***language* for the high resolution display, or **/usr/lib/raster/dflt/b/l/***language* for the low resolution display where *language* is the value of the *$LANG* environment variable. If *$LANG* is not defined or an entry for the language does not exist, it defaults to **/usr/lib/raster/10x20/lp.8U** for very high resolution display, or **/usr/lib/raster/8x16/lp.8U** for high resolution display; for Series 300 low resolution displays, it defaults to **/usr/lib/raster/6x8/lp.8U**.

WMALTFONT
: Default font to load as alternate font for newly-created **term0** windows. HP-15 (2-byte) fonts cannot be used for the alternate font. Default: set to **/usr/lib/raster/dflt/a/v/***language* for the very high resolution display, **/usr/lib/raster/dflt/a/h/***language* for the high resolution display, or **/usr/lib/raster/dflt/a/l/***language* for the low resolution display where *language* is the value of the *$LANG* environment variable. If *$LANG* is not defined or an entry for the language does not exist, it defaults to **/usr/lib/raster/10x20/lp.b.8U** for very high resolution displays or **/usr/lib/raster/8x16/lp.b.8U** for high-resolution displays; for Series 300 low-resolution displays, it defaults to **/usr/lib/raster/6x8/lp.b.8I.**

KJINPUTFONT
: Font used in the Kanji input window for **term0** windows. *KJINPUTFONT* is used only by the Japanese *t0server* which supports the Kanji input method. Default: not set by *wmstart*; if undefined, *$WMBASEFONT* is used.

TERM Always set to "hp9836".

Window System Context

There are a number of different ways you can start up and exit the window system.

1. Directly from init to wmstart

This is appropriate for a single-user (single-display), single-uid system, or one with limited flexibility (always same user on each display; no security). You can go through *su*(1) if desired to run the window system's processes as other than super-user (for safety, if nothing else). *Init* can be told to restart *wmstart* if the window system terminates. No login status is recorded. Extra work is needed in inittab to set up environment variables, etc.

1.1. wmstart from init state 1

Wakes up immediately when the system is booted. May be dangerous if *fsck*(1M) is needed, because lots of file system activity occurs in the process of waking up the window system, which could corrupt a disc further. Greatly increases the dependence on the health of the system to get a shell, compared to vanilla getty/login.

1.2. wmstart from init state 2

The system wakes up a single-user (super-user) shell for fsck, on the ITE or on a normal terminal. May or may not require login while in state 1 (it's up to you). The "init 2" command is given by the super-user when ready to start the window system. The single-user shell may be automatically killed, or the super-user may have to exit it first.

2. From init via getty

Appropriate for a single- or multi-user (multi-display), multi-uid system. Not every terminal need be a bit-mapped display which supports windows. *Getty* and *login* run using the ITE. Users must log in normally. This can be done from init state 1 (wake up multi-user), or init 2 (wake up single-user, then switch).

2.1. User's home shell is wmstart

Windows start up automatically after login. All terminals must be bit-mapped displays, or window users may only log in on window devices (other users may or may not, at their choice). It's difficult to customize the environment per user without modifying *wmstart*. Doing a *wmstop* leaves you logged out.

To put *wmstart* in **/etc/passwd** as a login shell, the following must be done:

1. Make a custom version, and name it something that does **not** contain the letter **r**, for example, "wmstat".
2. In the custom version, explicitly set *SHELL* to the value desired:

   SHELL="bin/sh"; export SHELL
3. Put the name of this custom version in **/etc/passwd**.

2.2. Home shell is sh(1) or csh(1)

This allows you to customize the window environment for all users. You must set up the ITE environment first (e.g. using *tset*(1)).

2.2.1. Run wmstart from /etc/profile

All users get windows (all terminals are bit-mapped displays). Windows start up automatically after login. No per-user customization is possible (global only). Only works for *sh*(1) (until *csh*(1) supports global initialization).

2.2.2. Run wmstart from .profile or .login

Per-user customization and control is possible. Windows start up automatically after login. This can be made conditional on terminal type (bit-mapped display or not).

2.2.3. Run wmstart manually

You get a normal shell on the ITE after login. You can reinvoke *wmstart* to restart windows after exiting. This provides full user control, but windows aren't automatic.

For methods 2.2.1 – 2.2.3, there are two ways to start the window system:

2.2.A. Run as subprocess ('wmstart')

ITE shell waits for termination (an extra process). After *wmstop*, the user is still logged in to the ITE shell, and can manually restart windows.

2.2.B. Execute directly ('exec wmstart')

The ITE shell is replaced by the wmstart process; there is no waiting shell process. After *wmstop*, the user is logged out.

Note that clever combinations of *init*(1) and *wmstart* are possible. For example, you could require login to a single-display, multi-user system, then run *wmstart*, then switch to an init state which, for example, restarts a shell on the console window (wconsole) each time the old one terminates.

Suppose you want the window system to terminate when the first shell exits. To accomplish this, you can execute "trap wmstop 0" in that shell. To make this automatic, change a copy of *wmstart* so it invokes *wsh* with the **–g** (login shell) option, and put a .profile file containing the command in your home directory.

SEE ALSO

windows(1), wmready(1), wmstop(1), wsh(1), umask(1), rtprio(2), plock(2), pty(4).

DIAGNOSTICS

In general *wmstart* does not check for errors. If anything goes wrong, various commands may write to standard error, but the script might not terminate. If $*PATH* is strange, some commands may not be found. If the window manager is not ready after about 20 seconds, *wmstart* prints a message to standard error and quits waiting to start an initial window and shell.

BUGS

If you invoke *wmstart* from the ITE in the background (with "**&**"), the ITE shell does not wait. Instead it puts out a prompt before blocking, waiting for the next command line. Depending on timing, this prompt may appear on the windows screen, and can only be erased by repainting.

NAME

wmstop – stop the window system on one display

SYNOPSIS

wmstop

DESCRIPTION

This command stops the window system for one display, normally the one from which it was invoked. It may be called from any process connected to any window in that window system.

wmstop actually calls a library routine (*wmkill*(3W)) to stop the window system whose window manager's special file is in the directory specified by the environment variable $*WMDIR*. Calling the library routine causes the window manager to terminate gracefully, destroying all windows and clearing the screen.

After *wm* terminates, a signal (SIGHUP) propagates to all server processes for windows in the window group, thence to user processes in the group. This normally causes them to also exit (gracefully or not). Processes which ignore the signal (such as those started with *nohup*(1)) may continue to run, but their output is lost or overwrites portions of the screen asynchronously unless it was redirected away from a window. Meanwhile, the internal terminal emulator (ITE) regains control of the keyboard for input.

SEE ALSO

windows(1), wmstart(1), wmready(1), wmkill(3W).

DIAGNOSTICS

The following diagnostic values are returned by this routine:

0 If no errors are detected.

1 Prints a message to standard error and returns 1 if it can't open the special file (within two seconds), or if the library call fails. If invoked from a process connected to a window, and if successful, *wmstop* may get a signal and terminate before returning 0.

NAME

wselect – connect a window to the keyboard

SYNOPSIS

wselect [window_spec]

DESCRIPTION

This command makes the specified window the selected window, that is, the one attached to the keyboard and the buttons on the optional mouse or stylus switch on the optional tablet. By default, if you give no *window_specs*, the window selected is the one which is the standard input to *wselect*. See *windows*(1) for an explanation of *window_spec*.

The selected window is highlighted by having a line through the center of its border.

EXAMPLES

wselect Select the window connected to standard input. If it was not already selected, this can only be done from a shell script.

wselect win8
Attach the keyboard to window "win8".

SEE ALSO

windows(1), wborder(1), wcreate(1), wdestroy(1), wdisp(1), wfont(1), wlist(1), wmove(1), wsh(1), wsize(1), wautoselect(3W), wselect(3W).

DIAGNOSTICS

The following diagnostic values are returned by this routine:

0 If the routine is succesfull.

1 Prints a message to standard error and returns 1 if any error is detected, such as bad invocation, standard input is not a window, can't open the window, etc.

NAME

wsh – create new shells in new or existing windows

SYNOPSIS

wsh [**–w** type] [**–kboiTtMmNnv**] [**–l** x,y] [**–s** w,h] [**–r** w,h] [**–gad**] [**–c** cmdline] [window_spec...]

wsh –e [**–gad**] [**–c** cmdline] window_spec...

DESCRIPTION

This command creates new windows with the given names, sets up reasonable environments in them (see below), and starts shells associated with them. If you don't give *window_spec*, *wsh* creates one window with a default name. See *windows*(1) for an explanation of *window_spec*.

The shell program used is defined by the *$SHELL* environment variable. It can be a full pathname, or be relative to one of the directories in *$PATH*. If *$SHELL* is null or not defined, the default shell is **/bin/sh**.

By default (no **–e** option), *wsh* creates all the needed windows at once, before creating any shells.

The options for *wsh* are identical to those for *wcreate*, with the addition of:

–e Create shells and attach them to existing windows (or terminals...), which must not be already affiliated to any program(s). With this option, none of the options normally passed to *wcreate* are allowed, and you must supply at least one *window_spec*.

–g Make the new shell for each window a login shell. The literal effect is to prepend a "–" to the first argument (arg0, program basename) passed to the shell program. This causes some programs to do extra initialization, e.g. **/bin/sh** runs **/etc/profile** and $HOME/**.profile.**

–c Give the new shell for each window a command line to execute. This option and its argument are passed (as two separate arguments) to each new shell.

–a Causes the window to be automatically destroyed when its device interface (special file) is closed by every process that has opened it. See *wdestroy*(1) for details on the **–a** option.

–d Causes the window to be automatically destroyed when its device interface (special file) is closed by every process **and** only when a new window has been created. See *wdestroy*(1) for details on the **–d** option.

WARNING: Don't give the **–c** option unless the shell program recognizes it.

If you do not give a *window_spec*, *wsh* calls the window manager to pick a name for you before calling *wcreate*.

wsh waits until each shell is successfully executed from a child process before starting the next one.

Side Effects

wsh sets up a reasonable user environment for each shell. It performs only those actions which most users need, leaving all other initialization under user control.

wsh adds an entry to **/etc/utmp** for term0 windows. This means that *who*(1) will show the user as being logged in on a *pty* for every window shell still active.

Each new shell is a process group leader, affiliated to its window, with its standard input, output, and error connected to the window, opened for read/write, and with all other files closed. *wsh* checks affiliation by opening (and closing) **/dev/tty**. It refuses to start an unaffiliated shell, e.g., if there is already a shell connected to the window.

wsh sets signals SIGHUP, SIGINT, and SIGQUIT to SIG_DFL (see *signal*(2)). The first is necessary in case the program was started with *nohup*(1); each new shell must be able to receive SIGHUP from its window's server. The latter two are necessary in case the program was started in the background with "**&**"; they are safe because each new shell is affiliated to a new window.

wsh sets the window's special file permissions to 0622. (Ownership must already be set to the process's userid for this to succeed. Normally this is done automatically by *wcreate*.)

wsh calls *ioctl*(2) to set the line discipline for each window to the following (see *tty*(4) for details):

```
/* iflag */  BRKINT | IGNPAR | ICRNL  | IXON,
/* oflag */  OPOST  | ONLCR  | TAB0,
/* cflag */  B9600  | CS8    | CREAD  | CLOCAL,
/* lflag */  ISIG   | ICANON | ECHO   | ECHOE   | ECHOK,
/* line  */  000,

/* 0 INTR  */  003,  /* ^C */
/* 1 QUIT  */  034,  /* ^\ */
/* 2 ERASE */  010,  /* ^H */
/* 3 KILL  */  025,  /* ^U */
/* 4 EOF   */  004,  /* ^D */
/* 5 EOL   */  000,  /* ^@ */
/* 6 rsvd1 */  000,
/* 7 rsvd2 */  000,
```

wsh does not change environment variables, nor *alarm*(2) or *nice*(2) settings.

EXAMPLES

wsh Greg

Create a **term0** window named "Greg" at the default screen location, and start a shell in it.

SHELL=/usr/bin/vi wsh -kb -l100,200 -s80,20 termwin

Create an assigned (selected), bottom, **term0** window named *termwin* in the directory $*WMDIR*, with a "shell" of type **/bin/vi** connected. The upper left corner is at x = 100, y = 200 (pixels). The window is 80 columns and 20 rows in size.

wsh -wgraphics -gc'exec anvil' win1 win2

Create graphics windows "win1" and "win2", start a login shell for each window, and have them execute the command "exec anvil".

wsh -ea uhoh

Connect a shell to window "uhoh", which was created with *wcreate* (or whose shell terminated). In addition, automatically destroy the window when the new shell is terminated.

To cause a **term0** window to be destroyed when the connected shell exits, specify **-a** with the other *wsh* options.

SEE ALSO

windows(1), wborder(1), wcreate(1), wdestroy(1), wdisp(1), wfont(1), wlist(1), wmove(1), wselect(1), wsize(1), ioctl(2), signal(2), utmp(4), tty(7).

DIAGNOSTICS

wsh prints a message to standard error and returns non-zero if it detects any error.

wsh uses a close-on-exec pipe from its child processes to gain information on any error which occurs up through successful *exec*(2) of each shell.

Return values and possible meanings:

0	No errors detected.
1	No windows or shells created, due to: bad invocation error before calling wcreate failure to call wcreate wcreate returned anything other than 2 (including termination due to signal)
1	Aborted after possibly creating windows, due to problems while expanding a window_spec pattern.
2	Windows possibly created, but no shells (wcreate returned 2)
3	All windows and one or more shells possibly created, but failed to: open pipe (with write file descriptor greater than 2) set pipe to close-on-exec fork open window as file descriptor 0 affiliate to window dup 0 to file descriptors 1 and 2 set window permissions (using chmod) set window line discipline (using ioctl) execute shell program

WARNINGS

If the process which calls *wsh* has SIGCLD set to SIG_IGN, *wsh* thinks *wcreate* fails because *system*(3) returns -1.

NAME

wsize – change sizes of one or more windows

SYNOPSIS

wsize [**-s** w,h] [window_spec...]

DESCRIPTION

This command changes the sizes of the specified windows (by default, if you give no *window_specs*, the window connected to standard input). See *windows*(1) for an explanation of *window_spec*.

By default (if you give no **-s** option), *wsize* changes each window to its maximum possible size (width and height), which is its raster/buffer size less its pan (view) offsets. The upper left corners of the user units remain fixed, while the opposite corners change their locations. For **term0** windows, which do not support panning, the maximum size is the logical screen size, normally the same number of rows and columns as when the window was created. See *wcreate*(1) for details.

The option is:

-s *w,h* Set the new sizes to the given width and height, in units appropriate to type of each window, as in *wcreate*(1). (If you give more than one *window_spec*, this only makes sense if they are all of the same type.)

wsize silently limits the new size of each window to its maximum width and height. It also silently limits its minimum width and height when it has a normal border. If the window's border is thin, or if the window has no border, the minimum allowed size is one character cell on a side (for **term0** windows) or one pixel on a side (for other types).

This command does not let you change the logical screen size (distinct from window and buffer sizes) of a **term0** window.

EXAMPLES

wsize Change the window connected to standard input to its maximum size.

wsize -s400,500 win5 win2
: Change the size of **graphics** windows "win5" and "win2" to 400 pixels wide, 500 pixels high, or to their maximum widths and/or heights if smaller.

wsize -s40,10
: Change the size of the window connected to standard input, of type **term0**, to 40 columns by 10 rows.

HARDWARE DEPENDENCIES

Series 500:
: Borderless windows (**–T** option) are not supported on Series 500.

SEE ALSO

windows(1), wborder(1), wcreate(1), wdestroy(1), wdisp(1), wfont(1), wlist(1), wmove(1), wselect(1), wsh(1), wsize(3W).

DIAGNOSTICS

The following diagnostic values are returned by this command:

0 If no errors are detected.

1 Prints a message to standard error and returns 1 if it encounters an error which prevents sizing a window, including trouble while expanding a *window_spec* pattern.

2 Prints a message to standard error, continues, and later returns 2 if it encounters any error while trying to size one window.

LIBRARY ROUTINE SUMMARY

altfont_term0	set or inquire the Term0 alternate font
basefont_term0	set or inquire the Term0 base font
fontgetid_term0	inquire the ID of a Term0 font
fontgetname_term0	inquire the name of a Term0 font
fontload_term0	load a Term0 font
fontreplaceall_term0	replace the current base font and alternate font
fontsize_size0	Term0 font size
fontswap_term0	replace a Term0 font
fromxy_term0	convert from Term0 pixel units to characters
toxy_term0	convert Term0 character units to pixels
wautodestroy	autodestroy a window
wautoselect	autoselect a window
wautotop	autotop a window
wbanner	inquire about or control a window´s border
wbottom	bottom window
wconceal	conceal window
wcreate_graphics	create a **graphics** window type
wcreate_term0	create a **term0** window type
wdestroy	destroy a window
wdfltpos	default window/icon position
weventclear	clear window locator events
weventpoll	poll for window locator events
wget_hw_sprite_color	get hardware sprite colors
wget_see_thru	get see_thru color index value
wgetbcolor	get window border colors
wgetbcoords	get border coordinates
wgetcoords	get window coordinates
wgetecho	get echo
wgeticonpos	get current icon position
wgetlocator	get window echo/locator parameter
wgetname	get path name
wgetrasterecho	get raster echo
wgetscreen	get screen information
wgetsigmask	get window SIGWINDOW interrupt mask
wgskbd	set keyboard mode for graphics window type
whotspot_create	create a hotspot
whotspot_delete	delete a hotspot
whotspot_get	get hotspot data
whotspot_set	set hotspot data
wiconic	change a window to/from an icon
window	summary of window library routines
winit	initialize window device
winput_conf	read/set configuration of window input channel
winput_getroute	determine window input routing path
winput_read	read from window input channel
winput_setroute	routes input and events to another window
winput_widpath	get path name for a window id
wmenu_activate	activate a menu
wmenu_create	create a menu
wmenu_delete	delete a menu

wmenu_eventread	read the menu event
wmenu_item	specify or change a menu item
wminquire	get a window manager environment variable
wmkill	kill the window manager
wmove	move the location of a window
wmpathmake	build a path name using an environment variable
wmrepaint	repaint the desk top
wpan	pan the window
wpauseoutput	pause output to a window
wrecover	recover a window
wscrn_sprite_mode	set full-screen sprite control mode
wscroll_get	interrogate scroll bar information
wscroll_set	control of window scroll bar elevators and arrows
wselect	attach a keyboard to the specified window
wset_hw_sprite_color	set hardware sprite colors
wset_see_thru	set see_thru color index value
wsetbcolor	set window border colors
wsetecho	set echo
wseticon	set icon
wseticonpos	set icon position
wsetlabel	set label in window border
wsetlocator	set window locator position
wsetrasterecho	set raster echo
wsetsigmask	set window SIGWINDOW interrupt mask
wsfk_mode	switch to soft key mode
wsfk_prog	set programmable soft keys
wshuffle	shuffle windows′ relative locations within display stack
wsize	change the size of a window
wterminate	release window resources
wtop	move the window to the top of the window stack

NAME

altfont_term0 – set or inquire the Term0 alternate font

SYNOPSIS

int altfont_term0(fd,id);
int fd;
int id;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

id identifies the font which shall become the alternate font, or requests inquiry.

DISCUSSION

This routine has two distinct uses. If *id* has a value of -1, the font id of the current alternate font is returned. If *id* is not -1, this routine attempts to make the font specified by *id* become the new alternate font. HP-15 (2-byte) fonts cannot be used as the alternate font; attempting to do so will result in an error.

The screen is **not** repainted when this routine is used. Only future use of the alternate font is affected by this routine (i.e., changing the alternate font is not retroactive).

HARDWARE DEPENDENCIES

Series 500:

Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontgetid_term0(3W),fontload_term0(3W),basefont_term0(3W),fontswap_term0(3W), fontreplaceall(3W).

DIAGNOSTICS

A return value of -1 indicates failure. Otherwise, the font id is returned for successful inquiry, or the font id of the new alternate font is returned for successful selection.

NAME

basefont_term0 – set or inquire the Term0 base font

SYNOPSIS

int basefont_term0(fd,id);
int fd;
int id;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

id identifies the font which shall become the base font, or requests inquiry.

DISCUSSION

This routine has two distinct uses. If *id* has a value of -1, the font id of the current base font is returned. If *id* is not -1, this routine attempts to make the font specified by *id* become the new base font. The screen is not repainted. Only future use of the base font is affected by this routine (i.e., changing the base font is not retroactive).

HARDWARE DEPENDENCIES

Series 500:

Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontload_term0(3W), fontgetid_term0(3W), altfont_term0(3W), fontswap_term0(3W), fontreplaceall(3W).

DIAGNOSTICS

A return value of -1 indicates failure. Otherwise, the font id is returned for successful inquiry, or the font id of the new base font is returned for successful selection.

NAME

fontgetid_term0 – inquire the ID of a Term0 font

SYNOPSIS

```
int fontgetid_term0(fd,fontpath);
int fd;
char *fontpath;
```

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

fontpath
a pointer to a path name for the font in question.

DISCUSSION

This routine returns the font id of the named font, if it is currently loaded. If the font is loaded more than once, the minimum font id which describes the font is returned. This routine is intended to support programs wishing to prevent multiple instances of the same font in Term0's cache. The values returned for *id* have no meaning as fast alpha or font manager font ids.

HARDWARE DEPENDENCIES

Series 500:
Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontgetname(3W),fontsize_term0(3W).

DIAGNOSTICS

A return of -1 indicates and error; otherwise the font id is returned. Possible errors can occur if the requested font is not present in Term0's cache. See *errno*(2) for more information.

NAME

fontgetname_term0 – inquire the name of a Term0 font

SYNOPSIS

#include <stdio.h>
char *fontgetname_term0(fd,id);
int fd, id;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

id identifies the font whose path name is desired.

DISCUSSION

This routine returns the name of the specified font, if it is currently loaded. Its return value is a pointer to static storage which is modified every time this routine is called.

This routine is intended to support programs wishing to use font manager inquiry routines in conjunction with Term0 font routines.

HARDWARE DEPENDENCIES

Series 500:

Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontgetid_term0(3W),fm_fileinfo(3W).

DIAGNOSTICS

A return of NULL indicates and error; otherwise the font name is returned. Errors can occur if the requested font is not present in Term0's cache. See *errno*(2) for more information.

NAME

fontload_term0 – load a Term0 font

SYNOPSIS

int fontload_term0(fd,fontpath);
int fd;
char *fontpath;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

fontpath
a pointer to a path name for the font to be loaded.

DISCUSSION

This routine causes a font file to be loaded from the file system into Term0's font cache. This action prepares the font to be used by Term0 later, by means of the *basefont_term0* or *altfont_term0* routines or via escape code sequences. The id of the newly loaded font is returned. The values returned for *id* have no meaning as font manager parameters.

It is possible to load the same font path name more than once, in which case *fontload_term0* returns a unique id each time.

HARDWARE DEPENDENCIES

Series 500:
Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontswap_term0(3w),fontreplaceall(3w),basefont_term0(3w),altfont_term0(3w)
fontgetid_term0(3w).

DIAGNOSTICS

A return of -1 indicates and error; otherwise the font id is returned. Errors include attempting to load when there is not enough space in the font cache, and attempting to load a font which is not the same cell size as the other font(s) currently in use. See *errno*(2) for more information.

NAME

fontreplaceall_term0 – replace the current base font and alternate font

SYNOPSIS

```
int fontreplaceall_term0(fd,bfpath,afpath);
int fd;
char *bfpath,*afpath;
```

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

bfpath path name to the new base font.

afpath path name to the new alternate font, or NULL.

DISCUSSION

This routine causes all of the fonts currently being used to be removed and replaced by the specified base font and alternate font. If the alternate font pathname parameter is not supplied, the font designated by *bfpath* is used for both the base font and the alternate font. HP-15 (2-byte) fonts are not allowed to be alternate fonts. If *afpath* is a HP-15 font or if *afpath* is set to NULL and *bfpath* is a HP-15 font, an error will be returned and no fonts will have changed.

All characters printed in the old alternate font are changed to use the new alternate font, named *afpath.* Then all other characters are changed to use the new base font, named *bfpath.* The screen is repainted, using the new font or fonts.

HARDWARE DEPENDENCIES

Series 500:

Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontswap_term0(3W),fontload_term0(3W).

DIAGNOSTICS

A return of -1 indicates failure, see *errno*(2) for further information. Otherwise, 0 is returned. It is an error for *bfpath* to be NULL. See *errno*(2) for more information.

NAME

fontsize_size0 – Term0 font size

SYNOPSIS

int fontsize_term0(fd,wptr,hptr);
int fd;
int *wptr,*hptr;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

wptr,hptr
the pixel width and height components of the fonts' common cell size.

DISCUSSION

This routine sets the caller's variables to the current cell size of the 1-byte characters, which is common to all fonts in the cache. The 2-byte characters are twice as wide as the 1-byte characters. This size is in units of "square pixels", common to all display hardware.

HARDWARE DEPENDENCIES

Series 500:
Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontreplaceall(3W).

DIAGNOSTICS

A return of -1 indicates failure, see *errno*(2) for further information; otherwise, 0 is returned. See *errno*(2) for more information.

NAME

fontswap_term0 – replace a Term0 font

SYNOPSIS

int fontswap_term0(fd,newpath,oldid);
int fd;
char *newpath;
int oldid;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

newpath
path name to the new font.

oldid identifies the old font to be replaced.

DISCUSSION

This routine causes the old TERM0 font with font id of *oldid* to be removed and replaced by a new one specified by *newpath.*

All characters displayed in the old font are changed to use the new font. If the old font was the base font, the new font becomes the base font. If the old font was the alternate font and the new font is not a HP-15 (2-byte) font, the new font becomes the alternate font. If the new font is a HP-15 font, an error will occur and the fonts will not be swapped. HP-15 fonts cannot be the alternate font. The font id of the new font is returned.

HARDWARE DEPENDENCIES

Series 500:
Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fontload_term0(3W),fontreplaceall_term0(3W).

DIAGNOSTICS

A return value of -1 indicates failure. Otherwise, the fontid is returned. It is an error for *newpath* to be NULL. See *errno*(2) for more information.

NAME

fromxy_term0 – convert from Term0 pixel units to characters

SYNOPSIS

int fromxy_term0(fd,x,y,colptr,rowptr);
int fd;
int x,y;
int *colptr,*rowptr;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

x,y are the coordinates, in pixels relative to upper left corner of window, of the point to be mapped.

colptr,rowptr
are pointers to the coordinates, in character units, after mapping.

DISCUSSION

This routine provides a mapping from pixel units to column and row in a TERM0 window. The column and row returned are the coordinates of the 1 byte character cell which contains the point *x,y*. The current font cell size applies to this conversion. Note that 2-byte characters take up two columns. Both coordinate systems (pixels and characters) define 0,0 to be at the upper left corner of the window (which may be offscreen). Note that character 0,0 in the window may not be character 0,0 in the scroll buffer.

This routine is useful to convert coordinates from locator input devices into screen positions.

HARDWARE DEPENDENCIES

Series 500:
Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

toxy_term0(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

toxy_term0 – convert Term0 character units to pixels

SYNOPSIS

int toxy_term0(fd,xptr,yptr,col,row);
int fd;
int *ptr,*yptr;
int col, row;

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

xptr,yptr
pointers to pixel coordinates of the point after mapping, whose values are set by this routine.

col,row
the coordinates (in character units) of the point to be mapped.

DISCUSSION

This routine provides a mapping from character units (columns and rows) to pixels in a Term0 window. The coordinates of the upper-left-most pixel in the character are used. Both coordinate systems (pixels and characters) define 0,0 to be at the upper left corner of the window (which may be offscreen). Note that character 0,0 in the window may not be character 0,0 in the scroll buffer.

This routine is useful to convert screen positions into x,y locations suitable for use with sprites or other pixel-oriented tracking schemes.

HARDWARE DEPENDENCIES

Series 500:
Term0 routines do not support HP-15 (2-byte) fonts on Series 500.

SEE ALSO

fromxy_term0(3W).

DIAGNOSTICS

A return of -1 indicates failure, see *errno*(2) for further information; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wautodestroy – autodestroy a window

SYNOPSIS

#include <window.h>
int wautodestroy (fd, value)
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for a window's opened device interface.

value is an integer which determines the action of this routine.

DISCUSSION

This call inquires or sets whether the window will automatically be destroyed upon the last close operation on that window. Note that this routine is ineffectual unless *wrecover*(3W) has been called to make the window recoverable.

If *value* is set to -1, then the window's current autodestroy status is returned. A 1 value returned indicates that the window denoted by this file descriptor is currently autodestroy, while a 0 value returned indicates that it is not.

If *value* is set to 0, then the window will **not** be destroyed when its device interface is closed by every process that has opened the window. This is the default state at window create time.

Setting *value* to 1 causes the window to be destroyed upon the last close.

SEE ALSO

wrecover(3W).

DIAGNOSTICS

A value of 0 or 1 is returned unless *fd* does not refer to a window, in wich case -1 is returned. See *errno*(2) for further information.

NAME

wautoselect – autoselect a window

SYNOPSIS

```
#include <window.h>
int wautoselect (fd, value)
int fd;
int value;
```

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

value is an integer which determines what action the routine will take.

DISCUSSION

This call inquires or sets, depending on the *value* parameter, whether the *term0* window will automatically become selected when output is sent to its device interface. This routine does **not** work with graphics windows.

If *value* is -1, then the current autoselect status is returned. A 1 value returned indicates that the window denoted by this file descriptor is currently autoselect while a 0 value returned indicates that it is not.

Setting *value* to 0 will turn off autoselect–the window won't automatically be selected when output is sent to its device interface. This is the default state at window create time.

Setting *value* to 1 causes the window to become selected automatically upon output to its device interface.

Sending output to a window that has auto-selection enabled has an additional side effect: it turns auto-selection off. Therefore, to automatically select a window when output is sent to it, call *wautoselect* whenever output is sent to the window.

DIAGNOSTICS

A value of 0 or 1 is returned unless auto-select does not apply or *fd* does not refer to a window, in which case -1 is returned. See *errno*(2) for further information.

NAME

wautotop – autotop a window

SYNOPSIS

#include <window.h>
int wautotop (fd, value)
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for an opened *term0 window type* device interface.

value is an integer which determines what action the routine should take.

DISCUSSION

This call inquires or sets whether the window will automatically become top most upon output to its device interface. This routine does **not** work with graphics window types.

If *value* is -1, the current autotop state is returned. A 1 value returned indicates that the window indicated by this file descriptor is currently autotop while a 0 value returned indicates that it is not.

If *value* is 0, then the window is **not** moved to the top upon output. This is the default state at window create time.

If *value* is 1, then the window is automatically moved to the top when output is sent to its device interface (special file). If the window is currently in an iconic state, it will become normal the next time output is directed to the window.

Note that the autotop attribute is turned off after output is sent to a window. Therefore, to automatically top a window every time output is sent to it, be sure to call *wautotop* whenever output is sent to the window.

DIAGNOSTICS

A value of 0 or 1 is returned unless autotop does not apply or *fd* does not refer to a window, in which case -1 is returned. See *errno*(2) for further information.

NAME

wbanner – inquire about or control a window's border

SYNOPSIS

int wbanner(fd,value);
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

value is the set/interrogation parameter; following are valid values for this parameter.

-1	return whether the window's border status: a return value of 0 means the window has a thin border; 1 means the window has a normal (thick) border; 2 means that the window has no border.
0	display a thin border with the window.
1	display a border with the window; this is the default state for windows created via the pop-up menu or window system commands.
2	do not display a border around the window (null border type). This border type is supported only on the *graphics window type.*

DISCUSSION

This call inquires or sets whether a border is displayed around the specified window.

HARDWARE DEPENDENCIES

Series 500:

Only the *thin* and *normal* border types are supported on Series 500; the null border type (no border) is not supported.

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 or 1 is returned. See *errno*(2) for further information.

NAME

wbottom – bottom window

SYNOPSIS

int wbottom(fd,value);
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for an opened *window* device interface.

value is the set/interrogation parameter. Following are valid values and their effect:

-1	returns 1 if the window is the bottom window in the stack; returns 0 otherwise.
0	then the library procedure is ignored.
1	place the window or icon underneath all other windows. If the window is concealed, make it visible but below all others. Obscurity may very well make it invisible, but shuffling the windows would eventually bring it on top.

DISCUSSION

This call inquires or sets whether this window is bottom most.

SEE ALSO

wtop(3w),wshuffle(3w),wconceal(3w).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 or 1 is returned. See *errno*(2) for further information,

NAME

wconceal – conceal window

SYNOPSIS

int wconceal(fd,value);
int fd;
int value;

DESCRIPTION

fd	is an integer file descriptor for an opened *window* device interface.
value	is the set/interrogation parameter which determines the action taken by this routine; valid values are:

-1	return whether the window is concealed (0, means no; 1 means yes).
0	then the window's state is not changed–the routine does nothing.
1	set the window or icon to be concealed.

DISCUSSION

This call inquires or sets the value of the window's visibility. *wconceal* must be false for anything to be visible. Being not concealed does not guarantee visibility, because it might be obscured via another window or by the screen edges itself.

SEE ALSO

wtop(3W),wbottom(3W),wshuffle(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 or 1 is returned. See *errno*(2) for further information.

NAME

wcreate_graphics – create a **graphics** window type

SYNOPSIS

```
int wcreate_graphics(wmfd,wname,x,y,w,h,rasterw,rasterh, attributes, border);
int wmfd;
char *wname;
int x,y;
int w,h;
int rasterw,rasterh;
int attributes;
int border;
```

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

wname
a pointer to a path name to use as the name of the window's window type device interface. Also the base name (see *basename*(1)) is known as the window name. By default, the window name is used as the window label in the window's border. The window type device interface is used for communicating with the window. *wname* is also a pointer to a path name that should be use for doing Starbase graphics to the contents window. This is the device to which *gopen* is applied.

x,y the device coordinates for the upper left corner of the drawing area of the window with respect to the upper left corner of the screen.

w,h the width and height of the screen view into the virtual raster in device coordinates.

rasterw,rasterh
the width and height of the virtual raster. (Also the maximum values that w and h can have for the life of the window.)

attributes
specifies whether the window should be non-retained, retained, or IMAGE. A value of 0 specifies non-retained. A value of 1 specifies retained as byte/pixel. A value of 2 specifies retained as bit/pixel. A value of 4 specifies IMAGE which also implies non-retained. IMAGE means that the user area of the window is mapped into the image planes while the border is displayed in the overlay planes. This value only applies on the HP98730 display system. All other values are reserved for future use and are currently invalid.

border
chooses either thin, normal, or no border for the window. A value of 0 indicates a thin border. A value of 1 indicates a normal border. A value of 2 indicates no border (null border type). If a normal border is chosen, there is a minimum enforced size for the window. A normal border contains the window name and various symbols for manipulating the window. A thin border consists of a thin blank frame around the window with no name or symbols.

DISCUSSION

Create a graphics window type with default characteristics.

HARDWARE DEPENDENCIES

Series 500:
For **attributes**, called **retained** on Series 500, a non-zero value specifies retained, a zero value specifies non-retained.

Only the *thin* and *normal* border types are supported on Series 500; the *null* border type is not supported.

SEE ALSO

wcreate_term0(3W),wdestroy(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wcreate_term0 – create a **term0** window type

SYNOPSIS

int wcreate_term0(wmfd,name,x,y,wincols,winrows, scrncols, scrnrows, bufcols,
bufrows, basefont, altfont, colormode, border);
int wmfd;
char *name;
int x,y,wincols,winrows,scrncols,scrnrows,bufcols,bufrows;
char *basefont,*altfont;
int colormode,border;

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

name a pointer to a path name that is used to make the device node that is associated with the new window.

x,y the device coordinates for the upper left corner of the contents portion of the window (ignoring the window borders), with respect to the upper left corner of the screen.

wincols,winrows
the width and height of the *window* in columns and rows. Note that each 2-byte character takes up two columns. This specifies the size of the contents portion of the *window*, and does not include space for a label and borders.

scrncols,scrnrows
the width and height of the *terminal screen* being emulated in columns and rows.

bufcols,bufrows
the width and height of the *scroll buffer* in columns and rows.

basefont,altfont
pointers to path names for the base and alternate font files to be used. If *altfont* is NULL, *basefont* will be used for both the base and the alternate font. HP-15 (2-byte) fonts may not be used for the alternate font. If *altfont* is a HP-15 font or if *altfont* is NULL and *basefont* is a HP-15 font, an error will result and the window will not be created.

colormode
enables color mode on color systems. A value of 2 indicates a color system. This parameter should always be set to 2.

border
chooses either thin or normal borders for the terminal window. A value of 1 indicates normal borders. A value of 0 indicates thin borders. A value of 2 indicates no border (null border type). If normal borders are chosen, there is a minimum size for the terminal, which is enforced. A normal border contains the window name and various symbols for manipulating the window. A thin border consists of a thin blank frame around the window with no name or symbols.

DISCUSSION

Create a Term0 window type with default characteristics. The width and height of the *terminal screen* must be no larger than the *scroll buffer.* Similarly, the size of the *window* must be no larger than the *terminal screen.* The *terminal screen* is used for screen-relative cursor addressing.

The default characteristics are:

- The window is offscreen (that is, not visible).
- The keyboard is not attached to the terminal emulator.
- The 8 alpha color pairs are initialized identically to the color pairs of the HP2627A terminal.
- Color pair 0 (white characters on black background) is selected.
- The basename of the window path name is used as the border label, if the border is not "thin".
- The cursor is at column 0, row 0, and is turned on (enabled).
- The terminal screen being emulated is positioned at the upper left of the scroll buffer.
- The window is positioned at the upper left of the terminal screen.
- There are no characters in the scroll buffer.
- The softkeys are in USER state, labeled f1 through f8, but are not displayed.
- There are no fonts in the cache other than basefont and altfont, and the basefont is active.
- The terminal emulator is in ASCII 8-bit mode.
- The "Keyboard" field in the configuration menu is set to match the keyboard which is actually present.
- Display Functions is not enabled.
- The left and right margins are set to 0 and the largest column of the scroll buffer.
- Tabs are set at column 9 and every 8th column thereafter.
- The window has a default icon image resembling a terminal.
- The border color is set to the values of environment variables; see wmstart(1).
- Fonts used in the border, icon, softkeys, and popup menu are set to the values of environment variables; see wmstart(1).

HARDWARE DEPENDENCIES

Series 500:
HP-15 (2-byte) fonts are not supported on Series 500.

SEE ALSO

wcreate(1),wmstart(1).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wdestroy – destroy a window

SYNOPSIS

int wdestroy(wmfd,name);
int wmfd;
char *name;

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

name the path name of the window type to destroy.

DISCUSSION

This routine destroys the window specified by *name*.

SEE ALSO

wcreate_term0(3W),wcreate_graphics(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wdfltpos – default window/icon position

SYNOPSIS

int wdfltpos(fd,req,wx,wy,ix,iy,name);
int fd;
int req;
int *wx,*wy;
int *ix,*iy;
char *name;

DESCRIPTION

fd	is an integer file descriptor of an opened *window manager* or *window type* device interface.
req	determines which values are returned by this routine; values are specified by setting the appropriate bits in this parameter. Bits are defined as follows:

bit 1	return wx, wy
bit 2	return iconx, icony
bit 3	return name[16]

wx,wy	are pointers to the default position of a window.
ix,iy	are pointers to the default position of an icon.
name	is a pointer to a 16-character space in which a window system default name is returned. This name is only the basename (see *basename*(1)).

DISCUSSION

This routine returns a round robin default position for a window or icon. Windows stair step down from the upper left corner of the screen. Icons go down from the upper right or up from the lower left, depending on the icon placement parameter given to the window manager at powerup.

Note: *wdfltpos* assigns values to only the requested parameters; therefore, dummy space need not be declared for the *name* parameter.

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

weventclear – clear window locator events

SYNOPSIS

#include <window.h>
int weventclear(fd,mask);
int fd;
int mask;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

mask is used to specify conditions to interrupt the user process with the SIGWINDOW signal. *mask* is a set of bits defined by the following bit names, where the default mask setting is 0. The header file containing these defines is **/usr/include/window.h**; the *wsetsigmask*(3W) reference page lists the events also. SIGWINDOW is defined in **/usr/include/sys/signal.h**.

DISCUSSION

This routine will clear the events specified by the event mask, so that a subsequent weventpoll of those events will return a 0 count.

SEE ALSO

wgetsigmask(3W),wsetsigmask(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

weventpoll – poll for window locator events

SYNOPSIS

#include <window.h>
int weventpoll(fd,mask,count,x,y);
int fd;
int *mask,*count,*x,*y;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

mask is defined as in *wsetsigmask.* In this routine, however, it is both an input parameter and an output parameter.

If *mask* = 0,then the bit corresponding to the last event that has occurred will be set and passed back in *mask* (EVENT_ECHO will not be considered an event in this case). Events are not queued, so *weventpoll* will return only the most recent event since the last call to *weventpoll.*

If *mask* <> 0, then *weventpoll* looks only at the events whose bits are set in *mask*. If one of the specified events has occurred, then the bit will remain set; otherwise, if the event has not occurred, then the bit is cleared in *mask* on return. *count, x*, and *y* are returned for the most significant bit (event) set, and the count is cleared for that event. If none of the events specified in *mask* have occurred, then 0 is returned in *mask.*

Valid event bits are defined in **window.h**.

count is the number of times this event has occurred since the last time the *count*, *x*, *y* values were read for this event.

x,y contain event-specific information, see below.

DISCUSSION

This routine checks to see whether one or more specific events have occurred.

If the event was EVENT_SIZE, then *x* and *y* contain the new width and height of the window.

If the event was EVENT_MOVE, then *x* and *y* contain the new location of the window.

If the event was EVENT_REPAINT, EVENT_SELECT, EVENT_DESTROY, EVENT_BREAK, EVENT_ICON, or EVENT_ABORT then *x* and *y* both contain 0.

If the event was EVENT_MENU, then *x* contains the menu id of the menu from which a selection was made, and *y* contains the item id of the selected item.

If the event was EVENT_HOTSPOT, then *x* contains the *event_byte* specified for the hotspot in *whotspot_create*(3W) or *whotspot_set*(3W), and *y* contains the cause of the hotspot activation.

If the event was EVENT_ELEVATOR then an interactive move of an elevator has been requested by the user. In this case, *x* contains either SCROLLBAR_V or SCROLLBAR_H to identify the elevator, and *y* contains the requested value for the elevator. Note that the elevator hasn't actually been moved; this can be accomplished using *wscroll_set*(3W). Elevator events can only occur for elevators that are in user mode.

If the event was EVENT_SB_ARROW, then *x* contains the sum of all horizontal scroll bar arrow events that have occurred where a right arrow event adds 1 and a left arrow event adds -1. Likewise, *y* contains the sum of all vertical scroll bar arrows events that have occurred where a down arrow event adds 1 and an up arrow event adds -1.

If the event was EVENT_B1_DOWN through EVENT_B8_UP or EVENT_ECHO, *x* and *y* are the the echo x and y locations at the time of the last event of this type. Note that *x* and *y* are in pixel units.

HARDWARE DEPENDENCIES

Series 500:

The following events are not supported on Series 500:

EVENT_HOTSPOT
EVENT_DESTROY
EVENT_BREAK
EVENT_ICON
EVENT_ELEVATOR
EVENT_SB_ARROW
EVENT_ABORT

In addition, the Series 500 does not support hot spots or scroll bars; these are supported on Series 300 Only.

SEE ALSO

wgetsigmask(3W), whotspot_create(3W), whotspot_set(3W), wscroll_set(3W), wsetsigmask(3W), weventclear(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wget_hw_sprite_color get hardware sprite colors

SYNOPSIS

int wget_hw_sprite_color(fd,fgcolor,bgcolor);
int fd;
int *fgcolor,*bgcolor;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

fgcolor,**bgcolor**
the foreground sprite color, and background sprite color.

DISCUSSION

Returns the foreground and background colors given by the last **wset_hw_sprite_color** call. If no call has yet been made to **wset_hw_sprite_color**, then defaults will be returned.

This call can be made on all devices, but the returned value will only be valid on the HP98730.

SEE ALSO

wset_hw_sprite_color(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wget_see_thru - get see_thru color index value

SYNOPSIS

int wget_see_thru(wmfd,see_thru);
int wmfd;
int *see_thru;

DESCRIPTION

fd is an integer file descriptor for an opened *window manager* device interface.

see_thru,
the see_thru color index.

DISCUSSION

Returns the window system see_thru color index. The default is set at window system startup.

This call can be made on all devices, but the returned value will only be valid on the HP98730 or the HP98720.

SEE ALSO

windows(1), wmstart(1), wset_see_thru(3W).

DIAGNOSTICS

A return of −1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgetbcolor – get window border colors

SYNOPSIS

int wgetbcolor(fd,fgborder,bgborder);
int fd;
int *fgborder,*bgborder;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

fgborder,bgborder
the foreground label character color, and background label and border color.

DISCUSSION

Returns the current foreground and background border colors. The defaults at window creation time are determined by the WMBDRFGCLR and WMBDRBGCLR environment variables.

If double buffering color mode is enabled via the WMCONFIG environment variable, the colors returned are the lower N/2 bits of the colors actually used, where N is the number of planes on the display.

SEE ALSO

wsetbcolor(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgetbcoords – get border coordinates

SYNOPSIS

```
int wgetbcoords(fd,x,y,w,h);
int fd;
int *x,*y;
int *w,*h;
```

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

x,y are pointers to the screen pixel coordinates of the upper left corner of the border with respect to the physical screen. Positive, negative and zero values for x and y are allowed. (0,0) is the upper left corner of the screen.

w,h are pointers to the pixel width and height of the border.

DISCUSSION

Returns the border coordinates associated with the window type device indicated by *fd*.

SEE ALSO

wgetcoords(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgetcoords – get window coordinates

SYNOPSIS

```
int wgetcoords(fd,x,y,w,h,dx,dy,rw,rh);
int fd;
int *x,*y;
int *w,*h;
int *dx,*dy;
int *rw,*rh;
```

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

x,y are pointers to the screen pixel coordinates of the upper left corner of the window view with respect to the physical screen. This refers to the upper left corner of the contents portion, not the border. Positive, negative and zero values for *x* and *y* are allowed. (0,0) is the upper left corner of the screen.

w,h are pointers to the pixel width and height of the window view. For a *window type* these refer to the contents portion, not the border.

dx,dy are pointers to the pixel delta x and delta y offset of the view in the window raster. These refer to the contents portion, not the border. *dx* and *dy* can have positive values only and tell how much the window is panned.

rw,rh are pointers to the pixel width and height of the window raster. *rw* and *rh* limit the maximum values for *dx*, *dy*, *w*, and *h*.

DISCUSSION

Returns the window coordinates associated with the window device denoted by *fd*.

SEE ALSO

wgetbcoords(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgetecho – get echo

SYNOPSIS

int wgetecho(fd,echo_value,x2,y2,optimized);
int fd;
int *echo_value,*x2,*y2,*optimized;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

echo_value
is the type of echo to be used. Predefined types are listed below.

0 invisible echo.

1 device's best echo.

2 full screen cross hair.

3 small tracking cursor.

4 rubber band line, with anchor point at *x2*,*y2*.

5 rubber band rectangle, with anchor point at *x2*,*y2*.

6 alpha digital representation (for displaying characters on external devices such as button boxes).

7 user defined raster cursor. (Default at window creation time where raster is defined to be a small arrow.) To get back to the default raster, set *w* and *h* to 0 (zero) for *wsetrasterecho*.

8 box of width x2, height y2; where the upper left corner is the current cursor position

> 8 device-dependent representation.

x2,y2 are the echo anchor position or box width and height.

optimized
is a boolean with two possible values:

0 means echo is exactly as defined. And movement via *wsetlocator* will do exactly as specified.

1 means echo representation may be modified as per the device to make it track in the best way possible. Movement via *wsetlocator* may snap to device-dependent boundaries to take advantage of specialized hardware (such as a 4 X 4 pixel tile mover).

DISCUSSION

Get the window's current echo. An additional routine, *wgetrasterecho*, is needed to get information about a raster echo.

SEE ALSO

wsetecho(3W),wgetrasterecho(3W),wsetrasterecho(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgeticonpos - get current icon position

SYNOPSIS

int wgeticonpos(fd,x,y);
int fd;
int *x,*y;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

x,y are pointers to the screen pixel coordinates for the position of the upper left hand corner of the icon representation of a window type with respect to the physical screen. Positive, negative and zero values are allowed.

DISCUSSION

The *wgeticonpos* function returns the icon representation position associated with the window device indicated by *fd.*

(0,0) is the upper left hand corner of the screen.

SEE ALSO

wseticonpos(3W),wseticon(3),wiconic(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for further information.

NAME

wgetlocator – get window echo/locator parameter

SYNOPSIS

int wgetlocator(fd,x,y,buttons);
int fd;
int *x,*y,*buttons;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

x,y contain the current echo position. *x and y* are relative to the current window data space (not the screen). They are also in pixel units. The echo hot spot determines which part of the echo is actually at the coordinates, see *wsetrasterecho* and *wsetecho.* For a window type, they are relative to the contents portion, not the border.

buttons
specifies the current state of the locator buttons. The least significant bit is associated with the left most button. If a bit is set, then that button is down.

DISCUSSION

Get the current echo position and buttons state. For higher performance, do *gopen* to get *fd*, or use *wsetrasterecho* on *fd.*

SEE ALSO

wsetlocator(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgetname – get path name

SYNOPSIS

int wgetname(fd,path);
int fd;
char *path;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* or *window manager* device interface.

path is a pointer to the path name for the referenced file descriptor.

DISCUSSION

The path name of the window type device or window manager indicated by *fd* is returned in *name.*

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgetrasterecho – get raster echo

SYNOPSIS

int wgetrasterecho(fd,dx,dy,w,h,rule,mask_rule,mask,image);
int fd;
int *dx,*dy,*w,*h,*rule,*mask_rule;
char *mask,*image;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

dx,dy are the offset of the echo hot spot to the upper left corner of the echo, usually negative.

w,h are the echo's size.

rule,mask_rule
are the echo's replacement rules used when displaying a raster cursor. These rules are defined in the HP Starbase Documentation.

mask is a pointer to a bit-per-pixel array of 128 characters. This array is used to make the mask for the raster echo. The mask is placed on the screen before the image. Each bit represents two possible values for each pixel: zero or all ones.

image is a pointer to a byte-per-pixel array of 1024 characters. This array is used to make the image for the raster echo.

DISCUSSION

Get the window's raster echo type. An additional routine, *wgetecho*, is needed to get more information about basic echo attributes first.

SEE ALSO

wsetrasterecho(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgetscreen – get screen information.

SYNOPSIS

int wgetscreen(wmfd,w,h,b,cmapent,sfkh);
int wmfd;
int *w,*h,*b,*cmapent,*sfkh;

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

w,h are pointers to the pixel width and height of the screen.

b is a pointer to the number of bytes/pixel.

cmapent
is a pointer to the number of color map entries. Black and white has only two.

sfkh is a pointer to the height of the softkeys in pixels.

DISCUSSION

Returns the screen information associated with the specified window manager device indicated by *wmfd.*

SEE ALSO

wgetbcoords(3W),wgetcoords(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME
wgetsigmask – get window SIGWINDOW interrupt mask.

SYNOPSIS
#include <window.h>
int wgetsigmask(fd,mask);
int fd;
int *mask;

DESCRIPTION
fd is an integer file descriptor for an opened *window type* device interface.

mask is specifies conditions to interrupt the user process with the SIGWINDOW signal. Mask is a set of bits defined by *wsetsigmask.*

DISCUSSION
Interrogate the value of the SIGWINDOW interrupt mask for the specified window.

SEE ALSO
wsetsigmask(3W).

DIAGNOSTICS
A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wgskbd – set keyboard mode for graphics window type

SYNOPSIS

#include <window.h>
int wgskbd(fd,mode);
int fd;
int mode;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

mode A value of -1 inquires the current keyboard mode for the the graphics window. A value of 0, 1, or 2 sets the keyboard mode to that value.

DISCUSSION

This routine sets or inquires the keyboard mode for a graphics window type. The keyboard mode determines the type of processing done by the window type on input data prior to supplying the data to the application. The default mode is 0. Whenever the mode is changed, any data in the input queue is flushed.

When used to set the keyboard mode, *wgskbd* has the side effect of resetting all *winput_conf*(3W) settings to defaults which are dependent upon *mode*. See *winput_conf*(3W). In addition, *wgskbd* calls *ioctl*(2) to modify the line discipline for the window as follows (see *tty*(4) for details).

ICANON	disabled
BRKINT	enabled
VTIME	0
VMIN	1, 2, or sizeof(struct event_code), depending on whether *mode* is 0, 1, or 2, respectively.

Other than the side effect of resetting *winput_conf*(3W) settings, *wgskbd* has no effect if input routing is in effect for this window.

Mode 0

If *mode* has the value 0, ASCII mode is enabled wherein data is processed as per the normal keyboard input model for graphics windows, essentially that of a Term0 window in **transmit functions** mode (see the *"Window System Input"* and *"Graphics Softkeys"* chapters of the *HP Windows/9000 Programmer's Manual*). Note the window must be selected for keycodes to be available via *fd*.

Mode 2

If *mode* has the value of 2, the window type is put into packetized input mode as described in the *winput_read*(3W) man entry. Using a value of 2 is only supported in conjunction with *winput_read*(3W).

Whenever the keyboard mode is set to 2, a button press over that window will **not** cause that window to be moved automatically to the top and/or selected as would normally happen with a window. If this behavior is desired, the application can simulate it when appropriate button events are detected. The WMIUICONFIG environment variable should be interrogated using *wminquire*(3W) to determine which buttons cause the top and/or select, and whether the action is select only or top-and-select.

Mode 1

If *mode* has the value 1, the window type is put into 2-byte mode. A complete description of this mode follows. **NOTE:** This discussion only applies to non-Katakana ITF keyboards.

First are some definitions for terms used later - these are all essentially key-*types*: *Modifier* keys are the keys labeled CTRL, Shift (2), and Extend char (2). A *Normal* key is any key that has a single ASCII character for a label; some have two labels (unshifted and shifted). Also included is the key labeled DEL/ESC. A *Special* key is any key which is not a Normal key or Modifier key (this includes such things as keys labeled with words, function keys, cursor keys, and blank keys). An *Npad* key is any key which is part of the numeric keypad (18 keys grouped together on the right side of the ITF keyboard) and the 4 unlabeled keys above it. A *Roman8* key is any Normal key which is not also an Npad key, or the DEL/ESC key. Any given key may be in more than one of these sets; for example, the key labeled 0 in the numeric keypad is both a Normal key and an Npad key.

When a *read* is done on *fd*, keycode information is returned in 2-byte packets; all reads should be in multiples of two. The packet looks like:

```
struct gr_key_code {
        unsigned char control_byte;
        unsigned char data_byte;
};
```

The control_byte is a bit field interpreted by the following values:

K_SHIFT_B	Shift Bit
K_CONTROL_B	Control Bit
K_META_B	Meta (Extend Left)
K_EXTEND_B	Extend (Extend Right)
K_NPAD	Number Pad Key
K_SPECIAL	Special Keys

The K_NPAD and the K_SPECIAL flags indicate the *type* of key being pressed (see definitions for *Npad* and *Special* above).

The K_SHIFT_B, K_CONTROL_B, K_META_B, and K_EXTEND_B flags are keycode (data_byte) modifiers, and are the result of combination keypresses (multiple keys simultaneously). K_SHIFT_B applies to either Shift key being used; K_META_B applies to the Extend char key to the left of the spacebar; K_EXTEND_B applies to the Extend char to the right of the spacebar.

The data_byte for K_SPECIAL keys is *not* ASCII data. A complete list of **#define**s for these is provided in <window.h>. Any of the keypress modifiers can be applied to these keys, and no mapping is done; that is, the value of data_byte is independent of the modifier.

The data_byte for K_NPAD keys can be either Special (K_SPECIAL) or Normal. Any of the keypress modifiers can be applied to these keys also, and no mapping is done.

The data_byte for Normal keys (not K_SPECIAL or K_NPAD) is mapped according to the following rules:

(1) If no modifiers are present, the data_byte is ASCII, and the value is mapped to the appropriate shifted or unshifted value, determined by the current *Capsmode* state (initially unshifted, see below for discussion of the *Capsmode* toggle).

(2) If the K_SHIFT_B modifier is set, the data_byte is mapped to the appropriate shifted or unshifted value, determined on the current *Capsmode* state.

(3) If the K_CONTROL_B modifier is set, no mapping is done to reflect this.

(4) If the K_META_B modifier is set, no mapping is done to reflect this.

(5) If the K_EXTEND_B modifier is set and the key is a Roman8 key, the data_byte is mapped to the appropriate Roman-8 character (either shifted or unshifted depending on the K_SHIFT_B modifier and the *Capsmode* state). (see "Series 300 System Console" in *HP-*

UX Concepts and Tutorials: Facilities for Series 200, 300, and 500 for the ITF keyboard layout, and *roman8*(7) for data_byte values). If the key is **not** a Roman8 key, no mapping of the data_byte is done.

There are some special cases that should be noted:

(1) The graphics window type maintains the *Capsmode* state for the Normal keys which can be capitalized (alphabetic and some Roman-8 characters). Initially this state is *off*, or unshifted. The state is toggled by the Caps key, which returns either K_CAPS_ON or K_CAPS_OFF in data_byte when pressed, depending on the state. If the state is off, keys which can be capitalized are mapped to their down-shifted value, and are up-shifted when typed with a Shift key. If the state is on, such keys are mapped to their up-shifted value, and when typed with a Shift key are mapped to their down-shifted value.

(2) Pressing the Break key (or the Break key with any modifier except K_SHIFT_B) will cause the graphics window type to emit a gr_key_code with both the control_byte and the data_byte being null (0). Additionally, a TIOCBREAK ioctl call will be issued which will send a SIGINT to the user process if it has done a *setpgrp*(2) properly, otherwise the user process will only see the null gr_key_code.

(3) The K_SHIFT_B modifier cannot be read with the Select key - this keypress is intercepted by the window system and cannot be used.

(4) The K_CONTROL_B modifier applied to arrow-keys are intercepted by the window system, and will cause the sprite to move; then these keycodes are passed on to the user process.

HARDWARE DEPENDENCIES

Series 300:

Each packet of information sent in mode 2 has a time-stamp specifying exactly when the event or keypress occurred. Most of the time, packets will be time-ordered in the queue; that is, packets are ordered in the sequence in which they occurred--oldest first, latest last. **However**, the window processes locator data before keypress data; so in rare instances, it is possible that a locator data packet may precede a keypress packet, even though the keypress occurred before the locator packet. Nevertheless, the time-stamps **are** correct for both packets; thus the application can still determine the exact order of keypresses or events by looking at the time-stamp.

Series 500:

This routine is not supported on Series 500. None of the winput_* routines is supported either.

SEE ALSO

winput_conf(3W),winput_read(3W),winput_setroute(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise the current mode is returned. See *errno*(2) for more information.

NAME

whotspot_create – create a hotspot

SYNOPSIS

```
#include <window.h>
int whotspot_create(fd,bmask,x,y,w,h,event_byte);
int fd;
int bmask;
int x,y,w,h;
int event_byte;
```

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

bmask is a bit mask indicating which locator button(s) activate the hotspot. The bit mask is defined as follows:

HS_MASK_BUTTON1	Locator Button 1
HS_MASK_BUTTON2	Locator Button 2
HS_MASK_BUTTON3	Locator Button 3
HS_MASK_BUTTON4	Locator Button 4
HS_MASK_BUTTON5	Locator Button 5
HS_MASK_BUTTON6	Locator Button 6
HS_MASK_BUTTON7	Locator Button 7
HS_MASK_BUTTON8	Locator Button 8 (In Proximity)–i.e., the graphics tablet stylus or puck switch was placed on the graphics tablet
HS_MASK_SELECT	Keyboard SELECT
HS_MASK_ENTEREXIT	Locator enter/exit

x,y are the coordinates of the upper left corner of the rectangular hotspot region in device units relative to the upper left corner of the virtual raster of the window.

w,h are the width and height of the rectangular hotspot region in device units. These parameters must be greater than zero.

event_byte
is the event to occur when the hotspot is activated. Legal values are discussed below.

DISCUSSION

This call creates a hotspot for the window specified by the opened fd. The *hotspot_id* for the created hotspot is returned. The *hotspot_id* is then used to refer to the newly created hotspot in subsequent calls. If a -1 is returned the call was unsuccessful.

A hotspot is activated whenever one of the buttons or the SELECT key specified in *bmask* is pressed while the locator is within the rectangle defined by *x,y,w,h*. A hotspot will also be activated if the enter/exit bit in *bmask* is 1 and the locator enters or exits the specified hotspot. Any combination of locator buttons, SELECT key, and enter/exit may be specified in *bmask*.

The activation of a hotspot can be detected by enabling SIGWINDOW for hotspots via *wsetsigmask*(3W), then using *weventpoll*(3W) to determine which hotspot was activated. *Weventpoll*(3W) will return the *event_byte* for the last hotspot activated and the cause for the hotspot being activated in *x* and *y* respectively. The cause returned in *y* can be any one of the following:

EC_BUTTON1	Button 1 was pressed
EC_BUTTON2	Button 2 was pressed

EC_BUTTON3	Button 3 was pressed
EC_BUTTON4	Button 4 was pressed
EC_BUTTON5	Button 5 was pressed
EC_BUTTON6	Button 6 was pressed
EC_BUTTON7	Button 7 was pressed)
EC_BUTTON8	Button 8 (proximity) was pressed-i.e., the graphics tablet stylus or puck switch was placed on the graphics tablet.
EC_SELECT	SELECT was pressed
EC_ENTER	Hotspot was entered
EC_EXIT	Hotspot was exited

If packetized window input is enabled for the window (see *wgskbd*(3W)), the hotspot data can be read from the input queue as an *event_code* packet. The value for the *control_byte* will be K_EVENT, the value for *data_byte* will be K_ILLEGAL, the value for *event_byte* will be the value specified in *whotspot_create* or *whotspot_set*(3W), and the value for *event_cause* will be one of EC_BUTTON1 through EC_EXIT.

The value specified for *event_byte* must be one of K_MOVE_ST, K_SIZE_LR_ST, K_POPUP_ST, or a value between 128 and 255 inclusive. If the value is K_MOVE_ST, K_SIZE_LR_ST, or K_POPUP_ST, then an interactive move, size, or pop-up menu, respectively, will be started for that window. In those cases the hotspot activation will not be signalled via SIGWINDOW. If *event_byte* is between 128 and 255 inclusive, the hotspot activation will be signalled via SIGWINDOW and the *event_code* packet will also be available via the packetized input queue.

The rectangles defined by hotspots can overlap. When they overlap, the hotspots form a stack within the window of which they are a part. When a hotspot is created, it is placed at the top of the stack. The position of a hotspot in the stack cannot be changed, except by deleting it and creating it again to force it to the top. The locator can only be over one hotspot at one time; if the locator is over a region that is the overlap between two or more hotspots, the locator is considered to be only over the top-most of those hotspots.

Whenever a hotspot is activated with a locator button, the activation only occurs on the downstroke of the button, not the upstroke. The upstroke of that button is never transmitted.

An enter/exit hotspot does not require that the window be the selected window for it to be activated. A button press over a hotspot whose *bmask* includes that button in a window that is not the selected window will activate the hotspot but will not select the window.

HARDWARE DEPENDENCIES

Series 500:

Hotspot routines (*whotspot_**) are not supported on Series 500; they work only on the Series 300.

SEE ALSO

wgskbd(3W),whotspot_delete(3W), whotspot_get(3W), whotspot_set(3W), winput_read(3W), wsetsigmask(3W), weventpoll(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise the hotspot_id for the created hotspot is returned. See *errno*(2) for more information.

NAME

whotspot_delete – delete a hotspot

SYNOPSIS

int whotspot_delete(fd,hotspot_id);
int fd;
int hotspot_id;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

hotspot_id
is the identifier of the hotspot to be released.

DISCUSSION

This call deletes the hotspot identified by *hotspot_id* for the window specified by the opened *fd*. If the hotspot is to be activated on enter/exit and the locator is currently over the hotspot, the hotspot will not be activated when it is deleted.

HARDWARE DEPENDENCIES

Series 500:
Hotspot routines (*whotspot_**) are not supported on Series 500; they work only on Series 300.

SEE ALSO

whotspot_create(3W), whotspot_get(3W), whotspot_set(3W), winput_read(3W), wsetsigmask(3W), weventpoll(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

whotspot_get - get hotspot data

SYNOPSIS

int whotspot_get(fd,hotspot_id,bmask,x,y,w,h,event_byte);
int fd;
int hotspot_id;
int *bmask;
int *x,*y,*w,*h;
int *event_byte;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

hotspot_id
is the identifier for an existing hotspot.

bmask is a pointer to the button bit mask for activating the hotspot.

x,y are pointers to the coordinates of the upper left corner of the rectangular hotspot region in device units relative to the upper left corner of the virtual raster of the window.

w,h are pointers to the width and height of the rectangular hotspot region in device units.

event_byte
is a pointer to the value for *event_byte* for this hotspot.

DISCUSSION

This call inquires the data of the hotspot identified by *hotspot_id* for the window specified by the opened *fd.*

HARDWARE DEPENDENCIES

Series 500:
Hotspot routines (*whotspot_**) are not supported on Series 500; they work only on Series 300.

SEE ALSO

whotspot_create(3W), whotspot_delete(3W), whotspot_set(3W), winput_read(3W), wsetsigmask(3W), weventpoll(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

whotspot_set - set hotspot data

SYNOPSIS

int whotspot_set(fd,hotspot_id,bmask,x,y,w,h,event_byte);
int fd;
int hotspot_id;
int bmask;
int x,y,w,h;
int event_byte;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

hotspot_id
is the identifier for an existing hotspot.

bmask is the button bit mask for activating the hotspot.

x,y are the coordinates of the upper left corner of the rectangular hotspot region in device units relative to the upper left corner of the virtual raster of the window.

w,h are the width and height of the rectangular hotspot region in device units. These parameters must be greater than zero.

event_byte
is the event to occur when the hotspot is activated.

DISCUSSION

This routine sets the data for the hotspot identified by *hotspot_id* for the window specified by the opened *fd.*

A hotspot can be disabled, by setting *bmask* to 0 or by placing the hotspot outside of the virtual raster of the window (e.g., set *x* and *y* to -10,-10 and *w* and *h* to 1,1).

The parameters are defined more thoroughly in the *whotspot_create*(3W) page.

HARDWARE DEPENDENCIES

Series 500:
Hotspot routines (*whotspot_**) are not supported on Series 500; they work only on Series 300.

SEE ALSO

whotspot_create(3W),whotspot_delete(3W),whotspot_get(3W),winput_read(3W), wsetsigmask(3W), weventpoll(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wiconic – change a window to/from an icon

SYNOPSIS

int wiconic(fd,value);
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

value determines the action taken by this routine; valid values and their effects follow:

- -1 return the window's iconic state: 1 is returned if the window is iconic; otherwise, 0 is returned.
- 0 display the window normally (non-iconic). This is the default state at window create time.
- 1 display the window as an icon.

DISCUSSION

Inquires on or sets a window's iconic state.

SEE ALSO

wseticon(3W),wseticonpos(3W),wgeticonpos(3W).

DIAGNOSTICS

A value of 0 or 1 is returned unless *fd* does not refer to a window, in which case -1 is returned. See *errno*(2) for more information.

NAME

window – summary of window library routines

DISCUSSION

The window library, **/usr/lib/libwindow.a**, contains routines that do window functions normally done by commands, for example, moving, selecting, or changing the size of a window. Window library routines also allow programs to do functions not attainable through commands, for example, event detection, user-defined icons, or changing graphics window softkeys.

Programs that call window library routines must be sure to link the window library (lwindow).

The **/usr/include/window.h** header file contains many useful type and constant definitions used by window library routines. Programs should use these definitions when calling window library routines.

Window library routines are summarized below. For more information on each routine, consult its reference page.

Window Management Routines

wcreate_graphics(3W)	Create a *graphics* window type device interface.
wcreate_term0(3W)	Create a *Term0* window type device interface.
wdestroy(3W)	Destroy a window.
wdfltpos(3W)	Return the default location for the next window or icon to create; also return the next default name.
wgetname(3W)	Return the path name of a window's window type device interface.
wgetscreen(3W)	Get information (resolution, etc) about the current display screen device.
wget_see_thru(3W)	Get the see_thru color index.
winit(3W)	Initialize a communication path with the window manager or a window type device interface.
wminquire(3W)	Return the value of a window system environment variable.
wmkill(3W)	Kill the window manager and the window system.
wmpathmake(3W)	Build a path name from a window system environment variable and a user-supplied suffix.
wmrepaint(3W)	Repaint all windows and the desk top.
wset_see_thru(3W)	Set the see_thru color index.
wshuffle(3W)	Shuffle windows up or down through the display stack.
wterminate(3W)	Release window manager or window type resources allocated by *winit*(3W).

Window Manipulation Routines

wautodestroy(3W)	Set or determine a window's autodestroy status; used with *wrecover*(3W).
wautoselect(3W)	Set or determine a *Term0* window's autoselect status.
wautotop(3W)	Set or determine a *Term0* window's autotop status.
wbanner(3W)	Set or determine a window's border type (thin, normal, or no border).

wbottom(3W)	Display a window as the bottom window in the display stack, or determine if a window is bottom.
wconceal(3W)	Conceal a window, or determine if a window is concealed.
wgetbcolor(3W)	Get a window's border foreground and background colors.
wgetbcoords(3W)	Get coordinate information (*x*,*y* pixel location, and pixel width and height) for a window's border.
wgetcoords(3W)	Get coordinate information (*x*,*y* pixel location, pixel width and height, pan position, raster width and height) for a window's user (contents) area.
wmove(3W)	Change a window's *x*,*y* pixel location.
wpan(3W)	Change the view into a *graphics* window's raster.
wpauseoutput(3W)	Pause or resume output to a *Term0* window.
wrecover(3W)	Set or determine a window's recover state; used with *wautodestroy*(3W).
wselect(3W)	Set or determine a window's selected status; i.e., whether the keyboard is attached to a window.
wsetbcolor(3W)	Set the foreground and background colors of a window's border.
wsetlabel(3W)	Change a window's label, displayed in the window's border.
wsize(3W)	Change a window's size.
wtop(3W)	Display a window as the top window in the display stack, or determine if a window is top.

Icon Manipulation Routines

wgeticonpos(3W)	Return an icon's *x*,*y* pixel location.
wiconic(3W)	Set or determine whether a window is displayed as an icon or normal.
wseticon(3W)	Change a window's icon to the representation given in an icon file.
wseticonpos(3W)	Set a the *x*,*y* location of a window's icon.

Event Detection Routines

weventclear(3W)	Clear event(s) for a window.
weventpoll(3W)	Poll for event(s) that may have occurred in a window.
wgetsigmask(3W)	Return the current event mask for a window.
wsetsigmask(3W)	Set the event mask for a window.

Locator and Echo Routines

wgetecho(3W)	Get information about a window's echo (pointer).
wgetlocator(3W)	Return the locator's current *x*,*y* pixel location and a mask representing which locator buttons are currently pressed.
wgetrasterecho(3W)	Return information about the echo's image for a given window.
wget_hw_sprite_color(3W)	Get the color indexes used for the sprite when using the

	hardware support for sprites.
wscrn_sprite_mode(3W)	Set or determine whether a window is in full-screen sprite control mode.
wsetecho(3W)	Set a window's echo to a specific type; used with *wsetrasterecho*(3W) when creating user-defined echoes.
wsetlocator(3W)	Set the locator's position, relative to a window.
wsetrasterecho(3W)	Set a user-defined echo type for a window; used with *wsetecho*(3W).
wset_hw_sprite_color(3W)	Set the color indexes used for displaying the sprite when using hardware support.

Graphics Window Scroll Bar Routines

wscroll_get(3W)	Get information about a graphics window's scroll bars.
wscroll_set(3W)	Set information about a graphics window's scroll bars.

Graphics Window Hotspot Rectangle Routines

whotspot_create(3W)	Create a hotspot rectangle in a graphics window.
whotspot_delete(3W)	Delete a hotspot rectangle from a graphics window.
whotspot_get(3W)	Return information about a specific hotspot rectangle in a given graphics window.
whotspot_set(3W)	Set information for a specific hotspot rectangle in a given graphics window.

User-Defined Menu Routines

wmenu_activate(3W)	Activate a user-defined pop-up menu for a window.
wmenu_create(3W)	Create a pop-up menu for a window.
wmenu_delete(3W)	Delete (remove) a pop-up menu from a window.
wmenu_eventread(3W)	Determine which item was selected from a pop-up menu in a window.
wmenu_item(3W)	Specify or change an item in a window's pop-up menu.

Graphics Window Input Routines

wgskbd(3W)	Set or determine the graphics window input mode (Mode 0, 1, or 2).
winput_conf(3W)	Set or determine the configuration of a graphics window's input channel.
winput_getroute(3W)	Return the path name of a window to which a window's input is routed.
winput_read(3W)	Read *event_code* packets (defined in **window.h**) from a graphics window.
winput_setroute(3W)	Reroute a graphics window's input to a different graphics window.
winput_widpath(3W)	Return the path name of a graphics window, given the window's window id.

Graphics Window Softkey Routines

wsfk_mode(3W)	Turn graphics window softkeys on or off.
wsfk_prog(3W)	Define the softkeys for a graphics window.

Term0 Font Management Routines

altfont_term0(3W)	Set or determine the current Term0 alternate font.
basefont_term0(3W)	Set or determine the current Term0 base font.
fontgetid_term0(3W)	Return the font ID of a Term0 font, if the font is currently loaded.
fontgetname_term0(3W)	Return the path name of a font file for a loaded Term0 font.
fontload_term0(3W)	Load a Term0 font.
fontreplaceall_term0(3W)	Replace the current base and alternate font with new base and alternate fonts.
fontsize_term0(3W)	Return the pixel width and height of all loaded fonts. (All loaded fonts must be the same size.)
fontswap_term0(3W)	Replace a loaded Term0 font with a font that is currently not loaded.
fromxy_term0(3W)	Convert *x,y* pixel coordinates to column and row coordinates, based on the current font size.
toxy_term0(3W)	Convert column and row coordinates to *x,y* pixel coordinates, based on the current font size.

EXAMPLES

The following example compiles a program, named *winprog.c*, that calls window library routines.

```
cc winprog.c -lwindow
```

SEE ALSO

windows(1).

NAME

winit – initialize window device

SYNOPSIS

int winit(fd);
int fd;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* or *window manager* device interface.

DISCUSSION

Initialize window device for subsequent calls to the window device. This is called just after *open*(2) or Starbase *gopen*(3S).

SEE ALSO

wterminate(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

winput_conf - read/set configuration of window input channel

SYNOPSIS

```
#include <window.h>
int winput_conf(fd,param,value);
int fd;
int param,value;
```

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

param the parameter to set or inquire.

value the value to set the desired parameter. If *value* is -1, then the parameter will only be inquired. Otherwise, the parameter will be set (enabled) or cleared (disabled) depending on whether *value* is 1 or 0, respectively. The value of the specified parameter is returned by winput_conf on a successful call, otherwise a -1 is returned.

DISCUSSION

This library call is supported by only the **graphics** window type. It sets or returns an input configuration parameter for a window. This routine should only be called in input mode 2.

The following parameters, defined in **window.h**, can be supplied as **param**:

K_TRACK

If set, report all locator moves. The default is cleared. Locator moves are reported only when the keyboard is connected to the window. Locator moves during an interactive size/move or window system pop-up menu are not reported. Moves will be reported relative to the current window virtual raster in pixel coordinates. **Warning**: enabling K_TRACK will cause window system performance to degrade.

K_LANGUAGE

Language nationality of the keyboard. The **value** parameter can be changed to a language supported on the current keyboard family. The default language is that of the keyboard attached. The supported values currently are:

K_I_USASCII	ITF United States.
K_I_BELGIAN	ITF Belgian.
K_I_CANENG	ITF Canadian English.
K_I_DANISH	ITF Danish.
K_I_DUTCH	ITF Dutch.
K_I_FINNISH	ITF Finnish.
K_I_FRENCH	ITF French (AZERTY).
K_I_CANFRENCH	ITF Canadian French.
K_I_SWISSFRENCH	ITF Swiss French.
K_I_GERMAN	ITF German.
K_I_SWISSGERMAN	ITF Swiss German.
K_I_ITALIAN	ITF Italian.
K_I_NORWEGIAN	ITF Norwegian.
K_I_EUROSPANISH	ITF European Spanish.
K_I_LATSPANISH	ITF Latin Spanish.
K_I_SWEDISH	ITF Swedish.
K_I_UNITEDK	ITF United Kingdom.
K_I_KATAKANA	ITF Katakana.
K_I_SWISSFRENCH2	ITF Swiss French II.

K_I_SWISSGERMAN2	ITF Swiss German II.
K_I_KANJI	ITF Kanji.

K_CAPSMODE
: CAPS Key. If set (the default) then characters will automatically be converted to upper case when the CAPS key is pressed. If cleared, disable handling of CAPS. (In any case, pressing the CAPS key will cause a key code to be sent.)

K_EXTEND
: Alternate Keyboards. For some languages, the EXTEND key can be used to toggle between normal and alternate keyboards; for other languages, the EXTEND key is a modifier to get additional codes. This parameter controls whether the EXTEND key will perform the language dependent function. If set (the default) then the language dependent function will be enabled. If cleared, the function will be disabled.

K_CONTROL
: Control Collapsing of Printables. If set (the default for *wgskbd*(3W) mode 0), then the control key will cause characters from 64 to 127 decimal to be collapsed to their control values before being received. If cleared (the default for *wgskbd*(3W) mode 1 or 2) then collapsing will be disabled. (In any case, pressing the CONTROL key will cause the control bit to be set when the associated key code is sent.)

K_SHIFT
: Shift Collapsing of Capitals. If set (the default), then the shift key will cause keys subject to the CAPS key to have their case inverted before being received. If cleared, this inversion is disabled. (In any case, pressing the SHIFT key will cause the shift bit to be set when the associated key code is sent.)

K_META
: Enable META Modifiers. If set (the default), the presence of meta keys will be recognized by setting the appropriate meta bits when key codes are sent. A zero parameter will disable this capability.

K_META_EXTEND
: Enable Left-EXTEND as META Modifier. This is effective only when K_META is set. If this parameter is set (the default for *wgskbd*(3W) mode 1 or 2), the left-EXTEND key will become the meta key. For the Katakana keyboard it will also switch the keyboard to the ROMAN keyboard at the same time. If cleared (the default for *wgskbd*(3W) mode 0), the left-EXTEND key is simply treated as an EXTEND key. (In any case, pressing the left-EXTEND key will cause a key code to be sent.)

K_CAPSLOCK
: CAPS Mode State. Set means locked; cleared (the default) means not locked.

K_KANAKBD
: Katakana Keyboard. If set, means that the alternate Katakana keyboard is currently active. If cleared (the default), means it is not in effect. This bit is only meaningful for a Katakana language keyboard.

K_KANJI
: Enable KANJI mode. This parameter is effective only when the keyboard language is K_META_EXTEND. If this parameter is set, the left extend key is used to toggle the state of K_KANJIKBD. If cleared (the default), means it is not in effect.

K_KANJIKBD
: Kanji input mode. If set, left meta key will be used as a key only. If cleared (the default), the left meta key will be used as a meta key.

HARDWARE DEPENDENCIES

Series 500:

Graphics window input routines (*winput_**) are not supported on Series 500; they only work on Series 300.

SEE ALSO

winput_read(3W), wgskbd(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise the current value of the specified parameter is returned. See *errno*(2) for more information.

NAME

winput_getroute – determine window input routing path

SYNOPSIS

int winput_getroute(fd,routepath);
int fd;
char *routepath;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

routepath
is a pointer to a space to be filled with the null-terminated path name of the window type device interface currently receiving all input and events that would normally go to the window represented by **fd**. The path name string cannot exceed 40 characters in length.

DISCUSSION

This routine determines whether any input or event routing is being done for the window referenced by **fd** (see *winput_setroute*(3W)). If input routing is enabled for the window, then **routepath** will point to the path name of the window receiving this input. Otherwise **routepath** will point to a null (zero-length) string.

In a *multi-hop* route, this routine will return the path name of the immediate window to which its input is routed, not necessarily the final destination.

HARDWARE DEPENDENCIES

Series 500:
Graphics window input routines (*winput_**) are not supported on Series 500; they work only on Series 300.

SEE ALSO

winput_read(3W), winput_setroute(3W), winput_widpath(3W).

DIAGNOSTICS

A return value of -1 indicates failure; see *errno*(2) for more information. Otherwise, the window id, **wid**, of the window referenced by **fd** is returned.

NAME

winput_read - read from window input channel

SYNOPSIS

```
#include <window.h>
int winput_read(fd,bufadr,count);
int fd;
struct event_code *bufadr;
int count;
```

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

bufadr
the address of the buffer to read data into.

count number of packets to read.

DISCUSSION

This routine may only be used in conjunction with *wgskbd*(3W) mode 2, known as *packetized input mode.*

Winput_read reads *event_code packets* (described below) into a buffer pointed to by **bufadr**. Each packet placed in the buffer represents a single key press or an event.

Winput_read attempt to read **count** packets into the **bufadr** buffer. The number of packets read is returned, and could be less than the number requested, or even zero. For optimal performance, set **count** to 25.

If more than 25 events occur between calls to *winput_read*, some event information may be lost. When this occurs, an *event_code* packet is sent with its **event_byte** field set to K_OVERFLOW. Any key presses or hot spot events occuring after the overflow condition will be lost. However, the **most recent** event for all other types of events will not be lost. For example, suppose K_OVERFLOW occurs and the user moves a window several times before the program calls *winput_read.* Then all window move events between the K_OVERFLOW and the final window move will be lost. *But* the last window move won't be lost; the program will still receive a packet for the last window move.

If *winput_read* is called from a window that has no data available, then the action taken by *winput_read* depends on the O_NDELAY value, set when the window type device interface was opened. See *open*(2) and *fcntl*(2) for details on O_NDELAY. If O_NDELAY is set, *winput_read* returns 0, meaning that no event packets have yet occurred since the last call to *winput_read.* If O_NDELAY is not set, *winput_read* will block until a signal or an event occurs in the window, at which point *winput_read* returns an *event_code* packet for the event. If a signal occurred, but not an event, *winput_read* will return 0.

The packets returned in **bufadr** are defined by the *event_code* structure (found in **window.h**):

```
struct event_code {
    unsigned char control_byte;
    unsigned char data_byte;
    unsigned char event_byte;
    unsigned char event_cause;
    unsigned int timestamp;
    unsigned int wid;
    int x;
    int y;
};
```

timestamp
A 32-bit integer number that specifies when, in milliseconds, the packet was received.

wid The window id that the packet originated in. (The *winput_widpath*(3W) routine can be used to convert this into a window path name.)

control_byte
is a bit field consisting of the OR of the following:

K_SHIFT_B	Shift
K_CONTROL_B	Control
K_META_B	META
K_EXTEND_B	Right Extend
K_UP	UP==1 DOWN==0
K_NPAD	Number Pad Key
K_EVENT	Key==0 Event==1
K_SPECIAL	Special Keys such as CLR LINE, ENTER, f1, etc.

data_byte
is either a special key (if K_SPECIAL is true), or it is a processed character value appropriate for the current configuration of the input channel (as set by *winput_conf*(3W)).

Below is a list of all the unique keys (other than alpha-numeric) that are currently supported for the K_SPECIAL data case. (Notice that with the addition of the control bits, every possible unique normal physical mechanical key combination can be represented.)

K_ILLEGAL	if no character, data byte will contain ILLEGAL
K_CAPS_ON	Only when K_CAPSMODE enabled
K_CAPS_OFF	Only when K_CAPSMODE enabled
K_GO_ROMAN	Only when language is Katakana or Kanji
K_GO_KATAKANA	Only when language is Katakana or Kanji
K_GO_KANJI	Only when language is Katakana or Kanji
K_GO_NOKANJI	Only when language is Katakana or Kanji
K_CAPS_LOCK	Only when K_CAPSMODE disabled
K_TAB	
K_F1	
K_F2	
K_F3	
K_F4	
K_F5	
K_F6	
K_F7	
K_F8	
K_DOWN_ARROW	ROLL_DOWN when shifted
K_UP_ARROW	ROLL_UP when shifted
K_LEFT_ARROW	ROLL_LEFT when shifted
K_RIGHT_ARROW	ROLL_RIGHT when shifted
K_INSERT_LINE	
K_DELETE_LINE	
K_INSERT_CHAR	
K_DELETE_CHAR	
K_BACKSPACE	

K_RETURN	
K_EXTEND_LEFT	
K_EXTEND_RIGHT	
K_META_LEFT	
K_META_RIGHT	
K_BUTTON1	
K_BUTTON2	
K_BUTTON3	
K_BUTTON4	
K_BUTTON5	
K_BUTTON6	
K_BUTTON7	
K_BUTTON8	In proximity–i.e., the graphics tablet stylus or puck switch was placed on the graphics tablet.
K_BREAK	RESET_KEY when shifted
K_STOP	
K_SELECT	
K_NP_ENTER	
K_NP_K0	
K_NP_K1	
K_NP_K2	
K_NP_K3	
K_HOME_ARROW	
K_PREV	
K_NEXT	
K_ENTER	PRINT when shifted K_SYSTEM (USER when shifted)
K_MENU	
K_CLR_LINE	
K_CLR_DISP	

event_byte
is used if the K_EVENT bit is set in **control_byte**. It can have any of the following values:

K_MOVE_CT	Window Move Completion
K_SIZE_LR_CT	Size Lower Right Completion
K_ICON_SHK	Shrink to an icon
K_ICON_EXP	Expand from an icon
K_PAUSE	Pause Toggle Bit
K_DESTROY	Window is trying to be destroyed
K_SELECTED	Window was just selected
K_USELECTED	Window was just unselected
K_REPAINT	Window needs to be repainted
K_FSSM_ABORT	Full screen sprite mode aborted
K_MOUSE_MOVE	Mouse has moved
K_BUTTON	Button press/release
K_MENU_ITEM	User popup menu item selected
K_ELEV_CT	Elevator move complete
K_SB_ARROW	Scroll bar arrow event
K_OVERFLOW	Overflow occurred - events lost
K_USER_HS	First user-defined hotspot

K_USER_HS+1	Second user-defined hotspot
K_USER_HS+*n*	*n*th user-defined hotspot
K_USER_HS+127	Last user-defined hotspot

x,y are used if the K_EVENT bit is set in **control_byte**. For events (other than hotspots) that generate the SIGWINDOW signal, **x** and **y** are set to the values described in *weventpoll*(3W). For hotspot events, **x** and **y** provide the position of the locator, relative to location 0,0 of the window's raster.

event_cause
is used if the K_EVENT bit is set in **control_byte** and if the event was caused by a hotspot or valid menu item selection. It indicates which locator button or keyboard key was pressed, or if a hotstop was entered or exited. **event_cause** can have any one of the following values:

EC_NONE	No button pressed and no hotspot entered.
EC_BUTTON1	Button 1 was pressed
EC_BUTTON2	Button 2 was pressed
EC_BUTTON3	Button 3 was pressed
EC_BUTTON4	Button 4 was pressed
EC_BUTTON5	Button 5 was pressed
EC_BUTTON6	Button 6 was pressed
EC_BUTTON7	Button 7 was pressed
EC_BUTTON8	Button 8 was pressed, in proximity-i.e., the graphics tablet stylus or puck switch was placed on the graphics tablet
EC_SELECT	SELECT was pressed
EC_ENTER	Hotspot was entered
EC_EXIT	Hotspot was exited

HARDWARE DEPENDENCIES

Series 500:

Graphics window input routines (*winput_**) are not supported on Series 500; they work only on Series 300.

SEE ALSO

open(2), fcntl(2), select(2), winput_conf(3W), winput_getroute(3W), winput_setroute(3W), winput_widpath(3W), wgskbd(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise the number of packets read is returned. See *errno*(2) for more information.

NAME

winput_setroute – routes input and events to another window

SYNOPSIS

int winput_setroute(fd,routepath);
int fd;
char *routepath;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

routepath
is a pointer to a null-terminated path name of the window to receive all input and events occurring in the window represented by *fd.* This string can be a maximum of 40 characters in length.

DISCUSSION

This routine reroutes a (source) window's input to another (destination) window. After calling this routine, all keystrokes and events that would normally go to the source window will instead be sent to the destination window. The **fd** parameter is the file descriptor returned from opening the source window's device interface; **routepath** points to the path name of the destination window's window type device interface.

On receiving input from the source window, the destination window will handle the data in a way appropriate to its input mode, as set by the *wgskbd*(3W) routine, regardless of the input mode of the source window.

Each window has its own set of *winput_conf*(3W) configuration parameters. When using input routing, it is normally desirable to have all windows set to the same configuration. This can be accomplished either by setting each window to the same *wgskbd*(3W) mode or by explicitly calling *winput_conf*(3W) to set the parameters of each window.

Although it is desirable to have the same *winput_conf* configuration parameters for each window, it is not absolutely necessary. If the source and destination windows have different *winput_conf* parameters, then input from the destination window will conform to the *winput_conf* parameters of the source window.

In **Mode 2** (*packetized input mode*), the **wid** field of the *event_code* structure will be set to the window id of the window from which the packet originated. Because of this, **wid** is useful to determine which window an event or keypress actually occurred in. In all other modes, the process that reads data from the **routepath** window cannot determine which window the input data originated from.

The call will return the window id, **wid**, for the window referenced by *fd* if it is successful. Otherwise -1 will be returned, and *errno*(2) will be set appropriately.

If a window which is the target of one or more input routes is destroyed, all input routing for those windows is cancelled. Input and event routing can also be disabled by passing a null value for **routepath**.

When input routing is enabled for a window, any data waiting in the input queue at the time of the *winput_setroute* is flushed. This is typically done immediately after an *open*(2) call, to ensure routing of important input and events.

Multi-hop routing is allowed. For example, the input to window **a** can be routed to window **b**, which in turn can be routed to window **c**. In this example, any input to windows **a**, **b**, or **c** will be routed to window **c**.

Note, however, that routing loops are not allowed. For example, routing **a** to **b**, **b** to **c**, and **c** to **a** will cause an error. Also, routing a window to itself (e.g., window **d** to window **d**) is a loop and is not allowed.

If routing is disabled in one of the windows in a multi-hop route, then routing still is valid for the windows that precede the disabled window in the route path. For example, suppose **a** routes to **b**, **b** routes to **c**, and **c** routes to **d**. Then, if the route from **c** to **d** is disabled, any input originating from windows **a**, **b**, or **c** is still available only from window **c**, while input originating from window **d** is still available only from window **d**.

HARDWARE DEPENDENCIES

Series 500:

Graphics window input routines (*winput_**) are not supported on Series 500; they work only on Series 300.

SEE ALSO

wgskbd(3W),winput_conf(3W),winput_getroute(3W),winput_read(3W), winput_widpath(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise the window id, **wid**, of the window referenced by *fd* is returned. See *errno*(2) for more information.

NAME

winput_widpath - get path name for a window id

SYNOPSIS

int winput_widpath(wmfd,wid,wname);
int wmfd;
int wid;
char *wname;

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

wid is the window id for a window as returned by either *winput_setroute*(3W) or *winput_getroute*(3W). **Note** that a window id is **not the same** as a window's file descriptor.

wname
is a pointer to a space to be filled with the path name of the window referred to by **wid**. This string will be null-terminated and will not exceed 40 characters in length.

DISCUSSION

This routine works only with graphics windows. On return, the **wname** parameter points to the null-terminated path name of the window whose window id is **wid**. To get the window id for a path name, see *winput_getroute*(3W). A window's window is is also returned by *winput_setroute*(3W).

HARDWARE DEPENDENCIES

Series 500:
Graphics window input routines (*winput_**) are not supported on Series 500; they work only on Series 300.

SEE ALSO

winput_getroute(3W), winput_read(3W), winput_setroute(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wmenu_activate – activate a menu

SYNOPSIS

wmenu_activate (fd, menuid, value)
int fd;
int menuid;
int value;

DESCRIPTION

fd the file descriptor for an opened *window type* device interface.

menuid
id of the menu whose activation status will be changed or inquired.

value determines the action take by this routine:

-1	inquires (returns) present value: 0 means that automatic pop-up is disabled; 1 means that automatic pop-up is enabled
0	disable automatic pop-up of the menu
1	will pop-up the menu automatically upon next press of an enabled start menu button
2	pops up the menu immediately, thereby waiting for a selection of an item

See **window.h** for named definitions of the values.

DISCUSSION

This routine inquires or changes the menu activation state, as determined by the *value* parameter described above.

SEE ALSO

wmenu_create(3W), wmenu_destroy(3W), wmenu_eventread(3W), wmenu_item(3W).

DIAGNOSTICS

If *fd, menuid,* or *value* is invalid; a -1 is returned. See *errno*(2) for further information.

NAME

wmenu_create - create a menu

SYNOPSIS

```
#include <window.h>
int wmenu_create (fd, cbits, button_mask, parent_menuid, parent_item)
int fd;
int cbits;
int button_mask;
int parent_menuid, parent_item;
```

DESCRIPTION

fd is the file descriptor of the window for which the menu is to be created.

cbits control bits, see **window.h** for supported options.

button_mask
the enabling mask for the start of a menu and the selection of an item within the menu. The lower byte represents the start menu enable mask where 0x7F are buttons 1-7, button one being the least significant bit, and 0x80 enables the Select key. Likewise the byte above that represents the select item enable mask where 0x7F00 are buttons 1-7, button one being the least significant bit, and 0x8000 enables the Select key.

parent_menuid
menu id of the parent menu to this one. MENU_NOPARENT, from **window.h**, indicates there is no parent menu. At this time, parent menus are NOT supported, so always use MENU_NOPARENT.

parent_item
item id of the parent item within the parent menu. Always use MENU_NOPARENT, from **window.h**, for this parameter.

DISCUSSION

This routine creates a user-defined pop-up menu according to the parameters described above. It returns as its value the id of the created menu.

SEE ALSO

wmenu_activate(3W), wmenu_delete(3W), wmenu_eventread(3W), wmenu_item(3W).

DIAGNOSTICS

If any of the following conditions occur, -1 is returned; otherwise 0 is returned:

* *fd* is invalid
* there are no buttons enabled for selecting an item
* *parent_menuid* is anything greater than -1
* no more menus can be created.

See *errno*(2) for further information.

NAME

wmenu_delete - delete a menu

SYNOPSIS

wmenu_delete (fd, menuid)
int fd;
int menuid;

DESCRIPTION

fd file descriptor of the *window type* device interface from which the menu is to be deleted.

menuid
id of the menu to be deleted from the window.

DISCUSSION

This routine deletes the specified menu.

SEE ALSO

wmenu_activate(3W), wmenu_create(3W), wmenu_eventread(3W), wmenu_item(3W).

DIAGNOSTICS

If *fd* or *menuid* is invalid, a -1 is returned; otherwise 0 is returned. See *errno*(2) for further information.

NAME

wmenu_eventread – read the menu event

SYNOPSIS

#include <window.h>
wmenu_eventread (fd, menuid, itemno)
int fd;
int *menuid;
int *itemno;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

menuid
id of the menu from which an item was selected.

itemno
id of the item selected; -1 indicates the menu was somehow aborted.

DISCUSSION

This routine returns menu event data. It's the responsibility of the application to sort menu event data by menuid. A return value of 0 indicates that no menu event data was available from *fd.* A return value of greater than 0 indicates the number of queued menu data events the first of which contained is returned in *menuid* and *itemno.*

This routine is typically used in conjunction with notification by SIGWINDOW and *weventpoll*(3W) that a menu event has happened. The EVENT_MENU bit (see **window.h**) is used with *wsetsigmask*(3W).

SEE ALSO

wmenu_activate(3W), wmenu_create(3W), wmenu_delete(3W), wmenu_item(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise the number of queued menu data events is returned. See *errno*(2) for further information.

NAME

wmenu_item – specify or change a menu item

SYNOPSIS

#include <window.h>
int wmenu_item (fd, menuid, itemno, type, disp_sel, type_struct)
int fd;
int menuid;
int itemno;
int type;
int disp_sel;
char *type_struct;

DESCRIPTION

fd is a file descriptor for an opened *window type* device interface.

menuid
id of the menu to which an item will be added or changed.

itemno
id of the item to be affected or MENU_NEWITEM, see **window.h,** which establishes a new item within the menu.

type determines what type this item will be, valid values include, MENU_STRING and MENU_SEPARATOR, see **window.h** for a complete list.

disp_sel
Sets the select, display, and tracking attributes of this item. Selection attributes are MENU_NOTSELECTABLE or MENU_SELECTABLE. Display attributes are MENU_DISPNORM or MENU_DISPGREY. Track attributes are MENU_TRACKNOCHNG or MENU_TRACKINV. see **window.h** for a complete list.

type_struct
This is the data associated with the type. For MENU_STRING pass the text string to be displayed. For MENU_SEPARATOR pass a single character whose ordinal value is the thickness of the separator line.

For example, if you pass a space (' ') character as this parameter, and the type is MENU_SEPARATOR, then the thickness of the line will be 32 pixels (because an ASCII space character is 32 in decimal).

Note that values of 0 or less default to a thickness of two. For example, if you pass a null string (''), then the separator thickness defaults to 2 pixels.

DISCUSSION

This routine changes or adds an item to the specified menu. It returns as its value the id of the item.

SEE ALSO

wmenu_activate(3W), wmenu_create(3W), wmenu_delete(3W), wmenu_eventread(3W).

DIAGNOSTICS

If either of the following occur, -1 is returned; otherwise 0 is returned:

* *fd, menuid, itemno,* or *type* is invalid
* there is no room for the item. See *errno*(2) for further details.

NAME

wminquire – get a window manager environment variable

SYNOPSIS

int wminquire(fd,environ,target);
int fd;
char *environ,*target;

DESCRIPTION

fd is an integer file descriptor for an opened *window manager* or *window type* device interface.

environ
is a pointer to a null terminated character string with the name of an environment variable known to the window manager.

target is a pointer to sufficient space to place the resulting string. No string will be longer than 40 characters.

DISCUSSION

Place the value of environment variable name, as used by the window manager, *environ*, in string space pointed to by *target.*

SEE ALSO

wmpathmake(3W),wmstart(1).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NANE

wmkill – kill the window manager

SYNOPSIS

int wmkill(wmfd);
int wmfd;

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

DISCUSSION

This routine kills the window manager, all associated windows and affiliated processes.

SEE ALSO

wmstart(1),wmready(1),wmstop(1).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wmove - move the location of a window

SYNOPSIS

int wmove(fd,x,y);
int fd;
int x,y;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

x,y are the screen pixel coordinates for the position of the upper- corner of window view with respect to the physical screen.

DISCUSSION

Move the window's position on the physical screen. This position can be anywhere on or off the screen.

x and *y* refer to the upper left corner of the contents portion, not the border. Positive, negative and zero values for *x* and *y* are allowed. (0,0) is the upper left hand corner of the screen.

SEE ALSO

curses(3X), wsize(3W),wpan(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

WARNING

The HP Windows/9000 library (-lwindow) and the curses library (-lcurses) both define the name *wmove*(). If both libraries are needed by the same program, the program must be compiled in pieces, with the window and curses pieces linked individually with their respective libraries.

For example, the following will not work:

```
cc main.c curses_piece.c window_piece.c -lcurses -lwindow   # won't work
```

Instead use:

```
cc -c curses_piece.c window_piece.c
ld -r curses_piece.o -lcurses -o curses.o -h _wmove
ld -r window_piece.o -lwindow -o window.o -h _wmove
cc main.c curses.o window.o
```

If Starbase libraries were required for a device such as the 9837 bit-mapped display, the last cc could be:

```
cc main.c curses.o window.o -ldd9837 -lwindow -lsb1 -lsb2
```

NAME

wmpathmake – build a path name using an environment variable

SYNOPSIS

int wmpathmake(environ,suffix,target);
char *environ, *suffix, *target;

DESCRIPTION

environ
is a pointer to a null terminated character string with the name of an environment variable containing a directory path.

suffix is a pointer to a null terminated character string with a path name in it.

target is a pointer to sufficient space, minimum of 40 chars, to return the resulting path name.

DISCUSSION

Wmpathmake expands the given suffix and environment variable to an absolute path name. Rules of processing are: unless the suffix begins with "/", "./", or "../", the suffix is appended to the value of the named environment variable. In all cases the path name is expanded to an absolute path name. Note the following:

- The space pointed at by *target*, must be a minimum of 40 characters.
- If the value of the environment variable duplicates the leading portion of the suffix, then no appending is done. For example, "x/y/z" cannot be appended to "x/y".
- *wmpathmake* requires **stdin** and **stdout** to be open in order to expand "./" or "../", see *getcwd(3C)*.

Typically *wmpathmake* is used to attach the value of the $*WMDIR* environment variable to "wm" or some window name. Care should be taken when $*WMDIR* is "./" or "../", in that $*WMDIR* must be the same for both the window system and the user, otherwise unexpected results will occur.

SEE ALSO

wminquire(3W).

DIAGNOSTICS

A return value of -1 indicates that the resulting path name was truncated at 40 characters; otherwise a return value of 0 indicates no error occurred.

NAME

wmrepaint – repaint the desk top

SYNOPSIS

int wmrepaint(wmfd);
int wmfd;

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

DISCUSSION

This routine sends a repaint event to all window units, then totally repaints the desk top. This library call exists in order to repair damage caused to the display, should such damage occur. For example, if the following is executed to a device with several windows displayed,

```
echo "lsakdjflaksdj" > /dev/console
```

the display may not act as expected–i.e., it is damaged.

Graphics windows may or may not be repaired by this library call. It depends upon if the application program using the windows has the ability to repaint upon receiving a repaint event (see *weventpoll*).

SEE ALSO

weventpoll(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wpan – pan the window

SYNOPSIS

int wpan(fd,dx,dy);
int fd;
int dx,dy;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

dx,dy is the pixel delta x and delta y offset in the window raster of the upper left hand corner of the view that is being shown.

DISCUSSION

Changes the position of the view within the raster, in graphics terminology this is called *panning.*

This operation is limited by the current position of the view into the raster and the raster's size. Under no circumstance will the pan cause the size of the view of the window to change. If an out-of-range operation is attempted, nothing will happen and a -1 will be returned.

SEE ALSO

wsize(3W),wmove(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wpauseoutput – pause output to a window

SYNOPSIS

```
int wpauseoutput(fd,value);
int fd;
int value;
```

DESCRIPTION

fd is an integer file descriptor for an opened *Term0 window type* device interface.

value is the set/interrogate parameter. Following are valid values for this parameter and the resulting effect:

-1 returns 0 if the window is not paused, otherwise it returns 1.

0 resume output to the window. On a *Term0* window this is equivalent to XON. This is the default state at window create time.

1 all output to the window is stopped (a paused condition). On a *Term0* window this is equivalent to an XOFF.

DISCUSSION

This call inquires or sets whether to pause output to a window. This routine works only with *Term0* window types.

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 or 1 is returned. See *errno*(2) for further information.

NAME

wrecover – recover a window

SYNOPSIS

```
#include <window.h>
int wrecover (fd, value)
int fd;
int value;
```

DESCRIPTION

fd is an integer file descriptor for an opened window device interface.

value is an integer which determines the action of this routine.

DISCUSSION

This call inquires or sets a window's recover state.

Setting a window's state to *recovered* will cause the window to be automatically destroyed when its device interface is no longer open by any process. The exact time that the window is destroyed is determined by the *wautodestroy*(3W) routine. If a window is recovered **and** it is set to be automatically destroyed, then the window will be immediately destroyed when its device interface is closed by every process that had it opened; if a window is recovered **and** it is not set for auto-destruction, then the window is destroyed when its device interface is closed by every process that had it opened **and only when** a new window is about to be created.

Setting *value* to -1 causes this routine to return whether the window is recoverable or not. A 1 value returned indicates that the window denoted by this file descriptor is currently recoverable, while a 0 value returned indicates that it is not.

If *value* is set to 0, then the window is **not** recovered upon next creation of a window. This is the default state at window create time.

Setting *value* to 1 causes the window to be recovered.

SEE ALSO

wautodestroy(3W).

DIAGNOSTICS

A value of 0 or 1 is returned unless *fd* does not refer to a window, in which case -1 is returned. See *errno*(2) for further information.

NAME

wscrn_sprite_mode – set full-screen sprite control mode

SYNOPSIS

#include <window.h>
int wscrn_sprite_mode(fd,value);
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

value is the set/interrogation parameter for which the following values are valid:

-1	return the window's current screen sprite control mode.
0	disable full-screen sprite control mode. This is the default mode.
1	enable full-screen sprite control mode. See discussion below.

DISCUSSION

This routine inquires or sets the window's screen sprite control mode. If the window is in full-screen sprite control mode, the window manager will set the locator echo for the full-screen to that of the window specified by *fd*. When the locator is moved over other windows, the desktop, or window borders, the echo will not change. In addition, a locator button press or SELECT key press while in full-screen sprite control mode will not cause the default actions; rather, the button press will be transmitted to the specified window as a button-press event.

If the user presses any key other than the SELECT key or buttons while in full-screen sprite control mode, full-screen sprite control mode is aborted. The SIGWINDOW signal is then sent to all processes that have used *wsetsigmask*(3W) to enable SIGWINDOW on EVENT_ABORT for this window.

Note that **all** button press events are sent to the application when full-screen sprite mode is in effect, regardless of whether the locator is over a window or the background (desk top). Therefore, if the application is to be consistent with the semantics of the window manager (e.g., give a system pop-up menu if the SELECT button is pressed over the desk top pattern), it will take extra work to do so. This extra work probably involves using *wminquire*(3W) to get the value of WMIUICONFIG and to act appropriately to button presses, based on the value of WMIUICONFIG.

HARDWARE DEPENDENCIES

Series 500:

Full-screen sprite control mode is not supported on Series 500; it only works on Series 300.

SEE ALSO

wsetecho(3W),wsetrasterecho(3W),wsetsigmask(3W),weventpoll(3W),wmstart(1).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 or 1 is returned. See *errno*(2) for more information.

Full-screen sprite control mode aborts if the interactive operation times-out, just like other interactive operations (controlled by the WMIATIMEOUT variable; see *wmstart*(1)).

NAME

wscroll_get – interrogate scroll bar information

SYNOPSIS

#include <window.h>
int wscroll_get(fd,which,mode,value,min,max,size);
int fd;
int which;
int *mode,*value,*min,*max,*size;

DESCRIPTION

fd is an integer file descriptor of an opened *graphics window type* device interface.

which identifies whether to interrogate the vertical or horizontal scroll bar, and must be one of:

SCROLLBAR_V
Vertical scroll bar (in the right hand border)

SCROLLBAR_H
Horizontal scroll bar (in the bottom border)

mode returns the mode of the specified scroll bar, and will consist of the OR of:

SCROLLBAR_ELEVATOR
the elevator is enabled.

SCROLLBAR_ARROWS
the arrows are enabled.

SCROLLBAR_USERMODE
The scroll bar is enabled in user mode (otherwise it is in pan mode). If SCROLLBAR_USERMODE and SCROLLBAR_ELEVATOR are both set, the following parameters are also returned:

value	The position (on a scale of **min** to **max**) of the elevator.
min	The value to be associated with the upper or left end of the elevator.
max	The value to be associated with the lower or right end of the elevator.
size	The size (on a scale of **min** to **max**) of the elevator.

DISCUSSION

Wscroll_get returns information about the vertical or horizontal scroll bar elevators and arrows in the border of a window.

HARDWARE DEPENDENCIES

Series 500:
Scroll bar capabilities are not supported on Series 500; they work only on Series 300.

SEE ALSO

wscroll_set(3W), wsetsigmask(3W), weventpoll(3W), winput_read(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wscroll_set - control of window scroll bar elevators and arrows

SYNOPSIS

int wscroll_set(fd,which,mode [**,value** [**,min,max,size**]] **);**
int fd;
int which;
int mode;
int value;
int min,max,size;

DESCRIPTION

fd is an integer file descriptor of an opened *graphics window type* device interface.

which identifies whether vertical, horizontal, or both scroll bars are to be affected. This parameter can be set by OR'ing one or both of the following values, defined in **window.h**.

SCROLLBAR_V	vertical scroll bar (in the right-hand border)
SCROLLBAR_H	horizontal scroll bar (in the bottom border)

mode controls the *mode* of the specified scroll bar. This parameter is set by OR'ing one or more of the following bit values, defined in **window.h**.

SCROLLBAR_ELEVATOR
enable the elevator.

SCROLLBAR_ARROWS
enable the arrows.

SCROLLBAR_USERMODE
enable *user mode*. If this is not set, then the default *pan mode* is used. If both SCROLLBAR_ELEVATOR and SCROLLBAR_USERMODE are set, then the **value** parameter must also be supplied.

SCROLLBAR_SCALE
meaningful only if the SCROLLBAR_USERMODE and SCROLLBAR_ELEVATOR are specified. If this is set, then the **min**, **max**, and **size** parameters must also be supplied.

value The position where the elevator should appear in the scroll bar within the **min** and **max** parameters, discussed next.

min The value to be associated with the upper (for SCROLLBAR_V) or left (for SCROLLBAR_H) end of the scroll bar.

If both SCROLLBAR_V and SCROLLBAR_H are set in the **which** parameter, then **min** is the same for both the horizontal and vertical scroll bar.

max The value to be associated with the lower (for SCROLLBAR_V) or right (for SCROLLBAR_H) end of the elevator. **min** must be less than **max.**

The elevator cannot be moved outside the range specified by **min** and **max.**

If both SCROLLBAR_V and SCROLLBAR_H are set in the **which** parameter, then **min** is the same for both the horizontal and vertical scroll bar.

size The size (on a scale of **min** to **max** of the elevator). For example, if **min** and **max** are -100 and +100, respectively, then setting **size** to 50 will cause the elevator to be one fourth the size of the scroll bar.

DISCUSSION

Wscroll_set enables or disables vertical and/or horizontal scroll bar elevators and arrows in the border of a window. Enabling or disabling scroll bar elevators or arrows has the side effect of turning on or off their visual representation.

There are two modes of operation for the scroll bars: *pan mode* and *user mode*. In *pan mode*, the size and position of an elevator are determined by the size and position of the view with respect to the raster. Completion of an interactive move of the elevator or activation of scroll arrows cause the window to pan. Conversely, panning the window via *wpan*(3W) causes the elevators to move.

In *user mode*, the size and position of the elevator are specified by the user in a user-supplied integer coordinate system and movement of an elevator or activation of a scroll arrow causes an event to occur. If the scale information is never supplied, defaults of 0 to 100 are used for **min** and **max**, and 10 is used for **size**.

To interactively move an elevator while in *pan mode*, position the pointer over the elevator and press a select button. This initiates an interactive operation which is echoed as a dotted box in the elevator "shaft." Move the locator to position the box to the desired location and press a select button to complete the move. Elevator operations can be aborted in the same manner as any other interactive operation (e.g., press a key or timeout).

The activation of a *user mode* scroll bar arrow or elevator can be detected by enabling SIGWINDOW for the desired event via *wsetsigmask*(3W), then using *weventpoll*(3W) to determine what event occurred. It can also be detected using *winput_read*(3W) while the graphics window is in **Mode 2** (see *wgskbd*(3W)). For elevator events, *weventpoll*(3W) will return which scroll bar (SCROLLBAR_V or SCROLLBAR_H) and the coordinate for the most recent elevator movement in x and y respectively. For arrow events, *weventpoll*(3W) will return in x and y the accumulated number of arrow events (x positive is right, y positive is down) since the last *weventpoll*(3W).

In *user mode*, activation of an elevator does not cause the elevator to move. The application must move the scroll bar by calling *wscroll_set* again with the **value** parameter set to the appropriate value.

When using elevators in *pan mode*, you may wish to reverse the sense of the border arrows; see the WMIUICONFIG environment variable in *wmstart*(1) for details on how to do this.

HARDWARE DEPENDENCIES

Series 500:

Scroll bar capabilities are not supported on Series 500; they work only on Series 300.

SEE ALSO

wscroll_get(3W), wsetsigmask(3W), weventpoll(3W), winput_read(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wselect – attach a keyboard to the specified window

SYNOPSIS

int wselect(fd,value);
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

value is the set/interrogation parameter; the following values are valid for this parameter:

-1 returns 0 if a keyboard is not attached to this window, otherwise it returns a 1 if a keyboard is attached.

0 the keyboard is detached from this window. The keyboard is then automatically attached to the topmost window unless the window specified is the topmost window. In this case the keyboard is attached to the next window down in the stack. If there is only one window the keyboard remains attached to it.

1 the keyboard is attached to the specified window. This has the side effect of detaching the keyboard from its current window.

DISCUSSION

This call inquires or sets whether the keyboard is attached to this window device.

There can only be one window at a time connected to the keyboard.

The window attached to the keyboard is also the window that is currently subject to manipulation by the interactive human interface.

SEE ALSO

wautoselect(3WS), wshuffle(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 or 1 is returned. See *errno*(2) for further information.

NAME

wset_hw_sprite_color – set hardware sprite colors

SYNOPSIS

int wset_hw_sprite_color(fd,fgcolor,bgcolor);
int fd;
int fgcolor,bgcolor;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

fgcolor,bgcolor
the foreground sprite color, and background sprite color. Colors are taken from the system (Starbase) color map.

DISCUSSION

Defaults for *fgcolor* and *bgcolor* are 1 and 0 respectively for the default at window creation time, except for border sprites that are set to the window's border fgcolor and bgcolor.

The sprite is displayed as follows. All *bgcolor* index values in the sprite data will be displayed as *bgcolor,* and all *non-bgcolor* index values will be displayed as *fgcolor.*

If a color is out of range for the device an error is generated. Normally the valid range is from 0 to (2^N – 1), where N is the number of planes.

This call can be made on all devices, but will only have an effect on the HP98730 when using the hardware support for sprites. The use of the hardware for sprites can be disabled with the WMCONFIG environment variable.

SEE ALSO

wget_hw_sprite_color(3W), wmstart(1).

DIAGNOSTICS

A return of –1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wset_see_thru - set see_thru color index value

SYNOPSIS

int wset_see_thru(wmfd,see_thru);
int wmfd;
int see_thru;

DESCRIPTION

fd is an integer file descriptor for an opened *window manager* device interface.

see_thru,
the see_thru color index. Colors are taken from the system (Starbase) color map.

DISCUSSION

This tells the window system what color index to use in the overlay planes when displaying **see_thru** and IMAGE windows. This routine will modify the system color map by setting the old see_thru index back to the last known color map values and setting the new index to see_thru (transparent).

Defaults for **see_thru** are set at window creation time.

If a color is out of range for the device, an error is generated. Normally the valid range is from 0 to (2^N - 1), where N is the number of planes.

This call can be made on all devices, but will only have an effect on the HP98730.

SEE ALSO

wget_see_thru(3W), windows(1), wmstart(1).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAMER

wsetbcolor – set window border colors

SYNOPSIS

int wsetbcolor(fd,fgborder,bgborder);
int fd;
int fgborder,bgborder;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

fgborder,bgborder
the foreground label character color, and background label and border color. Colors are taken from the system (Starbase) color map.

DISCUSSION

Defaults for *fgborder* and *bgborder* are determined from the WMBDRFGCLR and WMBDRBGCLR window system environment variables.

If a color is out of range for the device, it is modified to be within the valid range. Normally the valid range is from 0 to (2^N - 1), where N is the number of planes. If double buffering color mode is enabled via the WMCONFIG environment variable , then the valid range is from 0 to (2^(N/2) - 1), and the color actually displayed is of the form (C << (N/2) + C), where C is the color to display.

SEE ALSO

wgetbcolor(3w).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wsetecho - set echo

SYNOPSIS

int wsetecho(fd,echo_value,x2,y2,optimized);
int fd;
int echo_value,x2,y2,optimized;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

echo_value
is the type of echo to be used. Predefined types are listed below:

0	invisible window
1	device's best echo
2	full screen cross hair
3	small tracking cursor
4	rubber band line, with anchor point at*x2*,*y2*
5	rubber band rectangle, with anchor point at *x2*,*y2*
6	alpha digital representation (for displaying characters on external devices such as button boxes)
7	user defined raster cursor
8	box of width x2, height y2; where the upper left corner is the current cursor position
>8	device dependent representation

x2,y2 are the echo anchor position or box width and height.

optimized
is a boolean with two possible values:

0	means echo is exactly as defined. Movement via *wsetlocator* will do exactly as specified.
1	means echo representation may be modified as per the device to make it track in the best way possible. Movement via *wsetlocator* may snap to device dependent boundaries to take advantage of specialized hardware (such as a 4 X 4 pixel tile mover).

DISCUSSION

Set the window's echo to the specified Starbase compatible echo type. An additional routine, *wsetrasterecho*, is needed to set up a raster echo.

SEE ALSO

wgetecho(3W), wscreen_sprite_mode(3W), wset_hw_sprite_color(3W), wsetrasterecho(3W).

DIAGNOSTICS

A -1 indicates failure; otherwise, 0 is returned. See *errno*(2) for further information.

NAME

wseticon – set icon

SYNOPSIS

```
#include <fonticon.h>
int wseticon(fd,imode,lmode,iconfile);
int fd;
int imode,lmode;
char *iconfile;
```

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

imode is the mode controlling which parts of the icon are displayed; following are valid values for this paramter:

0 then do not display the picture portion of the icon.

1 display the type-dependent picture for the icon.

2 display the icon picture referenced by *iconfile.*

lmode is the mode controlling the display choice of the label:

0 do not display a label.

1 display the label last specified by *wsetlabel.*

iconfile
is the full path name of an icon file. See the iconstruct definition in **/usr/include/fonticon.h**.

DISCUSSION

wseticon sets the iconic representation for the window. The default at creation depends upon the window type. The foreground and background colors are the same as the window border colors.

Note that *imode* and *lmode* cannot both equal 0.

SEE ALSO

wiconic(3W),wseticonpos(3W),wgeticonpos(3W),wsetlabel(3W).

DIAGNOSTIC

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wseticonpos – set icon position

SYNOPSIS

```
int wseticonpos(fd,x,y);
int fd;
int x,y;
```

DESCRIPTION

fd	is an integer file descriptor for an opened *window type* device interface.
x,y	are the screen pixel coordinates for the position of the upper left hand corner of iconic representation of the window type with respect to the physical screen. Positive, negative and zero values for x and y are allowed.

DISCUSSION

wseticonpos sets the icon representation position associated with the window device indicated by *fd.*

(0,0) is the upper left hand corner of the screen.

SEE ALSO

wiconic(3W),wgeticonpos(3W),wseticon(3W).

DIAGNOSTICS

A -1 indicates failure; otherwise a 0 is returned. See *errno*(2) for further information.

NAME

wsetlabel – set label in window border

SYNOPSIS

int wsetlabel(fd,label);
int fd;
char *label;

DESCRIPTION

fd	is an integer file descriptor for an opened *window type* device interface.
label	is a pointer to a null-terminated character string to be displayed as the label in the border of the window, the title in the SFK area, the title in the popup menu for this window, and the label in the icon for the window.

DISCUSSION

This call sets the label to be displayed in the border of a window, the title in the SFK area, the title in the popup menu for the window, and the label in the icon for the window.

Only the first 12 bytes of a label are displayed, except for the label in the border of a *graphics window type*, in which up to 128 bytes are displayed. If the last byte in a label is the first byte of a 2-byte character, then the character is ignored and not displayed.

HARDWARE DEPENDENCIES

Series 500:

Graphics window labels can contain no more than 12 characters on Series 500.

HP-15 (2-byte) characters are not supported on Series 500.

SEE ALSO

wborder(1).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wsetlocator – set window locator position

SYNOPSIS

init wsetlocator(fd,x,y);
int fd;
int x,y;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

x,y contain the current locator position.

DISCUSSION

Set the window system locator position relative to the specified window unit. The definition of the locator will switch to whatever the locator is defined for the object in the window system that it is over.

x and **y** are relative to the current window data space (not the screen). They are also in pixel units. The locator hot spot determines which part of the locator is actually at the coordinates, see *wsetrasterecho* and *wsetecho.* For a window type, they are relative to the contents portion, (not the border).

SEE ALSO

wgetlocator(3W),wscreen_sprite_mode(3W),wsetrasterecho(3W),wsetecho(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wsetrasterecho – set raster echo

SYNOPSIS

int wsetrasterecho(fd,dx,dy,w,h,rule,mask_rule,mask,image);
int fd;
int fd,dx,dy,w,h,rule,mask_rule;
char *mask,*image;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

dx,dy are the offset of the echo hot spot to the upper left corner of the echo, usually negative.

w,h are the echo's size. Setting both to 0 (zero) caused the default echo to be used (small arrow).

rule,mask_rule
are the echo's replacement rules used when displaying a raster cursor. These rules are defined in the HP Starbase documentation.

mask is a pointer to a bit-per-pixel array of 128 characters. This array is used to make the mask for the raster echo. The mask is placed on the screen before the image. Each bit represents two possible values for each pixel: zero or all ones.

image is a pointer to a byte-per-pixel array of 1024 characters. This array is used to make the image for the raster echo.

DISCUSSION

Set the window's raster echo to the specified STARBASE compatible echo type. An additional routine, *wsetecho*, is needed to set up more basic echo attributes first.

SEE ALSO

wgetrasterecho(3W), wset_hw_sprite_color(3W), wsetecho(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wsetsigmask – set window SIGWINDOW interrupt mask

SYNOPSIS

#include <window.h>
#include <sys/signal.h>
int wsetsigmask(fd,mask);
int fd;
int mask;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

mask is used to specify conditions to interrupt the user process with the SIGWINDOW signal. *mask* is a set of bits defined by the following bit names, where the default mask setting is 0. The header file containing these defines is **/usr/include/window.h**. SIGWINDOW is defined in **/usr/include/sys/signal.h**.

EVENT_B1_DOWN	button 1 pressed
EVENT_B1_UP	button 1 released
EVENT_B2_DOWN	button 2 pressed
EVENT_B2_UP	button 2 released
EVENT_B3_DOWN	button 3 pressed
EVENT_B3_UP	button 3 released
EVENT_B4_DOWN	button 4 pressed
EVENT_B4_UP	button 4 released
EVENT_B5_DOWN	button 5 pressed
EVENT_B5_UP	button 5 released
EVENT_B6_DOWN	button 6 pressed
EVENT_B6_UP	button 6 released
EVENT_B7_DOWN	button 7 pressed
EVENT_B7_UP	button 7 released
EVENT_B8_DOWN	button 8 pressed (in proximity)–i.e., the puck switch or stylus was placed on the graphics tablet
EVENT_B8_UP	button 8 released (not in proximity)–i.e, the puck switch or stylus was picked up from the graphics tablet
EVENT_ECHO	echo moved while selected
EVENT_MOVE	window moved
EVENT_SIZE	window sized
EVENT_SELECT	KBD Attach state changed
EVENT_REPAINT	window needs to be repainted
EVENT_MENU	selection made from user-defined menu
EVENT_HOTSPOT	a hotspot has been activated
EVENT_DESTROY	window destroyed
EVENT_BREAK	BREAK key pressed
EVENT_ICON	iconic state changed
EVENT_ELEVATOR	window border elevator moved
EVENT_SB_ARROW	scroll bar arrow selected
EVENT_ABORT	interactive operation was aborted

DISCUSSION

This command is used set up signal generation events. Each process that desires to receive SIGWINDOW for a set of mask bits must call *wsetsigmask* separately. One window can send

SIGWINDOW to up to three processes. The receiving program still has to catch SIGWINDOW with the *signal* function call. The default signal behavior is SIG_IGN (ignore signal).

EVENT_HOTSPOT, EVENT_DESTROY, EVENT_BREAK, EVENT_ICON, EVENT_ELEVATOR, EVENT_SB_ARROW, and EVENT_ABORT are supported only by *graphics* windows.

If the limit of three processes per window is exceeded then ENOSPC error will be returned.

HARDWARE DEPENDENCIES

Series 500:

The Series 500 does not support the following events:

EVENT_HOTSPOT
EVENT_DESTROY
EVENT_BREAK
EVENT_ICON
EVENT_ELEVATOR
EVENT_SB_ARROW
EVENT_ABORT

SEE ALSO

weventclear(3W), weventpoll(3W), wgetsigmask(3W), signal(2).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wsfk_mode - switch to soft key mode

SYNOPSIS

```
int wsfk_mode(fd,mode);
int fd;
int mode;
```

DESCRIPTION

fd is an integer file descriptor for an opened *graphics window type* device interface.

mode is the mode to which the softkeys will be set:

0 softkey labels are turned off.

1 softkey labels are turned on.

DISCUSSION

Switch the soft key labels for the specified window to the specified mode. If the mode specified is soft key labels on, the labels will be displayed whenever the window becomes the selected window. If the mode is off, the no labels will be displayed whenever the window becomes the selected window.

SEE ALSO

wsfk_prog(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wsfk_prog – set programmable soft keys

SYNOPSIS

int wsfk_prog (fd,key,label,separator);
int fd;
int key;
char *label;
char separator;

DESCRIPTION

fd is an integer file descriptor for an opened *graphic window type* device interface.

key is the key label number to set. 1 thru 8 are for keys f1 thru f8 respectively.

label a pointer to a null terminated character string. Only the first 16 bytes are used. If less than 16 characters are supplied, blanks are assumed for the remaining characters. Keys 1 through 8 are displayed as two rows of eight 1-byte characters. If the eighth byte is the first byte of a 2-byte character, then all bytes from the eighth byte on are shifted up by one (eight goes to nine, nine goes to ten, etc) and the eighth byte is replaced with a space. This causes the 2-byte character to be wrapped to the second line instead of being split across the lines. If the last byte is the first byte of a 2-byte character, the byte is truncated.

separator
indicates whether or not to display a horizontal line to separate the the shifted and unshifted text for the sfk. If *separator* is non-zero, the separator line is displayed. If *separator* is zero, the separator line is not displayed.

DISCUSSION

Set the displayed string for a particular programmable soft key label. When the soft keys are displayed, the specified string will be displayed for that soft key. If a soft key is selected via SELECT or locator button, the key code for that soft key will be sent.

HARDWARE DEPENDENCIES

Series 500:
HP-15 (2-byte) characters are not supported on Series 500.

SEE ALSO

wsfk_mode(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wshuffle – shuffle windows' relative locations within display stack

SYNOPSIS

int wshuffle(wmfd,value);
int wmfd;
int value;

DESCRIPTION

wmfd is an integer file descriptor for an opened *window manager* device interface.

value determines how the windows are shuffled through the display stack:

0 means to shuffle the top window to the bottom, e.g., deal from the top of the deck.

1 means to shuffle the bottom window to the top, e.g., deal from the bottom of the deck.

DISCUSSION

This call shuffles the whole visible stack of windows. It always attaches the keyboard to the resulting top most window.

SEE ALSO

wtop(3W), wbottom(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 or 1 is returned. See *errno*(2) for further information.

NAME

wsize – change the size of a window

SYNOPSIS

```
int wsize(fd,w,h);
int fd;
int w,h;
```

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

w,h is the width and height of the window view measured in pixels. Width and height always refer to the contents portion, not the border.

DISCUSSION

This procedure is used to change the size of the window, which means to change the view into the virtual raster.

This operation is limited by the current position of the view into the raster and the raster's size. Under no circumstance will the size cause the delta x,y into the raster to change. Also the size operation may cause a portion of the window to be off screen.

A request to set the size of the window to less than the minimum width or height will set the window size to the minimum width or height. In the same way, a request to set the size to more than the maximum width or height will set the window size to the maximum width or height. The maximum size for a graphics *window type* is its raster size minus the current pan delta set via *wpan*. The maximum size for a term0 *window type* is its initial creation size. If the border type set via *wbanner*(3W) is "normal" (i.e. display a border), then the minimum size for both a term0 and graphics *window type* is such that at least one character of the label is visible. If the border type is "thin", then the minimum size is 1 pixel for graphics and 1 character for term0.

SEE ALSO

wpan(3W),wbanner(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wterminate – release window resources

SYNOPSIS

int wterminate(fd);
int fd;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* or *window manager* device interface.

DISCUSSION

This call releases window resources that were allocated by a call to *winit*(3W), when window communication was started with the window or window manager.

SEE ALSO

winit(3W).

DIAGNOSTICS

A return of -1 indicates failure; otherwise 0 is returned. See *errno*(2) for more information.

NAME

wtop – move the window to the top of the window stack

SYNOPSIS

int wtop(fd,value);
int fd;
int value;

DESCRIPTION

fd is an integer file descriptor for an opened *window type* device interface.

value is the set/interrogation parameter for which the following values are valid:

-1	return the window's current top status. If the window is the top window in the stack, then 1 is returned; otherwise, 0 is returned.
0	causes the routine to do nothing.
1	places the window or icon on top of all other windows. If the window is concealed make it visible and on top.

DISCUSSION

This call inquires or sets whether this window is top most.

SEE ALSO

wbottom(3W),wshuffle(3W).

DIAGNOSTICS

A value of 0 or 1 is returned unless *fd* does not refer to a window, in which case -1 is returned. See *errno*(2) for more information.

COMMAND/KEYWORD INDEX

WHAT IT IS!

This is an "in context" index. It is sometimes called a permuted index. It is generated from the **NAME** part, i.e., the command, or routine name, and its description part, of the reference pages. Each significant word in the command/description line is used as an index entry, a keyword. It is "in context" because the words in the command/description line surrounding, i.e., preceding and/or following, the keyword are included with it. In certain cases, especially when more than one command is included in the **NAME** part of a reference page, some license is used to select command line information.

HOW IT IS!

These conventions apply:

commands

- a command, or routine name, is printed in *italics* and followed by a colon (:)

context

- the context for the keyword is contained in the left and center columns
- to read the context of a keyword, start with the *command* or, if the command part has been truncated, with the right brace (}); read to the end of the center column and wrap around to the beginning of the left column; read across to the *command* or either brace

keyword

- the keyword, or look-up word, for an index entry is the left word in the center column, i.e., the one under the ↓
- keywords are in alphabetical order; uppercase is folded into lowercase
- if a keyword is a command it is preceded by an asterisk (∗)

PAGE NAME

- the PAGE NAME, where the command is presented in this reference, appears in the right column; the number and/or letter in parentheses identifies the section where this referenced PAGE NAME is located

special characters

∗	an asterisk indicates the keyword is a command
:	a colon separates a command or routine name from its description
{	a left brace indicates where the end of the command line is truncated
}	a right brace indicates where the beginning of the command line is truncated

truncation

- if a command line, including the command and the description, is too long to fit in the context area, the end and/or the beginning of the line is truncated
- braces are used to indicate where a truncation occurs

EXAMPLE

Here is a command/description from this reference manual:

fontreplaceall_term0 - replace the current base font and alternate font

And here are the entries produced for the index:

	↓ **command*/keyword	PAGE_NAME
the current base font and	alternate font }replace	FONTREPLACEALL_TERM0(3W)
}replace the current	base font and alternate font	FONTREPLACEALL_TERM0(3W)
font }replace the	current base font and alternate	FONTREPLACEALL_TERM0(3W)
}replace the current base	font and alternate font	FONTREPLACEALL_TERM0(3W)
current base font and alternate	font }replace the	FONTREPLACEALL_TERM0(3W)
the current base font and{	**fontreplaceall_term0*: replace	FONTREPLACEALL_TERM0(3W)
and{ *fontreplaceall_term0*:	replace the current base font	FONTREPLACEALL_TERM0(3W)

: separates command from description; **}** indicates location of leading truncation; **{** indicates location of trailing truncation;

	↓ **command*/keyword	PAGE_NAME
elevators and{ *wscroll_set*:	control of window scroll bar	WSCROLL_SET(3W)
color, label *wborder*:	control window border style,	WBORDER(1)
to characters *fromxy_term0*:	convert from Term0 pixel units	FROMXY_TERM0(3W)
pixels *toxy_term0*:	convert Term0 character units to	TOXY_TERM0(3W)
wgetbcoords: get border	coordinates	WGETBCOORDS(3W)
wgetcoords: get window	coordinates	WGETCOORDS(3W)
type *wcreate_graphics*:	create a **graphics** window	WCREATE_GRAPHICS(3W)
wcreate_term0:	create a **term0** window type	WCREATE_TERM0(3W)
whotspot_create:	create a hotspot	WHOTSPOT_CREATE(3W)
wmenu_create:	create a menu	WMENU_CREATE(3W)
existing windows *wsh*:	create new shells in new or	WSH(1)
wcreate:	create one or more new windows	WCREATE(1)
font }replace the	current base font and alternate	FONTREPLACEALL_TERM0(3W)
wgeticonpos: get	current icon position	WGETICONPOS(3W)
whotspot_get: get hotspot	data	WHOTSPOT_GET(3W)
whotspot_set: set hotspot	data	WHOTSPOT_SET(3W)
wdfltpos:	default window/icon position	WDFLTPOS(3W)
whotspot_delete:	delete a hotspot	WHOTSPOT_DELETE(3W)
wmenu_delete:	delete a menu	WMENU_DELETE(3W)
wmrepaint: repaint the	desk top	WMREPAINT(3W)
wdestroy:	destroy a window	WDESTROY(3W)
set autodestroy{ *wdestroy*:	destroy one or more windows or	WDESTROY(1)
path *winput_getroute*:	determine window input routing	WINPUT_GETROUTE(3W)
winit: initialize window	device	WINIT(3W)
relative locations within	display stack }shuffle windows'	WSHUFFLE(3W)
start the window system on one	display *wmstart*:	WMSTART(1)
stop the window system on one	display *wmstop*:	WMSTOP(1)
wdisp: change	displayability of windows	WDISP(1)
wgetecho: get	echo	WGETECHO(3W)
wgetrasterecho: get raster	echo	WGETRASTERECHO(3W)
wsetecho: set	echo	WSETECHO(3W)
wsetrasterecho: set raster	echo	WSETRASTERECHO(3W)
wgetlocator: get window	echo/locator parameter	WGETLOCATOR(3W)
}control of window scroll bar	elevators and arrows	WSCROLL_SET(3W)
wminquire: get a window manager	environment variable	WMINQUIRE(3W)
}build a path name using an	environment variable	WMPATHMAKE(3W)
wmenu_eventread: read the menu	event	WMENU_EVENTREAD(3W)
}routes input and	events to another window	WINPUT_SETROUTE(3W)
clear window locator	events *weventclear*:	WEVENTCLEAR(3W)
poll for window locator	events *weventpoll*:	WEVENTPOLL(3W)
wsh: create new shells in new or	existing windows	WSH(1)
wcreate_graphics: create a	**graphics** window type	WCREATE_GRAPHICS(3W)
wcreate_term0: create a	**term0** window type	WCREATE_TERM0(3W)
or inquire the Term0 alternate	font *altfont_term0*: set	ALTFONT_TERM0(3W)
}replace the current base	font and alternate font	FONTREPLACEALL_TERM0(3W)
set or inquire the Term0 base	font *basefont_term0*:	BASEFONT_TERM0(3W)
inquire the ID of a Term0	font *fontgetid_term0*:	FONTGETID_TERM0(3W)
inquire the name of a Term0	font *fontgetname_term0*:	FONTGETNAME_TERM0(3W)
fontload_term0: load a Term0	font	FONTLOAD_TERM0(3W)
current base font and alternate	font }replace the	FONTREPLACEALL_TERM0(3W)

: separates command from description; } indicates location of leading truncation; { indicates location of trailing truncation;

	↓ ∗*command*/keyword	PAGE_NAME
fontswap_term0: replace a Term0	font	FONTSWAP_TERM0(3W)
fontsize_size0: Term0	font size	FONTSIZE_TERM0(3W)
of a Term0 font	∗*fontgetid_term0*: inquire the ID	FONTGETID_TERM0(3W)
name of a Term0 font	∗*fontgetname_term0*: inquire the	FONTGETNAME_TERM0(3W)
font	∗*fontload_term0*: load a Term0	FONTLOAD_TERM0(3W)
the current base font and{	∗*fontreplaceall_term0*: replace	FONTREPLACEALL_TERM0(3W)
wfont: load and activate	fonts	WFONT(1)
wlist: list status of windows or	fonts	WLIST(1)
	∗*fontsize_size0*: Term0 font size	FONTSIZE_TERM0(3W)
font	∗*fontswap_term0*: replace a Term0	FONTSWAP_TERM0(3W)
pixel units to characters	∗*fromxy_term0*: convert from Term0	FROMXY_TERM0(3W)
wscrn_sprite_mode: set	full-screen sprite control mode	WSCRN_SPRITE_MODE(3W)
wgskbd: set keyboard mode for	graphics window type	WGSKBD(3W)
wget_hw_sprite_color: get	hardware sprite colors	WGET_HW_SPRITE_COLOR(3W)
wset_hw_sprite_color: set	hardware sprite colors	WSET_HW_SPRITE_COLOR(3W)
whotspot_get: get	hotspot data	WHOTSPOT_GET(3W)
whotspot_set: set	hotspot data	WHOTSPOT_SET(3W)
whotspot_create: create a	hotspot	WHOTSPOT_CREATE(3W)
whotspot_delete: delete a	hotspot	WHOTSPOT_DELETE(3W)
wgeticonpos: get current	icon position	WGETICONPOS(3W)
wseticonpos: set	icon position	WSETICONPOS(3W)
change a window to/from an	icon *wiconic*:	WICONIC(3W)
wseticon: set	icon	WSETICON(3W)
one or more windows or their	icons *wmove*: move	WMOVE(1)
fontgetid_term0: inquire the	ID of a Term0 font	FONTGETID_TERM0(3W)
get path name for a window	id *winput_widpath*:	WINPUT_WIDPATH(3W)
get see_thru color	index value *wget_see_thru*:	WGET_SEE_THRU(3W)
set see_thru color	index value *wset_see_thru*:	WSET_SEE_THRU(3W)
wgetscreen: get screen	information	WGETSCREEN(3W)
interrogate scroll bar	information *wscroll_get*:	WSCROLL_GET(3W)
winit:	initialize window device	WINIT(3W)
window *winput_setroute*: routes	input and events to another	WINPUT_SETROUTE(3W)
configuration of window	input channel }read/set	WINPUT_CONF(3W)
winput_read: read from window	input channel	WINPUT_READ(3W)
}determine window	input routing path	WINPUT_GETROUTE(3W)
window's border *wbanner*:	inquire about or control a	WBANNER(3W)
fontgetid_term0:	inquire the ID of a Term0 font	FONTGETID_TERM0(3W)
fontgetname_term0:	inquire the name of a Term0 font	FONTGETNAME_TERM0(3W)
altfont_term0: set or	inquire the Term0 alternate font	ALTFONT_TERM0(3W)
basefont_term0: set or	inquire the Term0 base font	BASEFONT_TERM0(3W)
information *wscroll_get*:	interrogate scroll bar	WSCROLL_GET(3W)
get window SIGWINDOW	interrupt mask *wgetsigmask*:	WGETSIGMASK(3W)
set window SIGWINDOW	interrupt mask *wsetsigmask*:	WSETSIGMASK(3W)
specify or change a menu	item *wmenu_item*:	WMENU_ITEM(3W)
wsfk_mode: switch to soft	key mode	WSFK_MODE(3W)
window type *wgskbd*: set	keyboard mode for graphics	WGSKBD(3W)
wselect: attach a	keyboard to the specified window	WSELECT(3W)
wselect: connect a window to the	keyboard	WSELECT(1)
wsfk_prog: set programmable soft	keys	WSFK_PROG(3W)
wmkill:	kill the window manager	WMKILL(3W)

: separates command from description; } indicates location of leading truncation; { indicates location of trailing truncation;

	↓ **command*/keyword	PAGE_NAME
wsetlabel: set	label in window border	WSETLABEL(3W)
window border style, color,	label *wborder*: control	WBORDER(1)
window: summary of window	library routines	WINDOW(3W)
wlist:	list status of windows or fonts	WLIST(1)
fontload_term0:	load a Term0 font	FONTLOAD_TERM0(3W)
wfont:	load and activate fonts	WFONT(1)
wmove: move the	location of a window	WMOVE(3W)
}shuffle windows' relative	locations within display stack	WSHUFFLE(3W)
weventclear: clear window	locator events	WEVENTCLEAR(3W)
weventpoll: poll for window	locator events	WEVENTPOLL(3W)
wsetlocator: set window	locator position	WSETLOCATOR(3W)
wminquire: get a window	manager environment variable	WMINQUIRE(3W)
wmready: tell if window	manager is awake and ready	WMREADY(1)
wmkill: kill the window	manager	WMKILL(3W)
get window SIGWINDOW interrupt	mask *wgetsigmask*:	WGETSIGMASK(3W)
set window SIGWINDOW interrupt	mask *wsetsigmask*:	WSETSIGMASK(3W)
wmenu_eventread: read the	menu event	WMENU_EVENTREAD(3W)
wmenu_item: specify or change a	menu item	WMENU_ITEM(3W)
wmenu_activate: activate a	menu	WMENU_ACTIVATE(3W)
wmenu_create: create a	menu	WMENU_CREATE(3W)
wmenu_delete: delete a	menu	WMENU_DELETE(3W)
wgskbd: set keyboard	mode for graphics window type	WGSKBD(3W)
set full-screen sprite control	mode *wscrn_sprite_mode*:	WSCRN_SPRITE_MODE(3W)
wsfk_mode: switch to soft key	mode	WSFK_MODE(3W)
their icons *wmove*:	move one or more windows or	WMOVE(1)
wmove:	move the location of a window	WMOVE(3W)
the window stack *wtop*:	move the window to the top of	WTOP(3W)
winput_widpath: get path	name for a window id	WINPUT_WIDPATH(3W)
fontgetname_term0: inquire the	name of a Term0 font	FONTGETNAME_TERM0(3W)
wmpathmake: build a path	name using an environment{	WMPATHMAKE(3W)
wgetname: get path	name	WGETNAME(3W)
wpauseoutput: pause	output to a window	WPAUSEOUTPUT(3W)
wpan:	pan the window	WPAN(3W)
get window echo/locator	parameter *wgetlocator*:	WGETLOCATOR(3W)
winput_widpath: get	path name for a window id	WINPUT_WIDPATH(3W)
variable *wmpathmake*: build a	path name using an environment	WMPATHMAKE(3W)
wgetname: get	path name	WGETNAME(3W)
determine window input routing	path *winput_getroute*:	WINPUT_GETROUTE(3W)
wpauseoutput:	pause output to a window	WPAUSEOUTPUT(3W)
fromxy_term0: convert from Term0	pixel units to characters	FROMXY_TERM0(3W)
convert Term0 character units to	pixels *toxy_term0*:	TOXY_TERM0(3W)
weventpoll:	poll for window locator events	WEVENTPOLL(3W)
wdfltpos: default window/icon	position	WDFLTPOS(3W)
wgeticonpos: get current icon	position	WGETICONPOS(3W)
wseticonpos: set icon	position	WSETICONPOS(3W)
wsetlocator: set window locator	position	WSETLOCATOR(3W)
wsfk_prog: set	programmable soft keys	WSFK_PROG(3W)
wgetrasterecho: get	raster echo	WGETRASTERECHO(3W)
wsetrasterecho: set	raster echo	WSETRASTERECHO(3W)
winput_read:	read from window input channel	WINPUT_READ(3W)

: separates command from description; } indicates location of leading truncation; { indicates location of trailing truncation;

	↓ *command*/keyword	PAGE_NAME
wmenu_eventread:	read the menu event	WMENU_EVENTREAD(3W)
window input{ *winput_conf*:	read/set configuration of	WINPUT_CONF(3W)
if window manager is awake and	ready *wmready*: tell	WMREADY(1)
wrecover:	recover a window	WRECOVER(3W)
wshuffle: shuffle windows'	relative locations within{	WSHUFFLE(3W)
wterminate:	release window resources	WTERMINATE(3W)
wmrepaint:	repaint the desk top	WMREPAINT(3W)
fontswap_term0:	replace a Term0 font	FONTSWAP_TERM0(3W)
and{ *fontreplaceall_term0*:	replace the current base font	FONTREPLACEALL_TERM0(3W)
wterminate: release window	resources	WTERMINATE(3W)
another window *winput_setroute*:	routes input and events to	WINPUT_SETROUTE(3W)
summary of window library	routines *window*:	WINDOW(3W)
determine window input	routing path *winput_getroute*:	WINPUT_GETROUTE(3W)
wgetscreen: get	screen information	WGETSCREEN(3W)
wscroll_set: control of window	scroll bar elevators and arrows	WSCROLL_SET(3W)
wscroll_get: interrogate	scroll bar information	WSCROLL_GET(3W)
wget_see_thru: get	see_thru color index value	WGET_SEE_THRU(3W)
wset_see_thru: set	see_thru color index value	WSET_SEE_THRU(3W)
}destroy one or more windows or	set autodestroy attributes	WDESTROY(1)
wsetecho:	set echo	WSETECHO(3W)
mode *wscrn_sprite_mode*:	set full-screen sprite control	WSCRN_SPRITE_MODE(3W)
wset_hw_sprite_color:	set hardware sprite colors	WSET_HW_SPRITE_COLOR(3W)
whotspot_set:	set hotspot data	WHOTSPOT_SET(3W)
wseticonpos:	set icon position	WSETICONPOS(3W)
wseticon:	set icon	WSETICON(3W)
window type *wgskbd*:	set keyboard mode for graphics	WGSKBD(3W)
wsetlabel:	set label in window border	WSETLABEL(3W)
alternate font *altfont_term0*:	set or inquire the Term0	ALTFONT_TERM0(3W)
font *basefont_term0*:	set or inquire the Term0 base	BASEFONT_TERM0(3W)
wsfk_prog:	set programmable soft keys	WSFK_PROG(3W)
wsetrasterecho:	set raster echo	WSETRASTERECHO(3W)
wset_see_thru:	set see_thru color index value	WSET_SEE_THRU(3W)
wsetbcolor:	set window border colors	WSETBCOLOR(3W)
wsetlocator:	set window locator position	WSETLOCATOR(3W)
mask *wsetsigmask*:	set window SIGWINDOW interrupt	WSETSIGMASK(3W)
windows *wsh*: create new	shells in new or existing	WSH(1)
locations within{ *wshuffle*:	shuffle windows' relative	WSHUFFLE(3W)
wgetsigmask: get window	SIGWINDOW interrupt mask	WGETSIGMASK(3W)
wsetsigmask: set window	SIGWINDOW interrupt mask	WSETSIGMASK(3W)
fontsize_size0: Term0 font	size	FONTSIZE_TERM0(3W)
wsize: change the	size of a window	WSIZE(3W)
wsize: change	sizes of one or more windows	WSIZE(1)
wsfk_mode: switch to	soft key mode	WSFK_MODE(3W)
wsfk_prog: set programmable	soft keys	WSFK_PROG(3W)
attach a keyboard to the	specified window *wselect*:	WSELECT(3W)
wmenu_item:	specify or change a menu item	WMENU_ITEM(3W)
}get hardware	sprite colors	WGET_HW_SPRITE_COLOR(3W)
}set hardware	sprite colors	WSET_HW_SPRITE_COLOR(3W)
}set full-screen	sprite control mode	WSCRN_SPRITE_MODE(3W)
locations within display	stack }shuffle windows' relative	WSHUFFLE(3W)

: separates command from description; } indicates location of leading truncation; { indicates location of trailing truncation;

	↓ ∗*command*/keyword	PAGE_NAME
window to the top of the window	stack *wtop*: move the	WTOP(3W)
display *wmstart*:	start the window system on one	WMSTART(1)
wlist: list	status of windows or fonts	WLIST(1)
display *wmstop*:	stop the window system on one	WMSTOP(1)
wborder: control window border	style, color, label	WBORDER(1)
routines *window*:	summary of window library	WINDOW(3W)
wsfk_mode:	switch to soft key mode	WSFK_MODE(3W)
windows: window	system concepts and commands	WINDOWS(1)
wmstart: start the window	system on one display	WMSTART(1)
wmstop: stop the window	system on one display	WMSTOP(1)
and ready *wmready*:	tell if window manager is awake	WMREADY(1)
}set or inquire the	Term0 alternate font	ALTFONT_TERM0(3W)
set or inquire the	Term0 base font *basefont_term0*:	BASEFONT_TERM0(3W)
toxy_term0: convert	Term0 character units to pixels	TOXY_TERM0(3W)
inquire the ID of a	Term0 font *fontgetid_term0*:	FONTGETID_TERM0(3W)
inquire the name of a	Term0 font *fontgetname_term0*:	FONTGETNAME_TERM0(3W)
fontload_term0: load a	Term0 font	FONTLOAD_TERM0(3W)
fontswap_term0: replace a	Term0 font	FONTSWAP_TERM0(3W)
fontsize_size0:	Term0 font size	FONTSIZE_TERM0(3W)
fromxy_term0: convert from	Term0 pixel units to characters	FROMXY_TERM0(3W)
wiconic: change a window	to/from an icon	WICONIC(3W)
wtop: move the window to the	top of the window stack	WTOP(3W)
wmrepaint: repaint the desk	top	WMREPAINT(3W)
character units to pixels	∗*toxy_term0*: convert Term0	TOXY_TERM0(3W)
create a **graphics** window	type *wcreate_graphics*:	WCREATE_GRAPHICS(3W)
create a **term0** window	type *wcreate_term0*:	WCREATE_TERM0(3W)
mode for graphics window	type *wgskbd*: set keyboard	WGSKBD(3W)
}convert from Term0 pixel	units to characters	FROMXY_TERM0(3W)
convert Term0 character	units to pixels *toxy_term0*:	TOXY_TERM0(3W)
wmpathmake: build a path name	using an environment variable	WMPATHMAKE(3W)
get see_thru color index	value *wget_see_thru*:	WGET_SEE_THRU(3W)
set see_thru color index	value *wset_see_thru*:	WSET_SEE_THRU(3W)
get a window manager environment	variable *wminquire*:	WMINQUIRE(3W)
a path name using an environment	variable *wmpathmake*: build	WMPATHMAKE(3W)
window	∗*wautodestroy*: autodestroy a	WAUTODESTROY(3W)
	∗*wautoselect*: autoselect a window	WAUTOSELECT(3W)
	∗*wautotop*: autotop a window	WAUTOTOP(3W)
control a window's border	∗*wbanner*: inquire about or	WBANNER(3W)
style, color, label	∗*wborder*: control window border	WBORDER(1)
	∗*wbottom*: bottom window	WBOTTOM(3W)
	∗*wconceal*: conceal window	WCONCEAL(3W)
windows	∗*wcreate*: create one or more new	WCREATE(1)
graphics window type	∗*wcreate_graphics*: create a	WCREATE_GRAPHICS(3W)
term0 window type	∗*wcreate_term0*: create a	WCREATE_TERM0(3W)
	∗*wdestroy*: destroy a window	WDESTROY(3W)
windows or set autodestroy{	∗*wdestroy*: destroy one or more	WDESTROY(1)
position	∗*wdfltpos*: default window/icon	WDFLTPOS(3W)
windows	∗*wdisp*: change displayability of	WDISP(1)
locator events	∗*weventclear*: clear window	WEVENTCLEAR(3W)
locator events	∗*weventpoll*: poll for window	WEVENTPOLL(3W)

: separates command from description; **}** indicates location of leading truncation; **{** indicates location of trailing truncation;

	↓ *command/keyword	PAGE_NAME
	*wfont: load and activate fonts	WFONT(1)
colors	*wgetbcolor: get window border	WGETBCOLOR(3W)
coordinates	*wgetbcoords: get border	WGETBCOORDS(3W)
coordinates	*wgetcoords: get window	WGETCOORDS(3W)
	*wgetecho: get echo	WGETECHO(3W)
hardware sprite colors	*wget_hw_sprite_color: get	WGET_HW_SPRITE_COLOR(3W)
position	*wgeticonpos: get current icon	WGETICONPOS(3W)
echo/locator parameter	*wgetlocator: get window	WGETLOCATOR(3W)
	*wgetname: get path name	WGETNAME(3W)
	*wgetrasterecho: get raster echo	WGETRASTERECHO(3W)
information	*wgetscreen: get screen	WGETSCREEN(3W)
color index value	*wget_see_thru: get see_thru	WGET_SEE_THRU(3W)
SIGWINDOW interrupt mask	*wgetsigmask: get window	WGETSIGMASK(3W)
graphics window type	*wgskbd: set keyboard mode for	WGSKBD(3W)
hotspot	*whotspot_create: create a	WHOTSPOT_CREATE(3W)
hotspot	*whotspot_delete: delete a	WHOTSPOT_DELETE(3W)
	*whotspot_get: get hotspot data	WHOTSPOT_GET(3W)
	*whotspot_set: set hotspot data	WHOTSPOT_SET(3W)
to/from an icon	*wiconic: change a window	WICONIC(3W)
wgetbcolor: get	window border colors	WGETBCOLOR(3W)
wsetbcolor: set	window border colors	WSETBCOLOR(3W)
label wborder: control	window border style, color,	WBORDER(1)
wsetlabel: set label in	window border	WSETLABEL(3W)
wgetcoords: get	window coordinates	WGETCOORDS(3W)
winit: initialize	window device	WINIT(3W)
wgetlocator: get	window echo/locator parameter	WGETLOCATOR(3W)
get path name for a	window id winput_widpath:	WINPUT_WIDPATH(3W)
}read/set configuration of	window input channel	WINPUT_CONF(3W)
winput_read: read from	window input channel	WINPUT_READ(3W)
winput_getroute: determine	window input routing path	WINPUT_GETROUTE(3W)
window: summary of	window library routines	WINDOW(3W)
weventclear: clear	window locator events	WEVENTCLEAR(3W)
weventpoll: poll for	window locator events	WEVENTPOLL(3W)
wsetlocator: set	window locator position	WSETLOCATOR(3W)
variable wminquire: get a	window manager environment	WMINQUIRE(3W)
ready wmready: tell if	window manager is awake and	WMREADY(1)
wmkill: kill the	window manager	WMKILL(3W)
wterminate: release	window resources	WTERMINATE(3W)
arrows wscroll_set: control of	window scroll bar elevators and	WSCROLL_SET(3W)
wgetsigmask: get	window SIGWINDOW interrupt mask .	WGETSIGMASK(3W)
wsetsigmask: set	window SIGWINDOW interrupt mask .	WSETSIGMASK(3W)
the window to the top of the	window stack wtop: move	WTOP(3W)
library routines	*window: summary of window	WINDOW(3W)
commands windows:	window system concepts and	WINDOWS(1)
wmstart: start the	window system on one display	WMSTART(1)
wmstop: stop the	window system on one display	WMSTOP(1)
wselect: connect a	window to the keyboard	WSELECT(1)
stack wtop: move the	window to the top of the window	WTOP(3W)
wiconic: change a	window to/from an icon	WICONIC(3W)
create a **graphics**	window type wcreate_graphics:	WCREATE_GRAPHICS(3W)

: separates command from description; } indicates location of leading truncation; { indicates location of trailing truncation;

	↓ **command*/keyword	PAGE_NAME
create a **term0**	window type *wcreate_term0*:	WCREATE_TERM0(3W)
set keyboard mode for graphics	window type *wgskbd*:	WGSKBD(3W)
wautodestroy: autodestroy a	window	WAUTODESTROY(3W)
wautoselect: autoselect a	window	WAUTOSELECT(3W)
wautotop: autotop a	window	WAUTOTOP(3W)
wbottom: bottom	window	WBOTTOM(3W)
wconceal: conceal	window	WCONCEAL(3W)
wdestroy: destroy a	window	WDESTROY(3W)
input and events to another	window *winput_setroute*: routes	WINPUT_SETROUTE(3W)
wmove: move the location of a	window	WMOVE(3W)
wpan: pan the	window	WPAN(3W)
wpauseoutput: pause output to a	window	WPAUSEOUTPUT(3W)
wrecover: recover a	window	WRECOVER(3W)
a keyboard to the specified	window *wselect*: attach	WSELECT(3W)
wsize: change the size of a	window	WSIZE(3W)
wdfltpos: default	window/icon position	WDFLTPOS(3W)
inquire about or control a	window's border *wbanner*:	WBANNER(3W)
wlist: list status of	windows or fonts	WLIST(1)
wdestroy: destroy one or more	windows or set autodestroy{	WDESTROY(1)
wmove: move one or more	windows or their icons	WMOVE(1)
within{ *wshuffle*: shuffle	windows' relative locations	WSHUFFLE(3W)
wcreate: create one or more new	windows	WCREATE(1)
wdisp: change displayability of	windows	WDISP(1)
and commands	**windows*: window system concepts	WINDOWS(1)
new shells in new or existing	windows *wsh*: create	WSH(1)
change sizes of one or more	windows *wsize*:	WSIZE(1)
	**winit*: initialize window device	WINIT(3W)
configuration of window input{	**winput_conf*: read/set	WINPUT_CONF(3W)
window input routing path	**winput_getroute*: determine	WINPUT_GETROUTE(3W)
input channel	**winput_read*: read from window	WINPUT_READ(3W)
and events to another window	**winput_setroute*: routes input	WINPUT_SETROUTE(3W)
for a window id	**winput_widpath*: get path name	WINPUT_WIDPATH(3W)
windows' relative locations	within display stack }shuffle	WSHUFFLE(3W)
fonts	**wlist*: list status of windows or	WLIST(1)
	**wmenu_activate*: activate a menu	WMENU_ACTIVATE(3W)
	**wmenu_create*: create a menu	WMENU_CREATE(3W)
	**wmenu_delete*: delete a menu	WMENU_DELETE(3W)
event	**wmenu_eventread*: read the menu	WMENU_EVENTREAD(3W)
menu item	**wmenu_item*: specify or change a	WMENU_ITEM(3W)
environment variable	**wminquire*: get a window manager	WMINQUIRE(3W)
	**wmkill*: kill the window manager	WMKILL(3W)
or their icons	**wmove*: move one or more windows	WMOVE(1)
window	**wmove*: move the location of a	WMOVE(3W)
using an environment variable	**wmpathmake*: build a path name	WMPATHMAKE(3W)
is awake and ready	**wmready*: tell if window manager	WMREADY(1)
	**wmrepaint*: repaint the desk top	WMREPAINT(3W)
on one display	**wmstart*: start the window system	WMSTART(1)
on one display	**wmstop*: stop the window system	WMSTOP(1)
	**wpan*: pan the window	WPAN(3W)
window	**wpauseoutput*: pause output to a	WPAUSEOUTPUT(3W)

: separates command from description; } indicates location of leading truncation; { indicates location of trailing truncation;

: separates command from description; } indicates location of leading truncation; { indicates location of trailing truncation;

HEWLETT
PACKARD

HP Part Number
97069-90022

Microfiche No. 97069-99022
Printed in U.S.A. E0488

97069-90609
For Internal Use Only