27. 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 the Windows section below). 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 the Menus section below).

Sometimes, a program needs to use a number of windows, displaying related information. The attached window facility (see the Attached Windows section below) 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.

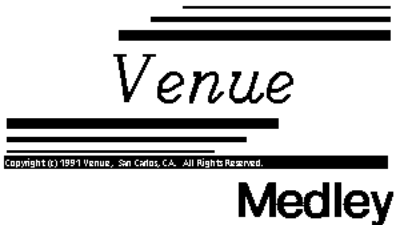
Warning: The window system assumes that all programs follow certain conventions concerning control of the screen. All user programs should use perform display operations using windows and menus. In particular, user programs should not perform operate directly on the screen bitmap; otherwise the window 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.

Using the Window System

When Medley 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, "Medley" 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 Chapter 13). 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 the TTY Process and the Caret section in this chapter).

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 ... EXP*) [NoSpread Function]

Clears the prompt window, and prints *EXP* through *EXP* in the prompt window.

(**CLRPROMPT**) [Function]

Clears the prompt window.

The Medley 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, pressing the RIGHT mouse button 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 button. In these cases, there is a convention that the default command menu may be brought up by depressing the RIGHT button when the cursor is in the header or border of a window (see the Mouse Activity in Windows section in this chapter). The operations are:

Close [Window Menu Command]

Closes the window, i.e, removes it from the screen. (See CLOSEW in the Opening and Closing Windows section in this chapter.)

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 button is down, bits are added. While the MIDDLE button 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 in the Redisplaying Windows section in this chapter.)

Hardcopy [Window Menu Command]

Prints the contents of the window to the printer. If the window has a window property HARDCOPYFN, 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 (see Chapter 29), 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 pressing and then releasing the LEFT button. During this time a ghost frame will indicate where the window will reappear when the key is released. (See GETBOXPOSITION in the Interactive Display Functions section below.)

Shape [Window Menu Command]

Allows the user to specify a new region for the existing window contents. If the LEFT button is used to specify the new region, the reshaped window can be placed anywhere. If the MIDDLE button 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 (see the Reshaping Windows section below).

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 in the Shrinking Windows into Icons section in this chapter) 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 in the Shrinking Windows into Icons section in this chapter.)

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 Chapter 12), 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 (see Chapter 12), 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 (see Chapter 29). 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 Chapter 23).

Various system utilities (TEdit, SEdit, 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.

Changing the Window System

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 APPLYs 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 APPLYed 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 EVALed.

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 the Menu Fields section below), 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 APPLYed 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 APPLYed 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 EVALed.

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 EVALed.

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* (see Chapter 30), 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.

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 Chapter 27), 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.

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.

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 (see Chapter 27).

(**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.

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 OPENFNs 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 CLOSEFNs 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 OPENFNs is the atom DON'T, the window will not be opened. Otherwise, the OPENFNs 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 CLOSEFNs are the atom DON'T, or if the value returned by any of the CLOSEFNs 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.

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, 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 REPAINTFNs 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 (see the Miscellaneous Window Functions section below) should not be used in REPAINTFNs because it resets the window's coordinate system. If a REPAINTFN wants to clear its region first, it should use DSPFILL (see Chapter 27).

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. 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.

DOSHAPEFNs 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: *WINDOW*; a bitmap that contains *WINDOW*'s previous screen image; and 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.

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, 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. 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.

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 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 MOVEFNs, 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.

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. 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.

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 (see the beginning of this chapter), 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 SHRINKFNs are the atom DON'T, or if the value returned by any of the SHRINKFNs is the atom DON'T, the window will not be shrunk.

EXPANDREGIONFN [Window property]

EXPANDREGIONFN, if non-NIL, should be the function to be called (with the window as its argument) before the window is actually expanded.

The EXPANDREGIONFN must return NIL or a valid region, and must not do any window operations (e.g., redisplaying). If NIL is returned, the window is expanded normally, as if the EXPANDREGIONFN had not existed. The region returned specifies the new region for the main window only, not for the group including any of its attached windows. The window will be opened in its new shape, and any attached windows will be repositioned or rejustified appropriately. The main window must have a REPAINTFN which can repaint the entire window under these conditions.

As with expanding windows normally, the OPENFN for the main window is not called.

Also, the window is reshaped without checking for a special shape function (e.g., a DOSHAPEFN).

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 explicitly giving a TOWHAT argument.

DEFAULTICONFN [Variable]

Changes how an icon is created when a window having no ICONFN is shrunk or when SHRINKW, with a TOWHAT argument of a string, is called. The value of DEFAULTICONFN is a function of two arguments (window text); text is either NIL or a string. DEFAULTICONFN returns an icon window.

The initial value of DEFAULTICONFN is MAKETITLEBARICON. It creates a window that is a title bar only; the title is either the text argument, the window's title, or "Icon made <date>" for titleless windows. MAKETITLEBARICON places the title bar at some corner of the main window.

An alternative behavior is available by setting DEFAULTICONFN to be TEXTICON. TEXTICON creates a titled icon window from the text or window's title.

You can now copy-select titled icons such as those used by FileBrowser, SEdit, TEdit, Sketch. The default behavior is that the icon's title is unread (via BKSYSBUF), but if the icon window has a COPYFN property, that gets called instead, with the icon window as its argument. For example, if the name displayed in an icon is really a symbol, and you want copy selection to cause the name to be unread correctly with respect to the package and read table of the exec you are copying into, you could put the following COPYFN property on the icon window:

```
(LAMBDA (WINDOW)
```

```
(IL:BKSYSBUF <fetch symbolic name from window> T ))
```

EXPANDFN [Window Property]

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

Creating Icons with ICONW

ICONW is a group of functions available for building small windows of arbitrary shape. These windows are principally for use as icons for shrinking windows; i.e., these functions are likely to be invoked from within the ICONFN of a window. An icon is specified by supplying its image (a bitmap) and a mask that specifies its shape. The mask is a bitmap of the same dimensions as the image whose bits are on (black) in those positions considered to be in the image, and off (white) in those positions where the background should show through. By using the mask and appropriate window functions, ICONW maintains the illusion that the icon window is nonrectangular, even though the actual window itself is rectangular. The illusion is not complete, of course. For example, if you try to select what looks like the background (or an occluded window) around the icon but still within its rectangular perimeter, the icon window itself is selected. Also, if you move a window occluded by an icon, the icon never notices that the background changed behind it. Icons created with ICONW can also have titles; some part of the image can be filled with text computed at the time the icon is created, or text may be changed after creation.

Creating Icons

Two types of icons can be created with ICONW, a borderless window containing an image defined by a mask and a window with a title.

(**ICONW** *IMAGE MASK POSITION NOOPENFLG*) [Function]

Creates a window at *POSITION*, or prompts for a position if *POSITION* is NIL. The window is borderless, and filled with *IMAGE*, as cookie-cut by *MASK*. If *MASK* is NIL, the image is considered rectangular (i.e., *MASK* defaults to a black bitmap of the same dimensions as *IMAGE*). If *NOOPENFLG* is T, the window is returned unopened.

(**TITLEDICONW** *ICON TITLE FONT POSITION NOOPENFLG JUST BREAKCHARS OPERATION*)

[Function]

Creates a titled icon at *POSITION*, or prompts for a position if *POSITION* is NIL. If *NOOPENFLG* is T, the window is returned unopened. The argument *ICON* is an instance of the record TITLEDICON, which specifies the icon image and mask, as with ICONW, and a region within the image to be used for displaying the title. Thus, the *ICON* argument is usually of the form

(create TITLEDICON ICON ← someIconImage

MASK ← iconMask TITLEREG ← someRegionWithinICON)

The title region is specified in coordinates relative to the icon, i.e., the lower-left corner of the image bitmap is (0, 0). The mask can be NIL if the icon is rectangular. The image should be white where it is covered by the title region. TITLEDICONW clears the region before printing on it. The title is printed into the specified region in the image, using *FONT*. If *FONT* is NIL it defaults to the value of DEFAULTICONFONT, initially Helvetica 10. The title is broken into multiple lines if necessary; TITLEDICONW attempts to place the breaks at characters that are in the list of character codes *BREAKCHARS*. *BREAKCHARS* defaults to (CHARCODE (SPACE ÿ)). In addition, line breaks are forced by any carriage returns in *TITLE*, independent of *BREAKCHARS*. *BREAKCHARS* is ignored if a long title would not otherwise fit in the specified region. For convenience, *BREAKCHARS* = FILE means the title is a file name, so break at file name field delimiters. The argument *JUST* indicates how the text should be justified relative to the region. It is an atom or list of atoms chosen from *TOP*, *BOTTOM*, *LEFT*, or *RIGHT*, which indicate the vertical positioning (flush to top or bottom) and/or horizontal positioning (flush to left edge or right). If *JUST* = NIL, the text is centered. The argument *OPERATION* is a display stream operation indicating how the title should be printed. If *OPERATION* is INVERT, then the title is printed white-on-black. The default *OPERATION* is REPLACE, meaning black-on-white. ERASE is the same as INVERT; PAINT is the same as REPLACE.

For convenience, TITLEDICONW can also be used to create icons that consist solely of a title, with no special image. If the argument *ICON* is NIL, TITLEDICONW creates a rectangular icon large enough to contain *TITLE*, with a border the same width as that on a regular window. The remaining arguments are as described above, except that a *JUST* of *TOP* or *BOTTOM* is not meaningful.

In the Medley release, TITLEDICONW can create icons with white text on a black background. To get this effect, your icon image must be black in the correct area, and you must specify the *OPERATION* argument as INVERT.

In Medley, you can copy- select the title of an icon.

Modifying Icons

(**ICONW.TITLE** *ICON TITLE*) [Function]

Returns the current title of the window *ICON*, which must be a window returned by TITLEDICONW. In addition, if *TITLE* is non-NIL, makes *TITLE* the new title of the window and repaints it accordingly. To erase the current title, make *TITLE* a null string.

(**ICONW.SHADE** *WINDOW SHADE*) [Function]

Returns the current shading of the window ICON, which must be a window returned by ICONW or TITLEDICONW. In addition, if *SHADE* is non-NIL, paints the texture *SHADE* on *WINDOW*. A typical use for this function is to communicate a change of state in a window that is shrunken, without reopening the window. To remove any shading, make *SHADE* be WHITESHADE.

Default Icons

When you shrink a window that has no ICONFN, the system currently creates an icon that looks like the window's title bar. You can make the system instead create titled icons by setting the global variable DEFAULTICONFN to the value TEXTICON.

(**TEXTICON** *WINDOW TEXT*) [Function]

Creates a titled icon window for the main window *WINDOW* containing the text *TEXT*, or the window's title if *TEXT* is NIL.

DEFAULTTEXTICON [Variable]

The value that TEXTICON passes to TITLEDICONW as its ICON argument. Initially it is NIL, which creates an unadorned rectangular window. However, you can set it to a TITLEDICON record of your choosing if you would like default icons to have a different appearance.

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 (see Chapter 27).

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, see Chapter 27) 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 (see the Miscellaneous Window Properties section below).

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 button 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 button 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 button 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, below).

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 below.

(**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 N *MWINDOW* 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 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, BITBLTs 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, SCROLLBYREPAINTFN will limit scrolling in the X and Y directions according to the value of the window property SCROLLEXTENTUSE.

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, see Chapter 27).

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. (see Chapter 27) 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. For most scrolling windows, the SCROLLFN function should be SCROLLBYREPAINTFN.

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.

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.

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.

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` 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 button is depressed in a window. More specifically, the RIGHTBUTTONFN is called instead of the BUTTONEVENTFN when (MOUSESTATE (ONLY RIGHT)). If the RIGHT button 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 , may be executed by pressing the RIGHT button when the cursor is in the header or border of a window. User RIGHTBUTTONFNs 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.

Terminal I/O and Page Holding

Each process has its own terminal i/o stream (accessed as the stream T, see Chapter 25). 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.

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 (see Chapter 23), 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 (see Chapter 23), the default

WINDOWENTRYFN property (see above). This implements the mechanism by which the keyboard is associated with different processes.

The window system uses a flashing caret () 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 Chapter 30), 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.

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.

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 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.

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])
)
```

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 button 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.

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.

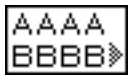
An item with subitems is marked in the menu by a grey, right pointing triangle following the label.

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. (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 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, 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.

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*.

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:

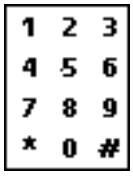
The strings following the YES and NO are help strings and will be printed if the cursor remains over one of the items for a period of time. This menu differs from the one above in that it distinquishes the NO case from the case where the user clicked outside of the menu. If the user clicks outside of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

Free Menus

Free Menus are powerful and flexible menus that are useful for applications needing menus with different types of items, including command items, state items, and items that can be edited. A Free Menu is part of a window. It can can be opened and closed as desired, or attached as a control menu to the application window.

Making a Free Menu

A Free Menu is built from a description of the contents and layout of the menu. As a Free Menu is simply a group of items, a Free Menu Description is simply a specification of a group of items. Each group has properties associated with it, as does each Free Menu Item. These properties specify the format of the items in the group, and the behavior of each item. The function `FREEMENU` takes a Free Menu Description, and returns a closed window with the Free Menu in it.

The easiest way to make a Free Menu is to define a specific function which calls `FREEMENU` with the Free Menu Description in the function. This function can then also set up the Free Menu window as required by the application. The Free Menu Description is saved as part of the specific function when the application is saved. Alternately, the Free Menu Description can be saved as a variable in your file; then just call `FREEMENU` with the name of the variable. This may be a more difficult alternative if the backquote facility is used to build the Free Menu Description.

Free Menu Formatting

A Free Menu can be formatted in one of four ways. The items in any group can be automatically laid out in rows, in columns, or in a table, or else the application can specify the exact location of each item in the group. Free Menu keeps track of the region that a group of items occupies, and items can be justified within that region. This way an item can be automatically positioned at one of the nine justification locations, top-left, top-center, top-right, middle-left, etc.

Free Menu Description

A Free Menu Description, specifying a group of items, is a list structure. The first entry in the list is an optional list of the properties for this group of items. This entry is in the form:

```
(PROPS   <PROP> <VALUE> <PROP> <VALUE> ...)
```

The keyword `PROPS` determines whether or not the optional group properties list is specified..

One important group property is `FORMAT`. The four types of formatting, `ROW`, `TABLE`, `COLUMN`, or `EXPLICIT`, determine the syntax of the rest of the Free Menu Description. When using `EXPLICIT` formatting, the rest of the description is any number of Item Descriptions which have `LEFT` and `BOTTOM` properties specifying the position of the item in the menu. The syntax is:

```
((PROPS FORMAT EXPLICIT ...)
 <ITEM DESCRIPTION>
 <ITEM DESCRIPTION> ...)
```

When using ROW or TABLE formatting, the rest of the description is any number of item groups, each group corresponding to a row in the menu. These groups are identical in syntax to an EXPLICIT group description. The groups have an optional PROPS list and any number of Item Descriptions. The items need not have LEFT and BOTTOM properties, as the location of each item is determined by the formatter. However, the order of the rows and items is important. The menu is laid out top to bottom by row, and left to right within each row. The syntax is:

```
((PROPS FORMAT ROW ...)      ; props of this group
 (<ITEM DESCRIPTION>         ; items in first row
  <ITEM DESCRIPTION> ...)
  ((PROPS ...)               ; props of second row
   <ITEM DESCRIPTION>        ; items in second row
   <ITEM DESCRIPTION> ...))
```

(The comments above only describe the syntax.)

For COLUMN formatting, the syntax is identical to that of ROW formatting. However, each group of items corresponds to a column in the menu, rather than a row. The menu is laid out left to right by column, top to bottom within each column.

Finally, a Free Menu Description can have recursively nested groups. Anywhere the description can take an Item Description, it can take a group, marked by the keyword GROUP. A nested group inherits all of the properties of its mother group, by default. However, any of these properties can be overridden in the nested groups PROPS list, including the FORMAT. The syntax is:

```
(               ; no PROPS list, default row format
(<ITEM DESCRIPTION>          ; first in row
(GROUP          ; nested group, second in row
   (PROPS FORMAT COLUMN ...) ; optional props
   (<ITEM DESCRIPTION> ...)  ; first column
   (<ITEM DESCRIPTION> ...))
    <ITEM DESCRIPTION>))     ; third in row
```

Here is an example of a simple Free Menu Description for a menu which might provide access to a simple data base:

```
(((LABEL LOOKUP SELECTEDFN MYLOOKUPFN)
  (LABEL EXIT SELECTEDFN MYEXITFN))
 ((LABEL Name: TYPE DISPLAY) (LABEL "" TYPE EDIT ID NAME))
 ((LABEL Address: TYPE DISPLAY) (LABEL "" TYPE EDIT ID ADDRESS))
 ((LABEL Phone: TYPE DISPLAY)
```

```
  (LABEL "" TYPE EDIT LIMITCHARS MYPHONEP ID PHONE)))
```

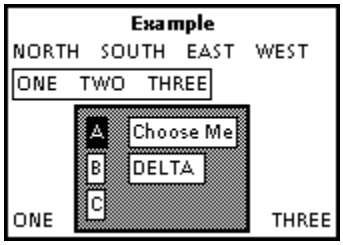
This menu has two command buttons, LOOKUP and EXIT, and three edit fields, with IDs NAME, PHONE, and ADDRESS. The Edit items are initialized to the empty string, as in this example they need no other initial value. The user could select the Name: prompt, type a person's name, and then press the LOOKUP button. The function MYLOOKUPFN would be called. That function would look at the NAME Edit item, look up that name in the data base, and fill in the rest of the fields appropriately. The PHONE item has MYPHONEP as a LIMITCHARS function. This function would be called when editing the phone number, in order to restrict input to a valid phone number. After looking up Perry, the Free Menu might look like:

```
LOOKUP EXIT
Name: Herbert Q Perry
Address: 13 Middleperry Dr
Phone: (411) 767-1234
```

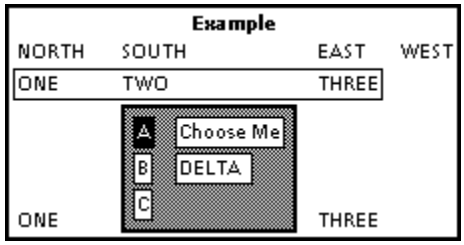
Here is a more complicated example:

```
((PROPS FONT (MODERN 10))
 ((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))
 ((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))
 ((PROPS ID ROW3 BOX 1)
  (LABEL ONE) (LABEL TWO) (LABEL THREE))
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
     ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT
T))
     (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
     (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
       ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
           INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
     (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
  (LABEL THREE)))
```

which will produce the following Free Menu:



And if the Free Menu were formatted as a Table, instead of in Rows, it would look like:



The following breakdown of the example explains how each part contributes to the Free Menu shown above.

`(PROPS FONT (MODERN 10))`

This line specifies the properties of the group that is the entire Free Menu. These properties are described in Section 28.7.4, Free Menu Group Properties. In this example, all items in the Free Menu, unless otherwise specified, will be in Modern 10.

`((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))`

This line of the Free Menu Description describes the first row of the menu. Since the `FORMAT` specification of a Free Menu is, by default, `ROW` formatting, this line sets the first row in the menu. If the menu were in `COLUMN` formatting, this position in the description would specify the first column in the menu.

In this example the first row contains only one item. The item is, by default, a type `MOMENTARY` item. It has its own Font declaration `(FONT (MODERN 10 BOLD))`, that overrides the font specified for the Free Menu as a whole, so the item appears bolded.

Finally, the item is justified, in this case centered. The `HJUSTIFY` Item Property indicates that the item is to be centered horizontally within its row.

`((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))`

This line specifies the second row of the menu. The second row has four very simple items, labeled NORTH, SOUTH, EAST, and WEST next to each other within the same row.

```
((PROPS ID ROW3 BOX 1)
 (LABEL ONE) (LABEL TWO) (LABEL THREE))
```

The third row in the menu is similar to the second row, except that it has a box drawn around it. The box is specified in the PROPS declaration for this row. Rows (and columns) are just like Groups in that the first thing in the declaration can be a list of properties for that row. In this case the row is named by giving it an ID property of ROW3. It is useful to name your groups if you want to be able to access and modify their properties later (via the function FM.GROUPPROP). It is boxed by specifying the BOX property with a value of 1, meaning draw the box one dot wide.

```
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
   ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
          (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
          (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
         ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
           INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
          (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
  (LABEL THREE)))
```

This part of the description specifies the fourth row in the menu. This row consists of: an item labelled ONE, a group of items, and an item labelled THREE. That is, Free Menu thinks of the group as an entry, and formats the rest of the row just as it it were a large item.

```
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
   ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
    (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
    (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
   ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
     INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
    (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
```

The second part of this row is a nested group of items. It is declared as a group by placing the keyword GROUP as the first word in the declaration. A group can be declared anywhere a Free Menu Description can take a Free Menu Item Description (as opposed to a row or column declaration).

The first thing in what would have been the second item declaration in this row is the keyword GROUP. Following this keyword comes a normal group description, starting with an optional list of properties, and followed by any number of things to go in the group (based on the format of the group).

This group's Props declaration is:

```
(PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4).
```

It specifies that the group is to be formatted as a number of columns (instead of rows, the default). The entire group will have a background shade of 23130, and a box of width 2 around it, as you can see in the sample menu. The BOXSPACE declaration tells Free Menu to leave an extra four dots of room between the edge of the group (ie the box around the group) and the items in the group.

The first column of this group is a Collection of NWAY items:

```
((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
 (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
 (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
```

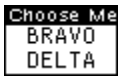
The three items, labelled A, B, and C are all declared as NWAY items, and are also specified to belong to the same NWAY Collection, Col1. This is how a number of NWAY items are collected together. The property NWAYPROPS (DESELECT T) on the first NWAY item specifies that the Col1 Collection is to have the Deselect property enabled. This simply means that the NWAY collection can be put in the state where none of the items (A, B, or C) are selected (highlighted). Additionally, each item is declared with a box whose width is one dot (pixel) around it.

The second column in this nested group is specified by:

```
((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
     INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
 (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35))
```

Column two contains two items, a STATE item and a DISPLAY item. The STATE item is labelled "Choose Me." A Label can be a string or a bitmap, as well as an atom. Selecting the STATE item will cause a pop-up menu to appear with two choices for the state of the item, BRAVO and DELTA. The items to go in the pop-up menu are designated by the MENUITEMS property.

The pop-up menu would look like:



The initial state of the "Choose Me" item is designated to be DELTA by the INITSTATE Item Property. The initial state can be anything; it does not have to be one of the items in the pop-up menu.

Next, the STATE item is Linked to a DISPLAY item, so that the current state of the item will be displayed in the Free Menu. The link's name is DISPLAY (a special link name for

STATE items), and the item linked to is described by the Link Description, (GROUP ALPHA). Normally the linked item can just be described by its ID. But in this case, there is more than one item whose ID is ALPHA (for the sake of this example), specifically the first item in the fourth row and the display item in this nested group. The form (GROUP ALPHA) tells Free Menu to search for an item whose ID is ALPHA, limiting the search to the items that are within this lexical group. The lexical group is the smallest group that is declared with the GROUP keyword (i.e., not row and column groups) that contains this item declaration. So in this case, Free Menu will link the STATE item to the DISPLAY item, rather than the first item in the fourth row, since *that* item is outside of the nested group. For further discussion of linking items, see Section 28.7.12, Free Menu Item Links.

Now, establish the DISPLAY item:

```
(TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)
```

We have given it the ID of Alpha that the above STATE item uses in finding the proper DISPLAY item to link to. This display item is used to display the current state of the item "Choose Me." Every item is required to have a Label property specified, but the label for this DISPLAY item will depend on the state of "Choose Me." That is, when the state of the "Choose Me" item is changed from DELTA to BRAVO, the label of the DISPLAY item will also change. The null string serves to hold the place for the changeable label.

A box is specified for this item. Since the label is the empty string, Free Menu would draw a very small box. Instead, the MAXWIDTH property indicates that the label, whatever it becomes, will be limited to a stringwidth of 35. The width restriction of 35 was chosen because it is big enough for each of the possible labels for this display item. So Free Menu draws the box big enough to enclose any item within this width restriction.

Finally we specify the final item in row four:

```
(LABEL THREE)
```

Free Menu Group Properties

Each group has properties. Most group properties are relevant and should be set in the group's PROPS list in the Free Menu Description. User properties can be freely included in the PROPS list. A few other properties are set up by the formatter. The macros FM.GROUPPROP or FM.MENUPROP allow access to group properties after the Free Menu is created.

ID — The identifier of this group. Setting the group ID is desirable, for example, if the application needs to get handles on items in particular groups, or access group properties.

FORMAT — One of ROW, COLUMN, TABLE, or EXPLICIT. The default is ROW.

FONT — A font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type. This will be the default font for each item

in this group. The default font of the top group is the value of the variable DEFAULTFONT.

COORDINATES — One of GROUP or MENU. This property applies only to EXPLICIT formatting. If GROUP, the items in the EXPLICIT group are positioned in coordinates relative to the lower left corner of the group, as determined by the mother group. If MENU, which is the default, the items are positioned relative to the lower left corner of the menu.

LEFT — Specifies a left offset for this group, pushing the group to the right.

BOTTOM — Specifies a bottom offset for this group, pushing the group up.

ROWSPACE — Specifies the number of dots between rows in this group.

COLUMNSPACE — Specifies the number of dots between columns in this group.

BOX — Specifies the number of dots in the box around this group of items.

BOXSHADE — Specifies the shade of the box.

BOXSPACE — Specifies the number of bits between the box and the items.

BACKGROUND — The background shade of this group. Nested groups inherit this background shade, but items in this group and nested groups do not. This is because, in general, it is difficult to read text on a background, so items appear on a white background by default. This can be overridden by the BACKGROUND Item Property.

Other Group Properties

The following group properties are set up and maintained by Free Menu. The application should probably not change any of these properties.

ITEMS — A list of the items in the group.

REGION — The region that is the extent of the items in the group.

MOTHER — The ID of the group that is the mother of this group.

DAUGHTERS — A list of ID of groups which are daughters to this group.

Free Menu Items

Each Free Menu Item is stored as an instance of the data type FREEMENUITEM. Free Menu Items can be thought of as objects, each item having its own particular properties, such as its type, label, and mouse event functions. A number of useful item types, described in Section 28.7.11, Predefined Item Types, are predefined by Free Menu. New types of items can be defined by the application, using

Display items as a base. Each Free Menu Item is created from a Free Menu Item Description when the Free Menu is created.

CAUTION: Edit (and thus Number) Freemenu Items do not perform well when boxed or when there is another item to the right in the same row. The display to the right of the edit item may be corrupted under editing and fm.changelabel operations.

Free Menu Item Descriptions

A Free Menu Item Description is a list in property list format, specifying the properties of the item. For example:

```
(LABEL Refetch SELECTEDFN MY.REFETCHFN)
```

describes a MOMENTARY item labelled Refetch, with the function MY.REFETCHFN to be called when the item is selected. None of the property values in an item description are evaluated. When constructing Free Menu descriptions that incorporate evaluated expressions (for example labels that are bitmaps) it is helpful to use the backquote facility. For instance, if the value of the variable MYBITMAP is a bitmap, then

```
(FREEMENU `(((LABEL A) (LABEL ,MYBITMAP))))
```

would create a Free Menu of one row, with two items in that row, the second of which has the value of MYBITMAP as its label.

Free Menu Item Properties

The following Free Menu Item Properties can be set in the Item Description. Any other properties given in an Item Description will be treated as user properties, and will be saved on the USERDATA property of the item.

TYPE The type of the item. Choose from one of the Free Menu Item type keywords MOMENTARY, TOGGLE, 3STATE, STATE, NWAY, EDITSTART, EDIT, NUMBER, or DISPLAY. The default is MOMENTARY.

LABEL An atom, string, or bitmap. Bitmaps are always copied, so that the original will not be changed. This property must be specified for every item.

FONT The font in which the item appears. The default is the font specified for the group containing this item. Can be a font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type.

ID May be used to specify a unique identifier for this item, but is not necessary.

LEFT and BOTTOM	When ROW, COLUMN, or TABLE formatting, these specify offsets, pushing the item right and up, respectively, from where the formatter would have put the item. In EXPLICIT formatting, these are the actual coordinates of the item, in the coordinate system given by the group's COORDINATES property.
HJUSTIFY	Indicates horizontal justification type: LEFT, CENTER, or RIGHT. Specifies that this item is to be horizontally justified within the extent of its group. Note that the main group, as opposed to the smaller row or column group, is used.
VJUSTIFY	Specifies that this item is to be vertically justified . Values are TOP, MIDDLE, or BOTTOM.
HIGHLIGHT	Specifies the highlighted looks of the item, that is, how the item changes when a mouse event occurs on it. See Section 28.7.12, Free Menu Item Highlighting, for more details on highlighting.
MESSAGE	Specifies a string that will be printed in the prompt window after a mouse cursor selects this item for MENUHELDWAIT milliseconds. Or, if an atom, treated as a function to get the message. The function is passed three arguments, ITEM, WINDOW, and BUTTONS, and should return a string. The default is a message appropriate to the type of the item.
INITSTATE	Specifies the initial state of the item. This is only appropriate to TOGGLE, 3STATE, and STATE items.
MAXWIDTH	Specifies the width allowed for this item. The formatter will leave enough space after the item for the item to grow to this width without collisions.
MAXHEIGHT	Similar to MAXWIDTH, but in the vertical dimension.
BOX	Specifies the number of bits in the box around this item. Boxes are made around MAXWIDTH and MAXHEIGHT dimensions. If unspecified, no box is drawn.
BOXSHADE	Specifies the shade that the box is drawn in. The default is BLACKSHADE.
BOXSPACE	Specifies the number of bits between the box and the label. The default is one bit.
BACKGROUND	Specifies the background shade on which the item appears. The default is WHITESHADE, regardless of the group's background.
LINKS	Can be used to link this item to other items in the Free Menu.

Mouse Properties

The following properties provide a way for application functions to be called under certain mouse events. These functions are called with the ITEM, the WINDOW, and the BUTTONS passed as arguments. These application functions do not interfere with any Free Menu system functions that take care of handling the different item types. In each case, though, the application function is called

after the system function. The default for all of these functions is NILL. The value of each of the following properties can be the name of a function, or a lambda expression.

SELECTEDFN — Specifies the function to be called when this item is selected. The Edit and EditStart items cannot have a SELECTEDFN. See the Edit Free Menu item description in Section 28.7.11, Predefined Item Types, for more information.

DOWNFN — Specifies the function to be called when the item is selected with the mouse cursor.

HELDFN — Specifies the function to be called repeatedly when the item is selected with the mouse cursor.

MOVEDFN — Specifies the function to be called when the mouse cursor moves off this item (mouse buttons are still depressed).

System Properties

The following Free Menu Item properties are set and maintained by Free Menu. The application should probably not change these properties directly.

GROUPID — Specifies the ID of the smallest group that the item is in. For example, in a row formatted group, the item's GROUPID will be set to the ID of the row that the item is in, not the ID of the whole group.

STATE — Specifies the current state of TOGGLE, 3STATE, or STATE items. The state of an NWAY item behaves like that of a toggle item.

BITMAP — Specifies the bitmap from which the item is displayed.

REGION — Specifies the region of the item, in window coordinates. This is used for locating the display position, as well as determining the mouse sensitive region of the item.

MAXREGION — Specifies the maximum region the item may occupy, determined by the MAXWIDTH and MAXHEIGHT properties (see Section 28.7.8, Free Menu item Properties). This is used by the formatter and the display routines.

SYSDOWNFN

SYSMOVEDFN

SYSSELECTEDFN — These are the system mouse event functions, set up by Free Menu according to the item type. These functions are called before the mouse event functions, and are used to implement highlighting, state changes, editing, etc.

USERDATA — Specifies how any other properties are stored on this list in property list format. This list should probably not need to be manipulated directly.

Predefined Item Types

MOMENTARY [Free Menu Item]

MOMENTARY items are like command buttons. When the button is selected, its associated function is called.

TOGGLE [Free Menu Item]

Toggle items are simple two-state buttons. When pressed, the button is highlighted; it stays that way until pressed again. The states of a toggle button are T and NIL; the initial state is NIL.

3STATE [Free Menu Item]

3STATE items rotate through NIL, T, and OFF, states each time they are pressed. The default looks of the OFF state are with a diagonal line through the button, while T is highlighted, and NIL is normal. The default initial state is NIL.

The following Item Property applies to 3STATE items:

OFF Specifies the looks of a 3STATE item in its OFF state. Similar to HIGHLIGHT. The default is that the label gets a diagonal slash through it.

NOTE: If you specify special highlighting (a different bitmap of string) for Toggle or 3State items AND use this item in a group formatted as a Column or a Table, the highlight looks of the item may not appear in the correct place.

STATE [Free Menu Item]

STATE items are general multiple state items. The following Item Property determines how the item changes state:

CHANGESTATE This Item Property can be changed at any time to change the effect of the item. If a MENU data type, this menu pops up when the item is selected, and the user can select the new state. Otherwise, if this property is given, it is treated as a function name, which is passed three arguments, ITEM, WINDOW, and BUTTONS. This function can do whatever it wants, and is expected to return the new state (an atom, string, or bitmap), or NIL, indicating the state should not change. The state of the item can automatically be indicated in the Free Menu, by setting up a DISPLAY link to a DISPLAY item in the menu (see Section 28.7.13, Free Menu Item Links). If such a link exists, the label of the DISPLAY item will be changed to the new state. The possible states are not restricted at all, with the exception of selections from a pop-up menu. The state can be changed to any atom, string, or bitmap, manually via FM.CHANGESTATE.

The following Item Properties are relevant to STATE items when building a Free Menu:

MENUITEMS If specified, should be a list of items to go in a pop-up menu for this item. Free Menu will build the menu and save it as the CHANGESTATE property of the item.

MENUFONT The font of the items in the pop-up menu.

MENUTITLE The title of the pop-up menu. The default title is the label of the STATE item.

NWAY [Free Menu Item]

NWAY items provide a way to collect any number of items together, in any format within the Free Menu. Only one item from each Collection can be selected at a time, and that item is highlighted to indicate this. The following Item Properties are particular to NWAY items:

COLLECTION An identifier that specifies which NWAY Collection this item belongs to.

NWAYPROPS A property list of information to be associated with this collection. This property is only noticed in the Free Menu Description on the first item in a COLLECTION. NWAY Collections are formed by creating a number of NWAY items with the same COLLECTION property. Each NWAY item acts individually as a Toggle item, and can have its own mouse event functions. Each NWAY Collection itself has properties, its state for instance. After the Free Menu is created, these Collection properties can be accessed by the macro FM.NWAYPROPS. Note that NWAY Collections are different from Free Menu Groups. There are three NWAY Collection properties that Free Menu looks at:

DESELECT If given, specifies that the Collection can be deselected, yielding a state in which no item in the Collection is selected. When this property is set, the Collection can be deselected by selecting any item in the Collection and pressing the right mouse button .

STATE The current state of the Collection, which is the actual item selected.

INITSTATE Specifies the initial state of the Collection. The value of this property is an Item Link Description

EDIT [Free Menu Item]

EDIT items are textual items that can be edited. The label for an EDIT item cannot be a bitmap. When the item is selected an edit caret appears at that cursor position within the item, allowing insertion and deletion of characters at that point. If selected with the right mouse button, the item is cleared before editing starts. While editing, the left mouse button moves the caret to a new position within the item. The right mouse button deletes from the caret to the cursor. CONTROL-W deletes the previous word. Editing is stopped when another item is selected, when the user moves the cursor into another TTY window and clicks the cursor, or when the Free Menu function FM.ENDEDIT is called (called when the Free Menu is reset, or the window is closed). The Free Menu editor will time out after about a minute, returning automatically. Because of the many ways in which editing can terminate, EDIT items are not allowed to have a SELECTEDFN, as it is not clear when this function should be called. Each EDIT item should have an ID specified, which is used when getting the state of the Free Menu, since the string being edited is defined as the state of the item, and thus cannot distinguish edit items. The following Item Properties are specific to EDIT items.

MAXWIDTH Specifies the maximum string width of the item, in bits, after which input will be ignored. If MAXWIDTH is not specified, the items becomes infinitely wide and input is never restricted.

INFINITEWIDTH This property is set automatically when MAXWIDTH is not specified. This tells Free Menu that the item has no right end, so that the item becomes mouse sensitive from its left edge to the right edge of the window, within the vertical space of the item.

In Medley, Changestate of an infinite width Edit item to a smaller item clears the old item properly.

LIMITCHARS The input characters allowed can be restricted in two ways: If this item property is a list, it is treated as a list of legal characters; any character not in the list will be ignored. If it is an atom, it is treated as the name of a test predicate, which is passed three arguments, ITEM, WINDOW, and CHARACTER, when each character is typed. This predicate should return T if the character is legal, NIL otherwise. The LIMITCHARS function can also call FM.ENDEDIT to force the editor to terminate, or FM.SKIPNEXT, to cause the editor to jump to the next edit item in the menu.

ECHOCHAR This item property can be set to any character. This character will be echoed in the window, regardless of what character is typed. However the item's label contains the actual string typed. This is useful for operations like password prompting. If ECHOCHAR is used, the font of the item must be fixed pitch. Unrestricted EDIT items should not have other items to their right in the menu, as they will be replaced. If the item is boxed, input is restricted to what will fit in the box. Typing off the edge of the window will cause the window to scroll appropriately. Control characters can be edited, including the carriage return and line feed, and they are echoed as a black box. While editing, the Skip/Next key ends editing the current item, and starts editing the next EDIT item in the Free Menu.

NUMBER [Free Menu Item]

NUMBER items are EDIT items that are restricted to numerals. The state of the item is coerced to the the number itself, not a string of numerals. There is one NUMBER- specific Item Property:

NUMBERTYPE If FLOATP (or FLOAT), then decimals are accepted. Otherwise only whole numbers can be edited.

EDITSTART [Free Menu Item]

EDITSTART items serve the purpose of starting editing on another item when they are selected. The associated Edit item is linked to the EditStart item by an EDIT link (see Free Menu Item Links below). If the EDITSTART item is selected with the right mouse button, the Edit item is cleared before editing is started. Similar to EDIT items, EDITSTART items cannot have a SELECTEDFN, as it is not clear when the associated editing will terminate.

In Medley, EDITSTART items linked to a Number item properly set number state when editing has completed.

DISPLAY [Free Menu Item]

DISPLAY items serve two purposes. First, they simply provide a way of putting dummy text in a Free Menu, which does nothing when selected. The item's label can be changed, though. Secondly, DISPLAY items can be used as the base for new item types. The application can create new item types by specifying DOWNFN, HELDFN, MOVEDFN, and SELECTEDFN for a DISPLAY item, making it behave as desired.

Free Menu Item Highlighting

Each Free Menu Item can specify how it wants to be highlighted. First of all, if the item does not specify a HIGHLIGHT property, there are two default highlights. If the item is not boxed, the label is simply inverted, as in normal menus. If the item is boxed, it is highlighted in the shade of the box. Alternatively, the value of the HIGHLIGHT property can be a SHADE, which will be painted on top of the item when a mouse event occurs on it. Or the HIGHLIGHT property can be an alternate label, which can be an atom, string or bitmap. If the highlight label is a different size than the item label, the formatter will leave enough space for the larger of the two. In all of these cases, the looks of the highlighted item are determined when the Free Menu is built, and a bitmap of the item with these looks is created. This bitmap is stored on the item's HIGHLIGHT property, and simply displayed when a mouse event occurs. The value of the highlight property in the Item Description is copied to the USERDATA list, in case it is needed later for a label change.

Free Menu Item Links

Links between items are useful for grouping items in abstract ways. In particular, links are used for associating EDITSTART items with their item to edit, and STATE items with their state display. The Free Menu Item property LINKS is a property list, where the value of each Link Name property is a pointer to another item. In the Item Description, the value of the LINK property should be a property list as above. The value of each Link Name property is a Link Description. A Link Description can be one of the following forms:

<ID> An ID of an item in the Free Menu. This is acceptable if items can be distinguished by ID alone.

(<GROUPID> <ID>) A list whose first element is a GROUPID, and whose second element is the ID of an item in that group. This way items with similar purposes, and thus similar ID's, can be distinguished across groups.

(GROUP <ID>) A list whose first element is the keyword GROUP, and whose second element is an item ID. This form describes an item with ID, in the same group that this item is in. This way you do not need to know the GROUPID, just which group it is in.

Then after the entire menu is built, the links are set up, turning the Link Descriptions into actual pointers to Free Menu Items. There is no reason why circular Item Links cannot be created, although such a link would probably not be very useful. If circular links are created, the Free Menu will not be garbage collected after it is not longer being used. The application is responsible for breaking any such links that it creates.

Free Menu Window Properties

FM.PROMPTWINDOW — Specifies the window that Free Menu should use for displaying the item's messages. If not specified, PROMPTWINDOW is used.

FM.BACKGROUND — The background shade of the entire Free Menu. This property can be set automatically by specifying a BACKGROUND argument to the function FREEMENU. The window border must be 4 or greater when a Free Menu background is used, due to the way the Window System handles window borders.

FM.DONTRESHAPE — Normally, Free Menu will attempt to use empty space in a window by pushing items around to fill the space. When a Free Menu window is reshaped, the items are repositioned in the new shape. This can be disabled by setting the FM.DONTRESHAPE window property.

Free Menu Interface Functions

(**FREEMENU** *DESCRIPTION TITLE BACKGROUND BORDER*) [Function]

Creates a Free Menu from a Free Menu Description, returning the window. This function will return quickly unless new display fonts have to be created.

Accessing Functions

(**FM.GETITEM** *ID GROUP WINDOW*) [Function]

Gets item *ID* in *GROUP* of the Free Menu in *WINDOW*. This function will search the Free Menu for an item whose *ID* property matches, or secondly whose LABEL property matches *ID*. If *GROUP* is NIL, then the entire Free Menu is searched. If no matching item is found, NIL is returned.

(**FM.GETSTATE** *WINDOW*) [Function]

Returns in property list format the ID and current STATE of every NWAY Collection and item in the Free Menu. If an item's or Collection's state is NIL, then it is not included in the list. This provides an easy way of getting the state of the menu all at once. If the state of only one item or Collection is needed, the application can directly access the STATE property of that object using the Accessing Macros described in Section 28.7.20, Free Menu Macros. This function can be called when editing is in progress, in which case it will provide the label of the item being edited at that point.

Changing Free Menus

Many of the following functions operate on Free Menu Items, and thus take the item as an argument. The *ITEM* argument to these functions can be the Free Menu Item itself, or just a reference to the item. In the second case, FM.GETITEM (see Section 28.7.16, Accessing Functions) will be used to find the item in the Free Menu. The reference can be in one of the following forms:

<ID> Specifies the first item in the Free Menu whose ID or LABEL property matches <ID>.

(<GROUPID> <ID>) Specifies the item whose ID or LABEL property matches <ID> within the group specified by <GROUPID>.

(**FM.CHANGELABEL** *ITEM NEWLABEL WINDOW UPDATEFLG*) [Function]

Changes an *ITEM*'s label after the Free Menu has been created. It works for any type of item, and STATE items will remain in their current state. If the window is open, the item will be redisplayed with its new appearance. *NEWLABEL* can be an atom, a string, or a bitmap (except for EDIT items), and will be restricted in size by the MAXWIDTH and MAXHEIGHT Item Properties. If these properties are unspecified, the *ITEM* will be able to grow to any size. *UPDATEFLG* specifies whether or not the regions of the groups in the menu are recalculated to take into account the change of size of this item. The application should not change the label of an EDIT item while it is being edited. The following Item Property is relevant to changing labels:

CHANGELABELUPDATE Exactly like UPDATEFLG except specified on the item, rather than as a function paramater.

(**FM.CHANGESTATE** *X NEWSTATE WINDOW*) [Function]

Programmatically changes the state of items and NWAY Collections. *X* is either an item or a Collection name. For items *NEWSTATE* is a state appropriate to the type of the item. For NWAY Collections, *NEWSTATE* should be the desired item in the Collection, or NIL to deselect. For EDIT and NUMBER items, this function just does a label change. If the window is open, the item will be redisplayed.

(**FM.RESETSTATE** *ITEM WINDOW*) [Function]

Sets an *ITEM* back to its initial state.

(**FM.RESETMENU** *WINDOW*) [Function]

Resets every item in the menu back to its initial state.

(**FM.RESETSHAPE** *WINDOW ALWAYSFLG*) [Function]

Reshapes the *WINDOW* to its full extent, leaving the lower-left corner unmoved. Unless *ALWAYSFLG* is T, the window will only be increased in size as a result of resetting the shape.

(**FM.RESETGROUPS** *WINDOW*) [Function]

Recalculates the extent of each group in the menu, updating group boxes and backgrounds appropriately.

(**FM.HIGHLIGHTITEM** *ITEM WINDOW*) [Function]

Programmatically forces an *ITEM* to be highlighted. This might be useful for *ITEM*s which have a direct effect on other *ITEM*s in the menu. The *ITEM* will be highlighted according to its HIGHLIGHT property, as described in Section 28.7.12, Free Menu Item Highlighting. This highlight is temporary, and will be lost if the *ITEM* is redisplayed, by scrolling for example.

Editor Functions

(**FM.EDITITEM** *ITEM WINDOW CLEARFLG*) [Function]

Starts editing an EDIT or NUMBER *ITEM* at the beginning of the *ITEM*, as long as the *WINDOW* is open. This function will most likely be useful for starting editing of an *ITEM* that is currently the null string. If *CLEARFLG* is set, the *ITEM* is cleared first.

(**FM.SKIPNEXT** *WINDOW CLEARFLG*) [Function]

Causes the editor to jump to the beginning of the next EDIT item in the Free Menu. If *CLEARFLG* is set, then the next item will be cleared first. If there is not another EDIT item in the menu, this function will simply cause editing to stop. If this function is called when editing is not in progress, editing will begin on the first EDIT item in the menu. This function can be called from any process, and can also be called from inside the editor, in a LIMITCHARS function.

(**FM.ENDEDIT** *WINDOW WAITFLG*) [Function]

Stops any editing going on in *WINDOW*. If *WAITFLG* is T, then block until the editor has completely finished. This function can be called from another process, or from a LIMITCHARS function.

(**FM.EDITP** *WINDOW*) [Function]

If an item is in the process of being edited in the Free Menu *WINDOW*, that item is returned. Otherwise, NIL is returned.

Miscellaneous Functions

(**FM.REDISPLAYMENU** *WINDOW*) [Function]

Redisplays the entire Free Menu in its *WINDOW*, if the *WINDOW* is open.

(**FM.REDISPLAYITEM** *ITEM WINDOW*) [Function]

Redisplays a particular Free Menu *ITEM* in its *WINDOW*, if the *WINDOW* is open.

(**FM.SHADE** *X SHADE WINDOW*) [Function]

X can be an item, or a group ID. *SHADE* is painted on top of the item or group. Note that this is a temporary operation, and will be undone by redisplaying. For more permanent shading, the application may be able to add a REDEDISPLAYFN and SCROLLFN for the window as necessary to update the shading.

(**FM.WHICHITEM** *WINDOW POSorX Y*) [Function]

Locates and identifies an item from its known location within the *WINDOW*. If *WINDOW* is NIL, (WHICHW) is used, and if *POSorX* is NIL, the current cursor location is used.

(**FM.TOPGROUPID** *WINDOW*) [Function]

Returns the ID of the top group of this Free Menu.

Free Menu Macros

These Accessing Macros are provided to allow the application to get and set information in the Free Menu data structures. They are implemented as macros so that the operation will compile into the actual access form, rather than figuring that out at run time.

(**FM.ITEMPROP** *ITEM PROP {VALUE}*) [Macro]

Similar to WINDOWPROP, this macro provides an easy access to the fields of a Free Menu Item. The function FM.GETITEM gets the *ITEM*, described in Section 28.7.16, Accessing Function. *VALUE* is optional, and if not given, the current value of the *PROP* property will be returned. If *VALUE* is given, it will be used as the new value for that *PROP*, and the old value will be returned. When a call to FM.ITEMPROP is compiled, if the *PROP* is known (quoted in the calling form), the macro figures out what field to access, and the appropriate Data Type access form is compiled. However, if the *PROP* is not known at compile time, the function FM.ITEMPROP, which goes through the necessary property selection at run time, is compiled. The TYPE and USERDATA properties of a Free Menu Item are Read Only, and an error will result from trying to change the value of one of these properties.

(**FM.GROUPPROP** *WINDOW GROUP PROP {VALUE}*) [Macro]

Provides access to the Group Properties set up in the *PROPS* list for each group in the Free Menu Description. *GROUP* specifies the ID of the desired group, and *PROP* the name of the desired property. If *VALUE* is specified, it will become the new value of the property, and the old value will be returned. Otherwise, the current value is returned.

(**FM.MENUPROP** *WINDOW PROP {VALUE}*) [Macro]

Provides access to the group properties of the top-most group in the Free Menu, that is to say, the entire menu. This provides an easy way for the application to attach properties to the menu as a whole, as well as access the Group Properties for the entire menu.

(**FM.NWAYPROP** *WINDOW COLLECTION PROP {VALUE}*) [Macro]

This macro works just like FM.GROUPPROP, except it provides access to the NWay Collections.

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 in the Window Operations and Attached Windows section below). 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 the Window Operations and Attached Windows section below). 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 don e 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 below) 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 below) 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).

(**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).

(**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).

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. 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.

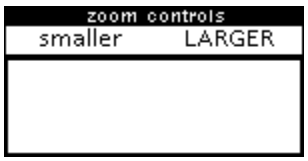
Examples:

```
(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.

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.

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 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.

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).

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 SHRINKFNs 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.

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.

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.

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 is the preferred way to access this property.

ATTACHEDWINDOWS [Window Property]

Pointer from a window to its attached windows. The function ATTACHEDWINDOWS 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.

Communication of Window Menu Commands between Attached Windows is dependent on the name of function used to implement the window command, e.g., CLOSEW implements CLOSE (refer to PASSTOMAINCOMS documentation under Attached Windows). Consequently, if an application intercepts a window command by changing WHENSELECTEDFN for an item in the WindowMenu (for example, to advise the application that a window is being closed), windows may not behave correctly when attached to other windows.

To get around this problem, the Medley release provides the variable *attached-window-command-synonyms*. This variable is an alist, where each element is of the form (new-command-function-name . old-command-function-name).

For example, if an application redefines the WindowMenu to call my-close-window when CLOSE is selected, that application should:

```
(cl:push '(my-close-window . il:closew) il:*attached-window-command-synonyms*)
```

in order to tell the attached window system that my-close-window is a synonym function for CLOSEW.