28. WINDOWS AND MENUS

Windows provide a means by which different programs can share a single display harmoniously. Rather than having every program directly manipulating the screen bitmap, all display input/output operations are directed towards windows, which appear as rectangular regions of the screen, with borders and titles. The Interlisp-D window system provides both interactive and programmatic constructs for creating, moving, reshaping, overlapping, and destroying windows in such a way that a program can use a window in a relatively transparent fashion (see page 28.12). This allows existing Interlisp programs to be used without change, while providing a base for experimentation with more complex windows in new applications.

Menus are a special type of window provided by the window system, used for displaying a set of items to the user, and having the user select one using the mouse and cursor. The window system uses menus to provide the interactive interface for manipulating windows. The menu facility also allows users to create and use menus in interactive programs (see page 28.37).

Sometimes, a program needs to use a number of windows, displaying related information. The attached window facility (page 28.45) makes it easy to manipulate a group of windows as a single unit, moving and reshaping them together.

This chapter documents the Interlisp-D window system. First, it describes the default windows and menus supplied by the window system. Then, the programmatic facilities for creating windows. Next, the functions for using menus. Finally, the attached window facility.

Warning: The window system assumes that all programs follow certain conventions concerning control of the screen. All user programs should use perform display operations using windows and menus. In particular, user programs should not perform operate directly on the screen bitmap; otherwise the window system will not work correctly. For specialized applications that require taking complete control of the display, the window system can be turned off (and back on again) with the following function:

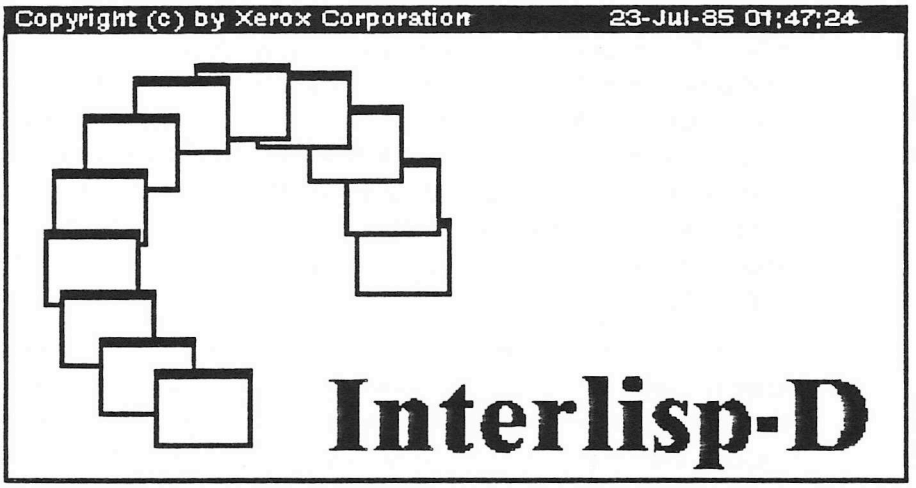
(WINDOWWORLD *FLAG***)** [NoSpread Function]

The window system is turned on if *FLAG* is T and off if *FLAG* is **NIL**. **WINDOWWORLD** returns the previous state of the window

system (T or **NIL**). If **WINDOWWORLD** is given no arguments, it simply returns the current state without affecting the window system.

28.1 Using The Window System

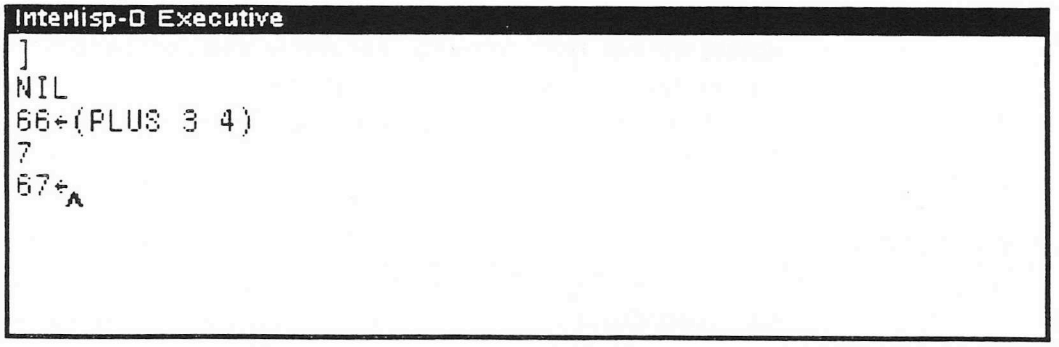
When Interlisp-D is initially started, the display screen lights up, showing a number of windows, including the following:



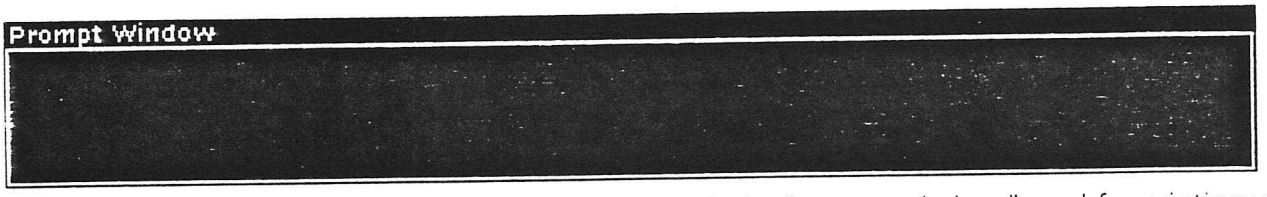
This window is the "logo window," used to identify the system. The logo window is bound to the variable **LOGOW** until it is closed. The user can create other windows like this by calling the following function:

(LOGOW *STRING WHERE TITLE ANGLEDELTA***)** [Function]

Creates a window formatted like the "logo window." *STRING* is the string to be printed in big type in the window; if **NIL**, "Interlisp-D" is used. *WHERE* is the position of the lower-left corner of the window; if **NIL**, the user is asked to specify a position. *TITLE* is the window title to use; if **NIL**, it defaults to the Xerox copyright notice and date. *ANGLEDELTA* specifies the angle (in degrees) between the boxes in the picture; if **NIL**, it defaults to 23 degrees.



This window is the "executive window," used for typing expressions and commands to the Interlisp-D executive, and for the executive to print any results (see page 13.1). For example, in the above picture, the user typed in **(PLUS 3 4)**, the executive evaluated it, and printed out the result, **7**. The upward-pointing arrow (↑) is the flashing caret, which indicates where the next keyboard typein will be printed (see page 28.30).



This window is the "prompt window," used for printing various system prompt messages. It is available to user programs through the following functions:

PROMPTWINDOW [Variable]

Global variable containing the prompt window.

(PROMPTPRINT EXP_1 ... EXP_N**)** [NoSpread Function]

Clears the prompt window, and prints EXP_1 through EXP_N in the prompt window.

(CLRPROMPT) [Function]

Clears the prompt window.

The Interlisp-D window system allows the user to interactively manipulate the windows on the screen, moving them around, changing their shape, etc. by selecting various operations from a menu.

For most windows, depressing the **RIGHT** mouse key when the cursor is inside a window during I/O wait will cause the window to come to the top and a menu of window operations to appear. If a command is selected from this menu (by releasing the right mouse key while the cursor is over a command), the selected operation will be applied to the window in which the menu was brought up. It is possible for an applications program to redefine the action of the **RIGHT** mouse key. In these cases, there is a convention that the default command menu may be brought up by depressing the **RIGHT** key when the cursor is in the header or border of a window (page 28.28). The operations are:

Close [Window Menu Command]

Closes the window, i.e, removes it from the screen. (See **CLOSEW**, page 28.15.)

Snap [Window Menu Command]

Prompts for a region on the screen and makes a new window whose bits are a snapshot of the bits currently in that region. Useful for saving some particularly choice image before the window image changes.

Paint [Window Menu Command]

Switches to a mode in which the cursor can be used like a paint brush to draw in a window. This is useful for making notes on a window. While the **LEFT** key is down, bits are added. While the **MIDDLE** key is down, they are erased. The **RIGHT** button pops up a command menu that allows changing of the brush shape, size and shade, changing the mode of combining the brush with the existing bits, or stopping paint mode.

Clear [Window Menu Command]

Clears the window and repositions it to the left margin of the first line of text (below the upper left corner of the window by the amount of the font ascent).

Bury [Window Menu Command]

Puts the window on the bottom of the occlusion stack, thereby exposing any windows that it was hiding.

Redisplay [Window Menu Command]

Redisplays the window. (See **REDISPLAYW**, page 28.16.)

Hardcopy [Window Menu Command]

Prints the contents of the window to the printer. If the window has a window property **HARDCOPYFN** (page 28.34), it is called with two arguments, the window and an image stream to print to, and the **HARDCOPYFN** must do the printing. In this way, special windows can be set up that know how to print their contents in a particular way. If the window does not have a **HARDCOPYFN**, the bitmap image of the window (including the border and title) are printed on the file or printer.

To save the image in a Press or Interpress-format file, or to send it to a non-default printer, use the submenu of the **Hardcopy** command, indicated by a gray triangle on the right edge of the **Hardcopy** menu item. If the mouse is moved off of the right of the menu item, another pop-up menu will apear giving the choices "To a file" or "To a printer." If "To a file" is selected, the user is prompted to supply a file name, and the format of the file (Press, Interpress, etc.), and the specified region will be stored in the file.

If "To a printer" is selected, the user is prompted to select a printer from the list of known printers, or to type the name of another printer. If the printer selected is not the first printer on **DEFAULTPRINTINGHOST** (page 29.4), the user will be asked whether to move or add the printer to the beginning of this list, so that future printing will go to the new printer.

Move [Window Menu Command]

Moves the window to a location specified by depressing and then releasing the **LEFT** key. During this time a ghost frame will indicate where the window will reappear when the key is released. (See **GETBOXPOSITION**, page 28.9.)

Shape [Window Menu Command]

Allows the user to specify a new region for the existing window contents. If the **LEFT** key is used to specify the new region, the reshaped window can be placed anywhere. If the **MIDDLE** key is used, the cursor will start out tugging at the nearest corner of the existing window, which is useful for making small adjustments in a window that is already positioned correctly. This is done by calling the function **SHAPEW** (page 28.16).

Occasionally, a user will have a number of large windows on the screen, making it difficult to access those windows being used. To help with the problem of screen space management, the Interlisp-D window system allows the creation of "icons." An icon is a small rectangle (containing text or a bitmap) which is a "shrunken-down" form of a particular window. Using the **Shrink** and **Expand** commands, the user can shrink windows not currently being used into icons, and quickly restore the original windows at any time.

Shrink [Window Menu Command]

Removes the window from the screen and brings up its icon. (See **SHRINKW**, page 28.21.) The window can be restored by selecting **Expand** from the window command menu of the icon.

If the **RIGHT** button is pressed while the cursor is in an icon, the window command menu will contain a slightly different set of commands. The **Redisplay** and **Clear** commands are removed, and the **Shrink** command is replaced with the **Expand** command:

Expand [Window Menu Command]

Restores the window associated with this icon and removes the icon. (See **EXPANDW**, page 28.22.)

If the **RIGHT** button is pressed while the cursor is not in any window, a "background menu" appears with the following operations:

Idle [Background Menu Command]

Enters "idle mode" (see page 12.4), which blacks out the display screen to save the phosphor. Idle mode can be exited by pressing any key on the keyboard or mouse. This menu command has subitems that allow the user to interactively set idle options to erase the password cache (for security), to request a password before exiting idle mode, to change the timeout before idle mode is entered automatically, etc.

SaveVM [Background Menu Command]

Calls the function **SAVEVM** (page 12.7), which writes out all of the dirty pages of the virtual memory. After a **SAVEVM**, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the **SAVEVM)** should you experience a system crash or other disaster.

Snap [Background Menu Command]

The same as the window menu command **Snap** described above.

Hardcopy [Background Menu Command]

Prompts for a region on the screen, and sends the bitmap image to the printer by calling **HARDCOPYW** (page 29.3). Note that the region can cross window boundaries.

Like the **Hardcopy** window menu command (above), the user can print to a file or specify a printer by using a submenu.

PSW [Background Menu Command]

Prompts the user for a position on the screen, and creates a "process status window" that allows the user to examine and manipulate all of the existing processes (see page 23.16).

Various system utilities (TEdit, DEdit, TTYIN) allow information to be "copy-inserted" at the current cursor position by selecting it with the "copy" key held down (Normally the shift keys are the "copy" key; this action can be changed in the key action table.) To "copy-insert" the bitmap of a snap into a Tedit document. If the right mouse button is pressed in the background with the copy key held down, a menu with the single item "**SNAP**" appears. If this item is selected, the user is prompted to select a region, and a bitmap containing the bits in that region of the screen is inserted into the current tty process, if that process is able to accept image objects.

Some built-in facilities and Lispusers packages add commands to the background menu, to provide an easy way of calling the different facilities. The user can determine what these new commands do by holding the **RIGHT** button down for a few seconds over the item in question; an explanatory message will be printed in the prompt window.

28.2 Changing Window Command Menus

The following functions provide a functional interface to the interactive window operations so that user programs can call them directly.

(DOWINDOWCOM *WINDOW***)** [Function]

If *WINDOW* is a **WINDOW** that has a **DOWINDOWCOMFN** window property, it **APPLY**s that property to **WINDOW**. Shrunken windows have a **DOWINDOWCOMFN** property that presents a window command menu that contains "expand" instead of "shrink".

If *WINDOW* is a **WINDOW** that doesn't have a **DOWINDOWCOMFN** window property, it brings up the window command menu. The initial items in these menus are described above. If the user selects one of the items from the provided menu, that item is **APPLY**ed to *WINDOW*.

If *WINDOW* is **NIL**, **DOBACKGROUNDCOM** (below) is called.

If *WINDOW* is not a **WINDOW** or **NIL**, **DOWINDOWCOM** simply returns without doing anything.

(DOBACKGROUNDCOM) [Function]

Brings up the background menu. The initial items in this menu are described above. If the user selects one of the items from the menu, that item is **EVAL**ed.

The window command menu for unshrunken windows is cached in the variable **WindowMenu**. To change the entries in this menu, the user should change the change the menu "command lists" in the variable **WindowMenuCommands**, and set the appropriate menu variable to a non-**MENU**, so the menu will be recreated. This provides a way of adding commands to the menu, of changing its font or of restoring the menu if it gets clobbered. The window command menus for icons and the background have similar pairs of variables, documented below. The "command lists" are in the format of the **ITEMS** field of a menu (see page 28.39), except as specified below.

Note: Command menus are recreated using the current value of **MENUFONT**.

WindowMenu [Variable]

WindowMenuCommands [Variable]

The menu that is brought up in response to a right button in an unshrunken window is stored on the variable **WindowMenu**. If **WindowMenu** is set to a non-**MENU**, the menu will be recreated from the list of commands **WindowMenuCommands**. The **CADR** of each command added to **WindowMenuCommands** should be a function name that will be **APPLY**ed to the window.

IconWindowMenu [Variable]

IconWindowMenuCommands [Variable]

The menu that is brought up in response to a right button in a shrunken window is stored on the variable **IconWindowMenu**. If it is **NIL**, it is recreated from the list of commands **IconWindowMenuCommands**. The **CADR** of each command added a function name that will be **APPLY**ed to the window.

BackgroundMenu [Variable]

BackgroundMenuCommands [Variable]

The menu that is brought up in response to a right button in the background is stored on the variable **BackgroundMenu**. If it is **NIL**, it is recreated from the list of commands **BackgroundMenuCommands**. The **CADR** of each command added to **BackgroundMenuCommands** should be a form that will be **EVAL**ed.

BackgroundCopyMenu [Variable]

BackgroundCopyMenuCommands [Variable]

The menu that is brought up in response to a right button in the background when the copy key is down is stored on the variable **BackgroundCopyMenu**. If it is **NIL**, it is recreated from the list of commands **BackgroundCopyMenuCommands**. The **CADR** of each command added to **BackgroundCopyMenuCommands** should be a form that will be **EVAL**ed.

28.3 Interactive Display Functions

The following functions can be used by programs to allow the user to interactively specify positions or regions on the display screen.

(GETPOSITION *WINDOW CURSOR***)** [Function]

Returns a **POSITION** that is specified by the user. **GETPOSITION** waits for the user to press and release the left button of the mouse and returns the cursor position at the time of release. If *WINDOW* is a **WINDOW**, the position will be in the coordinate system of *WINDOW*'s display stream. If *WINDOW* is **NIL**, the position will be in screen coordinates. If *CURSOR* is a **CURSOR** (page 30.14), the cursor will be changed to it while **GETPOSITION** is running. If *CURSOR* is **NIL**, the value of the system variable **CROSSHAIRS** will be used as the cursor:  .

(GETBOXPOSITION *BOXWIDTH BOXHEIGHT ORGX ORGY WINDOW PROMPTMSG***)** [Function]

Allows the user to position a "ghost" region of size *BOXWIDTH* by *BOXHEIGHT* on the screen, and returns the **POSITION** of the lower left corner of the region. If *PROMPTMSG* is non-**NIL**, **GETBOXPOSITION** first prints it in the **PROMPTWINDOW**. **GETBOXPOSITION** then changes the cursor to a box (using the global variable **BOXCURSOR**: ). If *ORGX* and *ORGY* are numbers, they are taken to be the original position of the region, and the cursor is moved to the nearest corner of that region. A ghost region is locked to the cursor so that if the cursor is moved, the ghost region moves with it. If *ORGX* and *ORGY* are numbers, the corner of the region formed by (*ORGX ORGY BOXWIDTH BOXHEIGHT*) that is nearest the cursor position is locked, otherwise the lower left corner is locked. The user can change to another corner by holding down the right button. With the right button down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the mouse will snap to the nearest corner, which will then become locked to the cursor. (The held corner can be changed after the left or middle button is down by holding both the original button and the right button down while the cursor is moved to the desired new corner, then letting up just the right button.) When the left or middle button is pressed and released, the lower left corner of the region at the time of release is returned. If *WINDOW* is a **WINDOW**, the returned position will be in *WINDOW*'s coordinate system; otherwise it will be in screen coordinates.

Example:

```
(GETBOXPOSITION 100 200 NIL NIL NIL
   "Specify the position of the command area.")
```

prompts the user for a 100 wide by 200 high region and returns its lower left corner in screen coordinates.

(GETREGION *MINWIDTH MINHEIGHT OLDREGION NEWREGIONFN NEWREGIONFNARG INITCORNERS***)** [Function]

Lets the user specify a new region and returns that region in screen coordinates. **GETREGION** prompts for a region by displaying a four-pronged box next to the cursor arrow at one corner of a "ghost" region:  . If the user presses the left button, the corner of a "ghost" region opposite the cursor is locked where it is. Once one corner has been fixed, the ghost region expands as the cursor moves.

To specify a region: (1) Move the ghost box so that the corner opposite the cursor is at one corner of the intended region. (2) Press the left button. (3) Move the cursor to the position of the opposite corner of the intended region while holding down the left button. (4) Release the left button.

Before one corner has been fixed, one can switch the cursor to another corner of the ghost region by holding down the right button. With the right button down, the cursor changes to a "forceps" () and the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the cursor will snap to the nearest corner of the ghost region.

After one corner has been fixed, one can still switch to another corner. To change to another corner, continue to hold down the left button and hold down the right button also. With both buttons down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the cursor will snap to the nearest corner, which will become the moving corner. In this way, the region may be moved all over the screen, before its size and position is finalized.

The size of the initial ghost region is controlled by the *MINWIDTH*, *MINHEIGHT*, *OLDREGION*, and *INITCORNERS* arguments.

If *INITCORNERS* is non-**NIL**, it should be a list specifying the initial corners of a ghost region of the form (*BASEX BASEY OPPX OPPY*), where (*BASEX*, *BASEY*) describes the anchored corner of the box, and (*OPPX*, *OPPY*) describes the trackable corner (in screen coordinates). The cursor is moved to (*OPPX*, *OPPY*).

If *INITCORNERS* is **NIL**, the ghost region will be *MINWIDTH* wide and *MINHEIGHT* high. If *MINWIDTH* or *MINHEIGHT* is **NIL**, 0 is used. Thus, for a call to **GETREGION** with no arguments specified, there will be no initial ghost region. The cursor will be in the lower right corner of the region, if there is one.

If *OLDREGION* is a region and the user presses the middle button, the corner of *OLDREGION* farthest from the cursor position is fixed and the corner nearest the cursor is locked to the cursor.

MINWIDTH and *MINHEIGHT*, if given, are the smallest **WIDTH** and **HEIGHT** that the returned region will have. The ghost image will not get any smaller than *MINWIDTH* by *MINHEIGHT*.

If *NEWREGIONFN* is non-**NIL**, it will be called to determine values for the positions of the corners. This provides a way of "filtering" prospective regions; for instance, by restricting the region to lie on an arbitrary grid. When the user is specifying a region, the region is determined by two of its corners, one that is fixed and one that is tracking the cursor. Each time the cursor moves or a mouse button is pressed, *NEWREGIONFN* is called with three arguments: *FIXEDPOINT*, the position of the fixed corner of the prospective region; *MOVINGPOINT*, the position of the opposite corner of the prospective region; and *NEWREGIONFNARG*. *NEWREGIONFNARG* allows the caller of **GETREGION** to pass information to the *NEWREGIONFN*.

The first time a button is pressed and when the user changes the moving corner via right buttoning, *MOVINGPOINT* is **NIL** and *FIXEDPOINT* is the position the user selected for the fixed corner of the new region. In this case, the position returned by *NEWREGIONFN* will be used for the fixed corner instead of the one proposed by the user. For all other calls, *FIXEDPOINT* is the position of the fixed corner (as returned by the previous call) and *MOVINGPOINT* is the new position the user selected for the opposite corner. In these cases, the value of *NEWREGIONFN* is used for the opposite corner instead of the one proposed by the user. In all cases, the ghost region is drawn with the values returned by *NEWREGIONFN*. *NEWREGIONFN* can be a list of functions in which case they are called in order with each being passed the result of calling the previous and the value of the last one used as the point.

(GETBOXREGION *WIDTH HEIGHT ORGX ORGY WINDOW PROMPTMSG***)** [Function]

Performs the same prompting as **GETBOXPOSITION** and returns the **REGION** specified by the user instead of the **POSITION** of its lower left corner.

(MOUSECONFIRM *PROMPTSTRING HELPSTRING WINDOW DON'TCLEARWINDOWFLG***)** [Function]

MOUSECONFIRM provides a simple way for the user to confirm or abort some action simply by using the mouse buttons. It prints the strings *PROMPTSTRING* and *HELPSTRING* in the window *WINDOW*, changes the cursor to a "little mouse" cursor:  (stored in the variable **MOUSECONFIRMCURSOR**), and waits for the user to press the left button to confirm, or any other button

to abort. If the left button was the last button released, returns **T**, else **NIL**.

If *PROMPTSTRING* is **NIL**, it is not printed out. If *HELPSTRING* is **NIL**, the string "Click LEFT to confirm, RIGHT to abort." is used. If *WINDOW* is **NIL**, the prompt window is used.

Normally, **MOUSECONFIRM** clears *WINDOW* before returning. If *DON'TCLEARWINDOWFLG* is non-**NIL**, the window is not cleared.

28.4 Windows

A window specifies a region of the screen, a display stream, functions that get called when the window undergoes certain actions, and various other items of information. The basic model is that a window is a passive collection of bits (on the screen). On top of this basic level, the system supports many different types of windows that are linked to the data structures displayed in them and provide selection and redisplaying routines. In addition, it is possible for the user to create new types of windows by providing selection and displaying functions for them.

Windows are ordered in depth from user to background. Windows in front of others obscure the latter. Operating on a window generally brings it to the top.

Windows are located at a certain position on the screen. Each window has a clipping region that confines all bits written to it to a region that allows a border around the window, and a title above it.

Each window has a display stream associated with it (see page 27.23), and either a window or its display stream can be passed interchangeably to all system functions. There are dependencies between the window and its display stream that the user should not disturb. For instance, the destination bitmap of the display stream of a window must always be the screen bitmap. The X offset, Y offset, and Clipping Region fields of the display stream should not be changed.

Windows can be created by the user interactively, under program control, or may be created automatically by the system.

Windows are in one of two states: "open" or "closed". In an "open" state, a window is visible on the screen (unless it is covered by other open windows or off the edge of the screen) and accessible to mouse operations. In a "closed" state, a window is not visible and not accessible to mouse operations. Any attempt to print or draw on a closed window will open it.

28.4.1 Window Properties

The behavior of a window is controlled by a set of "window properties." Some of these are used by the system. However, any arbitrary property name may be used by a user program to associate information with a window. For many applications the user will associate the structure being displayed with its window using a property. The following functions provide for reading and setting window properties:

(WINDOWPROP *WINDOW PROP NEWVALUE***)** [NoSpread Function]

Returns the previous value of *WINDOW*'s *PROP* aspect. If *NEWVALUE* is given, (even if given as **NIL**), it is stored as the new *PROP* aspect. Some aspects cannot be set by the user and will generate errors. Any *PROP* name that is not recognized is stored on a property list associated with the window.

(WINDOWADDPROP *WINDOW PROP ITEMTOADD FIRSTFLG***)** [Function]

WINDOWADDPROP adds a new item to a window property. If *ITEMTOADD* is **EQ** to an element of the *PROP* property of the window *WINDOW*, nothing is added. If the current property is not a list, it is made a list before *ITEMTOADD* added. **WINDOWADDPROP** returns the previous property. If *FIRSTFLG* is non-**NIL**, the new item goes on the front of the list; otherwise, it goes on the end of the list. If *FIRSTFLG* is non-**NIL** and *ITEMTOADD* is already on the list, it is moved to the front.

Many window properties (**OPENFN**, **CLOSEFN**, etc.) can be a list of functions. **WINDOWADDPROP** is useful for adding additional functions to a window property without affecting any existing functions. Note that if the order of items in a window property is important, the list can be modified using **WINDOWPROP**.

(WINDOWDELPROP *WINDOW PROP ITEMTODELETE***)** [Function]

WINDOWDELPROP deletes *ITEMTODELETE* from the window property *PROP* of *WINDOW* and returns the previous list if *ITEMTODELETE* was an element. If *ITEMTODELETE* was not a member of window property *PROP*, **NIL** is returned.

28.4.2 Creating Windows

(CREATEW *REGION TITLE BORDERSIZE NOOPENFLG***)** [Function]

Creates a new window. *REGION* indicates where and how large the window should be by specifying the exterior region of the window. The usable height and width of the resulting window will be smaller than the height and width of the region by twice the border size and further less the height of the title, if any. If

REGION is **NIL**, **GETREGION** is called to prompt the user for a region.

If *TITLE* is non-**NIL**, it is printed in the border at the top of the window. The *TITLE* is printed using the global display stream **WindowTitleDisplayStream**. Thus the height of the title will be **(FONTPROP WindowTitleDisplayStream 'HEIGHT)**.

If *BORDERSIZE* is a number, it is used as the border size. If *BORDERSIZE* is not a number, the window will have a border **WBorder** (initially 4) bits wide.

If *NOOPENFLG* is non-**NIL**, the window will not be opened, i.e. displayed on the screen.

The initial X and Y positions of the window are set to the upper left corner by calling **MOVETOUPPERLEFT** (page 27.14).

(DECODE.WINDOW.ARG *WHERESPEC WIDTH HEIGHT TITLE BORDER NOOPENFLG***)** [Function]

This is a useful function for creating windows. *WHERESPEC* can be a **WINDOW**, a **REGION**, a **POSITION** or **NIL**. If *WHERESPEC* is a **WINDOW**, it is returned. In all other cases, **CREATEW** is called with the arguments *TITLE BORDER* and *NOOPENFLG*. The *REGION* argument to **CREATEW** is determined from *WHERESPEC* as follows:

If *WHERESPEC* is a **REGION**, it is adjusted to be on the screen, then passed to **CREATEW**.

If *WIDTH* and *HEIGHT* are numbers and *WHERESPEC* is a **POSITION**, the region whose lower left corner is *WHERESPEC*, whose width is *WIDTH* and whose height is *HEIGHT* is adjusted to be on the screen, then passed to **CREATEW**.

If *WIDTH* and *HEIGHT* are numbers and *WHERESPEC* is not a **POSITION**, then **GETBOXREGION** is called to prompt the user for the position of a region that is *WIDTH* by *HEIGHT*.

If *WIDTH* and *HEIGHT* are not numbers, **CREATEW** is given **NIL** as a **REGION** argument.

If *WIDTH* and *HEIGHT* are used, they are used as interior dimensions for the window.

(WINDOWP *X***)** [Function]

Returns *X* if *X* is a window, **NIL** otherwise.

28.4.3 Opening and Closing Windows

(OPENWP *WINDOW*) [Function]

Returns *WINDOW*, if *WINDOW* is an open window (has not been closed); **NIL** otherwise.

(OPENWINDOWS) [Function]

Returns a list of all open windows.

(OPENW *WINDOW*) [Function]

If *WINDOW* is a closed window, **OPENW** calls the function or functions on the window property **OPENFN** of *WINDOW*, if any. If one of the **OPENFN**s is the atom **DON'T**, the window will not be opened. Otherwise the window is placed on the occlusion stack of windows and its contents displayed on the screen. If *WINDOW* is an open window, it returns **NIL**.

(CLOSEW *WINDOW*) [Function]

CLOSEW calls the function or functions on the window property **CLOSEFN** of *WINDOW*, if any. If one of the **CLOSEFN**s is the atom **DON'T** or returns the atom **DON'T** as a value, **CLOSEW** returns without doing anything further. Otherwise, **CLOSEW** removes *WINDOW* from the window stack and restores the bits it is obscuring. If *WINDOW* was closed, *WINDOW* is returned as the value. If it was not closed, (for example because its **CLOSEFN** returned the atom **DON'T**), **NIL** is returned as the value.

WINDOW can be restored in the same place with the same contents (reopened) by calling **OPENW** or by using it as the source of a display operation.

OPENFN [Window Property]

The **OPENFN** window property can be a single function or a list of functions. If one of the **OPENFN**s is the atom **DON'T**, the window will not be opened. Otherwise, the **OPENFN**s are called after a window has been opened by **OPENW**, with the window as a single argument.

CLOSEFN [Window Property]

The **CLOSEFN** window property can be a single function or a list of functions that are called just before a window is closed by **CLOSEW**. The function(s) will be called with the window as a single argument. If any of the **CLOSEFN**s are the atom **DON'T**, or if the value returned by any of the **CLOSEFN**s is the atom **DON'T**, the window will not be closed.

Note: If the **CAR** of the **CLOSEFN** list is a **LAMBDA** word, it is treated as a single function.

Note: A **CLOSEFN** should not call **CLOSEW** on its argument.

28.4.4 Redisplaying Windows

(REDISPLAYW *WINDOW REGION ALWAYSFLG***)** [Function]

Redisplay the region *REGION* of the window *WINDOW*. If *REGION* is **NIL**, the entire window is redisplayed.

If *WINDOW* doesn't have a **REPAINTFN** (page 28.16), the action depends on the value of *ALWAYSFLG*. If *ALWAYSFLG* is **NIL**, *WINDOW* will not change and the message "Window has no REPAINTFN. Can't redisplay." will be printed in the prompt window. If *ALWAYSFLG* is non-**NIL**, **REDISPLAYW** acts as if **REPAINTFN** was **NILL**.

REPAINTFN [Window Property]

The **REPAINTFN** window property can be a single function or a list of functions that are called to repaint parts of the window by **REDISPLAYW**. The **REPAINTFN**s are called with two arguments: the window and the region in the coordinates of the window's display stream of the area that should be repainted. Before the **REPAINTFN** is called, the clipping region of the window is set to clip all display operations to the area of interest so that the **REPAINTFN** can display the entire window contents and the results will be appropriately clipped.

Note: **CLEARW** (page 28.31) should not be used in **REPAINTFN**s because it resets the window's coordinate system. If a **REPAINTFN** wants to clear its region first, it should use **DSPFILL** (page 27.20).

28.4.5 Reshaping Windows

(SHAPEW *WINDOW NEWREGION***)** [Function]

Reshapes *WINDOW*. If the window property **RESHAPEFN** is the atom **DON'T** or a list that contains the atom **DON'T**, a message is printed in the prompt window, *WINDOW* is not changed, and **NIL** is returned. Otherwise, **RESHAPEFN** window property can be a single function or a list of functions that are called when a window is reshaped, to reformat or redisplay the window contents (see below). If the **RESHAPEFN** window property is **NIL**, **RESHAPEBYREPAINTFN** is the default.

If the region *NEWREGION* is **NIL**, it prompts for a region with **GETREGION** (page 28.10). When calling **GETREGION**, the function **MINIMUMWINDOWSIZE** is called to determine the minimum height and width of the window, the function

WINDOWREGION is called to get the region passed as the *OLDREGION* argument, the window property **NEWREGIONFN** is used as the **NEWREGIONFN** argument and *WINDOW* as the **NEWREGIONFNARG** argument. If the window property **INITCORNERSFN** is non-**NIL**, it is applied to the window, and the value is passed as the *INITCORNERS* argument to **GETREGION**, to determine the initial size of the "ghost region." These window properties allow the window to specify the regions used for interactive calls to **SHAPEW**.

If the region *NEWREGION* is a **REGION** and its **WIDTH** or **HEIGHT** less than the minimums returned by calling the function **MINIMUMWINDOWSIZE**, they will be increased to the minimums.

If *WINDOW* has a window property **DOSHAPEFN**, it is called, passing it *WINDOW* and *NEWREGION* (or the region returned by **GETREGION**). If *WINDOW* does not have a **DOSHAPEFN** window property, the function **SHAPEW1** is called to reshape the window. **DOSHAPEFN**s are provided to implement window groups and few users should ever write them. They are tricky to write and must call **SHAPEW1** eventually. The **RESHAPEFN** window property is a simpler hook into reshape operations.

(SHAPEW1 *WINDOW REGION***)** [Function]

Changes *WINDOW*'s size and position on the screen to be *REGION*. After clearing the region on the screen, it calls the window's **RESHAPEFN**, if any, passing it three arguments: (1) *WINDOW*, (2) a bitmap that contains *WINDOW*'s previous screen image and (3) the region of *WINDOW*'s old image within the bitmap.

RESHAPEFN [Window Property]

The **RESHAPEFN** window property can be a single function or a list of functions that are called when a window is reshaped by **SHAPEW**. If the **RESHAPEFN** is **DON'T** or a list containing **DON'T**, the window will not be reshaped. Otherwise, the function(s) are called after the window has been reshaped, its coordinate system readjusted to the new position, the title and border displayed, and the interior filled with texture. The **RESHAPEFN** should display any additional information needed to complete the window's image in the new position and shape. The **RESHAPEFN** is called with four arguments: (1) the window in its reshaped form, (2) a bitmap with the image of the old window in its old shape, and (3) the region within the bitmap that contains the window's old image, and (4) the region of the screen previously occupied by this window. This function is provided so that users can reformat window contents or whatever. **RESHAPEBYREPAINTFN** (below) is the default and should be useful for many windows.

NEWREGIONFN [Window Property]

If **SHAPEW** calls **GETREGION** to prompt the user for a region, the value of the **NEWREGIONFN** window property is passed as the *NEWREGIONFN* argument to **GETREGION** (page 28.10).

INITCORNERSFN [Window Property]

If this window property is non-**NIL**, it should be a function of one argument, a window, that returns a list specifying the initial corners of a "ghost region" of the form **(***BASEX BASEY OPPX OPPY***)**, where (*BASEX*, *BASEY*) describes the anchored corner of the box, and (*OPPX*, *OPPY*) describes the trackable corner. If **SHAPEW** calls **GETREGION** to prompt the user for a region, this function is applied to the window, and the list returned is passed as the *INITCORNERS* argument to **GETREGION** (page 28.10), to specify the initial ghost region.

DOSHAPEFN [Window Property]

If this window property is non-**NIL**, it is called by **SHAPEW** to reshape the window (instead of **SHAPEW1**). It is called with two arguments: the window and the new region.

(RESHAPEBYREPAINTFN *WINDOW OLDIMAGE IMAGEREGION OLDSCREENREGION***)**
[Function]

This the default window **RESHAPEFN**. *WINDOW* is a window that has been reshaped from the screen region *OLDSCREENREGION* to its new region (available via **(WINDOWPROP WINDOW 'REGION)**). *OLDIMAGE* is a bitmap that contains the image of the window from its previous location. *IMAGEREGION* is the region within *OLDIMAGE* that contains the old image.

RESHAPEBYREPAINTFN BITBLTs the old region contents into the new region. If the new shape is larger in either or both dimensions, the newly exposed areas are redisplayed via calls *WINDOW*'s **REPAINTFN** window property (page 28.16). **RESHAPEBYREPAINTFN** may call the **REPAINTFN** up to four times during a single reshape.

The choice of which areas of the window to remove or extend is done as follows. If *WINDOW*'s new region shares an edge with *OLDSCREENREGION*, that edge of the window image will remain fixed and any addition or reduction in that dimension will be performed on the opposite side. If *WINDOW* has an **EXTENT** property and the newly exposed window area is outside of it, any extra will be added so as to show **EXTENT** that was previously not visible. An exception to these rules is that the current X,Y position is kept visible, if it was visible before the reshape.

28.4.6 Moving Windows

(MOVEW *WINDOW POSorX Y***)** [Function]

Moves *WINDOW* to the position specified by *POSorX* and *Y* according to the following rules:

If *POSorX* is **NIL**, **GETBOXPOSITION** (page 28.9) is called to read a position from the user. If *WINDOW* has a **CALCULATEREGION** window property, it will be called with *WINDOW* as an argument and should return a region which will be used to prompt the user with. If *WINDOW* does not have a **CALCULATEREGION** window property, the region of *WINDOW* is used to prompt with.

If *POSorX* is a **POSITION**, *POSorX* is used.

If *POSorX* and *Y* are both **NUMBERP**, a position is created using *POSorX* as the **XCOORD** and *Y* as the **YCOORD**.

If *POSorX* is a **REGION**, a position is created using its **LEFT** as the **XCOORD** and **BOTTOM** as the **YCOORD**.

If *WINDOW* is not open and *POSorX* is non-**NIL**, the window will be moved without being opened. Otherwise, it will be opened.

If *WINDOW* has the atom **DON'T** as a **MOVEFN** window property, the window will not be moved. If *WINDOW* has any other non-**NIL** value as a **MOVEFN** property, it should be a function or list of functions that will be called before the window is moved with the *WINDOW* and the new positon as its arguments. If it returns the atom **DON'T**, the window will not be moved. If it returns a position, the window will be moved to that position instead of the new one. If there are more than one **MOVEFN**s, the last one to return a value is the one that determines where the window is moved to.

If *WINDOW* is moved and *WINDOW* has an **AFTERMOVEFN** window property, it should be a function or a list of functions that will be called after the window is moved with *WINDOW* as an argument.

MOVEW returns the new position, or **NIL** if the window could not be moved.

Note: If **MOVEW** moves any part of the window from off-screen onto the screen, that part is redisplayed (by calling **REDISPLAYW**).

(RELMOVEW *WINDOW POSITION***)** [Function]

Like **MOVEW** for moving windows but the *POSITION* is interpreted relative to the current position of *WINDOW*. Example: The following code moves *WINDOW* to the right one screen point.

(RELMOVEW *WINDOW* **(create POSITION XCOORD ← 1 YCOORD ← 0))**

CALCULATEREGION [Window Property]

If **MOVEW** calls **GETBOXPOSITION** to prompt the user for a region, the **CALCULATEREGION** window property is called (passing the window as an argument. The **CALCULATEREGION** should returns a region to be used to prompt the user with. If **CALCULATEREGION** is **NIL**, the region of the window is used to prompt with.

MOVEFN [Window Property]

If the **MOVEFN** is **DON'T**, the window will not be moved by **MOVEW**. Otherwise, if the **MOVEFN** is non-**NIL**, it should be a function or a list of functions that will be called before a window is moved with two arguments: the window being moved and the new position of the lower left corner in screen coordinates. If the **MOVEFN** returns **DON'T**, the window will not be moved. If the **MOVEFN** returns a **POSITION**, the window will be moved to that position. Otherwise, the window will be moved to the specified new position.

AFTERMOVEFN [Window Property]

If non-**NIL**, it should be a function or a list of functions that will be called after the window is moved (by **MOVEW**) with the window as an argument.

28.4.7 Exposing and Burying Windows

(TOTOPW *WINDOW NOCALLTOTOPFNFLG***)** [Function]

Brings *WINDOW* to the top of the stack of overlapping windows, guaranteeing that it is entirely visible. If *WINDOW* is closed, it is opened. This is done automatically whenever a printing or drawing operation occurs to the window.

If *NOCALLTOTOPFNFLG* is **NIL**, the **TOTOPFN** of *WINDOW* is called (page 28.20). If *NOCALLTOTOPFNFLG* is **T**, it is not called, which allows a **TOTOPFN** to call **TOTOPW** without causing an infinite loop.

(BURYW *WINDOW***)** [Function]

Puts *WINDOW* on the bottom of the stack by moving all the windows that it covers in front of it.

TOTOPFN [Window Property]

If non-**NIL**, whenever the window is brought to the top, the **TOTOPFN** is called (with the window as a single argument). This function may be used to bring a collection of windows to the top together.

If the *NOCALLTOPWFN* argument of **TOTOPW** is non-**NIL**, the **TOTOPFN** of the window is not called, which provides a way of avoiding infinite loops when using **TOTOPW** from within a **TOTOPFN**.

28.4.8 Shrinking Windows Into Icons

Occasionally, a user will have a number of large windows on the screen, making it difficult to access those windows being used. To help with the problem of screen space management, the Interlisp-D window system allows the creation of *Icons*. An icon is a small rectangle (containing text or a bitmap) which is a "shrunken-down" form of a particular window. Using the **Shrink** and **Expand** window menu commands (page 28.5), the user can shrink windows not currently being used into icons, and quickly restore the original windows at any time. This facility is controlled by the following functions and window properties:

(SHRINKW *WINDOW TOWHAT ICONPOSITION EXPANDFN*) [Function]

SHRINKW makes a small icon which represents *WINDOW* and removes *WINDOW* from the screen. Icons have a different window command menu that contains "**EXPAND**" instead of "**SHRINK**". The **EXPAND** command calls **EXPANDW** which returns the shrunken window to its original size and place. The icon can also be moved by pressing the **LEFT** button in it, or expanded by pressing the **MIDDLE** button in it.

The **SHRINKFN** property of the window *WINDOW* affects the operation of **SHRINKW**. If the **SHRINKFN** property of *WINDOW* is the atom **DON'T**, **SHRINKW** returns. Otherwise, the **SHRINKFN** property of the window is treated as a (list of) function(s) to apply to *WINDOW*; if any returns the atom **DON'T**, **SHRINKW** returns.

TOWHAT, if given, indicates the image the icon window will have. If *TOWHAT* is a string, atom or list, the icon's image will be that string (currently implemented as a title-only window with *TOWHAT* as the title.) If *TOWHAT* is a **BITMAP**, the icon's image will be a copy of the bitmap. If *TOWHAT* is a **WINDOW**, that window will be used as the icon.

If *TOWHAT* is not given (as is the case when invoked from the **SHRINK** window command), then the following apply in turn: (1) If the window has an **ICONFN** property, it gets called with the two arguments *WINDOW* and *OLDICON*, where *WINDOW* is the window being shrunk and *OLDICON* is the previously created icon, if any. The **ICONFN** should return one of the *TOWHAT* entities described above or return the *OLDICON* if it does not want to change it. (2) If the window has an **ICON** property, it is used as the value of *TOWHAT*. (3) If the window has neither an

ICONFN or **ICON** property, the icon will be *WINDOW*'s title or, if *WINDOW* doesn't have a title, the date and time of the icon creation.

ICONPOSITION gives the position that the new icon will be on the screen. If it is **NIL**, the icon will be in the corner of the window furthest from the center of the screen.

In all but the default case, the icon is cached on the property **ICONWINDOW** of *WINDOW* so repeating **SHRINKW** reuses the same icon (unless overridden by the **ICONFN** described above). Thus to change the icon it is necessary to remove the **ICONWINDOW** property or call **SHRINKW** explicitly giving a *TOWHAT* argument.

(EXPANDW *ICONW***)** [Function]

Restores the window for which *ICONW* is an icon, and removes the icon from the screen. If the **EXPANDFN** window property of the main window is the atom **DON'T**, the window won't be expanded. Otherwise, the window will be restored to its original size and location and the **EXPANDFN** (or list of functions) will be applied to it.

SHRINKFN [Window Property]

The **SHRINKFN** window property can be a single function or a list of functions that are called just before a window is shrunken by **SHRINKW**, with the window as a single argument. If any of the **SHRINKFN**s are the atom **DON'T**, or if the value returned by any of the **SHRINKFN**s is the atom **DON'T**, the window will not be shrunk.

ICONFN [Window Property]

If **SHRINKW** is called without begin given a *TOWHAT* argument (as is the case when invoked from the **SHRINK** window command) and the window's **ICONFN** property is non-**NIL**, then it gets called with two arguments, the window being shrunk and the previously created icon, if any. The **ICONFN** should return one of the *TOWHAT* entities described above or return the previously created icon if it does not want to change it.

ICON [Window Property]

If **SHRINKW** is called without being given a *TOWHAT* argument, the window's **ICONFN** property is **NIL**, and the **ICON** property is non-**NIL**, then it is used as the value of *TOWHAT*.

ICONWINDOW [Window Property]

Whenever an icon is created, it is cached on the property **ICONWINDOW** of the window, so calling **SHRINKW** again will reuse the same icon (unless overridden by the **ICONFN**.

Thus, to change the icon it is necessary to remove the **ICONWINDOW** property or call **SHRINKW** (page 28.21) explicitly giving a *TOWHAT* argument.

EXPANDFN [Window Property]

The **EXPANDFN** window property can be a single function or a list of functions. If one of the **EXPANDFN**s is the atom **DON'T**, the window will not be expanded. Otherwise, the **EXPANDFN**s are called after the window has been expanded by **EXPANDW**, with the window as a single argument.

28.4.9 Coordinate Systems, Extents, And Scrolling

Note: The word "scrolling" has two distinct meanings when applied to Interlisp-D windows. This section documents the use of "scroll bars" on the left and bottom of a window to move an object displayed in the window. "Scrolling" also describes the feature where trying to print text off the bottom of a window will cause the contents to "scroll up." This second feature is controlled by the function **DSPSCROLL** (page 27.24.

One way of thinking of a window is as a "view" onto an object (e.g. a graph, a file, a picture, etc.) The object has its own natural coordinate system in terms of which its subparts are laid out. When the window is created, the X Offset and Y Offset of the window's display stream are set to map the origin of the object's coordinate system into the lower left point of the window's interior region. At the same time, the Clipping Region of the display stream is set to correspond to the interior of the window. From then on, the display stream's coordinate system is translated and its clipping region adjusted whenever the window is moved, scrolled or reshaped.

There are several distinct regions associated with a window viewing an object. First, there is a region in the window's coordinate system that contains the complete image of the object. This region (which can only be determined by application programs with knowledge of the "semantics" of the object) is stored as the **EXTENT** property of the window (below). Second, the clipping region of the display stream (obtainable with the function **DSPCLIPPINGREGION**, page 27.11) specifies the portion of the object that is actually visible in the window. This is set so that it corresponds to the interior of the window (not including the border or title). Finally, there is the region on the screen that specifies the total area that the window occupies, including the

border and title. This region (in screen coordinates) is stored as the **REGION** property of the window (page 28.34).

The window system supports the idea of scrolling the contents of a window. Scrolling regions are on the left and the bottom edge of each window. The **LEFT** key is used to indicate upward or leftward scrolling by the amount necessary to move the selected position to the top or the left edge. The **RIGHT** key is used to indicate downward or rightward scrolling by the amount necessary to move the top or left edge to the selected position. The **MIDDLE** key is used to indicate global placement of the object within the window (similar to "thumbing" a book). In the scroll region, the part of the object that is being viewed by the window is marked with a gray shade. If the whole scroll bar is thought of as the entire object, the shaded portion is the portion currently being viewed. This will only occur when the window "knows" how big the object is (see window property **EXTENT**, page 28.26).

When the button is released in a scroll region, the function **SCROLLW** is called. **SCROLLW** calls the scrolling function associated with the window to do the actual scrolling and provides a programmable entry to the scrolling operation.

(SCROLLW *WINDOW DELTAX DELTAY CONTINUOUSFLG***)** [Function]

Calls the **SCROLLFN** window property of the window *WINDOW* with arguments *WINDOW*, *DELTAX*, *DELTAY* and *CONTINUOUSFLG*. See **SCROLLFN** window property, page 28.26.

(SCROLL.HANDLER *WINDOW***)** [Function]

This is the function that tracks the mouse while it is in the scroll region. It is called when the cursor leaves a window in either the left or downward direction. If *WINDOW* does not have a scroll region for this direction (e.g. the window has moved or reshaped since it was last scrolled), a scroll region is created that is **SCROLLBARWIDTH** wide. It then waits for **SCROLLWAITTIME** milliseconds and if the cursor is still inside the scroll region, it opens a window the size of the scroll region and changes the cursor to indicate the scrolling is taking place.

When a button is pressed, the cursor shape is changed to indicate the type of scrolling (up, down, left, right or thumb). After the button is held for **WAITBEFORESCROLLTIME** milliseconds, until the button is released **SCROLLW** is called each **WAITBETWEENSCROLLTIME** milliseconds. These calls are made with the *CONTINUOUSFLG* argument set to **T**. If the button is released before **WAITBEFORESCROLLTIME** milliseconds, **SCROLLW** is called with the *CONTINUOUSFLG* argument set to **NIL**.

The arguments passed to **SCROLLW** depend on the mouse button. If the **LEFT** button is used in the vertical scroll region, *DY* is distance from cursor position at the time the button was released to the top of the window and *DX* is 0. If the **RIGHT** button is used, the inverse of this quantity is used for *DY* and 0 for *DX*. If the **LEFT** button is used in the horizontal scroll region, *DX* is distance from cursor position to left of the window and *DY* is 0. If the **RIGHT** button is used, the inverse of this quantity is used for *DX* and 0 for *DY*.

If the **MIDDLE** button is pressed, the distance argument to **SCROLLW** will be a **FLOATP** between 0.0 and 1.0 that indicates the proportion of the distance the cursor was from the left or top edge to the right or bottom edge.

Note: The scrolling regions will not come up if the window has a **SCROLLFN** window property of **NIL**, has a non-**NIL** **NOSCROLLBARS** window property, or if its **SCROLLEXTENTUSE** (page 28.26) property has certain values and its **EXTENT** is fully visible.

(SCROLLBYREPAINTFN *WINDOW DELTAX DELTAY CONTINUOUSFLG***)** [Function]

SCROLLBYREPAINTFN is the standard scrolling function which should be used as the **SCROLLFN** property for most scrolling windows.

This function, when used as a **SCROLLFN**, **BITBLT**s the bits that will remain visible after the scroll to their new location, fills the newly exposed area with texture, adjusts the window's coordinates and then calls the window's **REPAINTFN** on the newly exposed region. Thus this function will scroll any window that has a repaint function.

If *WINDOW* has an **EXTENT** property (page 28.26), **SCROLLBYREPAINTFN** will limit scrolling in the X and Y directions according to the value of the window property **SCROLLEXTENTUSE** (page 28.26).

If *DELTAX* or *DELTAY* is a **FLOATP**, **SCROLLBYREPAINTFN** will position the window so that its top or left edge will be positioned at that proportion of its **EXTENT**. If the window does not have an **EXTENT**, **SCROLLBYREPAINTFN** will do nothing.

If *CONTINUOUSFLG* is non-**NIL**, this indicates that the scrolling button is being held down. In this case, **SCROLLBYREPAINTFN** will scroll the distance of one linefeed height (as returned by **DSPLINEFEED**, page 27.12).

Scrolling is controlled by the following window properties:

EXTENT [Window Property]

Used to limit scrolling operations. Accesses the extent region of the window. If non-**NIL**, the **EXTENT** is a region in the window's display stream that contains the complete image of the object being viewed by the window. User programs are responsible for updating the **EXTENT**. The functions **UNIONREGIONS**, **EXTENDREGION**, etc. (page 27.2) are useful for computing a new extent region.

In some situations, it is useful to define an **EXTENT** that only exists in one dimension. This may be done by specifying an **EXTENT** region with a width or height of **-1**. **SCROLLFN** handling recognizes this situation as meaning that the negative **EXTENT** dimension is unknown.

SCROLLFN [Window Property]

If the **SCROLLFN** property is **NIL**, the window will not scroll. Otherwise, it should be a function of four arguments: (1) the window being scrolled, (2) the distance to scroll in the horizontal direction (positive to right, negative to left), (3) the distance to scroll in the vertical direction (positive up, negative down), and (4) a flag which is **T** if the scrolling button is being held down. For more information, see **SCROLL.HANDLER** (page 28.24). For most scrolling windows, the **SCROLLFN** function should be **SCROLLBYREPAINTFN** (page 28.25).

NOSCROLLBARS [Window Property]

If the **NOSCROLLBARS** property is non-**NIL**, scroll bars will not be brought up for this window. This disables mouse-driven scrolling of a window. This window can still be scrolled using **SCROLLW** (page 28.24).

SCROLLEXTENTUSE [Window Property]

SCROLLBYREPAINTFN uses the **SCROLLEXTENTUSE** window property to limit how far scrolling can go in the X and Y directions. The possible values for **SCROLLEXTENTUSE** and their interpretations are:

NIL This will keep the extent region visible or near visible. It will not scroll the window so that the top of the extent is below the top of the window, the bottom of the extent is more than one point above the top of the window, the left of the extent is to the right of the window and the right of the extent is to the left of the window. The **EXTENT** can be scrolled to just above the window to provide a way of "hiding" the contents of a window. In this mode the extent is either in the window or just of the top of the window.

T The extent is not used to control scrolling. The user can scroll the window to anywhere. Having the **EXTENT** window property

does all thumb scrolling to be supported so that the user can get back to the EXTENT by thumb scrolling.

LIMIT This will keep the extent region visible. The window is only allowed to view within the extent.

+ This will keep the extent region visible or just off in the positive direction in either X or Y (i.e. the image will be either be visible or just off to the top and/or right.)

- This will keep the extent region visible or just off in the negative direction in either X or Y (i.e. the image will be either be visible or just off to the left and/or bottom).

+ -
- + This will keep the extent region visible or just off in the window (i.e. the image will be either be visible or just off to the left, bottom, top or right).

(*XBEHAVIOR* . *YBEHAVIOR*) If the **SCROLLEXTENTUSE** is a list, the **CAR** is interpreted as the scrolling limit in the X behavior and the **CDR** as the scrolling limit in the Y behavior. *XBEHAVIOR* and *YBEHAVIOR* should each be one of the atoms **(NIL T LIMIT + - + - - +)**. The interpretations of the atoms is the same as above except that **NIL** is equivalent to **LIMIT**.

Note: The **NIL** value of **SCROLLEXTENTUSE** is equivalent to **(LIMIT . +)**.

Example: If the **SCROLLEXTENTUSE** window property of a window (with an extent defined) is **(LIMIT . T)**, the window will scroll uncontrolled in the Y dimension but be limited to the extent region in the X dimension.

28.4.10 Mouse Activity in Windows

The following window properties allow the user to control the response to mouse activity in a window. The value of these properties, if non-**NIL**, should be a function that will be called (with the window as argument) when the specified event occurs.

Note: these functions should be "self-contained", communicating with the outside world solely via their window argument, e.g., by setting window properties. In particular, these functions should not expect to access variables bound on the stack, as the stack context is formally undefined at the time these functions are called. Since the functions are invoked asynchronously, they perform any terminal input/output operations from their own window.

WINDOWENTRYFN [Window Property]

Whenever a button goes down in the window and the process associated with the window is not the tty process, the

WINDOWENTRYFN is called. The default is **GIVE.TTY.PROCESS** (page 23.13) which gives the process associated with the window the tty and calls the **BUTTONEVENTFN**. **WINDOWENTRYFN** can be a list of functions and all will be called.

CURSORINFN [Window Property]

Whenever the mouse moves into the window, the **CURSORINFN** is called. If **CURSORINFN** is a list of functions, all will be called.

CURSOROUTFN [Window Property]

The **CURSOROUTFN** is called when the cursor leaves the window. If **CURSOROUTFN** is a list of functions, all will be called.

CURSORMOVEDFN [Window Property]

The **CURSORMOVEDFN** is called whenever the cursor has moved and is inside the window. **CURSORMOVEDFN** can be a list of functions and all will be called. This allows a window function to implement "active" regions within itself by having its **CURSORMOVEDFN** determine if the cursor is in a region of interest, and if so, perform some action.

BUTTONEVENTFN [Window Property]

The **BUTTONEVENTFN** is called whenever there is a change in the state (up or down) of the mouse buttons inside the window. Changes to the mouse state while the **BUTTONEVENTFN** is running will not be interpreted as new button events, and the **BUTTONEVENTFN** will not be re-invoked.

RIGHTBUTTONFN [Window Property]

The **RIGHTBUTTONFN** is called in lieu of the standard window menu operation **(DOWINDOWCOM)** when the **RIGHT** key is depressed in a window. More specifically, the **RIGHTBUTTONFN** is called instead of the **BUTTONEVENTFN** when **(MOUSESTATE (ONLY RIGHT))**. If the **RIGHT** key is to be treated like any other key in a window, supply **RIGHTBUTTONFN** and **BUTTONEVENTFN** with the same function.

When an application program defines its own **RIGHTBUTTONFN**, there is a convention that the default **RIGHTBUTTONFN**, **DOWINDOWCOM** (page 28.7), may be executed by depressing the **RIGHT** key when the cursor is in the header or border of a window. User **RIGHTBUTTONFN**s are encouraged to follow this convention, by calling **DOWINDOWCOM** if the cursor is not in the interior region of the window.

BACKGROUNDBUTTONEVENTFN [Variable]

BACKGROUNDCURSORINFN [Variable]

BACKGROUNDCURSOROUTFN [Variable]

BACKGROUNDCURSORMOVEDFN [Variable]

These variables provide a way of taking action when there is cursor action and the cursor in in the background. They are interpreted like the corresponding window properties. If set to the name of a function, that function will be called, respectively, whenever the cursor is in the background and a button changes, when the cursor moves into the background from a window, when the cursor moved from the background into a window and when the cursor moves from one place in the background to another.

28.4.11 Terminal I/O and Page Holding

Each process has its own terminal i/o stream (accessed as the stream **T**, page 25.1). The terminal i/o stream for the current process can be changed to point to a window by using the function **TTYDISPLAYSTREAM**, so that output and echoing of type-in is directed to a window.

(TTYDISPLAYSTREAM *DISPLAYSTREAM***)** [Function]

Selects the display stream or window *DISPLAYSTREAM* to be the terminal output channel, and returns the previous terminal output display stream. **TTYDISPLAYSTREAM** puts *DISPLAYSTREAM* into scrolling mode and calls **PAGEHEIGHT** with the number of lines that will fit into *DISPLAYSTREAM* given its current Font and Clipping Region. The line length of **TTYDISPLAYSTREAM** is computed (like any other display stream) from its Left Margin, Right Margin, and Font. If one of these fields is changed, its line length is recalculated. If one of the fields used to compute the number of lines (such as the Clipping Region or Font) changes, **PAGEHEIGHT** is not automatically recomputed. **(TTYDISPLAYSTREAM (TTYDISPLAYSTREAM))** will cause it to be recomputed.

If the window system is active, the line buffer is saved in the old **TTY** window, and the line buffer is set to the one saved in the window of the new display stream, or to a newly created line buffer (if it does not have one). Caution: It is possible to move the **TTYDISPLAYSTREAM** to a nonvisible display stream or to a window whose current position is not in its clipping region.

(PAGEHEIGHT *N*) [Function]

If *N* is greater than 0, it is the number of lines of output that will be printed to **TTYDISPLAYSTREAM** before the page is held. A page is held before the *N*+1 line is printed to **TTYDISPLAYSTREAM** without intervening input if there is no terminal input waiting to be read. The output is held with the screen video reversed until a character is typed. Output holding is disabled if *N* is 0. **PAGEHEIGHT** returns the previous setting.

PAGEFULLFN [Window Property]

If the **PAGEFULLFN** window property is non-**NIL**, it will be called with the window as a single argument when the window is full (i.e., when enough has been printed since the last **TTY** interaction so that the next character printed will cause information to be scrolled off the top of the window.)

If the **PAGEFULLFN** window property is **NIL**, the system function **PAGEFULLFN** is called. **PAGEFULLFN** simply returns if there are characters in the type-in buffer for *WINDOW*, otherwise it inverts the window and waits for the user to type a character. **PAGEFULLFN** is user advisable.

Note: The **PAGEFULLFN** window property is only called on windows which are the **TTYDISPLAYSTREAM** of some process.

28.4.12 The TTY Process and the Caret

At any time, one process is designated as the TTY process, which is used for accepting keyboard input. The TTY process can be changed to a given process by calling **GIVE.TTY.PROCESS** (page 23.13), or by clicking the mouse in a window associated with the process. The latter mechanism is implemented with the following window property:

PROCESS [Window Property]

If the **PROCESS** window property is non-**NIL**, it should be a **PROCESS** and will be made the TTY process by **GIVE.TTY.PROCESS** (page 23.13), the default **WINDOWENTRYFN** property (page 28.27). This implements the mechanism by which the keyboard is associated with different processes.

The window system uses a flashing caret (A) to indicate the position of the next window typeout. There is only one caret visible at any one time. The caret in the current TTY process is always visible; if it is hidden by another window, its window is brought to the top. An exception to this rule is that the flashing caret's window is not brought to the top if the user is buttoning or has a shift key down. This prevents the destination window

(which has the tty and caret flashing) from interfering with the window one is trying to select text to copy from.

(CARET *NEWCARET***)** [Function]

Sets the shape that blinks at the location of the next output to the current process. *NEWCARET* should be one of the following:

a **CURSOR** object — If *NEWCARET* is a **CURSOR** object (see page 30.14), it is used to give the new caret shape

OFF — Turns the caret off

NIL — The caret is not changed. **CARET** returns a **CURSOR** representing the current caret

T — Reset the caret to the value of **DEFAULTCARET**. **DEFAULTCARET** can be set to change the initial caret for new processes.

The hotspot of *NEWCARET* indicates which point in the new caret bitmap should be located at the current output position. The previous caret is returned. Note: the bitmap for the caret is not limited to the dimensions **CURSORWIDTH** by **CURSORHEIGHT**.

(CARETRATE *ONRATE OFFRATE***)** [Function]

Sets the rate at which the caret for the current process will flash. The caret will be visible for *ONRATE* milliseconds, then not visible for *OFFRATE* milliseconds. If *OFFRATE* is **NIL** then it is set to be the same as *ONRATE*. If *ONRATE* is **T**, both the "on" and "off" times are set to the value of the variable **DEFAULTCARETRATE** (initially 333). The previous value of **CARETRATE** is returned. If the caret is off, **CARETRATE** return NIL.

28.4.13 Miscellaneous Window Functions

(CLEARW *WINDOW***)** [Function]

Fills *WINDOW* with its background texture, changes its coordinate system so that the origin is the lower left corner of the window, sets its X position to the left margin and sets its Y position to the base line of the uppermost line of text, ie. the top of the window less the font ascent.

(INVERTW *WINDOW SHADE***)** [Function]

Fills the window *WINDOW* with the texture *SHADE* in **INVERT** mode. If *SHADE* is **NIL**, **BLACKSHADE** is used. **INVERTW** returns *WINDOW* so that it can be used inside **RESETFORM**.

(FLASHWINDOW *WIN? N FLASHINTERVAL SHADE***)** [Function]

Flashes the window *WIN?* by "inverting" it twice. *N* is the number of times to flash the window (default is 1). *FLASHINTERVAL* is the length of time in milliseconds to wait between flashes (default is 200). *SHADE* is the shade that will be used to invert the window (default is **BLACKSHADE**).

If *WIN?* is **NIL**, the whole screen is flashed. In this case, the *SHADE* argument is ignored (can only invert the screen).

(WHICHW *X Y***)** [Function]

Returns the window which contains the position in screen coordinates of *X* if *X* is a **POSITION**, the position (*X*,*Y*) if *X* and *Y* are numbers, or the position of the cursor if *X* is **NIL**. Returns **NIL** if the coordinates are not in any window. If they are in more than one window, it returns the uppermost.

Example: **(WHICHW)** returns the window that the cursor is in.

(DECODE/WINDOW/OR/DISPLAYSTREAM *DSORW WINDOWVAR TITLE BORDER***)** [Function]

Returns a display stream as determined by the *DSORW* and *WINDOWVAR* arguments. If *DSORW* is a display stream, it is returned. If *DSORW* is a window, its display stream is returned. If *DSORW* is **NIL**, the litatom *WINDOWVAR* is evaluated. If its value is a window, its display stream is returned. If its value is not a window, *WINDOWVAR* is set to a newly created window (prompting user for region) whose display stream is then returned. If *DSORW* is **NEW**, the display stream of a newly created window is returned. If a window is involved in the decoding, it is opened and if *TITLE* or *BORDER* are given, the **TITLE** or **BORDER** property of the window are reset. The *DSORW* = **NIL** case is most useful for programs that want to display their output in a window, but want to reuse the same window each time they are called. The non-**NIL** cases are good for decoding a display stream argument passed to a function.

(WIDTHIFWINDOW *INTERIORWIDTH BORDER***)** [Function]

Returns the width of the window necessary to have *INTERIORWIDTH* points in its interior if the width of the border is *BORDER*. If *BORDER* is **NIL**, the default border size **WBorder** is used.

(HEIGHTIFWINDOW *INTERIORHEIGHT TITLEFLG BORDER***)** [Function]

Returns the height of the window necessary to have *INTERIORHEIGHT* points in its interior with a border of *BORDER* and, if *TITLEFLG* is non-**NIL**, a title. If *BORDER* is **NIL**, the default border size **WBorder** is used.

WIDTHIFWINDOW and **HEIGHTIFWINDOW** are useful for calculating the width and height for a call to **GETBOXPOSITION** for the purpose of positioning a prospective window.

(MINIMUMWINDOWSIZE *WINDOW***)** [Function]

Returns a dotted pair, the **CAR** of which is the minimum width *WINDOW* needs and the **CDR** or which is the minimum height *WINDOW* needs.

The minimum size is determined by the value of the window property **MINSIZE** of *WINDOW*. If the value of the **MINSIZE** window property is **NIL**, the width is 26 and the height is the height *WINDOW* needs to have its title, border and one line of text visible. If **MINSIZE** is a dotted pair, it is returned. If it is a litatom, it should be a function which is called with **WINDOW** as its first argument, which should return a dotted pair.

28.4.14 Miscellaneous Window Properties

TITLE [Window Property]

Accesses the title of the window. If a title is added to a window whose title is **NIL** or the title is removed (set to **NIL**) from a window with a title, the window's exterior (its region on the screen) is enlarged or reduced to accomodate the change without changing the window's interior. For example, **(WINDOWPROP** *WINDOW* **'TITLE "Results")** changes the title of *WINDOW* to be "Results". **(WINDOWPROP** *WINDOW* **'TITLE NIL)** removes the title of *WINDOW*.

BORDER [Window Property]

Accesses the width of the border of the window. The border will have at most 2 point of white (but never more than half) and the rest black. The default border is the value of the global variable **WBorder** (initially 4).

WINDOWTITLESHADE [Window Property]

Accesses the window title shade of the window. If non-**NIL**, it should be a texture which is used as the "backgound texture" for the title bar on the top of the window. If it is **NIL**, the value of the global variable **WINDOWTITLESHADE** (initially **BLACKSHADE**) is used. Note that black is always used as the background of the title printed in the title bar, so that the letters can be read. The remaining space is painted with the "title shade".

HARDCOPYFN [Window Property]

If non-**NIL**, it should be a function that is called by the window menu command **Hardcopy** (page 28.4) to print the contents of a window. The **HARDCOPYFN** property is called with two arguments, the window and an image stream to print to. If the window does not have a **HARDCOPYFN**, the bitmap image of the window (including the border and title) are printed on the file or printer.

DSP [Window Property]

Value is the display stream of the window. All system functions will operate on either the window or its display stream. This window property cannot be changed using **WINDOWPROP**.

HEIGHT [Window Property]

WIDTH [Window Property]

Value is the height and width of the interior of the window (the usable space not counting the border and title). These window properties cannot be changed using **WINDOWPROP**.

REGION [Window Property]

Value is a region (in screen coordinates) indicating where the window (counting the border and title) is located on the screen. This window property cannot be changed using **WINDOWPROP**.

28.4.15 Example: A Scrollable Window

The following is a simple example showing how one might create a scrollable window.

CREATE.PPWINDOW creates a window that displays the pretty printed expression **EXPR**. The window properties **PPEXPR**, **PPORIGX**, and **PPORIGY** are used for saving this expression, and the initial window position. Using this information, **REPAINT.PPWINDOW** simply reinitializes the window position, and prettyprints the expression again. Note that the whole expression is reformatted every time, even if only a small part actually lies within the window. If this window was going to be used to display very large structures, it would be desirable to implement a more sophisticated **REPAINTFN** that only redisplays that part of the expression within the window. However, this scheme would be satisfactory if most of the items to be displayed are small.

RESHAPE.PPWINDOW resets the window (and stores the initial window position), calls **REPAINT.PPWINDOW** to display the window's expression, and then sets the **EXTENT** property of the

window so that **SCROLLBYREPAINTFN** will be able to handle scrolling and "thumbing" correctly.

```
(DEFINEQ

(CREATE.PPWINDOW
 [LAMBDA (EXPR)          (* rrb " 4-OCT-82 12:06")
                 (* creates a window that displays
                  a pretty printed expression.)

  (PROG (WINDOW)
                 (* ask the user for a piece of the
                  screen and make it into a window.)
    (SETQ WINDOW (CREATEW NIL "PP window"))
                 (* put the expression on the
                  property list of the window so that
                  the repaint and reshape functions
                  can access it.)
    (WINDOWPROP WINDOW (QUOTE PPEXPR) EXPR)
                 (* set the repaint and reshape
                  functions.)
    (WINDOWPROP WINDOW (QUOTE REPAINTFN)
          (FUNCTION REPAINT.PPWINDOW))
    (WINDOWPROP WINDOW (QUOTE RESHAPEFN)
          (FUNCTION RESHAPE.PPWINDOW))
                 (* make the scroll function
                  SCROLLBYREPAINTFN, a system
                  function that uses the repaint
                  function to do scrolling.)
    (WINDOWPROP WINDOW (QUOTE SCROLLFN)
          (FUNCTION SCROLLBYREPAINTFN))
                 (* call the reshape function to
                  initially print the expression and
                  calculate its extent.)
    (RESHAPE.PPWINDOW WINDOW)
    (RETURN WINDOW])


(REPAINT.PPWINDOW
 [LAMBDA (WINDOW REGION)      (* rrb " 4-OCT-82 11:52")

    (* the repainting function for a window with a
     pretty printed expression.  This repainting
     function ignores the region to be repainted
     and repaints the entire window.)

                 (* set the window position to the
                  beginning of the pretty printing
                  of the expression.)
```

```
    (MOVETO (WINDOWPROP WINDOW (QUOTE PPORIGX))
        (WINDOWPROP WINDOW (QUOTE PPORIGY))
        WINDOW)
    (PRINTDEF (WINDOWPROP WINDOW (QUOTE PPEXPR))
        0 NIL NIL NIL WINDOW])


(RESHAPE.PPWINDOW
 [LAMBDA (WINDOW)          (* rrb "4-OCT-82 12:01")
                 (* the reshape function for a
                  window with a pretty printed
                  expression.)
  (PROG (BTM)

      (* set the position of the window so that the
       first character appears in the upper left corner
       and save the X and Y for the repaint function.)

      (DSPRESET WINDOW)
      (WINDOWPROP WINDOW (QUOTE PPORIGX)
          (DSPXPOSITION NIL WINDOW))
      (WINDOWPROP WINDOW (QUOTE PPORIGY)
          (DSPYPOSITION NIL WINDOW))
              (* call the repaint function to
               pretty print the expression in
               the newly cleared window.)
      (REPAINT.PPWINDOW WINDOW)

      (* save the region actually covered by the pretty
       printed expression so that the scrolling routines
       will know where to stop.  The pretty printing of
       the expression does a carriage return after the
       last piece of the expression printed so that the
       current position is the base line of the next line
       of text.  Hence the last visible piece of the
       expression (BTM) is the ending position plus the
       height of the font above the base line (its ASCENT).)

      (WINDOWPROP WINDOW (QUOTE EXTENT)
       (create REGION
          LEFT ← 0
          BOTTOM ← [SETQ BTM (IPLUS
                       (DSPYPOSITION NIL WINDOW)
                       (FONTPROP WINDOW (QUOTE ASCENT]
          WIDTH ←(WINDOWPROP WINDOW (QUOTE WIDTH))
          HEIGHT ←(IDIFFERENCE
                       (WINDOWPROP WINDOW (QUOTE
HEIGHT))
                       BTM])
```

28.5 Menus

A menu is basically a means of selecting from a list of items. The system provides common layout and interactive user selection mechanisms, then calls a user-supplied function when a selection has been confirmed. The two major constituents of a menu are a list of items and a "when selected function." The label that appears for each item is the item itself for non-lists, or its **CAR** if the item is a list. In addition, there are a multitude of different formatting parameters for specifying font, size, and layout. When a menu is created, its unspecified fields are filled with defaults and its screen image is computed and saved.

Menus can be either pop up or fixed. If fixed menus are used, the menu must be included in a window.

(MENU *MENU POSITION RELEASECONTROLFLG* **—)** [Function]

This function provides menus that pop up when they are used. It displays *MENU* at *POSITION* (in screen coordinates) and waits for the user to select an item with a mouse key. Before any mouse key is pressed, the item the mouse is over is boxed. After any key is down, the selected menu item is video reversed. When all keys are released, *MENU*'s **WHENSELECTEDFN** field is called with four arguments: (1) the item selected, (2) the menu, (3) the last mouse key released (**LEFT**, **MIDDLE**, or **RIGHT**), and (4) the reverse list of superitems rolled through when selecting the item and **MENU** returns its value. If no item is selected, **MENU** returns **NIL**. If *POSITION* is **NIL**, the menu is brought up at the value from *MENU*'s **MENUPOSITION** field, if it is a **POSITION**, or at the current cursor position. The orientation of *MENU* with respect to the specified position is determined by its **MENUOFFSET** field.

If *RELEASECONTROLFLG* is **NIL**, this process will retain control of the mouse. In this case, if the user lets the mouse key up outside of the menu, **MENU** return **NIL**. (Note: this is the standard way of allowing the user to indicate that they do not want to make the offered choice.) If *RELEASECONTROLFLG* is non-**NIL**, this process will give up control of the mouse when it is outside of the menu so that other processes can be run. In this case, clicking outside the menu has no effect on the call to **MENU**. If the menu is closed (for example, by right buttoning in it and selecting "**Close**" from the window menu), **MENU** returns **NIL**. Programmers are encouraged to provide a menu item such as

"cancel" or "abort" which gives users a positive way of indicating "no choice".

Note: A "released" menu will stay visible (on top of the window stack) until it is closed or an item is selected.

(ADDMENU *MENU WINDOW POSITION DONTOPENFLG***)** [Function]

This function provides menus that remain active in windows. **ADDMENU** displays *MENU* at *POSITION* (in window coordinates) in *WINDOW*. If the window is too small to display the entire menu, the window is made scrollable. When an item is selected, the value of the **WHENSELECTEDFN** field of *MENU* is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (**LEFT**, **MIDDLE**, or **RIGHT**). More than one menu can be put in a window, but a menu can only be added to one window at a time. **ADDMENU** returns the window into which *MENU* is placed.

If *WINDOW* is **NIL**, a window is created at the position specified by *POSITION* (in screen coordinates) that is the size of *MENU*. If a window is created, it will be opened unless *DONTOPENFLG* is non-**NIL**. If *POSITION* is **NIL**, the menu is brought up at the value of *MENU*'s **MENUPOSITION** field (in window coordinates), if it is a position, or else in the lower left corner of *WINDOW*. If both *WINDOW* and *POSITION* are **NIL**, a window is created at the current cursor position.

Warning: **ADDMENU** resets several of the window properties of *WINDOW*. The **CURSORINFN**, **CURSORMOVEDFN**, and **BUTTONEVENTFN** window properties are replaced with **MENUBUTTONFN**, so that *MENU* will be active. **MENUREPAINTFN** is added to the **REPAINTFN** window property to update the menu image if the window is redisplayed. The **SCROLLFN** window property is changed to **SCROLLBYREPAINTFN** if the window is too small for the menu, to make the window scroll.

(DELETEMENU *MENU CLOSEFLG FROMWINDOW***)** [Function]

This function removes *MENU* from the window *FROMWINDOW*. If *MENU* is the only menu in the window and *CLOSEFLG* is non-**NIL**, its window will be closed (by **CLOSEW**).

If *FROMWINDOW* is **NIL**, the list of currently open windows is searched for one that contains *MENU*. If none is found, **DELETEMENU** does nothing.

28.5.1 Menu Fields

A menu is a datatype with the following fields:

ITEMS [Menu Field]

The list of items to appear in the menu. If an item is a list, its **CAR** will appear in the menu. If the item (or its **CAR**) is a bitmap, the bitmap will be displayed in the menu. The default selection functions interpret each item as a list of three elements: a label, a form whose value is returned upon selection, and a help string that is printed in the prompt window when the user presses a mouse key with the cursor pointing to this item. The default subitem function interprets the fourth element of the list. If it is a list whose **CAR** is the litatom **SUBITEMS** , the **CDR** is taken as a list of subitems.

SUBITEMFN [Menu Field]

A function to be called to determine if an item has any subitems. If an item has subitems and the user rolls the cursor out the right of that item, a submenu with that item's subitems in it pops up. If the user selects one of the items from the submenu, the selected subitem is handled as if it were selected from the main menu. If the user rolls out of the submenu to the left, the submenu is taken down and selection resumes from the main menu.

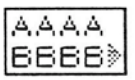
An item with subitems is marked in the menu by a grey, right pointing triangle following the label.

The function is called with two arguments: (1) the menu and (2) the item. It should return a list of the subitems of this item if any. (Note: it is called twice to compute the menu image and each time the user rolls out of the item box so it should be moderately efficient. The default **SUBITEMFN**, **DEFAULTSUBITEMFN**, checks to see if the item is a list whose fourth element is a list whose **CAR** is the litatom **SUBITEMS** and if so, returns the **CDR** of it.

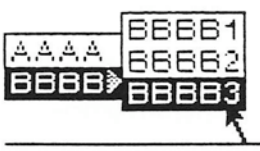
For example:

```
(create MENU
   ITEMS ← '(AAAA (BBBB 'BBBB "help string for BBBB"
                  (SUBITEMS BBBB1 BBBB2 BBBB3))))
```

will create a menu with items **A** and **B** in which **B** will have subitems **B1**, **B2** and **B3**. The following picture below shows this menu as it first appears:



The following picture shows the submenu, with the item **BBBB3** selected by the cursor ( ):



WHENSELECTEDFN [Menu Field]

A function to be called when an item is selected. The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (**LEFT**, **MIDDLE**, or **RIGHT**). The default function **DEFAULTWHENSELECTEDFN** evaluates and returns the value of the second element of the item if the item is a list of at least length 2. If the item is not a list of at least length 2, **DEFAULTWHENSELECTEDFN** returns the item.

Note: If the menu is added to a window with **ADDMENU**, the default **WHENSELECTEDFN** is **BACKGROUNDWHENSELECTEDFN**, which is the same as **DEFAULTWHENSELECTEDFN** except that **EVAL.AS.PROCESS** (page 23.17) is used to evaluate the second element of the item, instead of tying up the mouse process.

WHENHELDFN [Menu Field]

The function which is called when the user has held a mouse key on an item for **MENUHELDWAIT** milliseconds (initially 1200). The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (**LEFT**, **MIDDLE**, or **RIGHT**). **WHENHELDFN** is intended for prompting users. The default is **DEFAULTMENUHELDFN** which prints (in the prompt window) the third element of the item or, if there is not a third element, the string "This item will be selected when the button is released."

WHENUNHELDFN [Menu Field]

If **WHENHELDFN** was called, **WHENUNHELDFN** will be called: (1) when the cursor leaves the item, (2) when a mouse key is released, or (3) when another key is pressed. The function is called with the same three argument values used to call **WHENHELDFN**. The default **WHENUNHELDFN** is the function **CLRPROMPT** (page 28.3), which just clears the prompt window.

MENUPOSITION [Menu Field]

The position of the menu to be used if the call to **MENU** or **ADDMENU** does not specify a position. For popup menus, this is in screen coordinates. For fixed menus, it is in the coordinates of the window the menu is in. The point within the menu image that is placed at this position is determined by **MENUOFFSET**. If **MENUPOSITION** is **NIL**, the menu will be brought up at the cursor position.

MENUOFFSET [Menu Field]

The position in the menu image that is to be located at **MENUPOSITION**. The default offset is (0,0). For example, to bring up a menu with the cursor over a particular menu item, set

its **MENUOFFSET** to a position within that item and set its **MENUPOSITION** to **NIL**.

MENUFONT [Menu Field]

The font in which the items will be appear in the menu. Default is the value of **MENUFONT**.

TITLE [Menu Field]

If non-**NIL**, the value of this field will appear as a title in a line above the menu.

MENUTITLEFONT [Menu Field]

The font in which the title of the menu will be appear. If this is **NIL**, the title will be in the same font as window titles. If it is **T**, it will be in the same font as the menu items.

CENTERFLG [Menu Field]

If non-**NIL**, the menu items are centered; otherwise they are left-justified.

MENUROWS [Menu Field]

MENUCOLUMNS [Menu Field]

These fields control the shape of the menu in terms of rows and columns. If **MENUROWS** is given, the menu will have that number of rows. If **MENUCOLUMNS** is given, the menu will have that number of columns. If only one is given, the other one will be calculated to generate the minimal rectangular menu. (Normally only one of **MENUROWS** or **MENUCOLUMNS** is given.) If neither is given, the items will be in one column.

ITEMHEIGHT [Menu Field]

The height of each item box in the menu. If not specified, it will be the maximum of the height of the **MENUFONT** and the heights of any bitmaps appearing as labels.

ITEMWIDTH [Menu Field]

The width of each item box in the menu. If not specified, it will be the width of the largest item in the menu.

MENUBORDERSIZE [Menu Field]

The size of the border around each item box. If not specified, 0 (no border) is used.

MENUOUTLINESIZE [Menu Field]

The size of the outline around the entire menu. If not specified, a maximum of 1 and the **MENUBORDERSIZE** is used.

CHANGEOFFSETFLG [Menu Field]

(popup menus only) If **CHANGEOFFSETFLG** is non-**NIL**, the position of the menu offset is set each time a selection is confirmed so that the menu will come up next time in the same position relative to the cursor. This will cause the menu to reappear in the same place on the screen if the cursor has not moved since the last selection. This is implemented by changing the **MENUOFFSET** field on each use. If **CHANGEOFFSETFLG** is the atom **X** or the atom **Y**, only the X or the Y coordinate of the **MENUOFFSET** field will be changed. For example, by setting the **MENUOFFSET** position to (-1,0) and setting **CHANGEOFFSETFLG** to **Y**, the menu will pop up so that the cursor is just to the left of the last item selected. This is the setting of the window command menus.

The following fields are read only.

IMAGEHEIGHT [Menu Field]

Returns the height of the entire menu.

IMAGEWIDTH [Menu Field]

Returns the width of the entire menu.

28.5.2 Miscellaneous Menu Functions

(MAXMENUITEMWIDTH *MENU***)** [Function]

Returns the width of the largest menu item label in the menu *MENU*.

(MAXMENUITEMHEIGHT *MENU***)** [Function]

Returns the height of the largest menu item label in the menu *MENU*.

(MENUREGION *MENU***)** [Function]

Returns the region covered by the image of *MENU* in its window.

(WFROMMENU *MENU***)** [Function]

Returns the window *MENU* is located in, if it is in one; **NIL** otherwise.

(DOSELECTEDITEM *MENU ITEM BUTTON*) [Function]

Calls *MENU*'s **WHENSELECTEDFN** on *ITEM* and *BUTTON*. It provides a programmatic way of making a selection. It does not change the display.

(MENUITEMREGION *ITEM MENU*) [Function]

Returns the region occupied by *ITEM* in *MENU*.

(SHADEITEM *ITEM MENU SHADE DS/W*) [Function]

Shades the region occupied by *ITEM* in *MENU*. If *DS/W* is a display stream or a window, it is assumed to be where *MENU* is displayed. Otherwise, **WFROMMENU** is called to locate the window *MENU* is in. Shading is persistent, and is reapplied when the window the menu is in gets redisplayed. To unshade an item, call with a *SHADE* of 0.

(PUTMENUPROP *MENU PROPERTY VALUE*) [Function]

Stores the property *PROPERTY* with the value *VALUE* on a property list in the menu *MENU*. The user can use this property list for associating arbitrary data with a menu object.

(GETMENUPROP *MENU PROPERTY*) [Function]

Returns the value of the *PROPERTY* property of the menu *MENU*.

28.5.3 Examples of Menu Use

Example: A simple menu:

```
(MENU (create MENU ITEMS ← '((YES T) (NO (QUOTE NIL))) ))
```

Creates a menu with items **YES** and **NO** in a single vertical column:

If **YES** is selected, **T** will be returned. Otherwise, **NIL** will be returned.

Example: A simple menu, with centering:

```
(MENU (create MENU TITLE ← "Foo?"
      ITEMS ← '((YES T "Adds the Foo feature.")
                (NO 'NO "Removes the Foo feature."))
      CENTERFLG ← T))
```

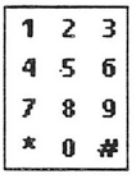
Creates a menu with a title **Foo?** and items **YES** and **NO** centered in a single vertical column:

The strings following the **YES** and **NO** are help strings and will be printed if the cursor remains over one of the items for a period of time. This menu differs from the one above in that it distinquishes the **NO** case from the case where the user clicked outside of the menu. If the user clicks outside of the menu, **NIL** is returned.

Example: A multi-column menu:

```
(create MENU ITEMS ← '(1 2 3 4 5 6 7 8 9 * 0 #)
        CENTERFLG ← T
        MENUCOLUMNS ← 3
        MENUFONT ← (FONTCREATE 'MODERN 10 'BOLD)
        ITEMHEIGHT ← 15
        ITEMWIDTH ← 15
        CHANGEOFFSETFLG ← T)
```

Creates a touch-tone-phone number pad with the items in 15 by 15 boxes printed in Modern 10 bold font:



If used in pop up mode, its first use will have the cursor in the middle. Subsequent use will have the cursor in the same relative location as the previous selection.

Example: A program using a previously-saved menu:

```
(SELECTQ [MENU
    (COND ((type? MENU FOOMENU)
        (* use previously computed menu.)
        FOOMENU)
      (T (* create and save the menu)
        (SETQ FOOMENU
          (create MENU
            ITEMS ← '((A 'A-SELECTED "prompt string for A")
                  (B 'B-SELECTED "prompt string for B"]
  (A-SELECTED (* if A is selected) (DOATHING))
  (B-SELECTED (* if B is selected) (DOBTHING))
  (PROGN (* user selected outside the menu) NIL)))
```

This expression displays a pop up menu with two items, **A** and **B**, and waits for the user to select one. If **A** is selected, **DOATHING** is called. If **B** is selected, **DOBTHING** is called. If neither of these is selected, the form returns **NIL**.

The purpose of this example is to show some good practices to follow when using menus. First, the menu is only created once, and saved in the variable **FOOMENU**. This is more efficient if the menu is used more than once. Second, all of the information about the menu is kept in one place, which makes it easy to

understand and edit. Third, the forms evaluated as a result of selecting something from the menu are part of the code and hence will be known to masterscope (as opposed to the situation if the forms were stored as part of the items). Fourth, the items in the menu have help strings for the user. Finally, the code is commented (always worth the trouble).

28.6 Attached Windows

The attached window facility makes it easy to manipulate a group of window as a unit. Standard window operations like moving, reshaping, opening, and closing can be done so that it appears to the user as if the windows are a single entity. Each collection of attached windows has one main window and any number of other windows that are "attached" to it. Moving or reshaping the main window causes all of the attached windows to be moved or reshaped as well. Moving or reshaping an attached window does not affect the main window.

Attached windows can have other windows attached to them. Thus, it is possible to attach window A to window B when B is already attached to window C. Similarly, if A has other windows attached to it, it can still be attached to B.

(ATTACHWINDOW *WINDOWTOATTACH MAINWINDOW EDGE POSITIONONEDGE WINDOWCOMACTION***)** [Function]

Associates *WINDOWTOATTACH* with *MAINWINDOW* so that window operations done to *MAINWINDOW* are also done to *WINDOWTOATTACH* (the exact set of window operations passed between main windows and attached windows is described on page 28.51). **ATTACHWINDOW** moves *WINDOWTOATTACH* to the correct position relative to *MAINWINDOW*.

Note: A window can be attached to only one other window. Attaching a window to a second window will detach it from the first. Attachments can not form loops. That is, a window cannot be attached to itself or to a window that is attached to it. **ATTACHWINDOW** will generate an error if this is attempted.

EDGE determines which edge of *MAINWINDOW* the attached window is positioned along: it should be one of **TOP**, **BOTTOM**, **LEFT**, or **RIGHT**. If *EDGE* is **NIL**, it defaults to **TOP**.

POSITIONONEDGE determines where along *EDGE* the attached window is positioned. It should be one of the following:

LEFT The attached window is placed on the left (of a **TOP** or **BOTTOM** edge).

RIGHT The attached window is placed on the right (of a **TOP** or **BOTTOM** edge).

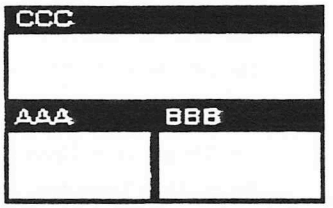
BOTTOM The attached window is placed on the bottom (of a **LEFT** or **RIGHT** edge).

TOP The attached window is placed on the top (of a **LEFT** or **RIGHT** edge).

CENTER The attached window is placed in the center of the edge.

JUSTIFY or **NIL** The attached window is placed to fill the entire edge. **ATTACHWINDOW** reshapes the window if necessary.

Note: The width or height used to justify an attached window includes any other windows that have already been attached to *MAINWINDOW*. Thus **(ATTACHWINDOW BBB AAA 'RIGHT 'JUSTIFY)** followed by **(ATTACHWINDOW CCC AAA 'TOP 'JUSTIFY)** will put **CCC** across the top of both **BBB** and **AAA**:



WINDOWCOMACTION provides a convenient way of specifying how *WINDOWTOATTACH* responds to right button menu commands. The window property **PASSTOMAINCOMS** determines which right button menu commands are directly applied to the attached window, and which are passed to the main window (see page 28.51). Depending on the value of *WINDOWCOMACTION*, the **PASSTOMAINCOMS** window property of *WINDOWTOATTACH* is set as follows:

NIL **PASSTOMAINCOMS** is set to **(CLOSEW MOVEW SHAPEW SHRINKW BURYW)**, so right button menu commands to close, move, shape, shrink, and bury are passed to the main window, and all others are applied to the attached window.

LOCALCLOSE **PASSTOMAINCOMS** is set to **(MOVEW SHAPEW SHRINKW BURYW)**, which is the same as when *WINDOWCOMACTION* is **NIL**, except that the attached window can be closed independently.

HERE **PASSTOMAINCOMS** is set to **NIL**, so all right button menu commands are applied to the attached window.

MAIN **PASSTOMAINCOMS** is set to **T**, so all right button menu commands are passed to the main window.

Note: If the user wants to set the **PASSTOMAINCOMS** window property of an attached window to something else, it must be done *after* the window is attached, since **ATTACHWINDOW** modifies this window property.

(DETACHWINDOW *WINDOWTODETACH***)** [Function]

Detaches *WINDOWTODETACH* from its main window. Returns a dotted pair (*EDGE* . *POSITIONONEDGE*) if *WINDOWTODETACH* was an attached window, **NIL** otherwise. This does not close *WINDOWTODETACH*.

(DETACHALLWINDOWS *MAINWINDOW***)** [Function]

Detaches and closes all windows attached to *MAINWINDOW*.

(FREEATTACHEDWINDOW *WINDOW***)** [Function]

Detaches the attached window *WINDOW*. In addition, other attached windows above (in the case of a **TOP** attached window) or below (in the case of a **BOTTOM** attached window) are moved closer to the main window to fill the gap.

Note: Attached windows that "reject" the move operation (see **REJECTMAINCOMS**, page 28.51) are not moved.

Note: **FREEATTACHEDWINDOW** currently doesn't handle **LEFT** or **RIGHT** attached windows.

(REMOVEWINDOW *WINDOW***)** [Function]

Closes *WINDOW*, and calls **FREEATTACHEDWINDOW** to move other attached windows to fill any gaps.

(REPOSITIONATTACHEDWINDOWS *WINDOW***)** [Function]

Repositions every window attached to *WINDOW*, in the order that they were attached. This is useful as a **RESHAPEFN** for main windows with attached window that don't want to be reshaped, but do want to keep their position relative to the main window when the main window is reshaped.

Note: Attached windows that "reject" the move operation (see **REJECTMAINCOMS**, page 28.51) are not moved.

(MAINWINDOW *WINDOW RECURSEFLG***)** [Function]

If *WINDOW* is not a window, it generates an error. If *WINDOW* is closed, it returns *WINDOW*. If *WINDOW* is not attached to another window, it returns *WINDOW* itself. If *RECURSEFLG* is **NIL** and *WINDOW* is attached to a window, it returns that window. If *RECURSEFLG* is **T**, it returns the first window up the "main window" chain starting at *WINDOW* that is not attached to any other window.

(ATTACHEDWINDOWS *WINDOW COM***)** [Function]

Returns the list of windows attached to *WINDOW*.

If *COM* is non-**NIL**, only those windows attached to *WINDOW* that do not reject the window operation *COM* are returned (see **REJECTMAINCOMS**, page 28.51).

(ALLATTACHEDWINDOWS *WINDOW***)** [Function]

Returns a list of all of the windows attached to *WINDOW* or attached to a window attached to it.

(WINDOWREGION *WINDOW COM***)** [Function]

Returns the screen region occupied by *WINDOW* and its attached windows, if it has any.

If *COM* is non-**NIL**, only those windows attached to *WINDOW* that do not reject the window operation *COM* are considered in the calculation (see **REJECTMAINCOMS**, page 28.51).

(WINDOWSIZE *WINDOW***)** [Function]

Returns the size of *WINDOW* and its attached windows (if any), as a dotted pair (*WIDTH* . *HEIGHT*).

(MINATTACHEDWINDOWEXTENT *WINDOW***)** [Function]

Returns the minimum size that *WINDOW* and its attached windows (if any) will accept, as a dotted pair (*WIDTH* . *HEIGHT*).

28.6.1 Attaching Menus To Windows

The following functions are provided to associate menus to windows.

(MENUWINDOW *MENU VERTFLG***)** [Function]

Returns a closed window that has the menu *MENU* in it. If *MENU* is a list, a menu is created with *MENU* as its **ITEMS** menu field (see page 28.39). Otherwise, *MENU* should be a menu. The returned window has the appropriate **RESHAPEFN**, **MINSIZE** and **MAXSIZE** window properties to allow its use in a window group.

If both the **MENUROWS** and **MENUCOLUMNS** fields of *MENU* are **NIL**, *VERTFLG* is used to set the default menu shape. If *VERTFLG* is non-**NIL**, the **MENUCOLUMNS** field of *MENU* will be set to 1 (the menu items will be listed vertically); otherwise the **MENUROWS** field of *MENU* will be set to 1 (the menu items will be listed horizontally).

(ATTACHMENU *MENU MAINWINDOW EDGE POSITIONONEDGE NOOPENFLG***)** [Function]

Creates a window that contains the menu *MENU* (by calling **MENUWINDOW**) and attaches it to the window *MAINWINDOW*

on edge *EDGE* at position *POSITIONONEDGE*. The menu window is opened unless *MAINWINDOW* is closed, or *NOOPENFLG* is **T**.

If *EDGE* is either **LEFT** or **RIGHT**, **MENUWINDOW** will be called with *VERTFLG* = **T**, so the menu items will be listed vertically; otherwise the menu items will be listed horizontally. These defaults can be overridden by specifying the **MENUROWS** or **MENUCOLUMNS** fields in *MENU*.

(CREATEMENUEDWINDOW *MENU WINDOWTITLE LOCATION WINDOWSPEC***)** [Function]

Creates a window with an attached menu and returns the main window. *MENU* is the only required argument, and may be a menu or a list of menu items. *WINDOWTITLE* is a string specifying the title of the main window. *LOCATION* specifies the edge on which to place the menu; the default is **TOP**. *WINDOWSPEC* is a region specifying a region for the aggregate window; if **NIL**, the user is prompted for a region.

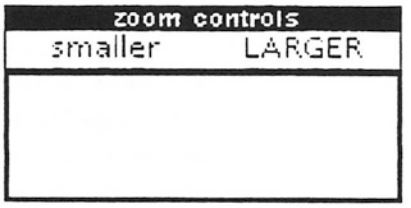
Examples:

```
(SETQ MENUW
 (MENUWINDOW
  (create MENU
   ITEMS ← '(smaller LARGER)
   MENUFONT ← '(MODERN 12)
   TITLE ← "zoom controls"
   CENTERFLG ← T
   WHENSELECTEDFN ← (FUNCTION ZOOMMAINWINDOW))))
```

creates (but does not open) a menu window that contains the two items "**smaller**" and "**LARGER**" with the title "**zoom controls**" and that calls the function **ZOOMMAINWINDOW** when an item is selected. Note that the menu items will be listed horizontally, because **MENUWINDOW** is called with *VERTFLG* = **NIL**, and the menu does not specify either a **MENUROWS** or **MENUCOLUMNS** field.

```
(ATTACHWINDOW MENUW
 (CREATEW '(50 50 150 50))
 'TOP
 'JUSTIFY)
```

creates a window on the screen and attaches the above created menu window to its top:



(CREATEMENUEDWINDOW

```
(create MENU
  ITEMS ← '(smaller LARGER)
  MENUFONT ← '(MODERN 12)
  TITLE ← "zoom controls"
  CENTERFLG ← T
  WHENSELECTEDFN ← (FUNCTION ZOOMMAINWINDOW))))
```

creates the same sort of window in one step, prompting the user for a region.

28.6.2 Attached Prompt Windows

Many packages have a need to display status information or prompt for small amounts of user input in a place outside their standard window. A convenient way to do this is to attach a small window to the top of the program's main window. The following functions do so in a uniform way that can be depended on among diverse applications.

(GETPROMPTWINDOW *MAINWINDOW #LINES FONT DONTCREATE***)** [Function]

Returns the attached prompt window associated with *MAINWINDOW*, creating it if necessary. The window is always attached to the top of *MAINWINDOW*, has **DSPSCROLL** set to **T**, and has a **PAGEFULLFN** of **NILL** to inhibit page holding. The window is at least *#LINES* lines high (default 1); if a pre-existing window is shorter than that, it is reshaped to make it large enough. *FONT* is the font to give the prompt window (defaults to the font of *MAINWINDOW*), and applies only when the window is first created. If *DONTCREATE* is true, returns the window if it exists, otherwise **NIL** without creating any prompt window.

(REMOVEPROMPTWINDOW *MAINWINDOW***)** [Function]

Detaches the attached prompt window associated with *MAINWINDOW* (if any), and closes it.

28.6.3 Window Operations And Attached Windows

When a window operation, such as moving or clearing, is performed on a window, there is a question about whether or not that operation should also be performed on the windows attached to it or performed on the window it is attached to. The "right" thing to do depends on the window operation: it makes sense to independently redisplay a single window in a collection of windows, whereas moving a single window usually implies moving the whole group of windows. The interpretation of window operations also depends on the application that the

window group is used for. For some applications, it may be desirable to have a window group where individual windows can be moved away from the group, but still be conceptually attached to the group for other operations. The attached window facility is flexible enough to allow all of these possibilities.

The operation of window operations can be specified by each attached window, by setting the following two window properties:

PASSTOMAINCOMS [Window Property]

Value is a list of window commands (e.g. **CLOSEW**, **MOVEW**) which, when selected from the attached window's right-button menu, are actually applied to the central window in the group, instead of being applied to the attached window itself. The "central window" is the first window up the "main window" chain that is not attached to any other window.

If **PASSTOMAINCOMS** is **NIL**, all window operations are directly applied to the attached window. If **PASSTOMAINCOMS** is **T**, all window operations are passed to the central window.

Note: **ATTACHWINDOW** (page 28.45) allows this window property to be set to commonly-used values by using its *WINDOWCOMACTION* argument. **ATTACHWINDOW** always sets this window property, so users must modify it directly only *after* attaching the window to another window.

REJECTMAINCOMS [Window Property]

Value is a list of window commands that the attached window will not allow the main window to apply to it. This is how a window can say "leave me out of this group operation."

If **REJECTMAINCOMS** is **NIL**, all window commands may be applied to this attached window. If **REJECTMAINCOMS** is **T**, no window commands may be applied to this attached window.

Note: The **PASSTOMAINCOMS** and **REJECTMAINCOMS** window properties affect right-button menu operations applied to main windows or attached windows, and the action of programmatic window functions (**SHAPEW**, **MOVEW**, etc.) applied to main windows. However, these window properties do *not* affect the action of window functions applied to attached windows.

The following list describes the behavior of main and attached windows under the window operations, assuming that all attached windows have their **REJECTMAINCOMS** window property set to **NIL** and **PASSTOMAINCOMS** set to **(CLOSEW MOVEW SHAPEW SHRINKW BURYW)** (the default if

ATTACHWINDOW is called with *WINDOWCOMACTION* = **NIL**, see page 28.45).

The behavior for any particular operation can be changed for particular attached windows by setting the standard window properties (e.g., **MOVEFN** or **CLOSEFN**) of the attached window. An exception is the **TOTOPFN** property of an attached window, that is set to bring the whole window group to the top and should not be set by the user (although users can *add* functions to the **TOTOPFN** window property).

Move — If the main window moves, all attached windows move with it, and the relative positioning between the main window and the attached windows is maintained. If the region is determined interactively, the prompt region for the move is the union of the extent of the main window and all attached windows (excluding those with **MOVEW** in their **REJECTMAINCOMS** window property).

If an attached window is moved by calling the function **MOVEW**, it is moved without affecting the main window. If the right-button window menu command **Move** is called on an attached window, it is passed on to the main window, so that all windows in the group move.

Reshape — If the main window is reshaped, the minimum size of it and all of its attached windows is used as the minimum of the space for the result. Any space greater than the minimum is distributed among the main window and its attached windows. Attached windows with **SHAPEW** on their **REJECTMAINCOMS** window property are ignored when finding the minimum size, creating a "ghost" region, or distributing space after a reshape.

If an attached window is reshaped by calling the function **SHAPEW**, it is reshaped independently. If the right-button window menu command **Shape** is called on an attached window, it is passed on to the main window, so the whole group is reshaped.

Note: Reshaping the main window will restore the conditions established by the call to **ATTACHWINDOW**, whereas moving the main window does not. Thus, if **A** is attached to the top of **B** and then moved by the user, its new position relative to **B** will be maintained if **B** is moved. If **B** is reshaped, **A** will be reshaped to the top of **B**. Additionally, if, while **A** is moved away from the top of **B**, **C** is attached to the top of **B**, **C** will position itself above where **A** used to be.

Close — If the main window is closed, all of the attached windows are closed also and the links from the attached windows to the main window are broken. This is necessary for the windows to be garbage collected.

If an attached window is closed by calling the function **CLOSEW**, it is closed without affecting the main window. If the right-button window menu command **Close** is called on an attached window, it is passed on to the main window. Note that closing an attached window detaches it.

Open — If the main window is opened, it opens all attached windows and reestablishes links from them to the main window.

Attached windows can be opened independently and this does not affect the main window. Note that it is possible to reopen a closed attached window and not have it linked to its main window.

Shrink — The collection of windows shrinks as a group. The **SHRINKFN**s of the attached windows are evaluated but the only icon displayed is the one for the main window.

Redisplay — The main or attached windows can be redisplayed independently.

Totop — If any main or attached window is brought to the top, all of the other windows are brought to the top also.

Expand — Expanding any of the windows expands the whole collection.

Scrolling — All of the windows involved in the group scroll independently.

Clear — All windows clear independently of each other.

28.6.4 Window Properties Of Attached Windows

Windows that are involved in a collection either as a main window or as an attached window have properties stored on them. The only properties that are intended to be set be set by the user are the **MINSIZE**, **MAXSIZE**, **PASSTOMAINCOMS**, and **REJECTMAINCOMS** window properties. The other properties should be considered read only.

MINSIZE [Window Property]

MAXSIZE [Window Property]

Each of these window properties should be a dotted pair (*WIDTH* . *HEIGHT*) or a function to apply to the window that returns a dotted pair. The numbers are used when the main window is reshaped. The **MINSIZE** is used to determine the size of the smallest region acceptable during reshaping. Any amount greater than the collective minimum is spread evenly among the windows until each reaches **MAXSIZE**. Any excess is given to the main window.

Note: If you give the main window of an attached window group a **MINSIZE** or **MAXSIZE** property, its value is moved to the

MAINWINDOWMINSIZE or **MAINWINDOWMAXSIZE** property, so that the main window can be given a size function that computes the minimum or maximum size of the entire group. Thus, if you want to change the main window's minimum or maximum size after attaching windows to it, you should change the **MAINWINDOWMINSIZE** or **MAINWINDOWMAXSIZE** property instead.

Note: This doesn't address the hard problem of overlapping attached windows side to side, for example if window **A** was attached as [**TOP**, **LEFT**] and **B** as [**TOP**, **RIGHT**]. Currently, the attached window functions do not worry about the overlap.

The default **MAXSIZE** is **NIL**, which will let the region grow indefinitely.

MAINWINDOW [Window Property]

Pointer from attached windows to the main window of the group. This link is not available if the main window is closed. The function **MAINWINDOW** (page 28.47) is the preferred way to access this property.

ATTACHEDWINDOWS [Window Property]

Pointer from a window to its attached windows. The function **ATTACHEDWINDOWS** (page 28.47) is the preferred way to access this property.

WHEREATTACHED [Window Property]

For attached windows, a dotted pair (*EDGE* . *POSITIONONEDGE*) giving the edge and position on the edge that determine how the attached window is placed relative to its main window.

The **TOTOPFN** window property on attached windows and the properties **TOTOPFN**, **DOSHAPEFN**, **MOVEFN**, **CLOSEFN**, **OPENFN**, **SHRINKFN**, **EXPANDFN** and **CALCULATEREGIONFN** on main windows contain functions that implement the attached window manipulation facilities. Care should be used in modifying or replacing these properties.