LISP Machine WINDOW SYSTEM MANUAL

Stallman, Moon, & Weinreb

LISP Machine

Window System Manual

Richard M. Stallman
David Moon
Daniel Weinreb

M.I.T. Artificial Intelligence Laboratory

Lisp Machine Window System Manual

Edition 1.1, System Version 95

August 1983

Richard Stallman
Daniel Weinreb
David Moon

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-80-C-0505.

Summary Table of Contents

1. Concepts 3
2. Visibility and Exposure of Windows. 10
3. Selection 31
4. Sizes and Positions 43
5. Input. 49
6. Output of Text 66
7. Fonts. 83
8. Drawing Graphics 93
9. Blinkers 103
10. The Mouse 112
11. Margins, Borders, and Labels 129
12. Frames 141
13. Miscellaneous Features 157
14. Choice Facilities. 173
15. Typeout Windows. 212
16. Text Scroll Windows. 219
17. General Scroll Windows. 228
Concept Index. 240
Operation Index. 242
Keyword Index 249
Flavor and Resource Index. 254
Variable Index. 256
Function Index 259

Table of Contents

1. Concepts . . . 3
1.1 Windows . . . 3
1.2 Capabilities of Windows . . . 4
1.3 Higher Level Window Facilities . . . 6
1.4 Windows as Flavor Instances . . . 6
1.5 Using a Window . . . 8
1.6 Creation of Windows . . . 9

2. Visibility and Exposure of Windows . . . 10
2.1 Hierarchy of Windows . . . 10
2.2 Screens . . . 13
2.3 Pixels . . . 14
2.4 Bit-Save Arrays . . . 15
2.5 Screen Arrays and Exposure . . . 17
2.6 Ability to Output . . . 21
2.7 Window Locking . . . 23
2.8 Temporary Windows . . . 24
2.9 The Screen Manager . . . 26
2.9.1 Control of Partial Visibility . . . 27
2.9.2 Priority among Windows for Exposure . . . 28
2.9.3 Delaying Screen Management . . . 30

3. Selection . . . 31
3.1 How Programs Select Windows . . . 32
3.2 Teams of Windows . . . 34
3.2.1 The System Menu Select Option . . . 35
3.2.2 Selection with Terminal and System Commands . . . 36
3.3 Selection Substitutes . . . 37
3.3.1 Non-Hierarchical Selection Substitutes . . . 39
3.4 The Status of a Window . . . 39
3.5 Windows and Processes . . . 40

4. Sizes and Positions . . . 43
4.1 Init Options for Sizes and Positions . . . 43
4.2 Flavor Operations for Sizes and Positions . . . 45
4.3 Low Level Edges Functions . . . 47

5. Input . . . 49
5.1 Input Buffers . . . 50
5.2 Blips . . . 52
5.3 Stream Input Operations . . . 52
5.4 I/O Buffers . . . 56
5.4.1 I/O Buffers and Type Ahead . . . 58
5.4.2 I/O Buffers as Input Buffers . . . 58
5.5 Intercepted Characters . . . 59
5.5.1 Synchronously Intercepted Characters . . . 60

5.5.2 Asynchronously Intercepted Characters 61
5.5.3 Global Asynchronous Characters 63
5.6 Polling The Keyboard Explicitly 65

6. Output of Text 66
6.1 How A Character Is Printed. 68
6.2 Stream Output Operations 68
6.3 Output Exceptions 70
6.3.1 Output Hold and End of Page Exceptions 71
6.3.2 **MORE** Exceptions 71
6.3.3 End of Line Exceptions 73
6.4 Cursor Motion 74
6.5 Erasing. 75
6.6 Inserting and Deleting Lines and Characters. 76
6.7 Anticipating the Effect of Output 77
6.8 Explicit (Non-Cursor) Output. 79
6.9 Window Parameters Affecting Output. 80

7. Fonts. 83
7.1 Specifying Fonts 83
7.1.1 Font Specifiers. 85
7.2 Attributes of Fonts 87
7.3 Format of Fonts 89
7.4 Color Fonts 92

8. Drawing Graphics 93
8.1 Alu Functions 93
8.2 Flavor Operations for Graphics. 94
8.3 Low-Level Graphics Using Subprimitives. 98
8.3.1 Subprimitives for Drawing. 101

9. Blinkers 103
9.1 Blinker Functions and Operations. 104
9.2 Blinker Flavors. 107

10. The Mouse 112
10.1 Encoding Mouse Clicks as Characters 113
10.2 Ownership of the Mouse. 114
10.2.1 Grabbing the Mouse 115
10.2.2 Usurping the Mouse 118
10.3 How Windows Handle the Mouse 119
10.4 Mouse Blinkers 121
10.4.1 Reusable Mouse Blinker Types 123
10.5 Mouse Scrolling. 123
10.5.1 Scrolling Protocol 124
10.5.2 Scroll Bars 125
10.5.3 Margin Scrolling 126
10.6 Mouse Parameters. 128

11. Margins, Borders, and Labels . . . 129
11.1 Borders . . . 130
11.2 Labels . . . 132
11.3 Margin Regions . . . 134
11.3.1 Margin Region Example . . . 136
11.4 Defining Margin Item Flavors . . . 138

12. Frames . . . 141
12.1 Constraint Frames . . . 142
12.1.1 Constraint Frame Flavors . . . 142
12.1.2 Examples of Specifications of Panes and Constraints . . . 143
12.1.3 Specifying Panes and Constraints . . . 146
12.1.4 Constraint Frame Operations . . . 153
12.2 Pane-Frame Interaction . . . 154
12.2.1 The Selected Pane . . . 155

13. Miscellaneous Features . . . 157
13.1 Notifications . . . 157
13.2 Lisp Listeners . . . 159
13.3 Editor Windows . . . 160
13.4 Window Flavors for Other Programs . . . 162
13.5 The Who Line . . . 163
13.6 The Color Screen . . . 165
13.6.1 Color Map Functions . . . 165
13.6.2 Operating on Pixels . . . 166
13.7 The System Menu . . . 167
13.8 Window Resources . . . 168
13.9 The Cold Load Stream . . . 170
13.10 The Window-Based Debugger . . . 171

14. Choice Facilities . . . 173
14.1 Menus . . . 173
14.1.1 Menu Items . . . 174
14.1.2 Easy Menu Interface . . . 177
14.1.3 Geometry . . . 178
14.1.4 Ordinary Menus . . . 181
14.1.5 Command Menus . . . 184
14.1.6 Dynamic Item List Menus . . . 185
14.1.7 Multiple Menus . . . 187
14.2 Multiple Choice Facility . . . 190
14.2.1 Functional Interface . . . 191
14.2.2 Flavors and Operations . . . 192
14.3 Choose-Variable-Values Facility . . . 194
14.3.1 Specifying the Variables . . . 194
14.3.2 Predefined Variable Types . . . 195
14.3.3 Functional Interface . . . 196
14.3.4 Defining Your Own Variable Type . . . 199
14.3.5 Making Your Own Window . . . 200

14.3.6 User Option Facility 204
14.4 Mouse-Sensitive Type Out. 207
14.5 Margin Choices 210

15. Typeout Windows. 212
15.1 Activation and Deactivation 213
15.2 Superiors of Typeout Windows 214
15.3 Delaying Redisplay After Typeout. 215

16. Text Scroll Windows. 219
16.1 Specifying the Item List 219
16.2 Bells and Whistles. 221
16.3 Item Generators. 222
16.4 Mouse Sensitive Text Scroll Windows 225

17. General Scroll Windows. 228
17.1 Specifying Items and Entries. 229
17.2 Using a Scroll Window 232
17.3 Inserting and Deleting Items. 234
17.4 Automatically Updating Items. 234
17.5 Representation of Items 237
17.6 Mouse Sensitive Scroll Windows. 238

Concept Index. 240

Operation Index. 242

Keyword Index 249

Flavor and Resource Index. 254

Variable Index. 256

Function Index 259

Preface

The Lisp Machine window system manual is intended to explain how you, as a programmer, can use the set of facilities in the Lisp Machine known collectively as the window system. Specifically, this document explains how to create windows, and what operations can be performed on them. It also explains how you can customize the windows you produce, by mixing together existing flavors to produce a window with the combination of functionality that your program requires and adding daemons to various operations.

It is assumed that you have a working familiarity with Zetalisp as documented in the Lisp Machine manual. It is also assumed that you have some experience with the user interface of the Lisp Machine, including the ways of manipulating windows, such as the Edit Screen, Split Screen, and Create commands from the system menu. To use the predefined flavors and methods, you need not be familiar with how methods are defined and combined, but you should understand what message passing is, how it is used on the Lisp Machine, what a flavor is, what a "mixin" flavor is, and how to define a new flavor by mixing existing flavors. To use the information provided here on where to add daemons, you must be thoroughly familiar with programming with flavors, and must be willing to refer to the window system source code as the final authority for all questions.

Any comments, suggestions, or criticisms will be welcomed. Please send Arpa network mail to BUG-LMMAN@MIT-MC.

Those not on the Arpanet may send U.S. snail to

Richard M. Stallman
545 Technology Square, Room 914
Cambridge, Mass. 02139

Note from Richard Stallman

The Lisp Machine is a product of the efforts of many people too numerous to list here and of the former unique cooperative environment of the M.I.T. Artificial Intelligence Laboratory. I believe that the commercialization of computer software hinders the further development of systems such as described herein. I consider proprietary software morally objectionable and plan to dedicate my career to promoting the sharing and free exchange of software.

Starting in December 1983 I plan to work on the development of GNU, a complete Unix-compatible software system for standard hardware architectures, to be shared freely with everyone just like EMACS. This will enable people to use computers without agreeing to the idea of proprietary software. This project has inspired a growing movement of enthusiastic supporters. If you would like to join it, write to me at the address on the previous page. Help get programmers sharing again! Contributions of part-time programming help will be very welcome, as will funding from philanthropists to support full-time workers, and donations or loans of computers.

The current implementation of the window system is based on flavors, and was designed and implemented primarily by Howard Cannon and Mike McMahon during 1980. It replaced an earlier version implemented by me, which was based on Smalltalk-like classes. The newer version is generally an improvement, but as Howard Cannon steadfastly refused to discuss the design with me I must decline responsibility for such counterintuitive aspects as the definition of exposure.

About a third of this manual is based on earlier documents written by Dave Moon and Daniel Weinreb. Sarah Smith of LMI helped to correct the manual, and Chris Schneider and Steve Strassman provided useful suggestions.

1. Concepts

The term *window system* refers to a large body of software used to manage communications between programs in the Lisp Machine and the user, via the Lisp Machine console. The console consists of a keyboard, a mouse, and one or more screens. All Lisp Machines have at least one high-resolution black-and-white screen, and some machines also have a color screen. The window system can handle any number of screens of various kinds.

The window system controls the keyboard, encoding the shifting keys, interpreting special commands such as the Terminal and System keys, and directing input to the right place. The window system also controls the mouse, tracking it on the screen, interpreting clicks on the buttons, and routing its effects to the right places. The most important part of the window system is its control of the screens, which it subdivides into windows so that many programs can coexist and even run simultaneously without getting in each other's way, sharing the screen space according to a set of established rules.

1.1 Windows

When you use the Lisp Machine, you can run many programs at once. You can have a Lisp listener, an editor, a mail reader, and a network connection program, or several of each, all running at the same time, and you can switch from one to the other conveniently. Interactive programs get input from the keyboard and the mouse, and send output to a screen. Since there is only one keyboard, it can only talk to one program at a time. However, each screen can be divided into regions, and one program can use one region while another uses another region. Furthermore, this division into regions can control which program the mouse talks to; if the mouse cursor position is in a region associated with a certain program, then mouse clicks are directed to that program, which is then allowed to decide what the clicks mean. Allocating access to screen space and input devices is the most important function of the window system.

The regions into which the screen is divided are known as *windows*. In your use of the Lisp Machine, you have encountered windows many times. Sometimes there is only one window visible on the screen; for example, when you cold-boot a Lisp Machine, it initially has only one window showing, and it is the size of the entire screen. If you start using the system menu's Create, Edit Screen, or Split Screen options. you can make windows in various places of various sizes and flavors. Usually windows have a border around them (a thin black rectangle around the edges of the window), and they also frequently have a label in the lower-left hand corner or on top. This is to help the user see where all the windows are, what parts of the screen they are taking up, and what kind of windows they are.

The next several sections begin to explain the detailed concepts of how windows work and what their internal state is. You should probably read over these quickly the first time, without worrying about all the details. You really don't have to understand all of the complexity to make simple use of the window system; it just helps if you know what sort of thing is going on.

1.2 Capabilities of Windows

A window may or may not be exposed, which means that output can be done on it (section 2.5, page 17). At any time at most one window can be *selected*, which means that input can be done through it (chapter 3, page 31). These two conditions constitute the window's *status*.

Another kind of state information that every window has is its edges: its size and its position (see chapter 4, page 43). You can specify these numerically, ask for the user to tell you (using the mouse), ask for a window to be near some point or some other window, and so on.

Windows can function as streams by accepting all the operations that streams accept. If you do input operations on windows, they read from the keyboard; if you do output operations on windows, they type out characters on the screen. The value of terminal-io (see section 21.5.9 of the Lisp Machine manual) is normally a window, and so input/output functions on the Lisp Machine do their I/O to windows by default.

A window whose flavor incorporates tv:stream-mixin supports all the standard input stream operations and may be passed as the input stream to functions such as read and readline (see chapter 5, page 49). Each such window has an *input buffer* holding characters that have been typed at the window but not read yet. You can *force keyboard input* into a window's input buffer; frequently two processes communicate by one process's forcing keyboard input into an input buffer from which another process is reading characters (see page 53).

Any window handles the standard output stream operations and can be passed as the output stream to functions such as print and format (see chapter 6, page 66). You can output characters at a cursor position, move the cursor around, selectively clear parts of the window, insert and delete lines and characters, and so on, by means of standard and not-so-standard stream operations. Output of text on windows provides additional features; for example, characters can be drawn in any of a large set of *fonts* (type faces), and you can switch from one to another within a single window (see chapter 7, page 83). Windows can define their own actions for exceptional conditions that affect output, such as reaching the right or bottom edge of the window, or printing more that a window-full without pausing (see section 6.3, page 70).

In addition to characters from fonts, you can also display graphics (pictures) on windows (see chapter 8, page 93). There are operations to draw lines, circles, triangles, rectangles, arbitrary polygons, circle sectors, and cubic splines.

Each window can have any number of *blinkers* (see chapter 9, page 103). Most windows have one blinker that follows the window's cursor position; this blinker normally appears as a blinking rectangle. But blinkers need not follow the cursor and need not actually blink (some do and some don't). For example, the editor shows you what character the mouse is pointing at; this blinker looks like a hollow rectangle. The arrow that follows the mouse is a blinker, too. Blinkers are used to add visible ornaments to a window, or temporary modifications to a window's normal display. Blinkers are flavor instances with their own standard operations.

Windows are the standard interface to the mouse (see chapter 10, page 112). Both mouse motion and mouse clicks are normally handled by messages sent to the window over which the mouse is positioned.

A window's area of the screen is divided into two parts. Around the edges of the window are the four *margins*; while the margins can have zero size, usually there is a margin on each edge of the window, holding a border and sometimes other things, such as a label. The rest of the window is called the *inside*; regular character output and graphics drawing all occur on the inside part of the window. The margins and inside of the window are managed separately so that mixins to add things to the margins can be independent of the program that draws in the window's inside. See chapter 11, page 129.

For greater flexibility in subdividing a window into multiple areas of different uses, you can create *inferior windows* or *panes* within the window. The main window is then called a *frame*. Each pane can be of a different flavor suitable to its own purpose; thus Peek uses a frame which has a menu and a scrolling window as its two panes. See section 2.1, page 10, for information on the hierarchy of windows, and chapter 12, page 141, for a description of frames.

The *asynchronously intercepted characters* (such as Control-Abort) which take effect instantaneously when typed are handled by the selected window. Each window can specify its own. See section 5.5.2, page 61.

A window can have an associated process. For example, when you type Control-Abort, the process aborted is the one associated with the selected window. Exactly how processes and windows relate depends on the flavor of the window, and, as usual, there are several operations to manipulate the connections. See section 3.5, page 40.

Notifications are a facility for displaying messages from events taking place asynchronously and not related to the program you are running (errors in background processes, qsends from other users, file servers planning to go down, etc.). Notifications work through operations on the selected window, so each window can decide how to display a notification. See section 13.1, page 157.

Screens are represented by flavor objects also; these are not windows, but share some of the operations and instance variables of windows (see section 2.2, page 13). Windows and screens collectively are called *sheets*. Each screen object usually corresponds to a particular piece of display hardware. Screens can be either black-and-white or color. Color screens have more than one bit for each pixel, and most operations on windows do something reasonable on color screens. But the extra bits give you extra flexibility, and so there are some more powerful things you can do to manipulate colors. Color screens also have a *color map* which specifies which values of the pixels display which colors. See section 13.6, page 165.

The *who line* at the bottom of the screen shows the user something about the state of the Lisp Machine. The window system software implements the who line as a separate screen even though it appears on the same TV monitor as the main screen and its windows. This is why you cannot move the mouse into the who line area, or make windows on the main screen hide the who line. See section 13.5, page 163 for more information on the who line and how it is interfaced and implemented.

1.3 Higher Level Window Facilities

The higher level window facilities are window flavors that combine the basic capabilities of windows appropriately to provide directly usable techniques for particular common applications. These facilities include menus and other choice windows, typeout windows, and scrolling windows.

Menus allow the user to choose one or several of a fixed set of items. The system menu that you get from double-click-right is an example of one. See section 14.1, page 173. *Multiple choice* windows allow the user to specify an answer to each of a set of similar multiple-choice questions (see section 14.2, page 190). The editor command Meta-X Kill or Save Buffers shows an example of one.

Choose-variable-values windows allow the user to view and modify the values of a set of variables, each variable printed and read according to its own range of possible values. One variable might allow only numbers, while another variable's value might be restricted to a list of pathnames. See section 14.3, page 194.

Typeout windows allow windows such as scroll windows and editor windows, which normally present displays reflecting permanent data bases, to print output in response to individual commands. The typeout window is an inferior of the other window, and exposes itself when output is done on it.

Scrolling windows allow the programmer to define a display which the user can then scroll through. The scrolling window facility provides for scrolling, redisplay, and interaction with the mouse, requiring the programmer only to specify the entire contents to be scrolled through. There are two types of scrolling windows, *text scroll windows* (chapter 16, page 219) and *general scroll windows* (chapter 17, page 228), the former being less powerful but simpler. Note that there is a standard interface protocol for the mouse to request scrolling (see section 10.5.1, page 124). You need not use one of the standard scrolling window facilities to make a window that can scroll if you are willing to implement the scrolling yourself. For example, editor windows and menus can also scroll.

1.4 Windows as Flavor Instances

In the Lisp world, each window is a flavor instance, an instance of some flavor of window. There are many different window flavors available; some of them are described in this manual. All of them contain the component tv:minimum-window.

tv:minimum-window *Flavor*

The flavor on which all window flavors are built. Any window flavors you define should include this component. This flavor itself is made of the components tv:essential-window, tv:essential-activate, tv:essential-expose, tv:essential-set-edges and tv:essential-mouse. tv:minimum-window has no methods of its own; all are inherited from those components. So you will at times (in the debugger) run across methods of those component flavors. You will also run across methods of tv:sheet, a component of tv:essential-window. However, there is no need for you as a programmer to pay attention to the distinctions among these flavors, and in this manual all the operations, instance variables and init options of tv:minimum-window are documented as being "of

windows" rather than of any specific flavor.

tv:window *Flavor*

This flavor of window has several mixins that provide much generally useful functionality, including the ability to select the window, graphics operations, labels and borders.

```
(defflavor tv:window ()
   (tv:stream-mixin tv:borders-mixin tv:label-mixin
    tv:select-mixin tv:delay-notification-mixin
    tv:graphics-mixin tv:minimum-window))
```

The operations of these mixins are specifically identified in this manual. Use the mixins, or use the flavor tv:window, if you want the operations to be available.

It is often necessary to mix flavors to get the desired window behavior. When doing this, you must pay attention to the correct ordering of flavor components. The earlier components will override later ones. For example, if you want to make a window that will print out notifications on itself by mixing in tv:notification-mixin, you must put it in front of tv:window:

```
(defflavor my-window () (tv:notification-mixin tv:window))
```

If you put them in the other order, as in

```
(defflavor my-window () (tv:window tv:notification-mixin))
```

you get something equivalent to tv:window. The tv:notification-mixin's effect is completely lost. The whole point of tv:notification-mixin is that it should override some methods of tv:window (inherited from tv:delay-notification-mixin), and in fact it defines the same operations in a different way. It follows that if tv:notification-mixin comes last, it will be overridden instead.

It is almost always correct to put mixins first in the ordering so that they will override whatever they are added to. One exception occurs with flavors of margin item; there, the ordering is used to control the spatial position of the margin items.

Screens are also represented by flavor instances, which share some of the characteristics of windows because they share the component flavor tv:sheet. Screens are described fully in section 2.2, page 13.

tv:sheet *Flavor*

tv:sheet is a flavor that windows and screens share. It is also what provides the structure required by the microcode display primitives. Operations defined by this flavor are documented as being "of windows and screens" in this manual.

Much of the contents of this manual is devoted to describing the instance variables and operations of various flavors of window. They are grouped below by functionality.

There is a vague convention sometimes followed for naming flavors of windows. Here the word *frobboz* is used to stand for any feature, attribute, or class of windows that would appear in a flavor name (e.g. peek, lisp-listener, or delayed-redisplay-label). Naming conventions are different for *instantiable* flavors (which are complete and can support instances of themselves) and *mixin* flavors (which are incomplete and only supply one particular aspect of behavior).

frobboz An instantiable flavor whose most distinguishing characteristic is that it is a *frobboz*. *frobboz* is preferred to *frobboz*-window except when it is necessary to make a distinction.

frobboz-mixin A flavor which provides the *frobboz* feature when mixed into other flavors, but is not instantiable. Such mixins often have no components, just :required-flavors.

basic-*frobboz* This form of name is used instead of *frobboz*-mixin when the flavor is regarded as altering the "essential character" of the window. It does not work to mix two "basic" flavors together unless they are designed to work together. In certain cases a basic-*frobboz* may contain tv:minimum-window as a component, and may even be instantiable, but usually it is a mixin that must be mixed with tv:minimum-window and other things in order to work.

essential-*frobboz*

essential-*frobboz*-mixin
A name like this is generally used for a component of *frobboz*-mixin, containing the heart of the *frobboz* facility but not its bells and whistles or its specific interface.

1.5 Using a Window

Many programs never need to create any new windows. Often it suffices to use the standard input, output and graphics operations on an existing window, such as a Lisp listener which is the value of terminal-io when your program is called. For example, here is a graphics demo that will draw a pattern of *xor*ed circles on any window which has tv:graphics-mixin (such as a Lisp listener).

```
(defun green-hornet (&optional (window terminal-io)
                               (separation 40))
 (hacks:with-real-time
  (send window ':clear-screen)
  (send window ':home-down)
  (multiple-value-bind (iw ih)
      (send window ':inside-size)
    (let ((center-x1 (- (truncate iw 2)
                        (truncate separation 2)))
          (center-x2 (+ (truncate iw 2)
                        (truncate separation 2)))
          (center-y (truncate ih 2)))
      (do ((i (- (min center-y center-x1) 10.)
              (1- i)))
          ((<= i 5))
        (send window ':draw-circle
              (if (bit-test 20 i) center-x1 center-x2)
              center-y
              i))))
  (send window ':tyi)
  t))
```

Such programs should try to stick to the most widely-implemented operations. The ideal is to use only the standard stream operations documented in section 21.5 of the Lisp Machine manual;

then your program will run even with streams that are not windows. With graphics programs such as green-hornet you are forced to use some windows-only operations, but it is still best to stick to the operations provided by the flavor tv:window.

1.6 Creation of Windows

When you want to create a flashy and sophisticated user interface, especially involving mouse-sensitivity or automatic updating, it is time to consider creating your own windows (and your own window flavors, perhaps).

To create a window, use the functions make-instance or instantiate-flavor. (Old programs usually use tv:make-window, which is now equivalent to make-instance but was different in the past).

make-instance *flavor-name* &rest *init-options*

Creates, initializes, and returns a new instance of the specified flavor. The *init-options* argument contains alternating keywords and values; the keywords must be *init options* accepted by the flavor you are using. The init options accepted by various window flavors are described in this manual.

Example:

```
(make-instance 'tv:lisp-listener
               ':borders 4
               ':font-map (list fonts:bigfnt)
               ':vsp 6
               ':edges-from ':mouse
               ':expose-p t)
```

creates an exposed Lisp listener with big characters and lots of vertical space between lines.

For more information on this function and on instantiate-flavor, see section 20.7 of the Lisp Machine manual.

tv:sheet-area *Variable*

The area in which windows are by default created.

:name *name* *Init option for windows*

Every window has a name, which is used primarily for printing the window as a Lisp object, but also serves as a default for the window's label. If you do not specify a name, the default is constructed from the flavor name and a counter (each flavor has its own) to make the name unique.

tv:name *Instance variable of windows*

The name of the window.

2. Visibility and Exposure of Windows

The most important piece of information about a window is whether it is actually *visible* on the screen. A related but different piece of information is whether the window is *exposed*. Understanding these basic concepts, the subjects of this chapter, is vital to any use of the window system.

Using the system menu Create option you can make two windows that partially overlap. (If you have never done so, you should try it.) The window system is forced to make a choice here: only one of those two windows can be the rightful owner of that piece of the screen. Of these two windows, only one can be (fully) *visible* at a time; the other one has to be not fully visible, but either partially visible or not visible at all. Only the fully visible window has an area of the screen to use.

If you play around with this, you will see that it looks as if the two windows were two overlapping pieces of paper on a desk, one of which must be on top of the other. Create two Lisp listeners using the Create command of the system menu so that they partially overlap, and then single-click-left on the one that is on the bottom. It will come to the top. Now single-click-left on the other one; it will come back up to the top. The one on top is fully visible, and the other one is not.

2.1 Hierarchy of Windows

Several Lisp Machine system programs and application programs present the user with a window that is split up into several sections, which are usually called "window panes" or "panes". For example, the inspector has six panes in its default configuration: the one you type forms into at the top, the menu, the history list, and the three inspection panes below the first three. The window debugger and ZMail also use elaborate windows with panes. Just as windows on a screen can subdivide the screen, a window's panes subdivide the screen space of the window. With programs such as the editor, inspector and ZMail, it may not be obvious that the windows you see are panes in another window because that window occupies the full screen. If you go into Edit Screen and reshape one of these, you will see clearly how there is a window with subwindows.

In fact, the panes in an inspector are related to the inspector's main window just as that window is related to the screen. Windows are arranged in a hierarchy, each window having a superior and a list of inferiors. Usually the top of the hierarchy is a screen. In the example above, the inspector window is an inferior of the screen, and the panes of the window are inferiors of the inspector window. The screen itself has no superior (if you were to ask for its superior, you would get nil).

A window's superior, its superior's superior, and so on, are collectively called its *ancestors*. A window's inferiors and their inferiors, and so on, are called its *descendants*.

The position of a window is remembered in terms of its relative position with respect to its superior. To figure out where a window is on the screen, we add this relative position to the absolute position of the superior (which is computed the same way, recursively; the recursion

terminates when we finally get to a screen). The important thing about this is that when a superior window is moved, all its inferiors are moved the same amount; they keep their relative position within the superior the same. You can see this if you play with the Move Window command in Edit Screen.

Normally Edit Screen edits the arrangement of the windows on a screen, but it can also edit the arrangement of inferiors (panes) of a window in the same fashion. If you click right on Edit Screen, you get a menu containing all the superiors of the window you pointed at, up to the screen. You can then edit the inferiors of whichever one you choose.

So, what Edit Screen really does is to manipulate a set of inferiors of some specific superior, which may or may not be a screen. The set of inferiors that you are manipulating is called the *active inferiors* set; each inferior in this set is said to be *active*. The active inferiors are all fighting it out for a chance to be visible on their superior. If no two active inferiors overlap, there is no problem; they can all be visible. However, whenever two overlap, only one of them can be on top. Edit Screen lets you change which active inferiors get to be on top. There is also a part of the window system called the *screen manager* whose basic job is to keep this competition straight. For example, it notices that a window that used to be covering up part of a second window has been reshaped, and so the second window is no longer covered and can be made visible. Inactive windows are never visible until they become active; when a window is inactive, it is out of the picture altogether. The screen manager will be discussed at length later (section 2.9, page 26).

Each superior keeps track of all of its active inferiors as a list in the instance variable tv:inferiors, and each inferior window keeps track of its superior, in the instance variable tv:superior. Superior windows do *not* keep track of their inactive inferiors; this is a purposeful design decision, in order to allow unused windows to be reclaimed by the garbage collector. So, when a window is deactivated, the window system doesn't touch it until it is activated again.

:activate *Operation on windows*

Makes the window active in its superior.

:deactivate *Operation on windows*

Makes the window cease to be active in its superior.

:activate-p *t-or-nil* *Init option for windows*

If this option is specified non-nil, the window is activated after it is created. The default is to leave it deactivated.

:kill *Operation on windows*

Killing a window deactivates it but also makes a positive effort to get rid of other entities such as processes or net connections that may be associated with the window. If a window has these things, it may not be satisfactory to just allow the window to be garbage collected; therefore, the :kill operation is provided. A command for the user to get rid of windows should use :kill rather than :deactivate.

:active-p *Operation on windows and screens*
t if this window is active in its superior. A screen is always considered active.

:inferiors *Operation on windows and screens*
Returns this window or screen's list of inferiors.

:superior *Operation on windows and screens*
Returns this window or screen's superior. For a screen, it is nil.

:set-superior *new-superior* *Operation on windows*
Makes this window an inferior of *new-superior*.

:superior *superior* *Init option for windows*
Makes the new window an inferior of *superior*. If this is not specified, the default is tv:mouse-sheet, which is initially the main black-and-white screen.

tv:inferiors *Instance variable of windows and screens*
A list of the active inferiors.

tv:superior *Instance variable of windows and screens*
In a window, the value is the window's superior. In a screen, the value is nil.

tv:sheet-superior *window-or-screen*
tv:sheet-inferiors *window-or-screen*
Accessor defsubsts for the corresponding instance variables.

tv:sheet-me-or-my-kid-p *sheet me*
t if *sheet* is an indirect inferior, zero or more levels down, of the sheet *me*.

tv:map-over-exposed-sheets *function*
Calls *function* on every exposed sheet, starting with the screens, their inferiors, and so on down.

tv:map-over-exposed-sheet *function sheet*
Calls *function* on every exposed inferior of *sheet*, to all levels, including *sheet* itself.

tv:map-over-sheets *function*
Calls *function* on every active sheet, starting with the screens, their inferiors, and so on down.

tv:map-over-sheet *function sheet*
Calls *function* on every active inferior of *sheet*, to all levels, including *sheet* itself.

2.2 Screens

The topmost nodes of the window hierarchy are actually screens rather than windows, a screen being an instance of the flavor tv:screen.

tv:screen *Flavor*

Screens are also flavor instances, whose flavors incorporate tv:screen. Screens are not windows, but they have much in common with windows, because both incorporate the flavor tv:sheet (page 7).

Usually each screen object represents an individual piece of display hardware. However, the main black-and-white physical screen that all Lisp Machines have is logically divided into two screens, with different screen objects. These are tv:main-screen and tv:who-line-screen. Because these are separate screens, windows on the main screen cannot be extended onto the who line, and the mouse cannot move onto the who line, etc.

Screens are the objects that know how to parse font specifiers (user-level names for fonts) into font objects that can be used for display. See page 85. Also, each screen can specify a font for each of the standard font purposes (:default, :label, :menu, etc.). See page 86.

tv:sheet-get-screen *sheet*

Returns the screen that *sheet* is an indirect inferior of (*sheet* itself, if it is a screen).

tv:main-screen *Variable*

The screen object that represents the Lisp Machine black-and-white display, except for the who line area. This is default superior for windows created with tv:make-window.

tv:who-line-screen *Variable*

The screen object that represents the who line area. Each field of the who line is a separate window on this screen.

tv:default-screen *Variable*

This is the screen that is "normally used". It is initialized to be the main screen. Certain functions that create a window without reference to the mouse use it as a default for the superior of the window, and window resources with a superior as parameter often create one window initially, with the default screen as the superior.

color:color-screen *Variable*

This is the color screen for the 4-bit-pixel color display that some Lisp Machines have. The screen object is always present, but is exposed only when the machine actually has a color screen. See section 13.6, page 165.

tv:all-the-screens *Variable*

A list of all screen objects. With this list, you can begin a tree walk to cover all active windows.

tv:set-tv-speed &optional (*speed* 60.5) (*wasted-lines* 0)

Sets the scanning rate of the main screen, in vertical sweeps per second, to *speed*. *speed* is usually a flonum.

The vertical size of the screen is inversely proportional to the number of vertical scans per second, because the display rate in horizontal scan lines per second is fixed.

A nonzero value of *wasted-lines* directs the system to refrain from using that many horizontal scan lines at the bottom of the screen. If you are using MIT software on a machine built by Symbolics, you may need to do this, since the screens are typically misaligned so that the who line is obscured by the screen's cabinet. A value of 20 to 30 generally does the trick.

2.3 Pixels

A screen displays an array of *pixels*. Each pixel is a little dot of some brightness and color; a screen displays a big array of these dots to form a picture. Everything you see on the screen, including borders, graphics, characters, and blinkers, is made up out of pixels.

Each physical screen has a display memory which stores the values of all the pixels. On regular black-and-white screens, each pixel has one of only two values, lit up or not lit up, so the pixel is represented in memory by one bit. Usually 0 is used for the background of a window and the characters or lines on it are made of 1's, so 1 can be considered "on" and 0 "off". On color screens, pixels have more than one bit. The usual sort of color screen has four bits per pixel. 0 is still often used as the background value and assigned the color black. There is no convention for the use of other pixel values.

Black and white screens have a hardware flag that controls the visual appearance of 1 and 0 pixels. In "black-on-white" mode, 1 is dark and 0 is bright, so windows appear with dark text on a white background. This mode is the default. In "white-on-black" mode, 1 is bright and 0 is dark. Users can switch between these modes with Terminal C.

An individual window can specify 1 for background and 0 for text; this is independent of white-on-black mode (which applies to the whole screen) and is requested with the :reverse-video-p init option or the :set-reverse-video-p operation (see page 81). These work by controlling the *alu functions* used for drawing and erasing characters; see section 8.1, page 93. Programs which use the window's recommended alu functions for their drawing and erasing will automatically display in reverse-video when this is specified. The who line mouse documentation window is an example of a window which uses reverse-video.

tv:black-on-white &optional (*screen* tv:default-screen)

Make *screen* display one-bits as black, with zero-bits as white. (This is the default mode.)

Note that this works by setting a bit in the display hardware; as a result, if done on the main screen, it applies to the who line as well.

tv:white-on-black &optional (*screen* tv:default-screen)
Make *screen* display one-bits as white, with zero-bits as black.

tv:complement-bow-mode &optional (*screen* tv:default-screen)
Toggle whether *screen* displays one-bits as white or as black. This is what Terminal C does.

tv:bits-per-pixel *Instance variable of* tv:screen
1 for a black-and-white screen, larger numbers for other kinds (4 for the standard color screen).

tv:buffer *Instance variable of* tv:screen
The address of the screen memory, as a fixnum.

tv:buffer-halfword-array *Instance variable of* tv:screen
An art-16b array containing the screen memory.

tv:control-address *Instance variable of* tv:screen
The address of the screen's control register which contains, among other things, the flag controlling black-on-white mode.

2.4 Bit-Save Arrays

The pixel values that make up a window's screen image are called its *contents*. When a window is fully visible, its contents are displayed on a screen so that they can be seen. When the window is not fully visible, its contents are lost unless there is a place to save them. Such a place is called a *bit-save array*.

A bit-save array is an array of bits of sufficient size to hold a copy of the window's contents. If a window has a bit save array, its contents are copied into the array when the window ceases to be fully visible. If the window becomes fully visible again, the contents are copied from the bit-save array back onto the screen. In the mean time, programs can use tv:sheet-force-access to do output into the bit-save array while the window is not visible (see page 23), and the window's inferiors, if any, can be exposed and do output (see section 2.5, page 17).

When a window with a bit-save array is partially visible, the visible parts can be displayed correctly by copying them from the bit-save array. This is the behavior you observe if you make a small Lisp listener window with Create and have a full-screen window such as the initial Lisp listener or a Zmacs frame partially visible around it. It happens because the Lisp listener or Zmacs frame has a bit-save array.

If a window does not have a bit-save array, then there is no place to put its contents when it is not visible, so they are lost. When the window becomes visible again, it will try to redraw its contents; that is, to regenerate the contents from some state information in the window. This is done by the :refresh operation documented below. Some windows can do this; for example, editor windows can regenerate their contents based on the editor buffers they are displaying. Other windows, such as Lisp listeners, do not remember what was displayed on them and cannot regenerate their previous contents. Such windows just leave their contents blank, except for the margins (see chapter 11, page 129), which all windows can regenerate.

The advantage of having a bit-save array is that losing and regaining visibility does not require the contents to be regenerated; this is desirable since regeneration may be computationally expensive, or even impossible. The disadvantage is that the bit-save array can be large and swapping it in can be slow.

When a frame is in use, giving the frame a bit-save array enables the contents of the frame and all the panes to be preserved if the frame ceases to be fully visible. Bit-save arrays for the panes would come into play only if panes were shuffled or substituted within the frame; in most applications, this happens never or rarely, and is accompanied by a thorough redisplay. So normally the frame gets a bit-save array and the panes do not.

tv:bit-array *Instance variable of windows*

This instance variable of all windows holds the window's bit array, or nil if it has none.

tv:sheet-bit-array *window*

Accessor defsubst for the corresponding instance variable.

:bit-array *Operation on windows*

Returns the window's bit array, or nil if it has none.

:save-bits *Operation on windows*

Returns non-nil if this window saves its bits when not exposed.

:set-save-bits *flag* *Operation on windows*

Tells this window to start or stop saving its bits when not exposed. *flag* is t to start or nil to stop.

:save-bits *flag* *Init option for windows*

flag may be t, nil or :delayed. :delayed causes the window to acquire a bit-save array the first time it is deexposed, but not before.

:refresh &optional (*type* ':complete-redisplay) *Operation on windows*

Restore the saved contents of the window or regenerate the contents, according to the value of *type* (and to whether the window has a bit-save array).

Here are the possible values of *type*:

:complete-redisplay

This is the default. The window's present bit image is completely discarded and regenerated from scratch. The margins are redrawn by invoking :refresh-margins. The default definition of :refresh just leaves the inside blank except for refreshing any exposed inferiors.

If the window has no bit-save array, *type* is ignored and the actions for :complete-redisplay are always used.

:use-old-bits The complete contents are restored from the bit-save array. This is specified by the system when a window is exposed.

:size-changed

This keyword is specified when the window's size has been changed. The

contents are restored from the bit-save array, and then the margins are refreshed with :refresh-margins.

:margins-only This keyword is specified when the inside portion of the window is completely undisturbed, and only the margins need to be refreshed. The system treats it just like :size-changed.

Window flavors ought when possible to provide :after daemons for :refresh, to complete the job of redrawing the window, which the system itself cannot know how to do. When these daemons run, the instance variable tv:restored-bits-p will be non-nil if the window contents were restored from a bit-save array. If this is so, there is no need for the :after daemons to do anything, except perhaps if the window's inside size has changed.

tv:restored-bits-p *Instance variable of windows*

In :after daemons of :refresh (and therefore also of :expose), this is t if the contents were restored from a bit-save array. If it is nil, the inside of the window was left blank and must be regenerated to whatever extent that is possible.

2.5 Screen Arrays and Exposure

This section discusses the concepts of screen arrays and of exposed windows. These have to do with how the system decides where to put a window's contents (its pixels), how the notion of visibility on the screen is extended into a hierarchy of windows, and how programs can control which windows are visible. Do not feel it is your fault if this seems complicated; you do not need to understand it fully on your first reading of the manual.

Each window or screen can have a *screen-array*, which is where output drawn on the window should go. Drawing characters or graphics is done by changing elements of the window's screen array. The screen array is stored in the instance variable tv:screen-array. The variable can also be nil, to say that the window does not have a screen array at the present time.

A screen normally has a screen array that is displaced to the special memory that the screen's hardware displays from. A window that is visible has a screen array; it is an indirect array that points into the area of the superior's screen-array where the inferior gets displayed on the superior. For example, consider a visible window whose superior is a screen and whose upper-left-hand corner is at location (100,100) in the screen. The window's screen-array would be an indirect array whose (0,0) element is the same as the (100,100) element of the screen. If you were to set a pixel in the window's screen-array, the corresponding pixel in the screen (found by adding 100 to each coordinate) would be set to that value.

A visible window more than one level down from the screen has a screen array that indirects more than once. The window's screen-array points into the middle of its superior's screen array, which points into the middle of the superior's superior's screen array, and so on until the screen is reached. When typeout is done on the window, it will appear on the screen, offset by the combined offsets of all the ancestors, so that it will appear in the correct absolute position on the screen.

Sometimes a window is unable to have a screen array that points to its superior's screen array. For now, let's not ask why this might happen, but consider instead what to do about the screen array when this does happen. There are two alternatives. If the window has a bit-save array, then the bit-save array is used as the screen array. If there is no bit-save array, there can be no screen array either. The window's tv:screen-array variable becomes nil, and there is nowhere for output on this window to go.

For a window *w* with a bit-save array, *w*'s inferiors are not affected by where *w*'s screen array points. *w* always has a screen array, and its inferiors' screen arrays can point to that.

But if *w* has no bit-save array, it may have nil instead of a screen array, and in that case it is impossible for *w*'s inferiors to have screen arrays pointing into *w*'s screen array. So they in turn must use their bit-save arrays, if any, as screen arrays, or not have screen arrays. The effect propagates down the hierarchy.

So we see one possible reason why a window may be unable to have a screen array that points into its superior's: if the superior doesn't have a screen array at all. There is one other reason: the superior may deny permission for this window to point its screen array into the superior. The superior has an instance variable tv:exposed-inferiors which record all the inferiors permitted to do this. (Only active inferiors are allowed.) This permission can be granted or revoked at any time, and is called *exposability*. Each window can be made exposable or not exposable using the :expose and :deexpose operations. So, if a window's superior does not have a screen array, or if the window is not exposable, then the window must scrounge up a screen array itself if it can.

A window is said to be *exposed* if it has a screen array that points into its superior's screen array. Note that a window must be exposable in order to be exposed, but the converse is not true. An exposable window is exposed as well if and only if its superior has a screen array.

An exposed window is not necessarily visible. A window is visible if its screen array points, through some number of levels of indirection, into the middle of the screen's screen array. An exposed window's screen array points into the middle of *something*, but that may be a bit-save array in a deexposed ancestor some number of levels up. A window that is exposed but not visible must have some ancestors that are not exposed, and at least one of them must not be exposable either. This diagram of a window w8 and its ancestors shows the pattern of exposed and deexposed windows and how it comes about.

```
s <-- w1 <-- w2 <-- w3 <-- w4 <-- w5 <-- w6 <-- w7 <-- w8
exposable    not!   exposable again...
                                  w5 has a bit-save array
exposed...   deexposed...              exposed...
visible...   invisible...           ...still invisible...
```

Output is allowed on a window whenever the window is exposed. Usually exposed windows are visible and the output can be seen on the screen. But output to an exposed window with a deexposed ancestor is also permitted. Then the output goes into the middle of that ancestor's bit-save array rather than onto the screen. Such output cannot actually be seen. But if the unexposable ancestor that must exist is made exposable, the bit-save array will be copied onto the screen and the output already done will be seen.

Output is not normally allowed on a deexposed window, even if the window has a screen array which is its bit-save array. However, in this case, you can use tv:sheet-force-access to override the prohibition and output onto the bit-save array. Use of :permit as the window's deexposed typeout action (see page 21) allows all output on such windows to proceed and draw in the bit-save array. A deexposed window with no bit-save array cannot have output done on it in any fashion since it has no contents.

The :expose operation makes a window exposable. If at that time its superior has a screen array, the window will become exposed as well. Or, if the superior later acquires a screen array, the window will become exposed then. This can happen if the superior itself is exposed, or if the superior is given a bit-save array with the :set-save-bits operation.

The :deexpose operation always makes the window unexposable and therefore not exposed.

It is possible for a screen to be deexposed. In particular, if a Lisp Machine does not have a color display physically attached to it, there is still a "color screen" Lisp object in the Lisp world, but it is deexposed (and so are any immediate inferiors it may have). This is so saved Lisp environments can be moved easily between machines with different hardware configurations. The screen object is left deexposed so that programs will not try to output to it. The screen is exposed whenever the Lisp Machine system is booted on a machine that actually has a color screen; then all its exposable inferiors become exposed too. For screens, there is no distinction between exposed and exposable, since there is no superior to have a say in the matter.

In order to maintain the model that windows are like pieces of paper on a desk, it is important that no two windows that both occupy some piece of screen space be exposed at the same time. To make sure that this is true, whenever a window becomes exposed, the system deexposes any of its exposed siblings that it overlaps. (Note: this is not true for temporary windows; see page 24.)

:expose *Operation on windows and screens*

&optional *inhibit-blinkers bits-action new-left new-top*

Makes the window exposable, and exposed if possible. This is a very useful operation to attach daemons to, but remember that this operation may be performed on a window that is already exposable. The daemons must not make the assumption that the window is just becoming exposable. If the window is not a direct inferior of the screen, it may not be becoming exposed either.

If the window is not active in its superior, it is first activated.

The arguments to the :expose operation are supplied by the system and usually of interest only to the system's methods. User invocations of this operation should usually supply no arguments.

If the window actually becomes visible, the window's blinkers normally appear with their deselected visibilities. If *inhibit-blinkers* is non-nil, the blinkers are not acted on. If the window is being exposed in order to select it, this is used to save time.

If the window actually becomes visible, *bits-action* controls how it is put back on the screen. It can be :noop, :restore or :clean. If it is :noop, the window's screen area is not touched. This is used only in very unusual cases. If it is :clean, the window is sent a :refresh message with argument :complete-redisplay, which should make the window redraw itself from scratch if it can. If *bits-action* is :restore, the window is sent a :refresh message with argument :use-old-bits, which should make the window copy its bit-save array onto the screen. nil as the *bits-action* is equivalent to :restore for windows with bit-save arrays and to :clean for windows without them.

new-left and *new-top* are the offsets within the superior at which to expose the window. They default to the window's current offsets. These arguments are for use by the :set-edges operation; you should not pass them.

A window cannot be made exposable unless its full size fits within the superior.

:deeexpose *Operation on windows and screens*

&optional *save-bits-p screen-bits-action remove-from-superior*

Makes the window not exposed and not exposable. This is a useful operation to add daemons to.

The arguments to the :deexpose operation are supplied by the system, and are usually of interest only to the system's methods.

save-bits-p defaults to :default. It can also be :force or nil. :default means the bits are saved if the window has a bit-save array. :force gives the window a bit-save array if it doesn't already have one, so that the bits are always saved. nil does not save the bits.

screen-bits-action controls what to do to the bits on the screen. It may be :noop to do nothing to them, or :clean to erase the area occupied by the window.

If *remove-from-superior* is nil, the window remains exposable. You should always use t (which is the default) for this argument. The window system uses nil as part of implementing deexposure of an exposable window whose superior loses its screen array. Use of nil at any other time would lead to incorrect results.

:expose-p *t-or-nil* *Init option for windows*

If this option is specified non-nil, the window is made exposable after it is created. The default is to leave it deexposed. If the value of the option is not t, it is used as the first argument to the :expose operation (the *inhibit-blinkers* argument).

:exposable-p *Operation on windows and screens*

t if the window is exposable.

:exposed-p *Operation on windows and screens*

t if the window is exposed.

:exposed-inferiors *Operation on windows and screens*
Returns a list of all exposable inferiors of this window or screen.

tv:with-sheet-deexposed (*sheet*) &body *body* *Special form*
Executes the *body* with *sheet* deexposed. If *sheet* had been exposed, it is reexposed when *body* exits. Operations that change things about the window often make use of this to reduce the complicated case of an exposed window to the simpler case of a deexposed one.

:screen-array *Operation on windows and screens*
Returns the window or screen's screen array, or **nil**.

tv:exposed-p *Instance variable of windows and screens*
t if the window is exposed.

tv:exposed-inferiors *Instance variable of windows and screens*
A list of all exposable inferiors of this window or screen.

tv:screen-array *Instance variable of windows and screens*
The screen array, or **nil** if there is none.

tv:sheet-exposed-p *window-or-screen*
tv:sheet-exposed-inferiors *window-or-screen*
tv:sheet-screen-array *window-or-screen*
Accessor defsubsts for the corresponding instance variables.

2.6 Ability to Output

Whether a window is exposed usually controls whether output can be done on it. In a deexposed window a flag called the *output hold flag* is normally 1. This causes an *output hold exception* if an attempt is made to output to the window. The normal result of an output hold exception is that the process doing output waits until the output hold flag is clear. The process wait state during this wait is **"Output Hold"**.

The output hold flag is also set in a window that has exposed inferiors, because output on the window would overwrite the inferiors.

tv:sheet-output-hold-flag *window*
1 to indicate an output hold exception, or 0 to permit output on the window. This is setf'able.

When a process attempts to type out on a window which is deexposed and has its output hold flag set, what happens depends on the window's *deexposed typeout action*. The deexposed typeout action can be any of certain keyword symbols, or it can be a list. After the specified action is taken, if the output hold flag is still set, the process will wait for it to clear. The interesting thing is that the action may affect the value of the output hold flag.

tv:deexposed-typeout-action *Instance variable of windows*
The window's deexposed typeout action.

:deexposed-typeout-action *Operation on windows*
:set-deexposed-typeout-action *action* *Operation on windows*
Get or set the window's deexposed typeout action.

:deexposed-typeout-action *action* *Init option for windows*
Initializes a window's deexposed typeout action to *action*.

tv:sheet-deexposed-typeout-action *window*
Accessor defsubst for the instance variable.

Here are the possible values of deexposed typeout action:

:normal This, the default, means "no action". Therefore, the process will always have to wait for the output hold flag to clear.

:expose The action is to send the window an :expose message. This may expose the window (if the superior has a screen-array), and if it does expose the window then the output hold flag will probably be cleared, allowing typeout to proceed immediately. If the superior is the screen, the :expose option provides a very different user interface from the :normal option.

:permit This means to permit typeout even though the window is not exposed, as long as the window has a screen array; i.e., it may type out on its own bit-save array even though it is not exposed. The next time the window is exposed, the updated contents will be retrieved from the bit-save array.

The action for :permit is to turn off the output hold flag if the window has a screen array. This mode has the disadvantage that output can appear on the window without anything being visible to the user, who might never see what is going on and might miss something interesting.

It is possible to request that output in this mode to partially visible windows be transferred to the screen periodically. See page 28.

:notify This means that the user should be notified when there is an attempt to do output on the window. The action taken is to send the :notice message to the window with the argument :output (see page 158). The default response to this is to notify the user that the window wants to type out and to put the window on a list for Terminal 0 S to select it. Supdup and Telnet windows have :notify deexposed typeout action by default.

:error The action is to signal an error.

a list, (*operation arguments...*)
The action is to send the window a message with *operation* and *arguments*.

Functions such as ed, whose purpose is to select a window for the user, should not return immediately. If ed returned immediately, then when called in a Lisp listener with its deexposed typeout action set to :expose, the printing of the value returned by ed would immediately switch

back to the Lisp listener, which defeats the purpose of ed. To avoid this behavior, ed calls tv:await-window-exposure.

tv:await-window-exposure

Wait until terminal-io is exposed (more precisely, until its :await-exposure operation returns).

:await-exposure *Operation on windows*

Does not return until the window is exposed. (Some window flavors implement it differently).

tv:sheet-force-access (*window*) *body* *Special form*

tv:sheet-force-access allows you to do typeout on a window that has a screen array even if its output hold flag is set. It works by turning off the output hold flag temporarily around the execution of the body. This is useful for drawing on a window while it is not visible. For example, changing the menu items of a menu redraws the menu contents immediately even if the menu is not visible; this is because when the menu does become visible it looks better to the user for it to become visible in one instant with the correct contents.

If the window is exposed, tv:sheet-force-access goes ahead and outputs to it. If the window is not exposed but has a bit-save array, the output goes there.

If the window is not exposed and has not bit-save array tv:sheet-force-access doesn't do anything at all; it just returns *without* evaluating its body.

Here is an example: when a text scroll window is given a new item generator, which completely changes the text that it should display, it redisplays the window in its bit-save array if necessary. *dont-prepare-flag* is t because the :clear-screen and :redisplay operations take care of preparing the sheet.

```
(defmethod (tv:text-scroll-window :set-item-generator)
           (new-item-generator)
  (setq item-generator new-item-generator)
  (tv:sheet-force-access (self)
    (send self ':clear-screen)
    (send self ':redisplay 0
          (tv:sheet-number-of-inside-lines))))
```

2.7 Window Locking

Each window or screen has a lock which is used to prevent two processes from operating on the window at once in a way that might cause inconsistent results. Outputting on the window, activating or deactivating the window, exposing or deexposing the window, and changing the window's shape all lock the window. This is done with process-lock, via tv:lock-sheet. Note that the window's inferiors must be locked too.

Another form of locking is called "temp-locking". A window is temp-locked when a temporary window (see page 24) is exposed on top of it. All the operations which lock the

window will have to wait if the window is temp-locked just as they would if the window were locked in the ordinary manner; however, the lock is not considered owned by a process but rather by the temporary windows that overlap the window. It will stay locked until the temporary windows are all deexposed. The :mouse-select operation and some other things know how to deexpose temporary windows when necessary to cause a window to become unlocked.

tv:lock-sheet (*window-or-screen*) &rest *body*... *Special form*

Executes *body* with *window-or-screen* locked by this process. Calls to tv:lock-sheet are found in wrappers for operations such as :expose, so you need not call it yourself, but you should be aware that it is being done.

tv:lock *Instance variable of windows and screens*

The lock. It is nil for an unlocked window, a process that has locked the window, or a list of covering temporary windows if this window is "temp-locked".

tv:lock-count *Instance variable of windows and screens*

The number of times the lock is locked. This counts the number of recursive lockings for the same process, for example.

tv:sheet-lock *window-or-screen*

Returns the contents of *window-or-screen*'s lock. This is a defsubst and can be setf'd. It is usually unmodular to use this.

tv:sheet-can-get-lock *window-or-screen* &optional (*lock-id* current-process)

Returns t if this window or screen could right now be locked by *lock-id*; essentially, if it is free or already locked that way (but in fact it is more complicated than this.)

Note that if you call this function with inhibit-scheduling-flag nil, you are likely to be susceptible to a timing error.

tv:sheet-clear-locks

Unlocks the locks of all active windows. For use in an emergency.

2.8 Temporary Windows

Normally, when a window is exposed in an area of the screen where there are already some other exposed windows, the windows that used to be there are deexposed automatically by the window system. This is because the window system normally doesn't leave two windows both exposed if they overlap. (In the absence of temporary windows, which we are about to introduce, the system never allows two overlapping windows to both be exposed.)

But sometimes there are windows that only get put up on the screen for a very short time. The most obvious examples of such windows are the momentary menus that only appear for long enough for you to select an item. It would be unfortunate if every time a momentary menu appeared, the windows under it had to be deexposed. The ones without bit-save arrays would have their screen image destroyed, forcing them to regenerate it or to reappear empty. The ones with bit-save arrays would not be damaged in this way, but they would have to be deexposed, and deexposure is a relatively expensive operation.

This problem is solved for momentary menus by making them *temporary windows*. Temporary windows work differently from other windows in the following way: when a temporary window is exposed, it saves away the pixels that it covers up. It restores these pixels when it is deexposed. These pixels may come from several different windows. This way it doesn't mess up the area of the screen that it uses, even if it covers up some windows that don't have bit-save arrays.

Also, a temporary window, unlike a normal window, does not deexpose the windows that it covers up. This way the covered windows need not try to save their bits away in their bit-save arrays (if they have them) nor ever have to try to regenerate their contents (if they don't). They never notice that the temporary window was (temporarily) there.

tv:temporary-window-mixin *Flavor*

This mixin makes a window a temporary window.

:temporary-bit-array *Operation on windows*

Non-nil if the window is a temporary window.

There would be some problems if temporary windows were this simple. Suppose there is a normal window, and a temporary window appears over it; some of the contents of the normal window are saved in an array inside the temporary window. Now, if the normal window were moved somewhere else, and possibly became deexposed or overlapped by other windows, and then the temporary window were deexposed, the temporary window would dump back its saved bits where the normal window used to be. This would clobber some other window.

Furthermore, even though normal window is still exposed, output on it must not be permitted, since that could overwrite the temporary window.

Because of problems like these, when a temporary window gets exposed on top of some other windows, all the windows that it covers up (fully or partially) become *temp-locked*. While a window is temp-locked, any attempt to type out on it will wait until it is no longer temp-locked. Furthermore, any attempt to deexpose, deactivate, move, or reposition a temp-locked window will wait until the window is no longer temp-locked. The temp-locking is undone when the temporary window is deexposed.

Because of temp-locking, you should never write a program that will put a temporary window up on the screen for a "long" time. There should be some action by the user, such as moving the mouse, which will make the temporary window deexpose itself. It is best if any attempt by the user to get the system to do something makes the temporary window go away. While the temporary window is in place, it blocks many important window system operations over its area of the screen. The windows it covers cannot be manipulated, and programs that try to manipulate them will end up waiting until the temporary window goes away.

It works fine to have two or more temporary windows exposed at a time. If you expose temporary window *a* and then expose temporary window *b*, and they don't overlap each other, they can be deexposed in either order, and any windows that both of them cover up will be temp-locked until both of them are deexposed. If *b* covers up *a*, then *a* will be temp-locked just like any other window, and so it will not be possible to deexpose *a* until *b* has been deexposed.

2.9 The Screen Manager

Sometimes not all of the screen is in use by fully visible windows. This does not happen in elementary use of the Lisp Machine, since the initial windows in the system are all full-screen-sized, but if you create a small Lisp listener with system menu Create the rest of the screen will be unclaimed by any fully visible window. The part of the window system responsible for dealing with unclaimed parts of the screen is called the *screen manager*.

The screen manager fills such unclaimed areas by looking for deexposed windows which fall entirely or partly within them. Only active immediate inferiors of the screen are considered, and in a specific priority order described in section 2.9.2, page 28.

A window that falls entirely within unclaimed areas can be made visible without deexposing any other windows. This is called *autoexposure*. Since the window is a direct inferior of the screen, exposing it always makes it visible. The screen manager goes on considering the remaining deexposed windows, but with less screen area unclaimed.

A window that overlaps the unclaimed areas but also overlaps a visible window cannot simply be exposed. So it becomes *partially visible*, which means simply that the screen manager copies the appropriate parts of the window's contents onto the unclaimed areas. The window is not treated as visible or exposed in any other sense. This gives the visual impression of overlapping pieces of paper on a desk top; the deexposed window is partially covered up by the visible windows, but you can still see those parts that aren't covered. The contents are copied from the window's bit-save array. Windows without bit-save arrays are by default ineligible for partial visibility, so other windows later in the order will get a chance for the same screen area; however, it is possible to arrange for windows without bit-save arrays to be partially visible (though the displayed contents may not be accurate).

Windows whose size and position are such that they do not fit within the bounds of the superior cannot be exposed, and the screen manager does not try to autoexpose such windows. However, they can be partially visible like any other windows.

The screen manager has one other job. At the same time that it does autoexposure, it can also select a window if there isn't any selected window at the time. This is called *autoselection*. A window is a candidate for autoselection if it is an exposed inferior of the screen and its :name-for-selection is non-nil (see page 35). For more information, see chapter 3, page 31.

The screen manager does not only manage the inferiors of screens; it can manage the inferiors of windows as well. The system invokes the screen manager on a sheet's inferiors by sending the sheet a :screen-manage message. This happens for *all* visible sheets regardless of flavor.

:screen-manage *Operation on windows and screens*

The default definition of this operation is to do autoexposure and display of partially visible windows among the active inferiors of this window or screen, as described above.

tv:no-screen-managing-mixin *Flavor*

Prevents the screen manager from dealing with the inferiors of a window by redefining the :screen-manage operation to do nothing.

When a frame is used by a single program, the program usually expects to have sole control over exposure of panes. Then this mixin can be used to tell the screen manager not to interfere. Constraint frames do not normally need to use this mixin because they avoid problems while changing configurations by deactivating any panes that do not belong in the configuration. Zmacs frames do use this mixin so that the screen manager will not autoexpose various editor windows that belong to the frame.

:screen-manage-autoexpose-inferiors *Operation on windows and screens*

Performs autoexposure of the active inferiors of this window or screen. Used by the default definition of :screen-manage.

2.9.1 Control of Partial Visibility

:screen-manage-deexposed-visibility *Operation on windows*

Should return non-nil if parts of this window ought to be displayed when the window is partially visible. The default definition returns non-nil if the window has a bit-save array.

tv:show-partially-visible-mixin *Flavor*

If a window has this flavor mixed in, then the screen manager will attempt to show it to the user when it is partially visible even if it doesn't have a bit-save array. Since there are no saved contents to display, the screen manager must give the window a screen array temporarily, send it a :refresh message so it will draw itself on the screen array, and then display whatever is found there. Often this means that you will see the label and borders of the window, but not the inside.

tv:gray-deexposed-right-mixin *Flavor*

tv:gray-deexposed-wrong-mixin *Flavor*

Make any visible parts of the window appear gray if the window is not fully visible. tv:gray-deexposed-wrong-mixin is faster, but does not work for windows that have inferiors. You would use these mixins in windows without bit-save arrays, as a cheaper alternative to tv:show-partially-visible-mixin, to provide something better than blankness when the window ought to be partially visible.

The precise kind of gray is controlled by the instance variable tv:gray-array, which comes with operations :gray-array and :set-gray-array and init option :gray-array. The value must be a two-dimensional array of bits that will be replicated by bitblt; its width must be a multiple of 32. Useful values for tv:gray-array include tv:75%-gray, tv:50%-gray, tv:33%-gray, tv:25%-gray, and tv:12%-gray.

tv:initially-invisible-mixin *Flavor*

Causes a window not to appear through screen management, even partially, until it has first been explicitly exposed. This is used in some window flavors (such as editor windows, Supdup windows, and others) of which instances are present in the saved system environment even without the user's ever having requested them. These windows

can be active, and available for System keys to select, but will not become partly visible if some other window is made smaller.

Recall that if a deexposed window has its deexposed typeout action set to :permit, output on the window can proceed but goes to the bit-save array rather than to the screen. If the window is partially visible, such output could modify the visible parts of the window. You can request that the screen manager check periodically for such output and copy the changed contents to the screen.

tv:screen-manage-update-permitted-windows *Variable*

Controls whether the screen manager looks for partially-visible windows with deexposed typeout actions of :permit and updates the visible portion of their contents on the screen. If the value is nil, which it is initially, the screen manager does not do this. Otherwise the value should be the interval between screen updates, in 60ths of a second.

2.9.2 Priority among Windows for Exposure

Suppose there is a section of the screen in which there are no exposed windows, and more than one active, deexposed window could be exposed to fill this area, but the two could not both be exposed (because they overlap). Which one gets to be exposed? Here's another issue: when the screen manager wants to display pieces of partially-visible windows, there may be more than one deexposed window that could be displayed in a given area of the screen. How does screen manager decide which window to display?

It decides on the basis of a priority ordering. All of the active inferiors of a window are maintained in a specific order, from highest to lowest priority. When there is a section of the screen on which more than one active inferior might be displayed, the inferior that is earliest in the ordering, and so has the highest priority, is the one that gets displayed. This ordering is like the relative heights of pieces of paper on a desk; the highest piece of paper at any point on the desk is the one that you see, and all the rest are covered up.

:order-inferiors *Operation on windows and screens*

Sorts the tv:inferiors list of active inferiors of this window or screen into the proper order for considering them for autoexposure or partial visibility.

The default definition of :order-inferiors uses a complicated algorithm which is designed to put the most recently exposed windows first, but also allows the programmer to specify priorities explicitly. If you do not need to know the details, you can safely skip the rest of this subsection.

The algorithm involves a value assigned to each window called its *priority*, which may be a fixnum or nil. The general idea is that windows with higher numerical priority values have higher priority to appear on the screen. The default value for the priority is nil, which is considered less than any numeric value.

tv:priority *Instance variable of windows*

The window's priority value, a number or **nil**.

:priority *Operation on windows*

:set-priority *new-priority* *Operation on windows*

Get or set the window's priority value.

:priority *priority* *Init option for windows*

Initializes the window's priority value.

The standard ordering of inferiors puts all exposable inferiors first, followed by the unexposable inferiors in order of decreasing priority. Each group of unexposable inferiors with the same priority is order by how recently they were exposable; the longer an inferior has gone without being exposable, the farther back it moves.

This is done by computing the current ordering based on the past ordering (as remembered by the old value of tv:inferiors). When the window system does anything which should change the ordering, such as making a window exposable or not exposable, it invokes the :order-inferiors operation to update the recorded ordering.

The ordering is updated by moving the exposable windows to the front and sorting the unexposable ones by priority. The sort is *stable*; that is, unexposable windows with the same priority value keep their previous ordering. Since most of the time numerical priorities are not used anyway (the priorities of most windows are **nil**), the ordering generally changes only as a result of exposure and deexposure of windows. When a window becomes exposable it gets pulled up to the front of the ordering; then when other windows become exposable instead, this window sinks back down. Thus, the ordering ends up showing simply how recently each window was exposable.

There is also an operation called *burying* a window, which deexposes the window and puts it at the end of its priority grouping in the ordering. A program typically buries its window when it thinks that the user is not interested in that window and would prefer to see some other windows. The **Bury** command in **Edit Screen** is a way for the user to bury a window.

:bury *Operation on windows*

Buries the window. See also **tv:deselect-and-maybe-bury-window**, a convenient interface to this operation (page 33).

Negative priorities have a special meaning. If the value of a window's priority is -1, then the window will not ever be visible at all even if it is only partially covered; however, it will still get autoexposed. If the value of priority is -2 or less, then the window will not even be autoexposed, and so it will simply never be seen unless sent an explicit **:expose** message.

2.9.3 Delaying Screen Management

The screen manager can potentially interfere with the actions of a program that explicitly deexposes windows. Suppose you send a :deexpose message to an exposed window. The screen manager will run, and will probably autoexpose that very window, canceling the effect of the :deexpose. That window certainly does not overlap any still-visible windows, and it is as recently-exposed as a window can get, so it will be the first candidate for autoexposure.

Explicit deexposure is usually done at the beginning of a sequence of window rearrangements. For example, moving an exposed window is done by deexposing it, changing its position (which is easy when it is deexposed) and reexposing it. We want the screen manager to run when the whole sequence is complete; it should not consider the transient intermediate states. Even if the screen manager did not directly interfere with the program's deliberate actions, it would waste time and confuse the user by displaying partially visible windows in temporarily-unclaimed screen areas for which the program is already preparing a new use. (This is a general phenomenon. Management is a useful auxiliary function, but managers have a tendency to interfere with work they don't understand if there is no way to shut them off.)

We shut the screen manager off with the special form tv:delaying-screen-management. While its body is being executed, events that would normally bring about screen management are recorded on a queue instead. When the tv:delaying-screen-management form is exited (whether normally or by throwing), the screen manager looks at the queue and does all necessary screen management in one blow.

Sometimes it happens that screen management cannot be done when the tv:delaying-screen-management form is exited, because relevant windows are locked by other processes. Then the entries are left on the queue. They are handled at some later time when the necessary locks are free by a background process called Screen Manager Background. So the necessary screen management always does eventually get done.

When tv:delaying-screen-management forms are nested, only the outermost one will do any screen management when it is exited.

tv:delaying-screen-management *body...* *Special form*

The body forms are evaluated sequentially with screen management delayed. The value of the last form is returned.

tv:without-screen-management *body...* *Special form*

The body forms are evaluated seqentially with screen management delayed. Moreover, if the body completes normally, the queue entries put on by its execution are removed from the queue, on the assumption that the body has itself done all appropriate screen redisplay. If the body terminates abnormally with a throw, the queued entries remain on the queue and are processed by the screen manager eventually.

3. Selection

At any time, keyboard input is directed to at most one window, designated by programs or by the system in response to user commands. This window is called the *selected* window. A process trying to do input through another window will normally wait until the window is selected (however, the window's deexposed typein action can change this; see below).

tv:selected-window *Variable*

The value of this variable is the selected window. You should not set this variable yourself, but use the defined interfaces described below.

A window's cursor blinker normally blinks only when the window is selected. This is how the user can tell which window is selected. (You can control what happens to each blinker when its window becomes selected; see page 103.)

The user can change the selected window using the Terminal and System keys or the system menu. Also, clicking the mouse on a window usually selects that window if it is meaningful to do so.

The simplest, and paradigmatic, case of window selection happens if you have several independent windows on the screen, such as Lisp listeners. One of them displays a blinking cursor, and input echoes there. The processes in the others remain in a keyboard input wait, as you can see if one of the windows on the screen is a Peek. The mouse, or the Terminal O command, can be used to select a different window.

The selected window needs to handle certain operations that windows in general do not need to handle. The flavor tv:select-mixin defines these operations, and should be used in flavors of windows that are going to be selected. (A window can be useful without being selectable. For example, menus cannot be selected.) The flavor tv:window includes tv:select-mixin.

If two processes try to read from the same window (or windows sharing an input buffer), it is unpredictable which one will get the input. If you are designing an application where this might happen, you must make sure than you will not have two processes actually active and reading input from the same source at the same time. In most applications there will be only one process that ever reads input from any one window or input buffer. In these applications you should use tv:process-mixin in the window flavor to tell the window which process is associated with it (see page 40).

The selected window controls the actions performed by the system at the instant a character is typed on the keyboard. Due to typed-ahead commands that switch windows (such as Control-Z in the editor), there is no way to know for certain which window will eventually read a character being typed at a given moment, so letting the selected window decide asynchronous processing for the character is the best that can be done. Asynchronous processing options include asynchronous intercepted characters (see section 5.5.2, page 61) and case conversion of control characters (see page 59).

Asynchronous intercepted characters such as Control-Abort which act on a process ask the selected window which process to operate on, with the :process operation (see page 42). The who line usually does the same thing to find the process whose run state should be displayed. If you use tv:process-mixin, :process returns the process associated with the window; otherwise, a default definition of :process is inherited from tv:select-mixin and returns whichever process last read input from the window (or from any other window sharing the same input buffer). This is fine for the who line, but can lead to weird results in Control-Abort. So *you should use* tv:process-mixin *whenever it makes sense.*

If a process tries to do input from a window whose input buffer is empty and not selected, it cannot get any input and must wait. (The input buffer is selected if this window, or any other window sharing the same input buffer, is the selected window). The wait ends when input appears in the buffer, or when the buffer becomes selected and there is keyboard input available. If the window is not even exposed, a notification may happen in addition. This is controlled by the window's *deexposed typein action*, which may be either :normal or :notify. :notify means that input from the window when it is deexposed should notify the user (see page 157) with a message like "Process X wants typein", and make the window "interesting" so that Terminal 0 S can select it.

:deexposed-typein-action *action* *Init option for windows*
Initializes the "deexposed typein action" (see page 32) of the window to *action*. It defaults to :normal.

:deexposed-typein-action *Operation on windows*
Returns the "deexposed typein action" (see page 32) of the window.

:set-deexposed-typein-action *action* *Operation on windows*
Sets the "deexposed typein action" (see page 32) of the window to *action*.

3.1 How Programs Select Windows

Programs change the selected window using the :select operation.

:select &optional (*remember-previous* t) *Operation on windows*
Makes this window (or its selection substitute, if any) the selected window. Unless remember-previous is nil, the previous selected window is entered on the list of previously selected windows for the Terminal and System keys to use.

Many application window flavors define daemons for this operation. Note, however, that the daemons will be run whenever this operation is invoked, even if the window is already selected.

tv:select-mixin *Flavor*
No window can actually be selected unless its flavor includes this mixin. tv:select-mixin is part of tv:window but not part of tv:minimum-window.

Windows whose flavors do not contain this mixin may be sent :select messages only if they have designated other windows as selection substitutes (see below). The ultimate substitute which is finally selected must have tv:select-mixin.

:selected-p *Operation on windows*

Returns t if this window is the selected window.

:mouse-select *args* *Operation on windows*

Selects a window, for a mouse click or for asynchronous keyboard input such as the Terminal command.

While mostly the same as sending :select to the window's alias for selected windows (see section 3.2.2, page 36), this operation also causes all type-ahead input to remain with the window that used to be selected (see page 59).

Note that the :select and :mouse-select operations should not be invoked in the mouse process. This means that if you want to use them in a :mouse-click or :mouse-buttons or :handle-mouse method, you must do

```
(process-run-function "Select" window-to-select ':mouse-select)
```

:deselect &optional (*restore-selected* t) *Operation on windows*

This operation is invoked by the window system when a window ceases to be selected. This can be because the window is no longer visible, or because some other window is being selected.

Many application window flavors defined daemons for the :deselect operation.

restore-selected controls what will be done with this window in the tv:previously-selected-windows array used by the Terminal S and System commands, and whether to select automatically some other window found in that array. The possible values are

:dont-save Do not put the window being deselected into the list anywhere, and do not select any other window.

nil or :beginning
Put the window being deselected at the front of the list, but do not select any other window.

:end Put the window being deselected at the end of the list, but do not select any other window.

:first Put the window being deselected at the front of the list, after selecting the window that used to be at the front of the list. This is like what Terminal S does.

:last or t Put the window being deselected at the end of the list, and select the window at the front of the list. This is the default.

tv:deselect-and-maybe-bury-window *window deselect-mode*

Deselects *window*, selecting the previously selected window. If that causes *window* to become deexposed, *window* is buried. *deselect-mode* is passed to the :deselect operation, where it controls where to put the window in the list of previously selected windows used by the Terminal and System commands.

tv:window-call (*window* [*exit-operation exit-args...*]) *body...* *Special form*
tv:window-mouse-call (*window* [*exit-operation exit-args...*]) *body...* *Special form*

Execute *body* with *window* selected. tv:window-call uses the :select operation to do this, while tv:window-mouse-call uses the :mouse-select operation; that is how they differ. On exit, they reselect the window that had been selected before, and send *window* a message with operation *exit-operation* and arguments *exit-arguments*. *exit-operation* is often :deactivate. It can also be omitted; then nothing is done to *window* except for deselecting it because some other window is selected.

These constructs are no longer as useful as they once were. For controlling selection among windows of a team, selection substitutes (see section 3.3, page 37) should be used now instead.

3.2 Teams of Windows

The simple paradigm of selection is based on windows that are independent competitors for the user's input, such as a pair of Lisp listeners. Another frequent situation is a to have a group of windows that act as a team. Usually the windows consist of a single frame and its panes, and usually they are managed by a single process, but neither of these is necessarily so. Often the windows of a team share an input buffer to make it easier for one process to read input from all of the windows at once (see section 5.1, page 50); this is an important technique which you should definitely read about if you are designing a team of windows.

The simple paradigm extends cleanly to teams if we imagine that the user regards each team as a unit of selection. In this extended paradigm, the user selects among entire teams as if they were single windows.

Teams are not actual Lisp objects, merely concepts understood by the user and programmer. The window system cannot have a selected team; some window of the team must be selected. Each team's program chooses a window of the team as the team's selection representative. The selected window should then be the selection-representative of the user's chosen team. The selected window can change when the user chooses a new team, or when the user's chosen team picks a new representative.

To implement this, the programmer of the team first picks one window of the team to be the "leader". This is not the same as the selection representative; that can change from moment to moment, but the leader must be fixed. When the team is a frame and its panes, it is natural to make the frame be the leader. Standard mixins are provided to make this easy to do. These mixins and the techniques of using them are described below, and in the following sections.

The selection representative is implemented as the leader's selection substitute (see section 3.3, page 37. Then the team can be selected with the :select operation on its leader window.

Even when the team allows the user some notion of selecting among the windows of the team—such as, when a Zmacs frame in two-window mode allows the user to mouse either of the editor windows to select it— this is implemented most cleanly by starting from the model of a team which does all selection under program control, and defining the appropriate mouse clicks as commands which tell the team's program to change its selection representative.

Usually you will want to have only a single item appear for the team in the system menu Select option's menu. If the team consists of a frame which is the leader and its panes, this can be done with tv:inferiors-not-in-select-menu-mixin in the frame's flavor. More complicated behavior is also possible; for example, Zmacs frames in two-window mode allow each editor window to have its own entry in the Select menu.

Also, Terminal and System commands should reselect the team by selecting its current selection representative. This is done by making them record and reselect the team's leader. If the team consists of a frame which is the leader and its panes, this can be done with tv:alias-for-inferiors-mixin in the frame's flavor. (In case you are curious, Zmacs frames follow this pattern exactly. The frame is the alias for any editor windows inside it.)

The following subsections describe the details of how these things are done.

3.2.1 The System Menu Select Option

When the Select option in the system menu is used, it gets the list of alternatives to offer by sending each screen a :selectable-windows message. This operation is normally defined to recurse down the window hierarchy and ask each window whether it wants to be included. Each window is sent a :name-for-selection message. The value should be either nil (omit this window) or a string, which is the string to display in the menu of windows.

:selectable-windows *Operation on windows*

Returns an alist of strings versus windows, which will become part of the alist that will be displayed in the system menu Select option's menu. The alist returned should describe this window and its inferiors, or whichever of them ought to appear in that menu.

The normal definition includes this window using its :name-for-selection as the car of the alist element, or omits this window if its :name-for-selection is nil. It then appends the :selectable-windows values obtained from the window's inferiors.

tv:inferiors-not-in-select-menu-mixin *Flavor*

This mixin redefines :selectable-windows to ignore the window's inferiors. They are not asked whether they should be included. This is an easy way to make a team of a frame and its panes have only one entry, the entry for the frame.

:name-for-selection *Operation on windows*

This operation is supposed to return a string to display in the system menu Select option's menu of windows for this window. It may also return nil, meaning do not list this window in the menu.

The default definition uses the window's label string if any, or else its name. Many applications redefine the operation. tv:not-externally-selectable-mixin redefines it to return nil.

If you want more complicated behavior from a team than simply having a single entry, you can get it by redefining this operation on the flavors of various windows in the team.

This operation also affects autoselection, done by the screen manager. A window can be autoselected only if its :name-for-selection is non-nil.

3.2.2 Selection with Terminal and System Commands

tv:previously-selected-windows *Variable*

This variable's value is an art-q-list array whose contents are all the active windows, not including the selected window, which the Terminal and System key commands for window selection should know about. The windows of a team are generally all represented by a single member of the team, which we can call the "leader". Typically the "leader" is a frame which contains the rest of the team as panes, but this is not required.

The Terminal S command can be thought of as acting on a combined list that contains the selected window followed by the previously selected windows. So, Terminal *n* S rotates the first *n* elements of this list, so that the selected window becomes the first previously-selected window, and the *n*th previously selected window becomes the selected window. The System key also uses this data base to find a window of the appropriate flavor to select, or to rotate through all the windows of that flavor.

Windows are put on tv:previously-selected-windows and taken off of it automatically when they are selected, deselected, activated or deactivated. Attention is required from the applications programmer only to identify teams of windows that should be treated as a unit. The system uses the :alias-for-selected-windows operation to inquire about this.

:alias-for-selected-windows *Operation on windows*

Should return the window to represent this one in tv:previously-selected-windows. When this window gets deselected, its alias is what will be recorded in that array. In the simple paradigmatic case of independent Lisp listeners, the alias of each Lisp listener is itself. For a window in a team, this should return the team's "leader" window.

The default definition of this operation is to return the superior's :alias-for-inferiors if that is non-nil, otherwise to return this window itself.

:alias-for-inferiors *Operation on windows*

Should return a window to serve as the alias for all inferiors to all levels of this window, if that is desired. Otherwise it should return nil.

The default definition returns this window's superior's :alias-for-inferiors. Thus, if an ancestor of this window says it wants to be an alias for all of its descendants, we pass on its request, but otherwise we allow the descendants to decide for themselves.

tv:alias-for-inferiors-mixin *Flavor*

This mixin makes a window be an alias for all of its inferiors. Thus, the window and all of its inferiors form a team considered as a unit by the Terminal and System commands, and this window is the "leader".

If two windows in a hierarchy, one above the other, both have tv:alias-for-inferiors-mixin, then the higher one "wins". Put another way, windows are grouped into the largest possible teams, and there are no subteams within teams.

Note also that no record is kept of which window in a team was actually selected most recently. tv:previously-selected-windows records only the alias or team leader window, and this is the window that will receive the :select message if a Terminal command is given to switch back to that team. The way to make sure that the proper window within the team is selected is to use selection substitutes, as described in the following section.

tv:not-externally-selectable-mixin *Flavor*

This mixin makes a window (and its descendants) have the window's superior as an alias, and keeps the window out of the Select menu.

Using this mixin, you can control more closely which windows are distinguished by the Select menu and by Terminal commands: instead of making the top of the team's hierarchy be an alias for *all* of its descendants, specifically chosen descendants are given this mixin so that they are not distinguished, and any other descendants remain distinguished.

An older name for this mixin, which still works, is tv:dont-select-with-mouse-mixin.

3.3 Selection Substitutes

Every window has the ability to designate a "selection substitute". If a window has a substitute, requests to select or deselect the original window will be passed along to the substitute. The substitute may have a substitute of its own, and so on. A window's selection substitute is remembered in the instance variable tv:selection-substitute, whose value is another window or nil.

tv:selection-substitute *Instance variable of windows*

The window's selection substitute, or nil.

The main use of selection substitutes is for controlling selection within a team of windows. The team has one window designated as the leader; all user requests to select the team come as :select messages to the team leader as a result of arrangements described in the previous section. As a result, the team's program can choose a selected window within the team by making it the leader's selection substitute.

The :alias-for-selected-windows of the substitute window should be the same as that of the window it substitutes for, to avoid paradoxical results from the Terminal command. With a hierarchical team of windows, this is usually arranged by using tv:alias-for-inferiors-mixin in the top window of the team. The substitute window should not appear in the system menu Select menu, for if it did, its entry and the entry for the window it substitutes for would be functional duplicates. tv:inferiors-not-in-select-menu-mixin in the top window of the team serves to prevent the duplicate entry.

These operations on windows are provided for working with selection substitutes:

:selection-substitute *Operation on windows*
Returns this window's selection substitute, or nil if the window does not currently have one.

:ultimate-selection-substitute *Operation on windows*
Returns this window's substitute's substitute... and so on until a window is reached that has no substitute. If this window has no substitute, it itself is returned.

:self-or-substitute-selected-p *Operation on windows*
t if this window, or its substitute, or its substitute's substitute, etc., is selected.

:set-selection-substitute *substitute* *Operation on windows*
Sets this window's selection substitute to *substitute* (another window or nil). If it was previously the case that this window or its substitute was selected, then the window's new substitute (or the window itself) will be selected afterward. Thus, the value of :self-or-substitute-selected-p on this window is not changed by this operation.

Note that when the team's program uses :set-selection-substitute on the team's leader window to change the selected pane within the team, it does not matter whether the team is currently selected. The "right" results will happen if the team is deselected and reselected at any time.

To switch the selected pane temporarily, use

tv:with-selection-substitute (*window for-window*) *body...* *Special form*
Executes *body* with *window* as the substitute for *for-window*. On exit, it sets *for-window* back to whatever it used to be, and deexposes or deactivates *window* if appropriate.

Also useful is

tv:preserve-substitute-status *window body...* *Special form*
Executes *body*, then selects *window* if *window* or its substitute had been selected to begin with.

:remove-selection-substitute *Operation on windows*
window-to-remove suggested-substitute
Makes sure that *window-to-remove* is not this window's substitute, suggesting *suggested-substitute* (possibly nil) as a substitute instead. The standard implementation of this operation simply sets the substitute to *suggested-substitute* if the substitute was *window-to-remove*. This operation is used and documented so that particular windows can define their own ways of calculating the new value for the substitute, perhaps ignoring *suggested-substitute*.

When a typeout window is deactivated, this operation is used to make sure that it ceases to be another window's substitute.

3.3.1 Non-Hierarchical Selection Substitutes

Some programs wish to "replace" one window with another temporarily. For example, the functions supdup and telnet can behave this way, giving the appearance of temporarily changing the Lisp listener or other window in which they are called into a Supdup or Telnet window. They do this by creating a suitable Supdup or Telnet window and making it the substitute for the original window. In this case, the substitute window will have the same edges and the same superior as the original window. It is not an inferior of the original window. It is not required that the "replacement" window be the same size as the original, either.

Non-inferior selection substitutes are usually established and deestablished by using tv:with-selection-substitute in a straightforward manner. The only thing that requires special attention is to make sure that the original window is the :alias-for-selected-windows of the substitute. In the case of supdup this is desirable to complete the illusion that the Lisp listener has "magically" changed temporarily into a Supdup window. Since the substitute window is not a descendant of the original one, it must have some other way to find the original window (such as an instance variable for the specific purpose) and a specially defined :alias-for-selected-windows method to return the original window.

3.4 The Status of a Window

A window's *status* is a keyword that encodes whether the window is selected, whether it is exposed, and whether it is active.

:status *Operation on windows*

Returns one of these symbols:

:selected Means this is the selected window.

:exposed Means this is exposed but not selected. It may not be visible.

:exposed-in-superior
Means this window is exposable but its superior has no screen array.

:deexposed Means this window is active in its superior but not exposable.

:deactivated Means this window is not even active.

:set-status *status* *Operation on windows*

Restores the window's status to *status*, by selecting or deselecting, exposing or deexposing, and activating or deactivating, as necessary. *status* must be one of the possible values of the :status operation.

The :status and :set-status operations are useful for selecting a window temporarily and then restoring everything as it was. :set-status is correct for this because it may be necessary to deexpose the window or deactivate it in addition to deselecting it.

3.5 Windows and Processes

A self-contained interactive system that has its own window(s) usually has its own process to drive the windows. Peek, Zmacs, ZMail and the inspector all do this when invoked through the System key. Usually each window you create has its own process; there is a Peek process for each Peek window, so different Peek windows run completely independently.

Whether a window is managed by a dedicated process or by various processes is not a crucial decision. The program that reads commands from the window and draws on the window can always be run in one dedicated process, or in different processes at different times (though if you run it in two processes at once, you had better be careful to keep them from confusing each other). The mechanisms of selection and exposure that control whether input and output are possible on a window at a given time work automatically on any process(es) that try to do the input or output. So when there is a dedicated process for a window, often the only connection between them is that the dedicated process is running a program that has a pointer to that window (typically the value of terminal-io in the process is that window).

For example, the inspector you get with System I has a dedicated process, whereas the one you get by calling inspect runs in the process that inspect is called in. Yet these two windows have the same flavor, and the same function, tv:inspect-command-loop, does the main work. The only differences are in deciding when to deexpose the window, what to do when that happens, when it can be reused, what to do if the user types End, and other things related directly to the difference in the two user interfaces for entering and exiting.

The inspector makes an instructive example for comparing these two ways of managing a window. The function inspect allocates a window out of a resource of reusable windows of the right flavor (see defresource, section 5.12 of the Lisp Machine manual). It sends the window some messages to initialize it for this particular session of use; this is how it tells the window about the object that is the argument to inspect. Then it selects the window manually using tv:window-call (see page 34) and calls the inspector program. When the user types End, the program returns, tv:window-call reselects the old window and deactivates the inspect window, and inspect returns. inspect uses an unwind-protect so that aborting outside of inspect for any reason brings back the old window.

Typing System I finds or creates an an inspect window of the same flavor. When no init options are specified, this flavor's default init plist specifies the creation of a process, which is initialized to call the inspector program. If the user types End and the inspector command loop returns, the top-level function in the dedicated process buries the inspector window and loops back to the beginning. That's all that is necessary to make System I work. The resource that inspect uses explicitly specifies an init option when it constructs a window, so that no process is made.

tv:process-mixin *Flavor*

Provides an instance variable tv:process which can remembers a process associated with the window. A window that will sometimes be used with a dedicated process should have this mixin.

The most valuable service that this flavor provides is an easy way to create and initialize a process for each window that is created, and inform the process which window it was created for. Once this is done, for the most part the desired results follow without special effort.

Selecting the window or making it visible will give the process a run reason. The window itself is used as the run reason. Also, this will reset the process if it is flushed (waiting with false as its wait function).

The :kill operation on the window will invoke the :kill operation on the process.

Use of tv:process-mixin guarantees that the :process operation will return the explicitly specified process, regardless of which process has most recently read from the window.

tv:process *Instance variable of* tv:process-mixin

The process associated with the window, or nil.

:process *process-or-description* *Init option for* tv:process-mixin

Specifies the process for this window. The argument can be a process, or it can be a list, which is used as a description for creating a process. The list looks like

(*initial-function make-process-options*)

When the process starts up, it will call *initial-function* with the window as its sole argument. Usually the initial function should bind terminal-io to the argument.

If this option is omitted or nil, the window starts out without a process.

:process *Operation on* tv:process-mixin

:set-process *process* *Operation on* tv:process-mixin

Gets or sets the process associated with this window. nil is a legal value, which means that the window has no process associated with it, even though it has the ability to have one.

:processes *Operation on windows*

Returns a list of processes dedicated to this window. :append method combination is used, so that all the processes mentioned by any of the methods are put into the final answer. These are the processes that the :kill operation will kill.

The default is to return nil. tv:process-mixin contributes a suitable method.

These process-related operations are defined on tv:select-mixin so that they are always supported by the selected window. Since windows lacking tv:process-mixin do not explicitly remember a process, a heuristic is used to come up with a process to operate on: it is the last process to have read input from this window's input buffer. (Think about the fact that the input buffer may be shared with other windows.)

tv:process-mixin is always put before tv:select-mixin in the components of a window flavor, so this method will be overridden.

:process *Operation on* tv:select-mixin

Gets a process somehow associated with this window, heuristically if necessary.

:set-process *process* *Operation on* tv:select-mixin

Records *process* in the place where the last process to read input from this window would normally be recorded.

:arrest *Operation on* tv:select-mixin

:un-arrest *Operation on* tv:select-mixin

Arrests or unarrests the process returned by the :process operation. The arrest reason used or revoked is not specified (it defaults).

:call *Operation on* tv:select-mixin

Selects an idle Lisp listener window (possibly this window, if it is an idle Lisp listener). If the window selected is not this one, arrest this window's process with arrest reason :call. This arrest reason is removed automatically by selecting this window again.

tv:reset-on-output-hold-mixin *Flavor*

Causes any process that tries to draw on this window when it has an output hold to be reset when it does so (see the :reset operation on processes, section 26.4.3 of the Lisp Machine manual).

tv:truncating-pop-up-text-window-with-reset *Flavor*

A temporary window that truncates lines and also resets processes that try to output on it when it has output hold. This flavor is what Terminal F uses.

4. Sizes and Positions

This chapter is about examining and setting the sizes and positions of windows. There are many different operations that let you express things in different forms that are convenient in varying applications. Usually, sizes are in units of pixels. However, sometimes we refer to widths in units of characters and heights in units of lines. The number of horizontal pixels in one character is called the character-width, and the number of vertical pixels in one line is called the line-height; these two quantities are explained on page 67.

A window has two parts: the inside and the margins. The margins include borders, labels, and other things; the inside is used for drawing characters and graphics. Some of the operations below deal with the outside size (including the margins) and some deal with the inside size.

Since a window's size and position are usually established when the window is created, we will begin by discussing the init-options that let you specify the size and position of a new window. To make things as convenient as possible, there are many ways to express what you want. The idea is that you specify various things, and the window figures out whatever you leave unspecified. For example, you can specify the right-hand edge and the width, and the position of the left-hand edge will automatically be figured out. If you underspecify some parameters, defaults are used. Each edge defaults to being the same as the corresponding inside edge of the superior window; so, for example, if you specify the position of the left edge, but don't specify the width or the position of the right edge, then the right edge will line up with the inside right edge of the superior. If you specify the width but neither edge position, the left edge will line up with the inside left edge of the superior; the same goes for the height and the top edge.

In order for a window to be exposed, its position and size must be such that it fits within the *inside* of the superior window. If a window is not exposed, then there are no constraints on its position and size; it may overlap its superior's margins, or even be outside the superior window altogether.

All positions are specified in pixels and are relative to the *outside* of the superior window.

4.1 Init Options for Sizes and Positions

:left *left-edge*	*Init option for windows*
:x *left-edge*	*Init option for windows*
:top *top-edge*	*Init option for windows*
:y *top-edge*	*Init option for windows*
:position (*left-edge top-edge*)	*Init option for windows*
:right *right-edge*	*Init option for windows*
:bottom *bottom-edge*	*Init option for windows*
:width *outside-width*	*Init option for windows*
:height *outside-height*	*Init option for windows*
:size (*outside-width outside-height*)	*Init option for windows*
:inside-width *inside-width*	*Init option for windows*
:inside-height *inside-height*	*Init option for windows*
:inside-size (*inside-width inside-height*)	*Init option for windows*

:edges (*left-edge top-edge right-edge bottom-edge*) *Init option for windows*

These options set various position and size parameters. The size and position of the window are computed from the parameters provided by these and other options, and the set of defaults described above. Note that all edge parameters are relative to the *outside* of the superior window.

:character-width *spec* *Init option for windows*

This is another way of specifying the width. *spec* is either a number of characters or a character string. The inside width of the window is made to be wide enough to display those characters, or that many characters, in font zero.

:character-height *spec* *Init option for windows*

This is another way of specifying the height. *spec* is either a number of lines or a character string containing a certain number of lines separated by carriage returns. The inside height of the window is made to be that many lines.

:integral-p *t-or-nil* *Init option for windows*

The default is nil. If this is specified as t, the inside dimensions of the window are made to be an integral number of characters wide and lines high, by making the bottom margin larger if necessary.

:edges-from *source* *Init option for windows*

Specifies that the window is to take its edges (position and size) from *source*, which can be one of:

a list — The elements of the list should be the four edges, *left*, *top*, *right* and *bottom*, all relative to this window's superior.

a string — The inside-size of the window is made large enough to display the string, in font zero.

a list (*left-edge top-edge right-edge bottom-edge*) — Those edges, relative to the superior, are used, exactly as if you had used the :edges init-option (see above).

:mouse — The user is asked to point the mouse to where the top-left and bottom-right corners of the window should go. (This is what happens when you use the Create command in the system menu, for example.)

a window — That window's edges are copied.

:minimum-width *n-pixels* *Init option for windows*

:minimum-height *n-pixels* *Init option for windows*

In combination with the :edges-from :mouse init option, these options specify the minimum size of the rectangle accepted from the user. If the user tries to specify a size smaller than one or both of these minima, he will be beeped at and prompted to start over again with a new top-left corner.

The group of operations below is used to examine or change the size or position of a window. Many operations that change the window's size or position take an argument called *option*. The reason that this argument exists is that certain new sizes or positions are not valid. One reason that a size may not be valid is that it may be so small that there is no room for the margins; for

example, if the new width is smaller than the sum of the sizes of the left and right margins, then the new width is not valid. A new setting of the edges is also invalid if the window is exposed and the new edges are not enclosed inside its superior. In all of the operations that take the *option* argument, *option* may be either nil or :verify. nil means that you really want to set the edges, and if the new edges are not valid, an error should be signalled. :verify means that you only want to check whether the new edges are valid or not, and you don't really want to change the edges. If the edges are valid, the operation with :verify returns t; otherwise it returns two values: nil and a string explaining what is wrong with the edges. (Note that it is valid to set the edges of a deexposed inferior window in such a way that the inferior is not enclosed inside the superior; you just can't expose it until the situation is remedied. This makes it more convenient to change the edges of a window and all of its inferiors sequentially; you don't have to be careful about what order you do it in.)

4.2 Flavor Operations for Sizes and Positions

:size *Operation on windows*

Returns two values, the outside width and outside height.

:height *Operation on windows*
:width *Operation on windows*

Return the window's height or its width.

:set-size *new-width new-height* &optional *option* *Operation on windows*

Sets the outside width and outside height of the window to *new-height* and *new-width*, without changing the position of the upper-left corner.

:inside-size *Operation on windows*

Returns two values, the inside width and the inside height.

:inside-height *Operation on windows*
:inside-width *Operation on windows*

Return the inside height of the window or the inside width.

:set-inside-size *Operation on windows*
new-inside-width new-inside-height &optional *option*

Set the inside width and inside height of the window to *new-inside-height* and *new-inside-width*, without changing the position of the upper-left corner. The margin sizes are recomputed according to their contents, which in simple cases means they will stay the same.

:position *Operation on windows*

Returns two values, the *x* and *y* positions of the upper-left corner of the window, in pixels, relative to the superior window.

:set-position *new-x new-y* &optional *option* *Operation on windows*

Sets the *x* and *y* position of the upper-left corner of the window, in pixels, relative to the superior window.

:edges *Operation on windows*

Returns four values, the left, top, right, and bottom edges, in pixels, relative to the superior window.

:set-edges *Operation on windows*

new-left new-top new-right new-bottom &optional *option*

Sets the edges of the window to *new-left*, *new-top*, *new-right*, and *new-bottom*, in pixels, relative to the superior window.

:inside-edges *Operation on windows*

Returns four values, the left, top, right, and bottom inside edges, in pixels, relative to the top-left corner of this window. This can be useful for clipping. Note that this operation is *not* analogous to the :edges operation, which returns the outside edges relative to the superior window.

:center-around *x y* *Operation on windows*

Without changing the size of the window, positions the window so that its center is as close to the point (*x*,*y*) as is possible without hanging off an edge. The coordinates are in pixels relative to the superior window.

:expose-near *mode* &optional (*warp-mouse-p* **t**) *Operation on windows*

If the window is not exposed, changes its position according to *mode* and exposes it (with the :expose operation; see page 19). If it is already exposed, does nothing.

mode should be a list; it may be any of the following:

(:point *x y*) Position the window so that its center is as at the point (*x*,*y*), in pixels, relative to the superior window, or as close as possible without hanging off an edge of the superior.

(:mouse) This is like the :point mode above, but the *x* and *y* come from the current mouse position instead of the caller. This is like what pop-up windows do. In addition, if *warp-mouse-p* is non-nil, the mouse is warped (see page 112) to the center of the window. (The mouse moves only if the window is near an edge of its superior; otherwise the mouse is already at the center of the window.)

(:rectangle *left top right bottom*)
The four arguments specify a rectangle, in pixels, relative to the superior window. The window is positioned somewhere next to but not overlapping the rectangle. In addition, if *warp-mouse-p* is non-nil, the mouse is warped (see page 112) to the center of the window.

(:window *window-1 window-2 window-3 ...*)
Position the window somewhere next to but not overlapping the rectangle that is the bounding box of all the *window-n*s. You must provide at least one *window*. Usually you only give one, and this means that the window

is positioned touching one edge of that window. In addition, if *warp-mouse-p* is non-nil, the mouse is warped (see page 112) to the center of the window.

:change-of-size-or-margins &rest *options* *Operation on windows*
This is the primitive operation for changing a window's size or the size of its margins. All the other operations to do so end up calling this one, after all error checking has been done.

This operation should not be called by users; to change the size, use :set-size or another higher-level operation, and the margin sizes should be managed by the flavors that are responsible for computing how big they should be (tv:borders-mixin, etc.).

However, this operation is a good place to add :after daemons to recompute other data structure or change the size of inferiors according to the window's new size. In the :after daemon, the window's size and margins will already be altered to their new values.

4.3 Low Level Edges Functions

tv:x-offset *Instance variable of windows*
tv:y-offset *Instance variable of windows*
The position of the window's outside left (or top) edge relative to the window's superior.

tv:width *Instance variable of windows*
tv:height *Instance variable of windows*
The total width or height of the window.

Recall that a *sheet* is either a window or a screen.

tv:sheet-width *window*
tv:sheet-height *window*
tv:sheet-x-offset *window*
tv:sheet-y-offset *window*
Return the value of the corresponding instance variable of *window*. These are accessor defsubsts created by the :outside-accessible-instance-variables option of defflavor. They can therefore be setf'd, but doing so is usually unwise.

tv:sheet-inside-width &optional (*window* self)
tv:sheet-inside-height &optional (*window* self)
Return the inside width or height of the window.

When used without an argument, these defsubsts refer directly to the instance variables, and therefore must be called from methods or functions which use (declare (:self-flavor ...)).

tv:sheet-number-of-inside-lines &optional (*window* self)
Returns the number of lines (of height equal to tv:line-height) that fit in the inside height of the window.

When used without an argument, these defsubsts refer directly to the instance variables, and therefore must be called from methods or functions which use (declare (:self-flavor ...)).

tv:sheet-calculate-offsets *window superior*
Returns the *x* and *y* positions of *window*'s upper left corner in *superior* as two values. *window* must be an indirect inferior of *superior*, zero or more levels down. If *window* and *superior* are the same window, the values are zero.

tv:sheet-overlaps-p *sheet left top width height*
tv:sheet-overlaps-edges-p *sheet left top right borrom*
t if *sheet* overlaps the specified rectangle. The edges specified are relative to *sheet*'s superior.

tv:sheet-overlaps-sheet-p *sheet-a sheet-b*
t if the two sheets overlap. This is a geometrical test, and it does not matter where in the hierarchy the two sheets are.

tv:sheet-within-p *sheet left top right bottom*
t if *sheet* is contained within the specified rectangle, given relative to *sheet*'s superior.

tv:sheet-within-sheet-p *sheet outer-sheet*
t if *sheet* is within *outer-sheet*'s area. This is a geometrical test, and it does not matter where in the hierarchy the two sheets are.

tv:sheet-bounds-within-sheet-p *left top width height outer-sheet*
t if the specified rectangle is within *outer-sheet*. The edges are specified relative to *outer-sheet*'s superior.

tv:sheet-contains-sheet-point-p *sheet top-sheet x y*
t if *sheet* contains the point (*x*,*y*) in *top-sheet*.

5. Input

Windows can be given the ability to function as input streams (see section 21.5 of the Lisp Machine manual). This is implemented by the mixin tv:stream-mixin, which is a component of tv:window. (Originally, both input and output stream operations were defined on this mixin, but now the output operations are available on all windows since a window is fairly useless if you don't draw on it.) Input characters normally come from the terminal keyboard, but can also come from mouse clicks, or anything else you may decide to program to generate input.

tv:stream-mixin *Flavor*

This mixin defines the standard input stream operations for doing input from the keyboard, as well as some nonstandard input operations defined in the following sections.

Keyboard input is done through windows so that selection of windows can control which process can read input at a given time. In fact, this is why the concept of selection exists: by making each process that does its output to a window also use that window to read input, and by making a single "selected" window the only window on which input operations can proceed, we enable the user to decide which process to direct his input to by selecting the corresponding window.

Reading characters from a window normally returns a fixnum that represents a character in the Lisp Machine character set, possibly with extra bits that correspond to the Control, Meta, Super, and Hyper keys. Character constants in code are written with the #\ or #/ construct and are described in the Lisp Machine manual in section 21.1 of the Lisp Machine manual.

Programs decode keyboard characters with ldb and dpb using the following byte fields:

%%kbd-char A name for the byte field that contains the basic character. This is the low eight bits, and the contents are a character that can go in a string.

%%kbd-control
A name for the byte field that contains the Control bit.

%%kbd-meta A name for the byte field that contains the Meta bit.

%%kbd-super A name for the byte field that contains the Super bit.

%%kbd-hyper A name for the byte field that contains the Hyper bit.

%%kbd-control-meta
A name for the four-bit byte field that contains the Hyper, Super, Meta and Control bits, in that order from most significant to least.

%%kbd-mouse
A name for the byte field that is 1 if this is a "mouse" character, a character that reports a click of a mouse button rather than a pressing of a keyboard key. (See page 113.) Note that mouse characters may contain Control bits, etc.

%%kbd-mouse-button
A name for the byte field that, in a mouse character, records the number of the button that was clicked. The left button is 0, the middle is 1, and the right is 2.

%%kbd-mouse-n-clicks

A name for the byte field that, in a mouse character, records the number of times whichever button was clicked, minus 1. It is 0 for a single click, 1 for a double click, etc.

Though keyboard input characters are currently fixnums, it is possible that a new, special data type for characters will exist in the future. The #\ construct will produce a character object rather than a fixnum, and the elements of a string will be character objects rather han fixnums. Characters will behave just like fixnums in arithmetic and = but will not be eq to fixnums. The :tyi and related stream operations will continue to return fixnums; new operations will be defined which return character objects instead. It will still be possible to use ldb and dpb with these byte field names on fixnums and character objects indiscriminately.

Note that reading characters from a window does not echo the characters; it does not type them out. If you want echoing, you can echo the characters yourself, or call the higher-level functions such as tyi, read, and readline; these functions accept a window as their stream argument and will echo the characters they read. This is in accord with the general Lisp Machine input stream conventions.

The console hardware actually sends codes to the Lisp Machine whenever a key is depressed or lifted; thus, the Lisp Machine knows at all times which keys are depressed and which are not. You can use the tv:key-state function to ask whether a key is down or up. Also, you can arrange for reading from a window to read the raw hardware codes exactly as they are sent, by putting a non-nil value of the :raw property on the property list of the input buffer; however, the format of the raw codes is complicated and dependent on the hardware implementation. It is not documented here.

tv:kbd-last-activity-time *Variable*

The value returned by the function time when the last input character was typed.

5.1 Input Buffers

Every window that generates input or from which input is read must have an *input buffer* that holds characters that are typed by the user before any program reads the characters. When you type a character, it enters the selected window's buffer. (This is not precisely true, but it's a good first mental model. See section 5.4, page 56.) Reading input from a window, with the :tyi operation for example, takes objects out of the window's input buffer. tv:stream-mixin gives the window an input buffer, but some other flavors (such as command menus) provide an input buffer without tv:stream-mixin. The input buffer lives in an instance variable of the window, called tv:io-buffer.

Input buffers are examples of *I/O buffers*, which are a general facility provided by the window system. You can explicitly manipulate input buffers in order to get certain advanced functionality, by using the :io-buffer init-option and the :io-buffer and :set-io-buffer operations. Another thing you can do is put properties on the I/O buffer's property list; this lets you request various special features. I/O buffers are explained on section 5.4, page 56.

A window can be thought of as generating input when the keyboard is used while the window is selected. This is the way that ordinary characters normally get into the input buffer. But input can be generated at any place in the program by means of the :force-kbd-input operation. For example, mouse clicks are often handled by forcing input which is read by the window's command-interpreting process (see page 113). Then we say that the mouse click also generates input.

All the input, no matter how generated, ends up mixed together in the same input buffer, in chronological order. All the input operations take input from the buffer in that order.

Normally each window that can generate input has its own input buffer. If a process is managing more than one window that can generate input, a program to look for input from all the windows at once would be cumbersome. So it is not done this way. Instead, all the windows are made to share a single input buffer. Then all input generated by all of the windows goes into that buffer, from which the input can be read through any one of the windows. The program simply reads input from one of the windows—always the same one, if the programmer prefers—and gets all the input intended for it. All the keyboard input directed at it, and all mouse clicks on its windows, get merged into a single chronological input stream.

The input buffer does not record which window was "responsible" for generating input read from a shared input buffer. For mouse clicks the program may need to know which window the mouse was clicked on in order to obey the command properly. The standard way to pass this information is to use a list as the input character and make the window clicked on one of the elements of the list.

The window(s) used for input operations must have tv:stream-mixin. The other windows need only be able to put input into the right input buffer. It is often easiest to use tv:stream-mixin for them as well, and generate the input with :force-kbd-input. However, it is sufficient for such windows to support the :io-buffer operation by returning the correct shared input buffer, and put the input they generate into that buffer in any way that works, such as with the function tv:io-buffer-put, or by invoking :force-kbd-input on another window known to have tv:stream-mixin and to share the same input buffer.

If a frame includes a pane that is handled by its own process (such as a Zmacs frame), that pane should not share the input buffer used by the rest of the panes. In general, there should be one input buffer for each process you are using, and that input buffer should be shared by the windows which go with that process.

In general, the way to make windows share an input buffer is to create one using tv:make-default-io-buffer and then specify it for the :io-buffer init keyword when each pane is created. There are also frame flavors that automatically make the panes share an input buffer.

tv:io-buffer *Instance variable of* tv:stream-mixin

The window's input buffer.

:io-buffer *spec* *Init option for* tv:stream-mixin

Initializes the input buffer of the window. *spec* may be an I/O buffer, a number or a list. If it is a number, an I/O buffer is made with that size, no input function, and the default output function. If it is a list, it is interpreted as

(*size input-function output-function*)

but if the *output-function* is nil or omitted, tv:kbd-default-output-function is used.

:io-buffer *Operation on* tv:stream-mixin

:set-io-buffer *io-buffer* *Operation on* tv:stream-mixin

Return or set the window's input buffer.

5.2 Blips

Input need not be made of characters; lists are often used as well for program-generated input, especially for representing mouse clicks in different kinds of mouse-sensitive areas. "Characters" which are lists are called *blips*. The car of the list is by convention a symbol which identifies the kind of blip. Look for "blip types" in the concept index to find the places in this manual that define various kinds of blips.

Caution: when using blips, you should keep in mind that the blips may be discarded if the process has called any function that does not know what to do with them. The debugger and break are such functions, so this can happen at any time. Blips either should describe mouse actions, which can safely be ignored if they happen when they are not meaningful, or should notify the process to check other data structures. A blip should not be used to indicate a request or response from another process, since this information must not be lost. Instead, put the data on a separate queue and have the process check the queue after every command. A blip that executes as a no-op command will serve to wake the process up if it is waiting for input when the data goes on the queue.

There is a technique you can use to cause blips to be handled even in the middle of calls to read, the debugger, and other programs that do not look for blips. It is to give your window flavor an :around method for :any-tyi. This :around method can look at the value being returned; if it is one of certain types of blips, you can handle it and then loop around, calling the original :any-tyi handler again without returning to the caller. If it is anything else, you just return it.

5.3 Stream Input Operations

:any-tyi &optional *eof-action* *Operation on* tv:stream-mixin

Reads and returns the next character of input from the window, waiting if there is none. The character comes from the window's input buffer if it contains any characters; otherwise, it comes from the keyboard. *eof-action* is ignored since "end-of-file" is not meaningful for windows; this argument exists only because it is part of the input stream protocol.

:tyi &optional *eof-action* *Operation on* tv:stream-mixin
Like :any-tyi but throws away any blips ("characters" which are lists) that it receives. It keeps on reading until it finds an actual character, and returns that. Discarded blips will never be seen as input.

:any-tyi-no-hang &optional *eof-action* *Operation on* tv:stream-mixin
Like :any-tyi if input is already present in the buffer, but returns nil right away if the buffer is empty. This is used by programs that continuously do something until a key is typed, then look at the key and decide what to do next.

:tyi-no-hang &optional *eof-action* *Operation on* tv:stream-mixin
Like :any-tyi but throws away any blips ("characters" which are lists) that it receives. It keeps on reading until it finds an actual character or the buffer empty; then it returns the character or nil. Discarded blips will never be seen as input.

:mouse-or-kbd-tyi *Operation on* tv:stream-mixin
:mouse-or-kbd-tyi-no-hang *Operation on* tv:stream-mixin
These are like the :tyi and :tyi-no-hang operations, except that blips of a certain kind are not discarded and do count as input. These are blips whose car is the symbol :mouse-button. In this case, the first value returned is the third element (caddr) of the blip, and the second value returned is the whole blip. By convention, the third element of such a blip is a character whose %%kbd-mouse bit is 1, which identifies the button that the user clicked (see page 113). All other blips are discarded, as they are by :tyi and :tyi-no-hang. The first value is always a fixnum.

:list-tyi *Operation on* tv:stream-mixin
This is the "opposite" of :tyi. It returns only blips and discards real characters.

:untyi *character* *Operation on* tv:stream-mixin
Put *character* back into the window's input buffer so that it will be the next character returned by :tyi. Note that *character* must be exactly the last character that was read, and that it is illegal to do two :untyi's in a row. This is used by parsers that look ahead one character, such as read.

:force-kbd-input *input* *Operation on* tv:stream-mixin
input is inserted into the window's input buffer, to be read by the :tyi or other input operation in its turn. *input* may be a character or a list (a blip). It may also be a string; then all the characters of the string are forced as input, one by one.

This is the standard way that blips are put into the input stream (see section 5.2, page 52).

:listen *Operation on* tv:stream-mixin
Returns t if there are any characters available to :tyi, or nil if there are not. For example, the editor uses this to defer redisplay until it has caught up with all of the characters that have been typed in.

:wait-for-input-with-timeout *timeout* *Operation on* tv:stream-mixin

Waits until either input is available or *timeout* 60ths of a second have elapsed.

:clear-input *Operation on* tv:stream-mixin

Clears this window's input buffer. This flushes all the characters that have been typed at this window, but have not yet been read.

:playback *Operation on* tv:stream-mixin

Returns an array describing the last *n* characters read from this window, for some value of *n* (which is the size of the array). The array elements are used in a circular fashion, the last one being followed by the first one, and array leader element 1 contains the index of the last slot stored into (the one containing the last character read). The editor command Help L uses this operation.

:rubout-handler *options function* &rest *args* *Operation on* tv:stream-mixin

Applies *function* to *args* inside an environment where inputting from this window will echo the characters typed and provide for simple input editing. This is documented in more detail in the Lisp Machine manual.

options is an assq list of keyword symbols and arguments to them. The options acceptable to windows are:

:full-rubout *flag*

If the user rubs out all of the characters that he has typed in, normally the rubout-handler just waits for more characters. If the :full-rubout option is supplied, the rubout handler returns to the caller in this situation. Two values are returned, nil and *flag*.

:initial-input *string*

Treat the characters in *string* as typeahead before reading anything from the keyboard.

:pass-through *ch1 ch2...*

Treat the characters *ch1*, *ch2*, etc. as ordinary characters even if they would normally be special commands to the rubout-handler.

:prompt *function*

function is a function to be called before reading any characters; typically it will display a prompt. The arguments to *function* are the window and a flag. When the rubout-handler is first entered the flag is nil, but if it is necessary to prompt again, for instance if the user cleared the screen, *function* is called with the character the user typed (e.g. #\clear-screen) as its flag argument.

function can also be a string; then it is simply printed as the prompt.

:reprompt *function*

The same as :prompt except that the function is not called the first time through. If both :prompt and :reprompt are used, the :prompt is used the first time and the :reprompt is used on reprinting.

:save-rubout-handler-buffer *Operation on* tv:stream-mixin

Returns a description of the rubout handler buffer's contents, and clears it out. Two values are returned: a string and a fixnum (which is the current cursor index in the string). This is used on entry to the function break so that typing the Break key interfaces properly with rubout handling.

:restore-rubout-handler-buffer *string index* *Operation on* tv:stream-mixin

Loads the rubout handler buffer contents from *string* and sets the cursor position to *index*. The arguments are usually two values obtained from :save-rubout-handler-buffer.

:refresh-rubout-handler *Operation on* tv:stream-mixin

&optional *discard-last-character*

Requests the rubout handler to reprint its buffer and reprompt. If *discard-last-character* is non-nil, the last character in the buffer is discarded first. This is used by :restore-rubout-handler-buffer.

If you are reading input using the rubout handler, but want to process certain characters immediately (perhaps the character Help) and not leave them as part of the ordinary input, use this operation with argument t.

tv:preemptable-read-any-tyi-mixin *Flavor*

This flavor defines the :preemptable-read operation.

:preemptable-read *Operation on* tv:preemptable-read-any-tyi-mixin

options function &rest *arguments*

You may have noticed that in the inspector and in the Window Error Handler, if you start typing in a Lisp expression, and then while in the middle of typing it you use the mouse to select an object by pointing at it, the program sees the object you moused. If nothing special were done, though, the blip sent by the mouse process would get put at the end of the input buffer and would not be seen because of the characters that you have typed. This mixin is what is used to solve the problem.

The :preemptable-read operation takes the same arguments as the normal :rubout-handler operation, and does the same thing if the mouse is not used. (In fact, it has nothing to do with the read function, despite the name.) The difference is that if any blip is sent to the window, the operation returns the blip as the first value and the symbol :mouse-char as the second value. (It does this even if the blip did not come from the mouse; most blips do.) The characters that were in the rubout-handler buffer when the blip arrived will come back the next time a :preemptable-read operation is used, so the user can keep typing his expression in.

These obsolete functions are still used in some old code:

kbd-tyi

Performs :tyi on terminal-io.

kbd-tyi-no-hang
Performs :tyi-no-hang on terminal-io.

kbd-char-available
Performs :listen on terminal-io.

5.4 I/O Buffers

An I/O buffer is an array of fixed size used as a ring buffer. Typically, characters are put into the buffer by one process and removed by another in FIFO order. The process that is removing characters can wait if the buffer is empty, and a process putting in characters can wait if the buffer is full (or it could throw away the characters). Each window with tv:stream-mixin has an input buffer which is an I/O buffer, and there is also one global I/O buffer for the keyboard itself.

Note that the things stored in an I/O buffer can be any Lisp objects. They do not have to be characters, in any sense. But in practice I/O buffers are in fact used for storing characters (which may be lists), so that is how this section is written.

An I/O buffer has these slots in its leader.

tv:io-buffer-size
The number of slots in the input buffer.

tv:io-buffer-input-pointer
The index at which the next character inserted should be stored.

tv:io-buffer-output-pointer
The index at which the next available character is present.

If the input and output pointers are equal, the buffer is empty. If the output pointer points at the slot after the input pointer, the buffer is considered full (in fact, one slot is still empty. It cannot be used).

tv:io-buffer-output-function
A function to be called when characters are removed, or nil. It is called with the buffer and the character as arguments. Its value should be a translated version of the character (this is usually the same as the argument). It can also return a non-nil second character, which says that the character should be discarded. In this case, tv:io-buffer-get will remove the following character, or wait for one.

In window input buffers this is usually a function that checks for and handles synchronous interception.

tv:io-buffer-input-function
A function to be called when characters are inserted, or nil. The window system does not actually use this feature. The calling conventions are the same as for the output function.

tv:io-buffer-state
This may be set to t, nil, :input or :output to control what can be done with the buffer. Characters can be put in if this is nil or :input, and can be removed if

this is nil or :output.

tv:io-buffer-plist
A property list containing various properties.

tv:io-buffer-last-input-process
The last process that put a character in this I/O buffer.

tv:io-buffer-last-output-process
The last process that removed a character from this I/O buffer.

tv:io-buffer-record
An array that records the last *n* characters read from this I/O buffer, for some fixed *n*. This array too is a ring buffer, but nothing is ever "removed" from it; after it is full, it contains the last *n* things stored into it. The accessor tv:io-buffer-record-pointer gets the index of the last slot stored into.

tv:io-buffer-empty-p *io-buffer*
tv:io-buffer-full-p *io-buffer*
Non-nil if the buffer is empty, or full.

tv:make-io-buffer *size input-function output-function plist state*
Creates and returns an I/O buffer, initializing some of the slots from its arguments and the others in a default or reasonable fashion. The buffer is initially empty.

tv:io-buffer-put *buffer character* &optional *no-hang-p*
Inserts *character* into *buffer*, waiting if it is full unless *no-hang-p*. This function also waits if the buffer's state does not permit input. It returns t if the character was inserted.

tv:io-buffer-get *buffer* &optional *no-hang-p*
Removes the next character from *buffer*. If the buffer is empty, normally we wait for a character to appear, but if *no-hang-p* is non-nil we return nil immediately. This function also waits if the buffer's state does not permit output. The character removed is put in *buffer*'s io-buffer-record array.

tv:io-buffer-unget *buffer character*
Inserts *character* into *buffer* as the next character to be removed rather than as the last one to be removed. This is used for undoing tv:io-buffer-get, and it is an error if *character* does not match the last character removed. *character* is removed from the io-buffer-record array, by backing up its pointer, just to avoid duplication when *character* is read a second time.

This function should not be used more than once between input operations.

tv:io-buffer-push *buffer character*
Inserts *character* into *buffer* as the *next* character to be removed; that is, in a LIFO manner. This is as opposed to tv:io-buffer-put which inserts a character at the end.

tv:io-buffer-clear *buffer*
Makes *buffer* empty.

tv:process-typeahead *buffer function*
Uses *function* as a filter for the characters in *buffer*. *function* is called once for each character, with the character as its sole argument. If *function* returns non-*nil*, that value is stored back in the buffer instead of the original character. If *function* returns *nil*, the character is deleted from the input buffer.

5.4.1 I/O Buffers and Type Ahead

We have said (see section 5.1, page 50) that keyboard input goes into the selected window's input buffer. This is not precisely true. Program-generated input made with :force-kbd-input does go directly into the window's input buffer, but keyboard input actually goes into another I/O buffer called the *keyboard input buffer*. (There is only one of these in the system.) The characters move from the keyboard input buffer to the selected window's input buffer whenever a program tries to read input from that buffer and it is empty. The keyboard input is not assigned to a selected window until the instant the program is ready to read it.

Asynchronous window-switching commands, such as Terminal S, and mouse clicks that select a window, actually copy the contents of the keyboard input buffer into the buffer of the window that is being deselected. If you type some commands to the editor, and then type System L before the editor has read its commands, those commands will still go to the editor, not to the Lisp listener you have selected.

By contrast, synchronous window-switching such as is done by the functions ed, supdup and inspect, and by "exit" commands in various programs, do not do this, since any further typed-ahead input should go to the program being switched to.

5.4.2 I/O Buffers as Input Buffers

tv:make-default-io-buffer
Creates and returns an I/O buffer of the sort used for window input buffers, with all slots suitably initialized. The output function used is tv:kbd-default-output-function.

tv:kbd-default-output-function *buffer char*
This is the default value for a window input buffer's output function. It checks the character against the value of tv:kbd-intercepted-characters and also checks tv:kbd-tyi-hook.

tv:kbd-io-buffer-get *buffer* &optional *no-hang-p* (*whostate* "Keyboard")
Removes a character from *buffer*, or possibly from the keyboard input buffer. The keyboard input buffer can be read from only if *buffer* is the input buffer of the selected window, and it is used only if *buffer* is empty. When a character is read from the keyboard input buffer, *buffer*'s output function is executed, as if the character had been put into *buffer* and then read from there.

whostate is passed as the first argument to process-wait if this function has to wait.

tv:kbd-wait-for-input-with-timeout *buffer timeout* &optional (*whostate* "Keyboard")
Waits until either tv:kbd-io-buffer-get would not hang on *buffer* or *timeout* elapses. *timeout* is in 60ths of a second. *whostate* appears in the who line while we wait.

tv:kbd-wait-for-input-or-deexposure *buffer window* &optional (*whostate* "Keyboard")
Waits until either tv:kbd-io-buffer-get would not hang on *buffer* or *window* is not exposed. *whostate* appears in the who line while we wait.

tv:kbd-snarf-input *buffer* &optional *not-from-hardware*
Transfer any characters that tv:kbd-io-buffer-get could now get from *buffer* right into *buffer*. This is what asynchronous selection commands use to make sure that type-ahead for the window being deselected remains with that window.

tv:kbd-char-typed-p
Non-nil if input is available in the selected window. This can be used in programs that loop with interrupts disabled, to tell when the user types a key.

The window system defines the meaning of certain properties on the tv:io-buffer-plist of a window input buffer. These are

:raw Non-nil to inhibit translation of characters from hardware codes to the Lisp Machine character set. The effect of this is hardware dependent.

:asynchronous-characters
An alist which controls which characters are intercepted asynchronously when this window is selected.

:dont-upcase-control-characters
Non-nil prevents the Control (etc.) keys from causing special treatment of alphabetic case. Normally, typing Control-Shift-A produces the character #\Control-/a with a lower case "a", while Control-A produces #\Control-A; and the same for Meta, Super and Hyper. If this property is non-nil, the two inputs are interchanged in meaning, so that Shift produces an upper case character with or without Control.

5.5 Intercepted Characters

There are several characters that are specially intercepted by the window system. Some are intercepted when a process tries to read them, and some are intercepted as soon as they are typed. The first kind are called *synchronously intercepted characters* and the second are called *asynchronously intercepted characters*. The latter come in two kinds: *global asynchronous characters* such as Terminal and System which are always available (see section 5.5.3, page 63), and others defined by the selected window, normally including Control-Abort and so on (see section 5.5.2, page 61).

5.5.1 Synchronously Intercepted Characters

Synchronous interception is performed by the io-buffer-output-function of the window input buffer (see page 56). By default, this function is tv:kbd-default-output-function, which uses the variable tv:kbd-intercepted-characters to decide which characters to intercept and how to handle them. A program can change its set of synchronously intercepted characters simply by binding this variable before reading input. Its default value specifies the characters Abort, Meta-Abort, Break, and Meta-Break.

tv:kbd-intercepted-characters *Variable*

The value is an alist specifying the characters to be intercepted synchronously (that is, when read by the program). Since the variable is looked at by a subroutine of the :tyi operation itself, what matters is current binding at the time the :tyi is done.

Each element of this list should look like

(*character function*) .

Then *function* will be called if *character* is read, with *character* as argument.

function should return two values. The second should be non-nil to say that the character has been handled by the function and should not be returned to the calling program as ordinary input. If the second value is nil, the first value should be a translated character to use as input instead of the character typed. (This can be and usually is the same character that was typed.) The first value is ignored if the second is non-nil. In practice, *function* usually returns its argument and t.

function should begin by setting inhibit-scheduling-flag to nil.

It is reasonable to add new entries to the top level value of this variable, and also for programs to bind the variable. It is probably unwise to remove the standard entries in the top level value.

tv:kbd-standard-intercepted-characters *Variable*

This is the that which is the initial value of tv:kbd-intercepted-characters.

tv:kbd-intercept-abort *char* &rest *ignore*
tv:kbd-intercept-abort-all *char* &rest *ignore*

These functions implement the standard meanings of the Abort and Control-Abort keys. They are suitable for use in tv:kbd-intercepted-characters. The first signals the sys:abort condition; the second resets the current process.

If terminal-io handles the :inhibit-output-for-abort-p operation and it returns non-nil, the string "[Abort]" will not be printed.

tv:kbd-intercept-break *char* &rest *ignore*
tv:kbd-intercept-error-break *char* &rest *ignore*

These functions implement the standard meanings of the Break and Control-Break keys. They are suitable for use in tv:kbd-intercepted-characters. The first calls break; the second invokes the debugger.

Furthermore, if the variable tv:kbd-tyi-hook is non-nil, then it is considered to be a user function that can intercept the character at this point; see page 61.

By convention, programs are all expected to use the Abort key as a command to abort things in some appropriate sense for that program. If you don't do anything special, Abort will be intercepted automatically. But some programs may want to do something specific when the user types Abort. The system default action can be replaced by binding the variable tv:kbd-intercepted-characters so that Abort goes to your own intercept routine instead of tv:kbd-intercept-abort, or so that Abort is read as an input character from the stream like any other and then is handled by your program.

tv:kbd-tyi-hook *Variable*

The default io-buffer-output-function (tv:kbd-default-output-function), before it does anything else, sees whether the value of tv:kbd-tyi-hook is non-nil; if so, it assumes that the value is a function of one argument, and it applies the function to the character that was typed. If the function returns a non-nil value, then the character will not be returned to callers of :tyi or other input operations; otherwise, the character is processed normally.

The idea is that you can write a function that intercepts anything passing through an input buffer that uses the default io-buffer-output-function. Your function gets passed the character, and returns nil if it doesn't want to handle it, or t if it has taken care of the character.

5.5.2 Asynchronously Intercepted Characters

Each window that has tv:stream-mixin can define a set of characters to be intercepted asynchronously when that window is selected. The interception is done through a different mechanism from that used for synchronous interception, but the same handling functions such as tv:kbd-intercept-abort can ultimately be used. By default, a window requests asynchronous interception of the four characters Control-Abort, Control-Meta-Abort, Control-Break, and Control-Meta-Break. The default meanings of these keys are given in Operating the Lisp Machine. You can change the set of such asynchronous keys on a per-window basis.

Since the interception is done by the keyboard process, the characters cannot straightforwardly be specified by a variable for the program to bind. So each window has a list of them (which is actually stored as the :asynchronous-characters property on the input buffer's property list).

:asynchronous-characters *alist* *Init option for* tv:stream-mixin

alist specifies the characters to be intercepted asynchronously while this window is selected, and what they should do.

Each element consists of a character, a function to call, and optionally some extra arguments to be passed to it. When the function is called, its arguments will be the character, the selected window, and any specified additional arguments from the alist element.

If the init option is not specified, the default comes from the value of tv:kbd-standard-asynchronous-characters, the initial value of which is

```
((#\c-abort tv:kbd-asynchronous-intercept-character
  (:name "Abort" :priority 50.)
  tv:kbd-intercept-abort)
 (#\c-m-abort tv:kbd-asynchronous-intercept-character
  (:name "Abort All" :priority 50.)
  tv:kbd-intercept-abort-all)
 (#\c-break tv:kbd-asynchronous-intercept-character
  (:name "Break" :priority 40.)
  tv:kbd-intercept-break)
 (#\c-m-break tv:kbd-asynchronous-intercept-character
  (:name "Error Break" :priority 40.)
  tv:kbd-intercept-error-break))
```

How these work is explained below.

:asynchronous-character-p *character* *Operation on* tv:stream-mixin

Returns non-nil if this window defines *character* for asynchronous interception.

:handle-asynchronous-character *character* *Operation on* tv:stream-mixin

Invokes the handler a defined for asynchronous interception of *character*. This runs the handler function in your current process. But since handler functions typically do process-run-function, it usually doesn't matter.

:add-asynchronous-character *Operation on* tv:stream-mixin
character handler-function &rest *additional-args*

Define *character* for asynchronous interception in this window, to be handled by *handler-function* and the *additional-args*. This adds an element

(*character handler-function* . *additional-args*)

to the alist on the input buffer's property list.

:remove-asynchronous-character *character* *Operation on* tv:stream-mixin

Removes *character*'s element from the alist, so that it is no longer intercepted asynchronously in this process.

Asynchronous interception is done by the Keyboard process, and the handler function runs in that process. Therefore, it must obey some strict conventions. It must not do any I/O, or wait for anything; it should not run for very long; it should not get an error. It is usually easiest to create another process and do the real work there, using process-run-function.

tv:kbd-asynchronous-intercept-character *character window process-options subhandler additional-subhandler-args*

This function is provided as a convenient way to set up the handling of an asynchronously intercepted character. It enables you to interface to the same functions used for synchronous interception. It is used with at least two additional arguments: the process name and options for process-run-function, and the function to call in the new process. Thus,

```
(#\c-break tv:kbd-asynchronous-intercept-character
 (:name "Break" :priority 40.)
 tv:kbd-intercept-break)
```

arranges to create a process named "Break" with priority 40, and call tv:kbd-intercept-

break in that process.

subhandler, which is tv:kbd-intercept-break in this example, is passed as arguments *character*, *window*, and the *additional-subhandler-args* if any.

5.5.3 Global Asynchronous Characters

The Terminal and System keys are also intercepted asynchronously, but since their functions do not usually relate to the selected window, they are not controlled by the selected window's alist of asynchronous characters. These are called global asynchronous characters.

tv:kbd-global-intercepted-characters *Variable*

This is an alist whose value controls the characters intercepted regardless of the selected window. Its elements look and work just like those of the alist specified in the :asynchronous-characters init option for a window.

The initial value is

```
((#\terminal tv:kbd-esc)
 (#\system tv:kbd-sys))
```

Terminal and System are defined to call functions that read another character and dispatch on it. The meaning of the second character is controlled by an alist so you can define new Terminal and System commands.

tv:*escape-keys* *Variable*

The value of this variable is an alist, each entry of which describes a subcommand of the Terminal key. (Escape is the old name for the Terminal key.) Rather than modifying the list yourself, use tv:add-escape-key or tv:remove-escape-key (below). Entries on the list are of the form:

(*char function documentation option1 option2* ...)

char is the character that should be typed after Terminal to get the new command. The character gets upper-cased before it is searched for in this list, so don't use lower case characters. *function* may either be a list to be evaluated, or a symbol, which is the name of a function to be applied to one argument. This is either the numeric argument specified by the user (as in Terminal 0 S), or nil if the user gave no argument.

documentation should be a string giving documentation, or a form that gets evaluated and returns either a string or nil. The string will be printed by Terminal Help, except that nil means to omit this character from the Terminal Help display.

Normally *function* is evaluated or applied in a new process created for the purpose, but if you give the :keyboard-process option it will run in the keyboard process. This option exists because certain of the built-in commands *must* work this way. If you add your own, you should not use this option, since you do not want to interfere with the operation of the keyboard process. The cost of creating a new process is quite low.

If the :typeahead option is specified, then everything typed before the Terminal key will be shoved into the selected I/O buffer, i.e. it will be treated as typeahead to the currently selected window. Use this option with commands that change the selected window, to

ensure that the user's typed input goes where he expected it to when he typed it.

Here is a sample element:

```
(#\clear-screen
 (tv:kbd-screen-redisplay)
 "Clear and redisplay all windows.")
```

tv:add-escape-key *char function documentation* &rest *options*
Adds an element to tv:*escape-keys*, and puts it in the right place alphabetically.

tv:remove-escape-key *char*
Removes any element for *char* from tv:*escape-keys*.

tv:*system-keys* *Variable*
The value of this variable is an alist, each entry of which describes a subcommand of the System key. Use the functions tv:add-system-key and tv:remove-system-key (below) to modify the list rather than doing it yourself. Entries are of the form:

(*char find documentation create*)

char is the character that should be typed after System to get the new command. The character gets upper-cased before it is searched for in this list, so don't use lower case characters. *documentation* should be a string to be printed by System Help.

If *find* is an instance of a flavor, then it should be a window, and the System command will select that particular window. However, normally *find* is the name of a flavor. If it is, the System command first searches the previously-selected-windows list for a window of that flavor, and selects one if it finds one. Otherwise, if the currently selected window is of that flavor, it beeps. Otherwise, it looks at *create* to figure out what to do. *find* can also be a list; then it is evaluated and the value should be a window or a flavor name to be used as described above.

If *create* is nil, it beeps. If *create* is t, a new window of flavor *find* is created by calling make-instance with no options, and is selected. If *create* is some other symbol, it is the name of the flavor of window to be created. (This can be different from the flavor to look for, which might be a mixin that is component of several different flavors all of which are suitable to select when this key is typed.) Otherwise, *create* is a form to be evaluated to create a window. The System command runs in a newly-created process and so the form is evaluated in its own process, not the keyboard process.

If the character typed after the System key is typed with the Control shift, existing windows are ignored and a new window is created according to *create*.

Here is a sample element:

```
(#/E zwei:zmacs-frame "Editor" t)
```

tv:add-system-key *char find documentation* &optional (*create* t)
Adds an element to tv:*system-keys*, and puts it in the right alphabetical position.

tv:remove-system-key *char*
Removes any element for *char* from tv:*system-keys*.

tv:find-window-of-flavor *flavor-name*
Returns a previously selected window of flavor *flavor-name*. Windows are found in tv:previously-selected-windows (page 36) and checked with typep.

tv:select-or-create-window-of-flavor *flavor-name*
Selects a previously selected window of flavor *flavor-name*, or, if none exists, creates a new one and selects it.

5.6 Polling The Keyboard Explicitly

Another way of using the keyboard, different from reading a stream of input characters from a window, is to treat it as a "random access" device and look at the instantaneous state of particular keys. Spacewar does this.

tv:key-state *key-name*
Returns t if the keyboard key named *key-name* is currently depressed, nil if it is not.

key-name may be the symbolic name of a shift key, from the table below, or the character code of a non-shift key, which is the character you get when you type that key without any shifts: a lower-case letter, a digit, or a special character. Shift keys that come in pairs have three symbolic names; one for the left-hand key, one for the right-hand key, and one for both, which is considered to be depressed if either member of the pair is. The shift key names are:

:shift	:left-shift	:right-shift
:greek	:left-greek	:right-greek
:top	:left-top	:right-top
:control	:left-control	:right-control
:meta	:left-meta	:right-meta
:super	:left-super	:right-super
:hyper	:left-hyper	:right-hyper
:caps-lock	:alt-lock	:mode-lock
:repeat		

6. Output of Text

All windows can function as output streams, displaying the output as if on the screen of an ordinary display terminal. The flavor tv:minimum-window implements the operations of the Lisp Machine output stream protocol (see section 21.5 of the Lisp Machine manual), as well as many additional output operations such as :insert-line. Every window has a current *cursor position*; its main use is to say where to put characters that are drawn. The way a window handles the operations asking it to type out is by drawing that character at the cursor position, and moving the cursor position forward past the just-drawn character.

Cursor position arguments to stream operations are always expressed in "inside" coordinates (see page 129); that is, coordinates relative to the top-left corner of the inside part of the window, so the margins don't count in cursor positioning. The cursor position always stays in the inside portion of the window—never in the margins. The point (0,0) is at the top-left corner of the window; increasing x coordinates are further to the right and increasing y coordinates are further towards the bottom. (Note that y increases in the down direction, not the up direction!)

tv:cursor-x *Instance variable of windows*
tv:cursor-y *Instance variable of windows*

The window's current cursor position. Note that these variables use "outside" coordinates, unlike the arguments to stream operations.

The x cursor position is the position of the left edge of the character box of the next character output. (The leftmost nonzero pixels of the character may be either left or right of the edge of the character box, according to the left-kern of the character; see page 88).

The y cursor position is the position of the top of the vertical extent for the line being output. If only a single font is in use, the top of the character box is at this vertical position.

In fact, characters are positioned so that their baselines come out on the baseline of the line. This way, characters of different fonts juxtaposed in one line come out with baselines aligned rather than with their top edges aligned. The position of the character's baseline is a property of its font. The window's baseline is computed from the set of fonts in use, to provide enough space above the baseline for any of the fonts (see page 85).

When a character is drawn, it is combined with the existing contents of the pixels of the window according to an *alu function*. The different alu functions are described in section 8.1, page 93. When characters are drawn, the value of the window's *char-aluf* is the alu function used. Normally, the *char-aluf* says that the bits of the character should be bit-wise logically *or*ed with the existing contents of the window (tv:alu-ior). This means that if you type a character, then set the cursor position back to where it was and type out a second character, the two characters will both appear, *or*ed together one on top of the other. This is called overstriking. Erasure is also done using an alu function which the window can specify, called the *erase-aluf*. Normally this is an alu function which *and*s the old pixel value with the complement of the area erased (tv:alu-andca).

tv:char-aluf *Instance variable of windows*
tv:erase-aluf *Instance variable of windows*
The window's *char-aluf* and *erase-aluf*.

Reverse-video windows work by interchanging the normal values of the *char-aluf* and *erase-aluf*, so that erasing an area sets it to one while drawing a character clears the character's pixels to zero.

Every window has a *font map*. A font map is an array of fonts in which characters on the window can be typed. At any time, one of these is the window's *current font*; the operations that type out characters always type in the current font. Details of fonts and the font map appear below (see chapter 7, page 83). For now, we describe fonts only enough to explain the *character-width* and *line-height* of the window; these two units are used by many of the operations documented in this section. The character-width is the *char-width* attribute—the width of a "typical" character—of the first font in the font map. The line-height is the sum of the *vsp* of the window and the maximum of the *char-height*s of all the fonts. The *vsp* is an attribute of the window that controls how much vertical spacing there is between successive lines of text. That is, each line is as tall as the tallest font is, and you can add vertical spacing between lines by controlling the *vsp* of the window. Operations for controlling the *vsp* are documented on page 80. There is no instance variable holding the *vsp*, but the system can recompute it from the line-height and the font map.

tv:char-width *Instance variable of windows*
tv:line-height *Instance variable of windows*
The character-width and line-height of the window. The line height is actually used for outputting a #\return character. The character width is not used at all for ordinary output, since each font determines its own widths. Both are used for interpreting cursor positions expressed in characters or lines.

Every window has a *current font*, which the operations use to figure out what font to type in. If you are not interested in fonts, you can ignore this and something reasonable will happen. In some fonts, all characters have the same width; these are called *fixed-width fonts*. The default font is an example. In other fonts, each character has its own width; these are called *variable-width fonts*. With variable-width fonts, it is not fully meaningful to express horizontal positions in numbers of characters, since different characters have different widths. Some of the functions below do use numbers of characters to designate widths; there are warnings along with each such use explaining that the results may not be meaningful if the current font has variable width.

tv:sheet-cursor-x. *window*
tv:sheet-cursor-y *window*
tv:sheet-char-aluf *window*
tv:sheet-erase-aluf *window*
tv:sheet-char-width *window*
tv:sheet-line-height *window*
Accessor defsubsts for the corresponding instance variables. It may be reasonable to setf the first four of them.

6.1 How A Character Is Printed

Typing out a character does more than just drawing the character on the screen. The cursor position is moved to the right place; non-printing characters are dealt with reasonably; if there is an attempt to move off the right or bottom edges of the screen, the typeout wraps around appropriately; *more* breaks are caused at the right time if *more processing* is enabled. Here is the complete explanation of what typing out a character does. You may want to remind yourself how the Lisp Machine character set works; see section 21.1 of the Lisp Machine manual. You don't have to worry much about the details here, but in case you ever need to know, here they are. If you aren't interested, skip ahead to the definitions of the operations.

First, any output exceptions that are present are dealt with, and made to go away. See section 6.3, page 70, for an explanation of this.

When all exceptions have been dealt with, the character finally gets typed out. If it is a printing character, it is typed in the current font at the cursor position and the cursor position is moved to the right by the width of the character. If it is one of the format effectors #\return, #\tab, and #\backspace, it is handled in a special way to be described in a moment. All other special characters have their names typed out in tiny letters surrounded by a lozenge, and the cursor position is moved right by the width of the lozenge. If an undefined character code is typed out, it is treated like a special character; its code number is displayed in a lozenge.

#\tab moves the cursor position to the right to the next tab stop, moving at least one character-width. Tab stops are equally spaced across the window. The distance between tab stops is *tab-nchars* times the *character-width* of the window. *tab-nchars* defaults to 8 but can be changed (see page 81).

Normally #\return moves the cursor position to the inside left edge of the window and down by one line-height, and clears the line (see page 75). It also deals with more processing and the end-of-page condition as described above. However, if the window's *cr-not-newline-flag* is on, the #\return character is not regarded as a format effector and is displayed as "return" in a lozenge, like other special characters.

If the character being typed out is a #\backspace, the result depends on the value of the window's *backspace-not-overprinting-flag*. If the flag is 0, as is the default, the cursor position is moved left by one character-width (or to the inside left edge, whichever is closer). If the flag is 1, #\backspaces are treated like all other special characters.

6.2 Stream Output Operations

:tyo *ch* &optional *font* *Operation on windows*

Type *ch* on the window, as described above. Basically, type the character *ch* in *font* or the current font at the cursor position, and advance the cursor position.

:string-out *string* &optional (*start* 0) (*end* nil) *Operation on windows*

Type *string* on the window, starting at the character *start* and ending with the character *end*. If *end* is nil, continue to the end of the string; if neither optional argument is given, the entire string is typed. This behaves exactly as if each character in the string (or the specified substring) were printed with the :tyo operation, but it is much faster.

:fat-string-out *string* &optional (*start* 0) (*end* nil) *Operation on windows*

Type the fat string *string* on the window. This is like :string-out except that the %%ch-font field of each character is used as the font to draw that character in. The window's current font is not used.

:line-out *string* &optional (*start* 0) (*end* nil) *Operation on windows*

Do the same thing as :string-out, and then advance to the next line (like typing a #\return character). The main reason that this operation exists is so that the stream-copy-until-eof function (see section 21.4 of the Lisp Machine manual) can, under some conditions, move whole lines from one stream to another; this is more efficient than moving characters singly. The behavior of this operation is not affected by the :cr-not-newline-flag init-option (see page 81).

:string-out-centered *string left right y-pos* *Operation on windows*

Output *string* (or the portion from *start* to *end*), centered between *x* positions *left* and *right*, at *y* position *y-pos* (which defaults to the current cursor position). The cursor is left at the end of the string. If the string is multiple lines, the entire rectangular shape it occupies is centered as a unit. To center lines individually, output each line individually with this operation.

:fresh-line *Operation on windows*

Get the cursor position to the beginning of a blank line. Do this in one of two ways. If the cursor is already at the beginning of a line (that is, at the inside left edge of the window), clear the line to make sure it is blank and leave the cursor where it was. Otherwise, advance the cursor to the next line and clear the line just as if a #\return had been output. The behavior of this operation is not affected by the :cr-not-newline-flag init-option (see page 81).

:beep &optional *beep-type* *Operation on windows*

Attempt to attract the user's attention, by either making a sound with the keyboard or flashing the screen into and out of inverse video or both.

If beep's value is nil, both are done. If the value is :beep, only the sound is made. If it is :flash, only flashing the screen is done.

No standard meanings have been assigned to *beep-type* yet.

beep &optional *beep-type* (*stream* standard-output)

Beeps by sending a :beep message to *stream*, passing *beep-type* as an argument. If the stream does not handle the :beep operation, a sound is made on the keyboard instead.

:display-lozenged-string *string* *Operation on windows*

Output *string* in a lozenge. This is how special characters are echoed.

tv:sheet-line-out *sheet string start end set-xpos set-ypos dwidth*

This is a complicated primitive whose interface is arranged to do exactly what the editor needs for buffer display, to make the editor as fast as possible.

It outputs part of *string* on *sheet* like the :fat-string-out operation, but stops if it reaches the right margin (outputting a right margin character if any output remains, if the window calls for that).

If *set-xpos* and *set-ypos* are non-nil, the cursor is moved there and a :clear-eol is done, before output starts. If one of these arguments is nil, that dimension of cursor position is not changed. If both are nil, the cursor is not moved and nothing is cleared.

If *dwidth* is non-nil, it should be a positive number. Output actually starts at index (1- *start*) in the string, and at *x* position *dwidth* less that the cursor position (as found or as set by *set-xpos*). However, if a :clear-eol is done, it starts at *set-xpos*. Non-nil *dwidth* is to be used if the previous character of the string is in an italic font, and is already present on the screen before the output now being done. It causes that character to be output again, presumably overprinting itself, in case a corner of it was erased accidentally because it protrudes to the right of its allocated space.

Returns two values, the final index in the string and the final *x* cursor position. The window's cursor is not guaranteed to be moved there; it is undefined on exit from this function. But the value will be correct.

6.3 Output Exceptions

Before doing output to a window, various exceptional conditions are checked for. If an exceptional condition is discovered, a standard operation is invoked to handle it. Redefining or adding daemons to these operations can change the handling of exceptions. For example, output with the cursor too close to the right margin causes an end of line exception; the handling of this exception is what moves the cursor to the next line, or truncates the line, or whatever the window's flavor arranges for.

The exceptions are actually indicated by flags, bits, set in the window. The operation to handle the exception should do nothing if it is invoked when the corresponding flag is not set, and should not return with the flag still set (or an error will be signaled). The end-of-page and more flags are set and cleared automatically by moving the cursor; as long as things are done properly, they will be set if and only if the cursor is in the right place for them. So the exception handler need only make sure to move the cursor to a good place. The output hold exception handler usually just waits for or brings about a situation in which the reason for the output hold is gone (usually because the window has been exposed).

:handle-exceptions *Operation on windows*

Performs the exception processing described by all the rest of this section. Exceptions are processed in this order:
Output Hold, End-of-Page, **MORE**, and End-of-Line.

6.3.1 Output Hold and End of Page Exceptions

First, if the window's output hold flag is set, an output hold exception happens. The operation :output-hold-exception is invoked to handle it.

tv:sheet-output-hold-flag *window*

Returns the output hold flag of *window*, which is 1 if there is a hold and 0 if not. This is a setf'able accessor defsubst.

:output-hold-exception *Operation on windows*

This operation should not return until the output hold is gone. It may wait for the output hold flag to be cleared, or try to cause it to be cleared. The default handler acts based according to the window's deexposed typeout action (see page 21).

Next, if the end-of-page flag is set (normally the case if the *y*-position of the cursor is less than one line-height above the inside bottom edge of the window), the :end-of-page-exception operation is invoked.

tv:sheet-end-page-flag *window*

Returns the end-of-page flag of *window*, which is 1 if the next output operation should wrap and 0 otherwise. This is a setf'able accessor defsubst.

:end-of-page-exception *Operation on windows*

This operation is invoked to handle the end-of-page exception when present. It should do nothing if invoked when the flag is zero.

The default definition is simply to move the cursor to the top line, clear that line, and set the vertical position for the next **MORE** if more-processing is enabled.

6.3.2 **MORE** Exceptions

Next, if the window's *more flag* is set, a *more exception* happens. The more flag gets set when the cursor is moved to a new line (e.g. when a #\return is typed) and the cursor position is thus made to be below the *more vpos* of the window. (If tv:more-processing-global-enable is nil, this exception is suppressed and the more flag is turned off.) The :more-exception operation is invoked to handle the exception.

tv:sheet-more-flag *window*

Returns the more flag, which is 1 if the next output operation should do a **MORE**, and 0 otherwise. This is a setf'able accessor defsubst.

tv:more-vpos *Instance variable of windows*

The vertical position at which the next **MORE** should happen in output on the window.

:more-vpos *Operation on windows*

Returns the window's tv:more-vpos.

tv:sheet-more-vpos *window*

Accessor defsubst for the preceding instance variable.

tv:more-processing-global-enable *Variable*

MORE processing does not happen if this variable is nil during the output operation in which the **MORE** would have happened.

:more-exception *Operation on windows*

The :more-exception handler in the tv:minimum-window flavor does a :clear-eol operation, types out **MORE**, reads a character using the :more-tyi operation, restores the cursor position to where it originally was when the :more-exception was detected, does another :clear-eol to wipe out the **MORE**, and resets the more vpos. The character read in is ignored.

This operation works by calling a subroutine, tv:sheet-more-handler, if the more flag is set. It should do nothing if the flag is zero. It is safe to redefine it to call that function with different arguments, or to do other things as well. It is very risky to write a new definition from scratch, as tv:sheet-more-handler is tricky.

tv:sheet-more-handler &optional (*operation* ':tyi) (*more-string* "**MORE**")

Implements the standard handling of *more* exceptions, described above, using *operation* to read the input and *more-string* as the output to be printed and then erased.

Note that the more flag is set only when the cursor moves to the next line, because a #\return is typed out, after a :line-out, or by the :end-of-line-exception handler described below. It is not set when the cursor position of the window is explicitly set (e.g. with :set-cursorpos); in fact, explicitly setting the cursor position clears the more flag. The idea is that when typeout is being streamed out sequentially to the window, *more*-exceptions happen at the right times to give the user a pause in which to read the text that is being typed, but when cursor positioning is being used the system cannot guess what order the user is reading things in and when (if ever) is the right time to stop. In this case it is up to the application program to provide any necessary pauses.

The algorithm for setting the more vpos is too complicated to go into here in all its detail, and you don't need to know exactly how it works, anyway. It is careful never to overwrite something before you have had a chance to read it, and it tries to do a **MORE** only if a lot of output is happening. But if output starts happening near the bottom of the window, there is no way to tell whether it will just be a little output or a lot of output. If there's just a little, you would not want to be bothered by a **MORE**. So it doesn't do one immediately. This may make it necessary to cause a **MORE** break somewhere other than at the bottom of the window. But as more output happens, the position of successive **MORE**s is migrated and eventually it ends up at the bottom.

tv:autoexposing-more-mixin *Flavor*

If you mix in this flavor, when a :more-exception happens, the window will be exposed (an :expose message will be sent to it). This is intended to be used in conjunction with having a deexposed typeout action of :permit (see page 22), so that a process can type out on a deexposed window and then have the window expose itself when a **MORE** break happens.

6.3.3 End of Line Exceptions

Finally, if the cursor is at or near the end of the line so that there is no room to output the next character, an end-of-line exception happens. The :end-of-line-exception operation is invoked to handle it. A flag is not used to trigger this exception since the condition depends on the width of the character to be output.

:end-of-line-exception *Operation on windows*

This operation is defined by default to advance the cursor to the next line, just as typing a #\return character does normally (see below). Doing so may, in turn, cause an :end-of-page-exception or a :more-exception to happen. Furthermore, if the *right margin character flag* is on (see page 81), then before going to the next line, an exclamation point in font zero is typed at the cursor position. When this flag is on, end-of-line exceptions are caused a little bit earlier, to make room for the exclamation point.

:tyo-right-margin-character *Operation on windows*

If a right-margin character is to be printed, this operation is invoked to print it. It can simply :tyo the character.

The way the cursor position goes to the next line when it reaches the right edge of the window is called *horizontal wraparound* or *continuation*. You can make windows that truncate lines instead of wrapping them around by using tv:line-truncating-mixin.

tv:line-truncating-mixin *Flavor*

This mixin gives a window the ability to truncate lines at the right margin instead of continuing output onto the next line as usual (see continuation, page 73). Truncation is performed if the window's truncate-line-out flag is set. When the cursor position is near the right-hand edge of the window and there is an attempt to type out a character, the character simply will not be typed out.

:truncate-line-out-flag *flag* *Init option for* tv:line-truncating-mixin

Initializes the truncate-line-out flag of the window to *flag*. One means truncate and zero means do not.

tv:sheet-truncate-line-out-flag *window*

Returns the truncate-line-out flag of the window, which is zero or one. One means truncate and zero means do not; however, the flag matters only if tv:line-truncating-mixin is in use. This is a defsubst which may be setf'd.

tv:truncating-window *Flavor*

This flavor is built on tv:window with tv:line-truncating-mixin mixed in. If you instantiate a window of this flavor, it will be like regular windows of flavor tv:window except that lines will be truncated instead of wrapping around.

6.4 Cursor Motion

The window's cursor position is where the upper left corner of the next output character will appear, with a vertical offset if necessary to match up the baselines of various fonts (see page 87). Recall that cursor position arguments and values of stream operations are relative to the inside upper left corner of the window.

:read-cursorpos &optional (*units* ':pixel) *Operation on windows*

Return two values: the *x* and *y* coordinates of the cursor position. These coordinates are in pixels by default, but if *units* is :character, the coordinates are given in character-widths and line-heights. (Note that character-widths don't mean much when you are using variable-width fonts.)

:increment-cursorpos *x y* &optional (*units* ':pixel) *Operation on windows*

Advances the cursor position the specified amount in each coordinate. The units may be specified as with :read-cursorpos. This operation is considered to be sequential motion of the cursor through a variable amount of space, rather than instantaneous jumping of the cursor. What this means is that exceptions happen, just as if output were being done. So the cursor wraps around at the margins (or does whatever this window does for :end-of-line-exception and :end-of-page-exception), and **MORE** processing happens at the appropriate place.

The following few operations do cursor *motion* rather than *advancing* the cursor. The end-of-page, *more* and end-of-line exception flags will be set if the cursor is moved to a position where they ought to be on, and can be cleared if they were previously on and the cursor is moved to a place where they ought to be off. Exception handling does not take place.

:set-cursorpos *x y* &optional (*units* ':pixel) *Operation on windows*

Moves the cursor position to the specified coordinates. The units may be specified as with :read-cursorpos. If the coordinates are outside the window, move the cursor position to the nearest place to the specified coordinates that is in the window.

:home-cursor *Operation on windows*

Moves the cursor to the upper left corner of the window.

:home-down *Operation on windows*

Moves the cursor to the lower left corner of the window.

:forward-char &optional ***char*** *Operation on windows*

Moves the cursor forward one character position, or the width of *char* in the current font if *char* is specified. Exceptions are processed, so this is like outputting a space which has the appropriate width.

:backward-char &optional *char* *Operation on windows*

Moves the cursor backward one character position, or the width of *char* in the current font if *char* is specified. Exceptions are processed, but there is no reverse-wraparound. At the left margin, the cursor does not move.

:size-in-characters *Operation on windows*

Returns two values, the dimensions of the window, in units of character-widths and line-heights. (Note that character-widths don't mean much when you are using variable-width fonts.)

:set-size-in-characters *Operation on windows*

width-spec height-spec &optional *option*

Sets the inside size of the window, according to the two specifications, without changing the position of the upper-left corner. *width-spec* and *height-spec* are interpreted the same way as arguments to the :character-width and :character-height init-options, respectively. *option* is passed along to :set-edges (page 46).

6.5 Erasing

All the erasing operations operate on the window pixels by drawing the area to be erased using the window's *erase-aluf* as the alu function (see page 67). This is by default tv:alu-andca, which clears the screen bits of the screen area drawn.

:clear-char &optional *char* *Operation on windows*

Erases the character at the current cursor position. When using variable-width fonts, you tell it the character code of the character you are erasing, so that it will know how wide the character is (it assumes the character is in the current font). If you don't pass the *char* argument, it simply erases a character-width, which is fine for fixed-width fonts.

:clear-string *string* &optional *start end* *Operation on windows*

Erases enough space, starting at the cursor, to contain *string* (or the portion of *string* from *start* to *end*), printed in the current font. The entire height of the line is erased, so it does not matter whether the text on the screen is *string* or something else. *string* determines only how far to erase. If a fixed-width font is in use, this is equivalent to doing :clear-char once for each character in *string*. This operation becomes desirable because of variable-width fonts.

:clear-eol *Operation on windows*

Erases from the current cursor position to the end of the current line; that is, erases a rectangle horizontally from the cursor position to the inside right edge of the window, and vertically from the cursor position to one line-height below the cursor position.

:clear-eof *Operation on windows*

Erases from the current cursor position to the bottom of the window. In more detail, first does a :clear-eol, and then clears all of the window past the current line.

:clear-screen *Operation on windows*

Erases the whole window and moves the cursor position to the upper left corner of the window.

:clear-between-cursorposes *start-x start-y end-x end-y* *Operation on windows*

Erases an area starting at cursor position *start-x* and *start-y*, wrapping around if necessary at the end of the line or the page, until *end-x* and *end-y* are reached.

Though the arguments are expressed as cursor positions, the cursor position of the window is not changed.

6.6 Inserting and Deleting Lines and Characters

Inserting a character means printing it at the cursor but pushing the rest of the text on the line toward the right margin. Similarly, deleting a character means pulling the following text on the line back toward the left so that the position occupied by the character is closed up. Inserting and deleting lines work the same way vertically, moving the lines below the cursor down or up.

The operations that take a numeric argument specifying the amount of space to insert or delete also take an argument specifying the unit (either :pixel or :character) in which the space has been measured. The unit argument's meaning is the same as in the :read-cursorpos operation (page 74) but the default is :character rather than :pixel.

:delete-char &optional (*n* 1) (*unit* ':character) *Operation on windows*

Without an argument, deletes the character at the current cursor position. Otherwise, deletes *n* characters (or *n* pixels if *unit* is :pixel), starting at the cursor position. Move the display of the part of the current line that is to the right of the deleted section leftwards to close the resultant gap. (If *unit* is :character, this assumes all characters are one character-width wide, and so will not do anything useful with variable-width fonts.)

:delete-string *string* &optional (*start* 0) (*end* nil) *Operation on windows*

This is for deleting specific strings in the current font. It is one of the things to use when dealing with variable-width fonts.

If *string* is a string, excise a region exactly as wide as that string, or a substring specified by *start* and *end*, and moves the display of the part of the current line that is to the right of the excised region leftwards to close the gap.

If *string* is a number, it is considered to be a character code. The single character is treated like a string containing that character.

:delete-line &optional (*n* 1) (*unit* ':character) *Operation on windows*

Without an argument, deletes the line that the cursor is on. Otherwise deletes *n* lines, or *n* rows of pixels if *unit* is :pixel, starting with the one the cursor is on. Moves the display below the deleted section up to close the resulting gap.

:insert-char &optional (*n* 1) (*unit* ':character) *Operation on windows*

Opens up a space the width of *n* characters (or *n* pixels if *unit* is :pixel) in the current line at the current cursor position. Shifts the characters to the right of the cursor further to the right to make room. Characters pushed past the right-hand edge of the window are lost. (If *unit* is :character, this assumes all characters are one character-width wide, and so will not do anything useful with variable-width fonts.)

:insert-string *Operation on windows*

string &optional (*start* 0) (*end* nil) (*type-too* t)

Inserts a string at the current cursor position, moving the rest of the line to the right to make room for it.

The string to insert is specified by *string*; a substring thereof may be specified with *start* and *end*, as with :string-out.

string may also be a number, in which case the character with that code is inserted.

If *type-too* is specified as nil, the string is not actually printed. The space opened up is big enough for the string, but is left blank.

:insert-line &optional (*n* 1) (*unit* ':character) *Operation on windows*

Takes the line containing the cursor and all the lines below it, and moves them down one line. The line containing the cursor is moved in its entirety, not broken, no matter where the cursor is on the line. A blank line is created at the cursor. If an argument *n* is given, opens up *n* blank lines, or *n* rows of pixels if *unit* is :pixel. Lines pushed off the bottom of the window are lost.

6.7 Anticipating the Effect of Output

The following operations do not output, but provide information about what would happen to the cursor and the screen if output were done.

:character-width *char* &optional (*font* tv:current-font) *Operation on windows*

Returns the width of the character *char*, in pixels. The current font is used if *font* is not specified. If *char* is a Backspace, :character-width can return a negative number. For Tab, the number returned depends on the current cursor position. If *char* is Return, the result is defined to be zero.

:compute-motion *Operation on windows*

string &optional (*start* 0) (*end* nil) (*x* tv:cursor-x) (*y* tv:cursor-y) (*cr-at-end-p* nil) (*stop-x* 0) *stop-y bottom-limit right-limit font line-height tab-width*

This is used to figure out where the cursor would end up if you were to output *string* using :string-out. It does the right thing if you give it just the string as an argument. *start* and *end* can be used to specify a substring as with :string-out. *x* and *y* can be used to start your imaginary cursor at some point other than the present position of the real cursor. If you specify *cr-at-end-p* as t, it pretends to do a :line-out instead of a :string-out. *stop-x* and *stop-y* define the size of the imaginary window in which the string is

being printed; the printing stops if the cursor becomes simultaneously ≥ both of them. These default to the lower left-hand corner of the window. (This corner is reached before the right-hand one, since output goes from left to right on each line.)

bottom-limit and *right-limit* are vertical and horizontal positions at which to wrap around; they default to the inside height and width of the window. They differ from the *stop-x* and *stop-y* in that these act independently when the cursor reaches either one, and they cause the cursor position to change rather than returning to the caller.

The computation normally uses *font*, or the window's current font if *font* is nil. However, if *string* is of type art-fat-string, each character's %%ch-font field is used as an index in the window's font map to find the font for that character, and *font* is ignored except possibly for defaulting the *tab-width*.

For vertical spacing, *line-height* is used. The default for *line-height* is *font*'s line height if *font* is non-nil, else the window's line-height.

tab-width specifies the distance between tab stops, in pixels. If it is omitted, the default is (tv:sheet-tab-width self) if no font is specified, or (* (tv:sheet-tab-nchars self) (tv:font-char-width *font*)) if a font is specified.

Four values are returned:

final-x
final-y The positions at which output stopped.

final-index The index in *string* at which output stopped, or nil if it reached the end of the string, or *t* if the string itself was processed but not the implicit Return which was supposed to follow the string.

maximum-x The largest *x* position value reached during processing.

All coordinates for this operation are cursor positions, relative to the window's inside edges. However, if you specify all the arguments you can use any origin of coordinate system you like, as long as you interpret the values in the same coordinate system.

:string-length *Operation on windows*

string &optional (*start* 0) (*end* nil) *stop-x* (*font* current-font) (*start-x* 0) *tab-width*

This is very much like :compute-motion, but works in only one dimension. It tells you how far the cursor would move if *string* were to be displayed in the current font starting at the left margin, or at *start-x* if that is specified. *start* and *end* work as with :string-out to specify a substring of *string*. If *stop-x* is not specified or nil, the window is assumed to have infinite width; otherwise the simulated display will stop when a position *stop-x* pixels from the left edge is reached.

The computation normally uses *font*, or the window's current font if *font* is nil. However, if *string* is of type art-fat-string, each character's %%ch-font field is used as an index in the window's font map to find the font for that character, and *font* is ignored except possibly for defaulting the *tab-width*.

tab-width specifies the distance between tab stops, in pixels. If it is omitted, the default is (tv:sheet-tab-width self) if no font is specified, or (* (tv:sheet-tab-nchars self) (tv:font-char-width *font*)) if a font is specified.

:string-length returns three values:

final-x Where the imaginary cursor ended up.

final-index The index of the next character in the string (the length of the string if the whole string was processed, or the index of the character which would have moved the cursor past *stop-x*),

maximum-x The maximum *x* coordinate reached by the cursor (this is the same as the first value unless there are Return or Backspace characters in the string).

6.8 Explicit (Non-Cursor) Output

A window includes some state information which changes as output is done. These include the cursor position, the current font, alu function, and exception flags. The presence of this information makes the window behave coherently as a stream, so that the output from one operation follows that of the previous operation. But sometimes this is not desirable. The "explicit" output operations use a window only for its position and size, with all additional information passed by the caller explicitly. This way, multiple streams of output to the same window can exist, which do not interfere with each other by trying to use a single cursor.

The *x* and *y* position arguments used by these operations are relative to the outside edges of the window. This is different from the stream and higher-level operations. It is because these operations are frequently used for drawing parts of the margins, such as labels and margin regions.

:string-out-explicit *Operation on windows*
string start-x start-y x-limit y-limit font alu &optional (*start* 0) *end multi-line-line-height*

Outputs *string* (or the portion from *start* to *end*) onto the window starting at *start-x* and *start-y*, neither using nor moving the window's cursor position. If *x-limit* or *y-limit* is non-*nil*, output stops if it reaches that position.

Output is done in *font* using alu function *alu*. The window's current font and alu function are not used or set. If there are Return characters in the output, and *multi-line-line-height* is nil, they are printed as "Return" in a lozenge. If *multi-line-line-height* is a number, that number is used as the line height, ignoring the window's line height, and the horizontal output position moves to *start-x* rather than the left margin for the next line of output.

Note that the arguments of tv:sheet-string-out-explicit are in a different order. The argument order of this operation was cleaned up.

The operation returns three values: the final *x* position, the final *y* position, and the final index in the string. You can use these to do multiple operations in consecutive places on the screen.

:string-out-centered-explicit *Operation on windows*
string &optional *left y-pos right y-limit font alu* (*start* 0) *end multi-line-line-height*

Outputs *string* (or the portion from *start* to *end*) centered between *x* positions *left* and *right*, at *y* position *y-pos*. If *y-limit* is reached, output stops. *left* and *right* default to the inside edges of the window.

Output is done in *font* and *alu*, which default to the ones current for the window, and lines are separated by *multi-line-line-height* (which defaults to the window's line height).

:string-out-x-y-centered-explicit *Operation on windows*
string &optional *left top right bottom font alu start end multi-line-line-height*

Displays *string* (or the portion from *start* to *end*) with the rectangle it occupies centered both horizontally and vertically. Horizontally it is centered between *left* and *right*, and vertically between *top* and *bottom*. The defaults for these arguments are the inside edges of the window.

Output is done in *font* and *alu*, which default to the ones current for the window, and lines are separated by *multi-line-line-height* (which defaults to the window's line height).

6.9 Window Parameters Affecting Output

The following operations and initialization options initialize, get, and set various window attributes which are relevant to the typing out of characters. (See also the operations to manipulate the current font, on page 84.)

:more-p *t-or-nil* *Init option for windows*
Initializes whether the window should have more processing. It defaults to t.

:more-p *Operation on windows*
Returns t if more processing (see page 71) is enabled; otherwise, return nil.

:set-more-p *more-p* *Operation on windows*
If *more-p* is nil, turns off more processing (see page 71); otherwise turns it on.

:vsp *n-pixels* *Init option for windows*
Initializes the window's *vsp*. It defaults to 2.

:vsp *Operation on windows*
Returns the value of *vsp* for this window (see page 67).

:set-vsp *new-vsp* *Operation on windows*
Sets the value of *vsp* for this window (see page 67) to *new-vsp*.

:reverse-video-p *Operation on windows*

Returns **nil** normally or **t** if the window displays in white on black rather than black on white. This is separate from the whole screen's inverse video mode, which is what Terminal C sets.

:set-reverse-video-p *t-or-nil* *Operation on windows*

Enables or disables reverse-video display. Changing this mode inverts all of the bits in the window.

:reverse-video-p *t-or-nil* *Init option for windows*

Initializes the use of reverse-video display.

:right-margin-character-flag *x* *Init option for windows*

If *x* is 1, the window should print an exclamation point in the right margin when :end-of-line-exception happens; if *x* is 0, it should not. The default is 0. See page 73.

tv:sheet-right-margin-character-flag &optional (*window* **self**)

Returns the flag which controls printing of characters at the right margin on wrap-around on *window*. This is a setf'able accessor macro.

:backspace-not-overprinting-flag *x* *Init option for windows*

If *x* is 0, output of #\backspace will move the cursor position backward; if it is 1, it will display "overstrike" in a lozenge (that is, #\backspace will be just like other special characters). The default is 0. See page 68.

tv:sheet-backspace-not-overprinting-flag &optional (*window* **self**)

Returns the flag which controls how Backspace prints on *window*. This is a setf'able accessor macro.

:cr-not-newline-flag *x* *Init option for windows*

If *x* is 0, output of #\return will move the cursor position to the beginning of the next line and clear that line; if it is 1, it will display "return" in a lozenge (that is, #\return will be just like other special characters). The default is 0. This flag does not affect the behavior of the :line-out nor the :fresh-line operations.

tv:sheet-cr-not-newline-flag &optional (*window* **self**)

Returns the flag which controls how Return prints on *window*. This is a setf'able accessor macro.

:tab-nchars *n* *Init option for windows*

n is the separation of tab stops on this window, in units of the window's char-width. This controls how the #\tab character prints. *n* defaults to 8.

tv:sheet-tab-nchars &optional (*window* **self**)

Returns the distance between tab stops, measured in units of *window*'s char-width.

tv:sheet-tab-width &optional (*window* self)

Returns the distance between tab stops, measured in pixels.

7. Fonts

Having used the Lisp Machine for a while, you have probably noticed that characters can be typed out in any of a number of different typefaces. Some text is printed in characters that are small or large, boldface or italic, or in different styles altogether. Each such type face is called a *font*. A font is conceptually an array, indexed by character code, of pictures showing how each character should be drawn on the screen.

A font is represented inside the Lisp Machine as a Lisp object. Each font has a name. The name of a font is a symbol, usually in the fonts package, and the symbol is bound to the font. A typical font name is tr8. In the initial Lisp environment, the symbol fonts:tr8 is bound to a font object whose printed representation is something like

```
#<font tr8 234712342>
```

The initial Lisp environment includes many fonts. Usually there are more fonts stored in QFASL files in file computers. New fonts can be created, saved in QFASL files, and loaded into the Lisp environment; they can also simply be created inside the environment.

Drawing of characters in fonts is done by microcode and is very fast. The internal format of fonts is arranged to make this drawing as fast as possible. This format is described later, but you almost certainly do not need to worry about it.

7.1 Specifying Fonts

You can control which font is used when output is done to a window. Every window has a *font map* and a *current font*. The font map is conceptually an array of fonts; with a small non-negative number, the font map associates a font. The current font of a window is always one of the fonts in the window's font map. Whenever output is done to a window, the characters are printed in the current font. You can change the font map and the current font of a window at any time with the appropriate operations.

The reason why the window has a font map rather than merely a current font is that it is necessary to know all the fonts that will be used before doing any output in order to know how to position the output properly (so that output in different fonts on the same line will look right).

In addition, certain output operations can accept fat strings (arrays of type art-fat-string) which contain 16-bit characters, and regard the top 8 bits of each character as a font number to look up in the font map. These include :compute-motion, :string-length and :fat-string-out.

:font-map *Operation on windows*

Returns the font map of the window. The object returned is the array that is actually being used to represent the font map inside the window. The elements are actual font objects.

You should not alter anything about this array, since the window depends on it in order to function correctly. To change the font map, use the :set-font-map operation.

:set-font-map *new-map* *Operation on windows*

Sets the font map to contain the fonts given in *new-map*. Returns the array of fonts that actually represents the font map inside the window (don't mess with this array!). *new-map* may be an array of font specifiers, in which case this array is installed as the new internal array of the window, and the font specifiers are replaced by fonts. Font specifiers are described in the following section; a font or the name of a font may be used.

new-map may also be a list of font specifiers, in which case the array is created from the list in the style of fillarray, with the last element of the list filling in the remaining elements of the array if any (the array is made at least 26. elements long, or long enough to hold all the elements of the list).

If *new-map* is nil, all the elements of the map are set to the default font of the screen.

The current font is set to zero (the first font in the list or array). The line height and baseline of the window are adjusted appropriately (see below).

The specified font specifiers are remembered so that the :change-of-default-font operation can cause the map to be recomputed from them. This is in case one of the specifiers is a purpose keyword.

tv:font-map *new-map* *Init option for windows*

This option lets you initialize the font map. *new-map* is interpreted the same way it is interpreted by the :set-font-map operation.

tv:font-map *Instance variable of windows*

The window's font map.

:current-font *Operation on windows*

Returns the current font, as a font object.

:set-current-font *new-font* *Operation on windows*

Sets the current font of the window. *new-font* may be a number, in which case that element of the font map becomes the current font. It may also be a font specifier, in which case the font that the specifier describes is used, unless that font is not in the font map, in which case an error is signalled. Only fonts already in the font map may be selected.

tv:current-font *Instance variable of windows*

The window's current font.

:baseline *Operation on windows*

Returns the maximum baseline of all the fonts in the font map. The bases of all characters will be aligned so as to be this many pixels below the *y* cursor position, which is top of the line on which the characters are printed. In other words, when a character is drawn, it will be drawn below the cursor position, by an amount equal to the difference between this number and the baseline of the font of the character.

tv:baseline *Instance variable of windows*

The position of the baseline of a text line, in pixels from the top of the line's vertical extent (its cursor position).

tv:sheet-font-map *window*
tv:sheet-baseline *window*
tv:sheet-current-font *window*

Accessor defsubsts for the corresponding instance variables.

You can use the List Fonts command in Zmacs to get a list of all of the fonts that are currently loaded into the Lisp environment. Here is a list of some of the useful fonts:

fonts:cptfont This is the default font, used for almost everything.

fonts:medfnt This is the default font in menus. It it a fixed-width font with characters somewhat larger than those of cptfont.

fonts:medfnb This is a bold version of medfnt. When you use Split Screen, for example, the Do It and Abort items are in this font.

fonts:hl12i This is a variable-width italic font. It is useful for italic items in menus; ZMail uses it for this in several menus.

fonts:tr10i This is a very small italic font. It is the one used by the inspector to say "*More above*" and "*More below*".

fonts:hl10 This is a very small font used for non-selected items in Choose Variable Values windows.

fonts:hl10b This is a bold version of hl10, used for selected items in Choose Variable Values windows.

7.1.1 Font Specifiers

Different kinds of screen require different kinds of fonts. The two kinds of screens currently supported are black-and-white screens with one bit per pixel, and color screens with four bits per pixel. Color screens with eight bits per pixel will certainly be supported in the near future, and other kinds of screen may appear. However, it is nice to be able to write programs that will work no matter what screen their window is created on. The problem is that if your program specifies which fonts to use by actually naming specific fonts, then the program will only work if the window that you are using is on the same kind of screen as the fonts you are using were designed for.

To solve this problem, a program does not have to specify the actual font to be used. Instead, it specifies a certain symbol that stands for a whole collection of fonts. All of these fonts are the same except that they work on different kinds of screens. The symbol that you use is the name of the member of the collection that works on the black-and-white screen. In other words, when you want to specify a font, always use the name of a black-and-white font rather than a font itself. Every screen knows how to understand these symbols and find an appropriate font to use. This symbol is called a *font specifier*, because it describes a font rather than actually being a font.

A font object may be supplied as a font specifier. This does not mean to use the font as specified; it means to use the font's name as a font specifier. Thus, if you supply the font object for the black-and-white font cptfont for a window on a color screen, the symbol fonts:cptfont is used as a font specifier, resulting in the color version of cptfont actually being used.

The functions that understand font specifiers have some cleverness in order to make life easier for you. If you pass in the name of a font that is not loaded into the Lisp environment, an attempt will be made to load it from the file server, using the name of the font as the name of the file, leaving the version and type unspecified, using the load function. The filename used is SYS: FONTS; *fontname* QFASL. Also, the color screen knows how to create color versions of fonts on the fly if they do not already exist. Either of these things may make your program run slowly the first time you run it, and so, if you care, you can load the file yourself and create a color version of the font yourself (see page 167).

Since different users like to use different fonts, we provide a facility called *font purposes*. Wherever a font specifier is used, the program can specify a *purpose* keyword instead. This means, "use whatever font the user likes to use for this particular purpose". The window remembers when a purpose was specified instead of a particular font, so that if the user changes the standard font for that purpose, all the existing windows that were told to use that purpose will change font. The user specifies a standard font for a purpose with tv:set-default-font, tv:set-standard-font or tv:set-screen-standard-font. Each screen has its own alist mapping font purposes to font names, but normally they are all altered in parallel. Defined purpose keywords include

:default This is the font name for ordinary output. It is also called the *default font*.

:menu This is the font name for use in most menu items.

:menu-standout
This is the font name for menu items that are supposed to stand out. It is normally an italic font.

:label This is the font name used by default for labels.

:margin-choice
This is the default font name for margin choice boxes (see page 210).

It is up to each program to decide when any of these purpose keywords is appropriate.

:parse-font-specifier *font-specifier* *Operation on* tv:screen
Parses a font specifier in the proper way for this window, according to the screen the window is on. The value is a font object.

:parse-font-name *font-specifier* *Operation on* tv:screen
Parses a font specifier in the proper way for this window, according to the screen the window is on. The value is a font name: a symbol which, evaluated repeatedly, ultimately produces a font.

tv:font-evaluate *font-name*
Returns the font that *font-name* is the name of; this is done by evaluating *font-name* repeatedly until the result is not a symbol.

tv:set-standard-font *purpose font-specifier*
Sets the standard font for purpose *purpose* on each screen based on *font-specifier*. *font-specifier* is turned into a font by each screen individually, and that font becomes the new standard font for *purpose* on that screen. All windows on the screen that were set up to use the standard font for this purpose will switch to using the newly specified font.

tv:set-default-font *font-specifier*
Sets the standard font for purpose :default.

tv:set-screen-standard-font *screen purpose font-specifier*
Sets the standard font for *purpose* on *screen* only.

:change-of-default-font *old-font new-font* *Operation on windows*
Informs the window that the meaning of some standard font-name symbols has changed. If the window uses any of them, it may need to recompute various things; for example, if that font is used in the label, the window's inside size may be changed; if it is used in the window's font map, the line height may be changed. Either one means the number of lines may change, and this may require adjustment of other data. This can be done by an :after daemon on this operation.

In addition, the operation must be passed along to all inferiors and potential inferiors.

7.2 Attributes of Fonts

Fonts, and characters in fonts, have several interesting attributes. One attribute of each font is its *character height*. This is a non-negative fixnum used to figure out how tall to make the lines in a window. We have mentioned earlier that each window has a certain *line height*. The line height is computed by examining each font in the font map, and finding the one with the largest character height. This largest character height is added to the *vsp* specified for the window (see page 67), and the sum is the line height of the window. The line height, therefore, is recomputed every time the font map is changed or the *vsp* is set. It works this way so that there will always be enough room on any line for the largest character of the largest font to be displayed, and still leave the specified vertical spacing between lines. One effect of this is that if you have a window that has two fonts, one large and one small, and you do output in only the small font, the lines will still be spaced far enough apart that characters from the large font will fit. This is because the window system can't predict when you might, in the middle of a line, suddenly switch to the large font.

Another attribute of a font is its *baseline*. The baseline is a non-negative fixnum that is the number of raster lines between the top of each character and the base of the character. (The "base" is usually the lowest point in the character, except for letters that descend below the baseline such as lower case "p" and "g".) This number is stored so that when you are using several different fonts side-by-side, they will be aligned at their bases rather than at their tops or bottoms. So when you output a character at a certain cursor position, the window system first

examines the baseline of the current font, then draws the character in a position adjusted vertically to make the bases of all the characters line up.

There is another attribute called the *character width*. This can be an attribute either of the font as a whole, or of each character separately. If there is a character width for the whole font, it is as if each character had that character width separately. The character width is the amount by which the cursor position should be moved to the right when a character is output on the window. This can be different for different characters if the font is a variable-width font, in which a "W" might be much wider than an "i". Note that the character width does not necessarily have anything to do with the actual width of the bits of the character (although it usually does); it is just defined to be the amount by which the cursor should be moved.

There is another attribute that is an attribute of each character separately; it is called the *left kern*. Usually it is zero, but it can also be a positive or negative fixnum. When the window system draws a character at a given cursor position, and the left kern is non-zero, then the character is drawn to the left of the cursor position by the amount of the left kern, instead of being drawn exactly at the cursor position. In other words, the cursor position is adjusted to the left by the amount of the left kern of a character when that character is drawn, but only temporarily; the left kern affects only where the single character is drawn and does not have any cumulative effect on the cursor position.

A font that does not have separate character widths for each character and does not have any non-zero left kerns is called a *fixed-width* font. The characters are all the same width and so they line up in columns, as in typewritten text. Other fonts are called *variable-width* because different characters have different widths and things do not line up in columns. Fixed-width fonts are typically used for programs, where columnar indentation is used, while variable-width fonts are typically used for English text, because they tend to be easier to read and to take less space on the screen.

Each font also has attributes called the *blinker width* and *blinker height*. These are two non-negative fixnums that tell the window system a nice-looking width and height to make a rectangular blinker for characters in this font. These attributes are completely independent of everything else and are used only for making blinkers. Using a fixed width blinker for a variable-width font is not the nicest-looking thing to do; instead, the editor actually re-adjusts its blinker width as a function of what character it is on top of, making a wide blinker for wide characters and a narrow blinker for narrow characters. But if you don't want to go to this trouble, or don't necessarily know just which character the blinker is on top of, you can just use the font's blinker width as the width of your blinker. For a fixed-width font there's no problem.

There is also an array for each font called the *char-exists* table. It is an art-1b array with a 1 for each character that actually exists in the font, and a 0 for other characters. This table is not used by the character-drawing software; it is just for informational purposes. Characters that do not exist have pictures with no bits "on" in them, just like the "space" character. Most fonts implement most of the printing characters in the character set, but some are missing some characters.

7.3 Format of Fonts

This section explains the internal format in which fonts are represented. Most users do not need to know anything about this format; you can skip this section without loss of continuity.

Fonts are represented as arrays. The body of the array holds the bits of the characters, and the array leader holds the attributes of the font and characters as well as information about the format of the body of the array. Note that there is only one big array holding all the characters, rather than a separate array for each character. The format in which the bits are stored is specially designed to maximize the speed of character drawing and to minimize the size of the data structure, and so it is not as simple you might expect.

FED operates on fonts by converting them into a different type of object containing the same data. This new object is called a *font descriptor*; it is simpler and easier to work with. See the files SYS: IO1; FNTDEF LISP for the format of font descriptors, and SYS: IO1; FNTCNV LISP for functions to operate on them, and to convert between font descriptors and fonts.

The font format works slightly differently depending on whether the font contains any characters that are wider than thirty-two bits. If there are any such characters, then the font is considered to be "wide", and a single character may be made up of several subcharacters to be drawn side by side. A wide font stores subcharacters instead of characters as such, and has a table indicating which subcharacters belong to each character of the character set. For the time being, we will discuss only narrow fonts in which there is no need to distinguish characters from subcharacters because each character is made of a single subcharacter.

Each character in a font has an array of bits stored for it. The dimensions of this array are called the *raster width* and *raster height*. The raster width and raster height are the same for every character of a font; they are properties of the font as a whole, not of each character separately. Consecutive rows are stored in the array; the number of rows per character is the raster height, and the number of bits per row is the raster width. An integral number of rows are stored in each word of the array; if there are any bits left over, those bits are unused. Thus no row is ever split over a word boundary. Rows are stored right-adjusted, from right to left. When there are more rows than will fit into a word, the next word is used; remaining bits at the left of the first word are ignored, and the next row is stored right-adjusted in the next word, and so on. An integral number of words is used for each character.

For example, consider a font in which the widest character is seven bits wide and the tallest character is six bits tall. The raster width of the font is seven and the raster height is six. Each row of a character is seven bits, and so four of them fit into a thirty-two bit word, with four bits wasted. The remaining two rows require a second word, the rest of which will be unused because the number of words per character must be an integer. So this font will have four rows per word, and two words per character. To find the bits for character three of the font, you multiply the character number, three, by the number of words per character, two, and find that the bits for character three start in word six. The rightmost seven bits of word six are the first row of the character, the next seven bits are the second row, and so on. The rightmost seven bits of the seventh word are the fifth row, and the next seven bits of the seventh word are the sixth and last row.

Note that we have been talking about "words" of the array. The character-drawing microcode does not actually care what type the array is; it only looks at machine words as a whole, unlike most of the array-referencing in the Lisp Machine. In a Lisp-object-holding array such as an art-q array, the leftmost eight bits are not under control of the user, and so these kinds of arrays are not suitable for fonts. In general, you need to be able to control the contents of every bit in the array, and so usually fonts are art-1b arrays. This means you need to know the internal storage layout of bits within an art-1b array in order to fully understand the font format, so here it is: the zeroth element of an art-1b array is the rightmost bit of the zeroth word, and successive elements are stored from right to left in that word. The thirty-third element is the rightmost bit in the next word, and so on.

Now, if there are any characters in the font that are wider than 32 bits, then even a single row of the font will not fit into a word. Such characters are divided into subcharacters no more than 32 bits wide, and the character is drawn by drawing all its subcharacters, one by one, side by side. The character drawing microcode can only handle ordinary narrow characters, and it is invoked once for each subcharacter in order to draw a wide character. In order to make this work, the wide font stores subcharacters in the same way a narrow font stores its characters.

In addition, the wide font has a *font indexing table* which gives the first subcharacter number for each character code. (In a narrow font, the font indexing table is nil.) The character W would be drawn by finding the value at index 127 (the code for W) in the font indexing table, and the value at index 130. Suppose that these are 171 and 173. Then W is made up of subcharacters 171 and 172. Either of these subcharacters' bits can be found in the same way that the bits for character code 171 or 172 would be found in a narrow font.

The array leader of a font is a structure defined by defstruct. Here are the names of the accessors for the elements of the array leader of a font.

tv:font-name *font*
: The name of the font. This is a symbol whose value is the font and which serves to name the font. The print-name of this symbol appears in the printed representation of the font.

tv:font-char-height *font*
: The character height of the font; a non-negative fixnum.

tv:font-char-width *font*
: The character width of the characters of the font; a non-negative fixnum. If the tv:font-char-width-table of this font is non-nil, then this element is ignored except that it is used to compute the distance between horizontal tab stops; it is typically the width of a lower-case "m".

tv:font-baseline *font*
: The baseline of this font; a non-negative fixnum.

tv:font-char-width-table *font*
If this is nil then all the characters of the font have the same width, and that width is given by the tv:font-char-width of the font. Otherwise, this is an art-q array of non-negative fixnums, one for each logical character of the font, giving the character width for that character. The array *must* be an art-q array for the sake of the sys:%string-translate function.

tv:font-left-kern-table *font*
If this is nil then all characters of the font have zero left kern. Otherwise, this is an array of fixnums, one for each logical character of the font, giving the left kern for that character.

tv:font-blinker-width *font*
The blinker width of the font.

tv:font-blinker-height *font*
The blinker height of the font.

tv:font-chars-exist-table *font*
This is an art-1b array with one element for each logical character of the file. The element is 1 if the character exists and 0 if the character does not exist.

tv:font-raster-height *font*
The raster height of the font; a positive fixnum.

tv:font-raster-width *font*
The raster width of the font; a positive fixnum.

tv:font-rasters-per-word *font*
The number of rows of a character stored in each word of the font; a positive fixnum.

tv:font-words-per-char *font*
The number of words stored for each character or subcharacter; a positive fixnum.

tv:font-indexing-table *font*
If this is nil, then no characters of this font are wider than thirty-two bits. Otherwise, this is the font indexing table of the font, an array indexed by character code, containing the number of the first subcharacter for that character code. There is an extra array element at an index one greater than the largest character code; it says where the subcharacters of the largest character code stop.

7.4 Color Fonts

We mentioned earlier that you need to use different fonts to draw on different kinds of screen. To draw on a color screen, you must use a color font. If you just pass in a font specifier when you specify an element of a font map, then a color version of that font will be created if there isn't one already, and it will be used as the font.

A color font is almost the same as a regular black-and-white font except that for each pixel there are many bits. For example, for a four-bit color display (the only type presently supported), there are four bits for each pixel. While nothing prevents each pixel of a font from having any value it wants, usually each pixel is either zero or one other specific value; that is, color fonts do not usually have multicolored characters in them, or two characters of different color.

Color fonts can be created from black-and-white fonts by the following function:

color:make-color-font *bw-font* &optional (*color* 17) (*suffix* "")

Creates and returns a new font. *bw-font* should be an existing black-and-white font. The new font has all the same attributes as *bw-font*, and each character has the same attributes as the corresponding character in *bw-font*. For each zero-valued pixel in *bw-font*, the pixel in the new font is zero as well. For each one-valued pixel in *bw-font*, the pixel in the new font has value *color*. The name of the new font is formed by appending "color-", the print-name of the name of *bw-font*, and *suffix* together to form a string, and then interning that string in the fonts package.

When a font specifier is examined and the window system decides to make a color version of the font, it calls color:make-color-font with only one argument, letting the others default. So, for example, if a color version of fonts:foo-font is automatically created, its name will be fonts:color-foo-font, and its pixels will have the value 17 wherever those in the original font have the value one. However, you can call color:make-color-font to make many color versions of the same black-and-white font, each in a different color.

Something to keep in mind when using color fonts is that when characters of a color font are drawn, onto a color window, and the *char-aluf* of the window is tv:alu-ior (as it normally is), then the bits of the pixels of the character will be bit-wise "or"'ed with the existing bits in the pixels of the window. If the existing bits (that is, the background against which the character is being drawn) are all zero, there's no problem. But if they are not, the resulting values of the pixels will be some color determined by a bit-wise "or" of two color values, which is unlikely to yield meaningful results. Unless this is actually what you want, you should make sure that the background is made of zeroes before drawing characters onto a color window.

8. Drawing Graphics

A window can be used to draw graphics (pictures). There is a set of operations for drawing lines, circles, sectors, polygons, cubic splines, and so on, implemented by the flavor tv:graphics-mixin. The tv:graphics-mixin flavor is a component of the tv:window flavor, and so the operations documented below will work on windows of flavor (or flavors built on) tv:window.

tv:graphics-mixin *Flavor*

Defines the standard window graphics operations.

There are also some operations in this section that are in tv:stream-mixin (page 49) rather than tv:graphics-mixin, because they are likely to be useful to any window that can draw characters, but such windows might not want the full functionality of tv:graphics-mixin. These operations are :draw-rectangle, and the :bitblt operation and its relatives. (If you are building on tv:window anyway, this doesn't affect you, since tv:window includes both of these mixins.)

The cursor position is not used by graphics operations; the operations explicitly specify all relevant coordinates. All coordinates are in terms of the inside size of the window, just like coordinates for typing characters; the margins don't count. Remember that the point (0,0) is in the upper left; increasing *y* coordinates are *lower* on the screen, not higher. Coordinates are always fixnums.

As with typing out text, before any graphics are typed the process must wait until it has the ability to output (see section 2.6, page 21). The "output hold flag" must be off and the window must not be temp-locked. The other exception conditions of typing out are not relevant to graphics.

All graphics functions *clip* to the inside portion of the window. This means that when you specify positions for graphic items, they need not be inside the window; they can be anywhere. Only the portion of the graphic that is inside the inside part of the window will actually be drawn. Any attempt to write outside the inside part of the window simply won't happen.

8.1 Alu Functions

Most graphics operations take an *alu* argument, which controls how the bits of the graphic object being drawn are combined with the bits already present in the window. In most cases this argument is optional and defaults to the window's *char-aluf* (see page 66), the same alu function as is used to draw characters, which is normally inclusive-or. The following variables have the most useful *alu* functions as their values:

tv:alu-ior *Variable*

Inclusive-or alu function. Bits in the object being drawn are turned on and other bits are left alone. This is the char-aluf of most windows. If you draw several things with this alu function, they will write on top of each other, just as if you had used a pen on paper.

tv:alu-andca *Variable*

And-with-complement alu function. Bits in the object being drawn are turned off and other bits are left alone. This is the erase-aluf of most windows. It is useful for erasing areas of the window or for erasing particular characters or graphics.

tv:alu-xor *Variable*

Exclusive-or alu function. Bits in the object being drawn are complemented and other bits are left alone. Many graphics programs use this. The graphics operations take quite a bit of care to do "the right thing" when an exclusive-or alu function is used, drawing each point exactly once and including or excluding boundary points so that adjacent objects fit together nicely. The useful thing about exclusive-or is that if you draw the same thing twice with this alu function, the window's contents are left just as they were when you started; so this is good for drawing objects if you want to erase them afterwards.

tv:alu-seta *Variable*

Alu function to copy the input bits into the output bits, ignoring the old values of the output bits. This is not useful with the drawing operations, because the exact size and shape of the affected region depend on the implementation details of the microcode. The seta function is useful with the bitblt operations, where it causes the source rectangle to be transferred to the destination rectangle with no dependency on the previous contents of the destination.

tv:alu-and *Variable*

"And" alu function. Like tv:alu-seta, this is not useful with the drawing operations, but can be useful with the bitblt operations. 1 bits in the input leave the corresponding output bit alone, and 0 bits in the input clear the corresponding output bit.

8.2 Flavor Operations for Graphics

:point *x y* *Operation on* tv:graphics-mixin

Returns the numerical value of the picture element at the specified coordinates. The result is 0 or 1 on a black-and-white TV. Clipping is performed; if the coordinates are outside the window, the result will be 0.

:draw-point *x y* &optional *alu value* *Operation on* tv:graphics-mixin

Draws *value* into the picture element at the specified coordinates, combining it with the previous contents according to the specified *alu* function (*value* is the first argument to the operation, and the previous contents is the second argument.) *value* should be 0 or 1 on a black-and-white TV. Clipping is performed; that is, this operation will have no effect if the coordinates are outside the window. *value* defaults to -1, which is a pixel with all bits 1.

:bitblt *Operation on* tv:stream-mixin

alu width height from-array from-x from-y to-x to-y

Copies a rectangle of bits from *from-array* onto the window. The rectangle has dimensions *width* by *height*, and its upper left corner has coordinates (*from-x,from-y*). It is transferred onto the window so that its upper left corner will have coordinates (*to-x,to-y*). The bits of the transferred rectangle are combined with the bits on the display according

to the Boolean function specified by *alu*. As in the bitblt function, if *from-array* is too small it is automatically replicated.

See the discussion of the bitblt function (section 8.7 of the Lisp Machine manual) for complete details. Note that *to-array* is constrained as described there. See also the tv:make-sheet-bit-array function below (page 102).

:bitblt-from-sheet *Operation on* tv:stream-mixin

alu width height from-x from-y to-array to-x to-y

Copies a rectangle of bits from the window to *to-array*. All the other arguments have the same significance as in :bitblt.

See the discussion of the bitblt function (section 8.7 of the Lisp Machine manual) for complete details. Note that *to-array* is constrained as described there. See also the tv:make-sheet-bit-array function below (page 102).

:bitblt-within-sheet *Operation on* tv:stream-mixin

alu width height from-x from-y to-x to-y

Copies a rectangle of bits from the window to some other place in the window. All the other arguments have the same significance as in :bitblt. Note that *width* or *height* may be negative, in which case the coordinates to be copied extend to lower values from the specified starting values, and copying is done in reverse order. The order bits are copied makes no difference when copying between different arrays but is important when copying between overlapping portions of one array.

:draw-char *font char x y* &optional *alu* *Operation on* tv:graphics-mixin

Displays the character with code *char* from font *font* on the window with its upper left corner at coordinates (*x,y*). This lets you draw characters in any font (not just the ones in the font map), and it lets you put them at any position without affecting the cursor position of the window.

:draw-line *Operation on* tv:graphics-mixin

x1 y1 x2 y2 &optional *alu* (*draw-end-point* t)

Draws a line on the window with endpoints (*x1,y1*) and (*x2,y2*). If *draw-end-point* is specified as nil, does not draw the last endpoint (that is, stops drawing just before that point instead of at it). This is useful with alu function tv:alu-xor when multiple connected lines are in use, since drawing an endpoint once each for two lines would cancel out.

:draw-lines *alu x0 y0 x1 y1 ... xn yn* *Operation on* tv:graphics-mixin

Draws *n* lines on the screen, the first with endpoints (*x0,y0*) and (*x1,y1*), the second with endpoints (*x1,y1*) and (*x2,y2*), and so on. The points between lines are drawn exactly once and the last endpoint, at (*xn,yn*), is not drawn.

:draw-dashed-line *Operation on* tv:graphics-mixin

x0 y0 x1 y1 alu dash-spacing space-literally-p offset dash-length

Draws a line divided into dashes. The first five arguments are the same as those of the :draw-line operation.

The argument *dash-spacing* specifies the period of repetition of the dashes; it is the length of a dash plus the length of a space between dashes. Its default value is 20. *dash-length* is the length of the actual dash; it defaults to half the spacing.

If *space-literally-p* is nil, the spacing between dashes is adjusted so that the dashes fit evenly into the length of line to be drawn. If it is non-nil, the spacing is used exactly as specified, even though that might put the end point in the middle of a space between dashes.

A nonzero *offset* is used if you want a space between the starting point and the beginning of the first dash. The value is the amount of space desired, in pixels. The same space will be provided at the end point, if *space-literally-p* is nil. *offset* defaults to zero.

:draw-curve *Operation on* tv:graphics-mixin

x-array y-array &optional *end alu closed-p*

Draws a sequence of connected line segments. The *x* and *y* coordinates of the points at the ends of the segments are in the arrays *x-array* and *y-array*. The points between line segments are drawn exactly once and the point at the end of the last line is not drawn at all; this is especially useful when *alu* is tv:alu-xor. The number of line segments drawn is 1 less than the length of the arrays, unless a nil is found in one of the arrays first in which case the lines stop being drawn. If *end* is specified it is used in place of the actual length of the arrays.

If *closed-p* is non-nil, the end point is connected back to the first point.

:draw-wide-curve *Operation on* tv:graphics-mixin

x-array y-array width &optional *end alu closed-p*

Like :draw-curve but *width* is how wide to make the lines.

:draw-rectangle *width height x y* &optional *alu* *Operation on* tv:stream-mixin

Draws a filled-in rectangle with dimensions *width* by *height* on the window with its upper left corner at coordinates (*x,y*).

:draw-triangle *x1 y1 x2 y2 x3 y3* &optional *alu* *Operation on* tv:graphics-mixin

Draws a filled-in triangle with its corners at (*x1,y1*), (*x2,y2*), and (*x3,y3*).

:draw-circle *Operation on* tv:graphics-mixin

center-x center-y radius &optional *alu*

Draws the outline of a circle centered at the point *center-x*, *center-y* and of radius *radius*.

:draw-circular-arc *Operation on* tv:graphics-mixin

center-x center-y radius start-theta end-theta &optional *alu*

Draws part of the outline of a circle centered at the point *center-x*, *center-y* and of radius *radius*.

The part of the circle to be drawn is specified by *start-theta* and *end-theta*. These angles are in radians; an angle of zero is the positive *x* direction, and angles increase counter-clockwise. The arc starts at *start-theta* and goes through increasing angles, passing through zero if necessary, to stop at *end-theta*.

:draw-filled-in-circle *Operation on* tv:graphics-mixin

center-x center-y radius &optional *alu*

Draws a filled-in circle specified by its center and radius.

:draw-filled-in-sector *Operation on* tv:graphics-mixin

center-x center-y radius theta-1 theta-2 &optional *alu*

Draws a "triangular" section of a filled-in circle, bounded by an arc of the circle and the two radii at *theta-1* and *theta-2*. These angles are in radians; an angle of zero is the positive-X direction, and angles increase counter-clockwise.

:draw-regular-polygon *Operation on* tv:graphics-mixin

x1 y1 x2 y2 n &optional *alu*

Draws a filled-in, closed, convex, regular polygon of (abs *n*) sides, where the line from (*x1,y1*) to (*x2,y2*) is one of the sides. If *n* is positive then the interior of the polygon is on the right-hand side of the edge (that is, if you were walking from (*x1,y1*) to (*x2,y2*), you would see the interior of the polygon on your right-hand side; this does *not* mean "toward the right-hand edge of the window").

:draw-cubic-spline *Operation on* tv:graphics-mixin

px py z &optional *curve-width alu c1 c2 p1-prime-x p1-prime-y pn-prime-x pn-prime-y*

Draws a cubic spline curve that passes through a sequence of points. The arrays *px* and *py* hold the *x* and *y* coordinates of the sequence of points; the number of points is determined from the active length of *px*. Through each successive pair of points, a parametric cubic curve is drawn with the :draw-curve operation, using *z* points for each such curve. If *curve-width* is provided, the :draw-wide-curve operation is used instead, with the given width. The cubics are computed so that they match in position and first derivative at each of the points. At the end points, there are no derivatives to be matched, so the caller must specify the boundary conditions. *c1* is the boundary condition for the starting point, and it defaults to :relaxed; *c2* is the boundary condition for the ending point, and it defaults to the value of *c1*. The possible values of boundary conditions are:

:relaxed — Makes the derivative zero at this end.

:clamped — Allows the caller to specify the derivative. The arguments *p1-prime-x* and *p1-prime-y* specify the derivative at the starting point, and are only used if *c1* is :clamped; likewise, *pn-prime-x* and *pn-prime-y* specify the derivative at the ending point, and are only used if *c2* is :clamped.

:cyclic — Makes the derivative at the starting point and the ending point be equal. If *c1* is :cyclic then *c2* is ignored. To draw a closed curve through *n* points, in addition to using :cyclic, you must pass in *px* and *py* with one more than *n* entries, since you must pass in the first point twice, once at the beginning and once at the end.

:anticyclic — Makes the derivative at the starting point be the negative of the derivative at the ending point. If *c1* is :anticyclic then *c2* is ignored.

tv:spline *px py z* &optional *cx cy c1 c2 p1-prime-x p1-prime-y pn-prime-x pn-prime-y*

This subroutine of the :draw-cubic-spline operation is also useful in its own right. It does the computation of the spline to be drawn, then converts it into a sequence of line segments, returning arrays of *x* and *y* coordinates of endpoints of lines. :draw-cubic-spline works by passing these arrays to the :draw-curve operation.

The function returns three values, an array of *x* coordinates, an array of *y* coordinates, and the number of active points in those arrays. (The arrays are not required to have fill pointers.)

The arrays to be used can be supplied as the *cx* and *cy* arguments, or else new arrays will be created. If arrays are supplied and too short, they will be made longer.

8.3 Low-Level Graphics Using Subprimitives

Drawing graphics on a window is usually done by sending messages to the window. However, there is a certain overhead in sending each message. If your application requires speed, you can go to some more trouble by writing your very own method to do graphics. It is a good idea not to do this until you know that using existing messages will not work; it is easier and less bug-prone to use the existing messages than to write handlers for new ones.

To write a new method you must have a flavor to which to attach that method. In this case, we want to add some graphics messages to existing kinds of windows. So, what we want here is a mixin flavor. You will define a new mixin flavor for your application. You will add methods to this flavor to do the things you need to do. Then, when you want to create an actual window to use, you will create a window of a new flavor; this new flavor will include, as one of its mixins, your new mixin. For a simple case, you might use the following flavor definitions:

```
(defflavor circus-mixin () ()
   (:required-flavors tv:essential-window))
   ;;This makes the instance variables of tv:essential-window accessible.

(defmethod (circus-mixin :draw-clown) (size weight happy-p)
  ...)

(defmethod (circus-mixin :draw-tent)
           (height &optional (number-of-rings 3))
  ...)

(defflavor circus-window () (circus-mixin tv:window))
```

Now you can instantiate windows of flavor circus-window, and they will support your new messages.

Within the definition of a primitive output operation you will use the graphics subprimitives such as sys:%draw-char rather than the high-level operations described in previous sections. To avoid errors, you should use these subprimitives only from within window methods that provide the error checking that the subprimitives lack.

In addition, the subprimitives must be used only within the body of a tv:prepare-sheet special form. An error is signaled if they are used elsewhere.

tv:prepare-sheet *body...* *Special form*

Executes *body* in an environment in which it is safe to draw on the window. tv:prepare-sheet waits until the window is not output-held or locked, and then opens all blinkers that could be on top of the window so that they will not interfere with the output (see page 103). It also turns off interrupts so that the window will remain unlocked and the blinkers will remain open.

Because interrupts are turned off, you must be careful in writing the *body*. It should execute for no longer than you would mind being unable to do a Control-Abort. It also *must* not wait for anything, since that would allow the blinkers to reappear and defeat the whole purpose of preparing the sheet.

The microcode subprimitives generally use coordinates relative to the outside edges of the window. This is unlike the high-level interfaces, which use cursor positions, in which the margins of the window do not count. Also, subprimitives do little or no clipping or other testing for coordinates that are out of bounds. The results of passing erroneous coordinates are unpredictable; in principle, the machine might halt.

Another place you can use the subprimitives is inside the :blink operation of a blinker. This operation is always invoked in a suitable environment for calling them, including interrupts off. Because blinkers are always drawn by xor'ing, it does not actually matter whether any other blinkers are present.

These instance variables and macros are useful in writing output primitives:

(tv:sheet-inside-left)
(tv:sheet-inside-right)

Return the positions of the inside left edge and the inside right edge, both relative to the outside left edge. If your operation is intended to output on the inside of the window, these may be useful for clipping, and also for converting cursor positions to low-level coordinates.

(tv:sheet-inside-top)
(tv:sheet-inside-bottom)

Return the positions of the inside top edge and the inside bottom edge, both relative to the outside top edge.

(tv:sheet-inside-width)
(tv:sheet-inside-height)

Return the inside size of the window.

tv:width
tv:height

The total width and height of the window, including the margins.

tv:cursor-x
tv:cursor-y

The current cursor position, expressed in *outside* coordinates. That is to say, these values are *not* "cursor positions" in the usual sense of that term, but they do describe the position of the cursor.

tv:screen-array
: The array of bits that hold the contents of the window. Usually this is an indirect array that points to part of the screen, although it may also point to the superior's bit-save array, as described in section 2.5, page 17. You can use ar-2-reverse and as-2-reverse on this array, indexed by coordinates relative to the outside edges, to examine and draw individual points. The dimensions of this array will be the width and height.

tv:char-aluf
tv:erase-aluf
: These are the alu function codes (see section 8.1, page 93) that are supposed to be used for normal drawing and erasing. :tyo, :string-out and so on all use tv:char-aluf and all the standard erase operations use tv:erase-aluf. If your operation is a kind of drawing or a kind of erasing, it may be correct for you to use one of these two.

  Usually tv:char-aluf is tv:alu-ior, which means to turn on (set to all ones) the corresponding bits in the array. tv:erase-aluf is usually tv:alu-andca, which means to turn off (set to zero) the relevant bits. However, they would be different if the window were in reverse video mode. Reverse video mode is not a highly-used feature, but by using these variables you can make your extensions work correctly in reverse video mode, so it is cleaner to use them.

  However, you may use any alu function. tv:alu-xor is often useful. tv:alu-seta is usually not wise to use, since it will often result in the alteration of bits that you did not expect to change, but which happen to fall in the same word as the ones you were working on.

tv:current-font
: This is the window's current font. If you are drawing characters, it is usually correct to use this font.

Here is an example from the tv:graphics-mixin flavor, changed by adding the tv: prefixes in the places where you would need them if you were to write this outside the tv package.

```
(defmethod (graphics-mixin :draw-point)
           (x y &optional (alu tv:char-aluf) (value -1))
  (tv:prepare-sheet (self)
    (setq x (+ x (tv:sheet-inside-left))
          y (+ y (tv:sheet-inside-top)))
    (if (not (or (< x (tv:sheet-inside-left))
                 (≥ x (tv:sheet-inside-right))
                 (< y (tv:sheet-inside-top))
                 (≥ y (tv:sheet-inside-bottom))))
        (setf (ar-2-reverse tv:screen-array x y)
              (boole alu value
                     (ar-2-reverse tv:screen-array x y))))))
```

This method takes its arguments in inside coordinates, and so it first converts them to outside coordinates. Then it compares them with the boundaries of the inside of the window, and does nothing if they are outside those boundaries. This is how it does clipping. Finally, if everything

is OK, it reads out the current value of the point, combines it with the new value using the specified alu function (which defaults to the char-aluf of the window), and stores it back into the array.

8.3.1 Subprimitives for Drawing

In addition to using as-2-reverse yourself, you can use these subprimitives, mostly microcoded. They are equivalent in principle to using as-2-reverse many times, but they are much faster and have much less error checking.

Some of these primitives will accept a sheet or an array. In window-system applications the argument is usually a sheet, but any suitable two-dimensional numeric array will do. (Suitable usually means that the width, times the number of bits per element, is a multiple of 32.) If an array is used, there is no need to worry about tv:prepare-sheet. If you are doing the output on a window, you should pass the window, not its screen array.

sys:%draw-rectangle *width height x-bitpos y-bitpos alu-function sheet-or-array*
Draws a rectangle of size *width* by *height* with its upper left corner at *x-bitpos*, *y-bitpos*. Alu function *alu-function* is used, so you can draw, erase or complement the rectangle with the same function. *sheet-or-array* is usually the sheet to be drawn on. There is no clipping or error checking.

tv:%draw-rectangle-clipped *width height x-bitpos y-bitpos alu-function sheet*
This is a little smarter, clipping to the edges of *sheet*. It does not work on arrays.

tv:draw-rectangle-inside-clipped *width height x-bitpos y-bitpos alu-function sheet*
This clips to the inside edges of *sheet*.

sys:%draw-line *x0 y0 x y alu draw-end-point-p sheet-or-array*
Draws a line from (*x0*,*y0*) to (*x*,*y*), all relative to the outside edges of the sheet, or indices in the array. The point at (*x*,*y*) is not drawn if *draw-end-point-p* is nil. No clipping or error checking is done.

sys:%draw-triangle *x1 y1 x2 y2 x3 y3 alu sheet-or-array*
Draws a triangle with the specified corners. No clipping or error checking is done.

tv:draw-char *font char x y alu sheet-or-array*
Draws the character with code *char* in *font* with its upper left corner at position (*x*,*y*) in outside coordinates. *alu* is used as the alu function, so you can either draw or erase. There is no clipping or error checking.

sys:%draw-char *font char x y alu sheet-or-array*
This is the actual microcoded primitive. It does not take into account the indexing table of a wide font, so when used on a wide font *char* is not the character code that the user actually wants to output. It is best to use tv:draw-char.

sys:%color-transform *n17 n16 n15 n14 n13 n12 n11 n10 n7 n6 n5 n4 n3 n2 n1 n0 width height array start-x start-y*

This function operates on a rectangular portion of an art-4b array. It examines each element of the array, and replaces the value of that element with *n0* if its previous value was 0, *n1* if its previous value was 1, and so on. The upper-left hand corner of the array is specified by *start-x* and *start-y*, and its size is specified by *width* and *height*. *array* must be an art-4b array and the specified rectangle must be within the bounds of the array.

bitblt *alu width height from-array from-x from-y to-array to-x to-y*

Copies or merges a rectangular portion of *from-array* to a congruent portion of *to-array*. *from-x* and *from-y* specify one corner of the rectangle in *from-array*, and *to-x* and *to-y* specify the corresponding point in *to-array*. The opposite corner is found by adding *width* and *height* to either of those two positions. The copying is done starting at the specified corner and proceeding toward the opposite one.

The width of each array, times the number of bits per element in that array, must be a multiple of 32.

When used in window system applications, one of the arrays will frequently be a window's screen array. Then the window must be prepared using tv:prepare-sheet.

The operation is not simply one of copying; the bits coming from *from-array* can be merged with those of *to-array*. This is controlled by the *alu* argument. Each pair of bits is combined according to that argument to get the new bit to put in *to-array*. If *alu* is tv:alu-seta, the old bits in *to-array* are ignored. If *alu* is tv:alu-ior, then the old bits and the incoming bits are or'ed together. And so on. bitblt is careful never to change bits in *to-array* outside the specified rectangle, which is why it is safe to use tv:alu-seta, whereas it is not safe to use it in the other subprimitives.

tv:make-sheet-bit-array *window x y* &rest *make-array-options*

This function creates a two-dimensional bit-array useful for bitblting to and from windows. It makes an array whose first dimension is at least *x* but is rounded up so that bitblt's restriction regarding multiples of 32. is met, whose second dimension is *y*, and whose type is the same type as that of the screen array of *window* (or the type it would be if *window* had a screen array). *make-array-options* are passed along to make-array (see section 8.2 of the Lisp Machine manual) when the array is created, so you can control other parameters such as the area.

9. Blinkers

Each window can have any number of *blinkers*. The kind of blinker that you see most often is a blinking rectangle the same size as the characters you are typing; this blinker shows you the cursor position of the window. In fact, a window can have any number of blinkers. They need not follow the cursor (some do and some don't); the ones that do are called *following* blinkers; the others have their position set by explicit operations.

Blinkers are instances of flavors, like windows, but they are different flavors, and support a different set of standard operations. The window system provides several kinds of blinkers, which differ in the way they appear on the screen.

tv:blinker *Flavor*

All flavors of blinkers incorporate this one.

Blinkers need not actually blink; for example, the mouse arrow does not blink. A blinker's *visibility* may be any of the following:

:blink	The blinker should blink on and off periodically. The rate at which it blinks is called the *half-period*, and is a fixnum giving the number of sixthieths of a second between when the blinker turns on and when it turns off.
:on or t	The blinker should be visible but not blink; it should just stay on.
:off or nil	The blinker should be invisible.

Usually only the blinkers of the selected window actually blink; this is to show you where your type-in will go if you type on the keyboard. This is because the blinker's visibility is generally controlled based on another attribute, the *deselected visibility*, combined with whether the window is selected. While the current visibility is frequently changed by hand by the program that is using the blinker, the deselected visibility is usually fixed and says something about how the blinker is generally used. Here are its possible values and their meanings:

:on	Solid when deselected, blinking when selected. This is the most commonly used value, and the default for the blinkers that show the cursor position of a window.
:off	Off when deselected, blinking when selected.
:blink	Blinking whether selected or not.
t	Solid whether selected or not.
nil	Off whether selected or not.

When the window is deselected, each blinker's visibility is initialized from its deselected visibility. When the window is selected, visibilities of :on or :off are changed to :blink. Blinkers whose visibility is t or nil or :blink are not affected.

Blinkers are used to add visible ornaments to a window; a blinker is visible to the user, but while programs are examining and altering the contents of a window the blinkers all go away. The way this works is that before characters are output or graphics are drawn, the blinker gets turned off; it comes back later. This is called *opening* the blinker. tv:prepare-sheet (page 99) is

responsible for doing this. You can see this happening with the mouse blinker when you type at a Lisp Machine. To make this work, blinkers are always drawn using exclusive ORing (see tv:alu-xor, page 94).

Every blinker is associated with a particular sheet (window or screen). The blinker is displayed on this sheet, so that its image can appear only within the sheet. When characters are output or graphics are drawn on a sheet, only the blinkers of that sheet and its ancestors are opened (since blinkers of other sheets cannot possibly be occupying screen space that might overlap this output or graphics). The mouse blinker is free to move all over whatever screen it is on; it is therefore associated with the screen itself, and so must be opened whenever anything is drawn on any window on the screen.

A blinker has a *position* which gives the location of the blinker's upper left corner relative to the blinker's sheet. The blinker's lower right corner is controlled by the blinker's size together with its position. The blinker position is constrained to be within the sheet's area. This does not force the blinker's lower right corner to be within the sheet's area, but if it is not, the blinker's image will probably be truncated and the part outside the sheet will not appear.

:blinker-p *t-or-nil* *Init option for windows*
:blinker-flavor *flavor-name* *Init option for windows*
:blinker-deselected-visibility *visibility* *Init option for windows*

These init options specify whether a cursor-following should be created for this window, and what its flavor and visibility should be. The defaults are t, tv:rectangular-blinker, and :on.

Any other blinkers you want for a window must be created manually in an :init method or elsewhere.

tv:blinker-list *Instance variable of windows and screens*

The list of all blinkers associated with this window or screen.

:blinker-list *Operation on windows and screens*

Returns the list of blinkers associated with this window or screen.

tv:sheet-blinker-list *sheet*

Accessor defsubst for the instance variable.

9.1 Blinker Functions and Operations

tv:make-blinker *window* &optional (*flavor* 'tv:rectangular-blinker) &rest *options*

Creates and returns a new blinker. The new blinker is associated with the given *window*, and is of the given *flavor*. Other useful flavors of blinker are documented below. The *options* are initialization-options to the blinker flavor. All blinkers include the tv:blinker flavor, and so init-options taken by tv:blinker will work for any flavor of blinker. Other init-options may only work for particular flavors.

tv:x-pos *Instance variable of* tv:blinker
tv:y-pos *Instance variable of* tv:blinker

The current position of the blinker on its window, or nil if the blinker should follow the window's cursor.

:x-pos *x* *Init option for* tv:blinker
:y-pos *y* *Init option for* tv:blinker

Set the initial position of the blinker within the window. These init-options are irrelevant for blinkers that follow the cursor. The initial position for non-following blinkers defaults to the current cursor position.

:read-cursorpos *Operation on* tv:blinker

Returns two values: the *x* and *y* components of the position of the blinker within the inside of the window.

:set-cursorpos *x y* *Operation on* tv:blinker

Sets the position of the blinker, relative to the inside of the window. If the blinker has been a following blinker (that is, one which follows the window's cursor) then it ceases to be one, and from this point on moves only when :set-cursorpos is done.

:size *Operation on* tv:blinker

Returns the width and height of the blinker area occupied by the blinker, in pixels, as two values. Each flavor of blinker implements this differently.

:set-size *new-width new-height* *Operation on* tv:blinker

Sets the size of the blinker's displayed pattern. Not all blinker flavors actually do anything, but they will all allow the operation. For example, character blinkers have no way to change their size because there is no mechanism for automatically scaling fonts.

:follow-p *t-or-nil* *Init option for* tv:blinker

Sets whether the blinker follows the cursor; if this option is non-nil, it does. By default, this is nil, and so the blinker's position gets set explicitly.

:set-follow-p *new-follow-p* *Operation on* tv:blinker

Sets whether the blinker follows the cursor. If this is nil, the blinker stops following the cursor and stays where it is until explicitly moved. Otherwise, the blinker starts following the cursor.

tv:visibility *Instance variable of* tv:blinker

The blinker's current visibility.

:visibility *Operation on* tv:blinker
:set-visibility *new-visibility* *Operation on* tv:blinker

Get or set the visibility of the blinker. The specified visibility should be one of :on, nil, :off, t, or :blink; their meanings are described above.

:visibility *visibility* *Init option for* tv:blinker
Initializes the visibility.

tv:deselected-visibility *Instance variable of* tv:blinker
The blinker's deselected visibility.

:deselected-visibility *symbol* *Init option for* tv:blinker
Sets the initial deselected visibility. By default, it is :on.

:deselected-visibility *Operation on* tv:blinker
:set-deselected-visibility *new-visibility* *Operation on* tv:blinker
Examine or change the deselected visibility of the blinker.

tv:half-period *Instance variable of* tv:blinker
The time interval in 60ths of a second between successive blinks of the blinker. This is relevant only if the visibility is :blink.

:half-period *Operation on* tv:blinker
:set-half-period *new-half-period* *Operation on* tv:blinker
Get or set the half-period of the blinker. The argument is in 60ths of a second.

:half-period *half-period* *Init option for* tv:blinker
Initialize the half-period. The default is 15.

tv:sheet *Instance variable of* tv:blinker
The window or screen this blinker moves on.

:sheet *Operation on* tv:blinker
Gets the window or screen that the blinker moves on.

:set-sheet *new-sheet* *Operation on* tv:blinker
Sets to *new-sheet* the window or screen on which the blinker moves. If the old window is an ancestor or descendant of *new-sheet*, adjusts the (relative) position of the blinker so that it does not move. Otherwise, moves it to the point (0,0).

tv:time-until-blink *Instance variable of* tv:blinker
The time interval in 60ths until the next time this blinker should blink. For a blinking blinker, this controls the next turning on or off.

A non-blinking blinker will not necessarily change its state at the specified time, but it will be checked at that time and displayed if it is supposed to be visible but is not. This is how blinkers reappear after being opened so that output can be done.

:defer-reappearance *Operation on* tv:blinker
This operation is invoked whenever a blinker is opened in order to prepare a sheet, if the visibility is not :blink and if the blinker is scheduled to reappear in less than 25/60 sec. By default, it is defined to delay the blinker's reappearance until 1/2 sec after the present.

tv:phase *Instance variable of* tv:blinker

t when the blinker is present on the screen, nil when it is not.

:phase *Operation on* tv:blinker

Returns t if the blinker is now displayed on the screen.

:blink *Operation on blinkers*

Draws or erases the blinker. Since the blinker is always drawn by xor'ing, drawing it and erasing it are usually exactly the same. The method can examine the instance variable tv:phase to tell which one is happening, but usually there is no need to know. The :blink operation may assume that the blinker's sheet is prepared for output. It is always called with interrupts disabled.

tv:with-blinker-ready *do-not-open body...* *Special form*

This macro is useful in writing methods of blinkers that change the size, position, shape or anything else that affects how the blinker appears. It executes *body* after preparing to remove the blinker self from the screen. If *do-not-open* is nil, the blinker is actually opened before *body* is executed. Otherwise, *body* may call tv:open-blinker if it wants the blinker open. Interrupts are disabled by this macro in any case, so that if the blinker is opened it remains open for the duration of *body*.

Once the blinker is opened, its instance variables may be set without special care.

tv:open-blinker *blinker*

Clears *blinker* off from the screen if it is currently drawn. This does not change *blinker*'s visibility. Blinkers that are supposed to be visible but are not on the screen are put back on the screen by the scheduler, every so often. Thus, a blinker can be relied on to stay open only as long as interrupts are disabled.

tv:sheet-following-blinker *window*

Returns a blinker that follows *window*'s cursor, or nil if that window has no such blinker. If there is more than one, it returns the first one it finds (it is pretty useless to have more than one, anyway).

tv:turn-off-sheet-blinkers *window*

Sets the visibility of all blinkers on *window* to :off.

9.2 Blinker Flavors

All the flavors in this section depend on tv:blinker.

For other blinker flavors and related considerations for use of a blinker for mouse tracking, see the section on mouse blinkers, section 10.4, page 121.

tv:rectangular-blinker *Flavor*

This is one of the flavors of blinker provided for your use. A rectangular blinker is displayed as a solid rectangle; this is the kind of blinker you see in Lisp listeners and editor windows. The width and height of the rectangle can be controlled.

:width *n-pixels* *Init option for* tv:rectangular-blinker
:height *n-pixels* *Init option for* tv:rectangular-blinker

Set the initial width and height of the blinker, in pixels. By default, they are set to the font-blinker-height and font-blinker-width (see page 91) of the zeroth font of the window associated with the blinker.

:set-size *new-width new-height* *Operation on* tv:rectangular-blinker

Sets the width and height of the blinker, in pixels.

:set-size-and-cursorpos *Operation on* tv:rectangular-blinker
new-width new-height x y

Sets the width and height of the blinker, in pixels, and also its position, at once. This avoids any chance that the blinker will appear on the screen with its old size and new position, or vice versa.

tv:hollow-rectangular-blinker (tv:rectangular-blinker) *Flavor*

This flavor of blinker displays as a hollow rectangle; the editor uses such blinkers to show you which character the mouse is pointing at. This flavor includes tv:rectangular-blinker, and so all of tv:rectangular-blinker's init-options and operations work on this too.

tv:box-blinker (tv:rectangular-blinker) *Flavor*

This flavor of blinker is like tv:hollow-rectangular-blinker except that it draws a box two pixels thick, whereas the tv:hollow-rectangular-blinker draws a box one pixel thick. This flavor includes tv:rectangular-blinker, and so all of tv:rectangular-blinker's init-options and operations work on this too.

tv:stay-inside-blinker-mixin *Flavor*

This mixin makes a rectangular blinker, or any modified version thereof, keep all of its corners inside the blinker's sheet. Normally a blinker only makes sure that its position (its upper left corner) is within the sheet. Trying to position this sort of blinker in a bad place positions it against the edge of the sheet, as near as possible to the requested place.

tv:ibeam-blinker *Flavor*

This flavor of blinker displays as an I-beam (like a capital I). Its height is controllable. The lines are two pixels wide, and the two horizontal lines are nine pixels wide.

:height *n-pixels* *Init option for* tv:ibeam-blinker

Sets the initial height of the blinker. It defaults to the *line-height* (see page 67) of the window.

tv:character-blinker *Flavor*

This flavor of blinker draws itself as a character from a font. You can control which font and which character within the font it uses.

:font *font* *Init option for* tv:character-blinker

Sets the font in which to find the character to display. This may be anything acceptable to the :parse-font-specifier operation (see page 86) of the window's screen. You must provide this.

:character *ch* *Init option for* tv:character-blinker

Sets the character of the font to display. You must provide this.

:character *Operation on* tv:character-blinker

Returns the character that this blinker is displaying as.

:set-character *new-character* &optional *new-font* *Operation on* tv:character-blinker

Sets the character to be displayed to *new-character*. Also, if *new-font* is provided, set the font to *new-font*. *new-font* may be anything acceptable to the :parse-font-specifier operation (see page 86) of the window's screen.

tv:character *Instance variable of* tv:character-blinker
tv:font *Instance variable of* tv:character-blinker

The character being displayed, and the font it is displayed in.

tv:bitblt-blinker (tv:mouse-blinker-mixin) *Flavor*

A blinker that displays by copying a two-dimensional array of pixels onto the screen. The array's size must be at least the blinker's size. As it happens, this flavor also includes the ability to be the mouse blinker.

:array *array* *Init option for* tv:bitblt-blinker

This option specifies the array of pixels to be used to display the blinker. Use make-pixel-array to create the array. If you do not specify this option, you must specify both the :height and :width options, which will be used to create an array.

:width *n-pixels* *Init option for* tv:bitblt-blinker
:height *n-pixels* *Init option for* tv:bitblt-blinker

Set the initial width and height of the blinker, in pixels.

:size *Operation on* tv:bitblt-blinker

Returns the width and height of the blinker. If this is less than the size of the blinker's array, then only part of the array, starting at the upper left corner, is used.

:set-size *width height* *Operation on* tv:bitblt-blinker

Sets the size of the blinker, making a new array if the old one is not as big as the new size.

:array *Operation on* tv:bitblt-blinker
:set-array *array* *Operation on* tv:bitblt-blinker
Get or set the array of pixels to be used to display the blinker.

tv:array *Instance variable of* tv:bitblt-blinker
tv:height *Instance variable of* tv:bitblt-blinker
tv:width *Instance variable of* tv:bitblt-blinker
These instance variables hold the special information of bitblt blinkers.

tv:magnifying-blinker (tv:bitblt-blinker) *Flavor*
A kind of bitblt blinker which automatically displays a "magnified" version of some of the dots underneath it. A small square of screen pixels is magnified by replacing each pixel with an *n* by *n* square of identical pixels, where *n* is the blinker's magnification factor.

The x-offset and y-offset which the blinker has by virtue of tv:mouse-blinker-mixin (see page 122) help determine the center of magnification. The position of the magnifying blinker is, as always, the position of its upper left corner. However, the cursor positions plus the offsets give the point which the blinker is indicating (this is the place where the mouse position would be, if this blinker were the mouse blinker). The magnification is done so as to keep that point on the screen fixed.

:magnification *factor* *Init option for* tv:magnifying-blinker
Specifies the magnification factor of the magnifying blinker. 3 is a good value to use. The height and width of the blinker should be multiples of the magnification. So should the offsets.

:magnification *Operation on* tv:magnifying-blinker
:set-magnification *factor* *Operation on* tv:magnifying-blinker
Get or set the magnification factor of the blinker.

tv:magnification *Instance variable of* tv:magnifying-blinker
The magnification factor of the blinker.

tv:reverse-character-blinker (tv:bitblt-blinker) *Flavor*
This flavor of blinker appears as a solid rectangle with a character removed from it. That is, a solid rectangle and the character are both drawn, and xor with each other. This flavor of blinker proved to be very confusing in the use for which it was originally implemented, but there seems no point in deleting it entirely.

All the operations and init options of tv:character-blinker are provided, though this flavor does not depend on that one.

The position of the blinker is at the upper left corner of the rectangle. The position of the upper left corner of the *character* with respect to the rectangle is specified with the init options :character-x-offset and :character-y-offset.

:character-x-offset *n-pixels* *Init option for* tv:reverse-character-blinker
:character-y-offset *n-pixels* *Init option for* tv:reverse-character-blinker

Specify the offset of the character's upper left corner to the right and down from the blinker position (the rectangle's upper left corner).

10. The Mouse

Programs and windows can use the mouse as an input device. The functions, variables, and flavors described below allow you to use the mouse to do some simple things. To get advanced mouse behavior in your own programs, like the way the editor gets the mouse to put a box around the character being pointed at, you have to define new methods for various window operations described in this chapter. Alternatively, you can invoke the built-in choice facilities, such as menus and multiple-choice windows; these high-level facilities are described later.

At any time the mouse is considered to be indicating a certain position on the screen, called the *mouse cursor position*. The mouse cursor is a conceptual entity which we think of as what moves, inside the machine, when the user moves the mouse.

The mouse cursor position is indicated on the screen by a blinker called the mouse blinker, an actual Lisp object of the sort described in the chapter on blinkers. Different blinkers can be the mouse blinker at different times, since each window can decide what to use as the mouse blinker when that window owns the mouse.

There can be more than one screen, but the mouse cursor position is limited to one screen, called the *mouse sheet* (it does not *have* to be a screen, but it normally is). Mouse cursor positions are usually represented relative to the outside of the mouse sheet, though in operations on windows they are sometimes represented relative to the particular window. The Terminal ≥ command can be used to set the mouse sheet to another screen if your Lisp Machine has more than one screen; there is also a system menu option for this.

tv:mouse-x *Variable*
tv:mouse-y *Variable*

These variables give the position of the mouse, in pixels, measured from the outside upper-left corner of the mouse sheet. They are maintained by the process handling the mouse, normally the mouse process.

tv:mouse-set-sheet *sheet*

Makes *sheet* be the mouse sheet, the one on which the mouse cursor moves. Only inferiors of the mouse sheet (to any number of levels) can own the mouse.

tv:mouse-sheet *Variable*

The mouse sheet.

tv:mouse-set-sheet-then-call *sheet function* &rest *args*

Applies *function* to *args* with *sheet* as the mouse-sheet.

Usually the mouse cursor moves only if the user moves the mouse. However, the program can move the mouse cursor, and change the logical position of the mouse, at any time. This is called *warping* the mouse. For example, double-click-left in the editor warps the mouse to where the editor cursor is currently located. Since there is no fixed association between positions of physical mouse on the table and spots on the screen, warping the mouse does not result in any inconsistency.

tv:mouse-warp *x y*

Warps the mouse to be at positions *x*, *y* with respect to the mouse sheet.

Tracking the mouse means examining the hardware mouse interface, noting how the mouse is moving, and adjusting the mouse cursor position and the mouse blinker accordingly. Mouse tracking is done by microcode within a window, and by a process called the *mouse process* when moving between windows. The mouse process also keeps track of which window *owns* the mouse at any time. For example, when the mouse enters an editor window, the editor window becomes the owner, and to indicate this, the blinker changes to a northeast arrow instead of a northwest arrow; this is all done by the mouse process.

In general, the mouse process decides how to handle the mouse based on the flavor of the window that owns the mouse. Some flavors handle the mouse themselves, running in the mouse process, in order to be able to put little boxes and such around things, usually to indicate what would happen if you were to click a button. The editor, the inspector, menus, and other system facilities do this. The flavor of the window owning the mouse is also what usually controls the effect of clicking the mouse buttons.

10.1 Encoding Mouse Clicks as Characters

Clicks on the mouse are sometimes *encoded* into characters. Such characters are normally forced into input buffers of windows (see page 53), and so they are distinguished from regular keyboard characters by having the %%kbd-mouse bit turned on. Encoding of clicks is done with tv:mouse-button-encode (see page 116). See page 49 for full information the fields of such a character.

Note that "mouse clicks" can also be done on the keyboard. See the variables tv:use-kbd-buttons and tv:*mouse-incrementing-keystates*, in section 10.6, page 128.

These standard mixins handle mouse clicks by forcing keyboard input describing the click:

tv:kbd-mouse-buttons-mixin *Flavor*

Handles mouse clicks by encoding them as characters which are forced into the window's input buffer. In more detail: if it is a double-click on the right button, the system menu is called forth. Otherwise, the encoded character representation of the click is forced into the input buffer of the window. Furthermore, if it is a single-click on the left button, the window is selected.

The state of the Control, Meta, Super and Hyper keys at the time of the click is included in the character, in the %%kbd-control, etc., fields (see page 49).

tv:list-mouse-buttons-mixin *Flavor*

This is just like tv:kbd-mouse-buttons-mixin except that a blip goes in the input buffer rather than just an encoded click. The blip looks like:

(:mouse-button *encoded-click window x y*)

This is more useful than just the encoded click: it tells you where the mouse was (relative to the outside part of the window), and which window the mouse was over (this is useful primarily if several windows are sharing the same input buffer).

The state of the Control, Meta, Super and Hyper keys is included in the encoded click, in the %%kbd-control, etc., fields.

The following subtle point may explain some difficulties you may have with the above flavors. It is a tricky point, and you can ignore it if you don't understand it. The characters (or blips) created by the flavors above go straight into the window's input buffer. Under some circumstances they may bypass pending characters that have been typed ahead at the keyboard. So if you type something and then mouse-click at something in rapid succession while your program is busy, the program may see the mouse-click before it sees the character from the keyboard. [This may be fixed in the future.] See section 5.4.1, page 58, for further discussion of these issues.

10.2 Ownership of the Mouse

Usually the mouse is handled according to the window that it is positioned over. We say that this window *owns the mouse*. The window that owns the mouse is the one that will receive the :handle-mouse, :mouse-moves and :mouse-click messages. So the usual case is that the window under the mouse owns the mouse.

Since windows are arranged in a hierarchy, generally a window, its superior, its superior's superior, and so on, are all under the mouse at the same time. So the window that owns the mouse is really the lowest window in the hierarchy (farthest in the hierarchy from the screen) that is visible (it and all its ancestors are exposed). If you move the window to part of the screen occupied by a partially-visible window, then one of its ancestors (often the screen itself) becomes the owner. The screen handles single-clicking on the left button by selecting the window under it; this is why you can select partially-visible windows with the mouse.

A greedy window can keep ownership of the mouse even if the mouse moves outside of it, by setting tv:window-owning-mouse to that window. This should be done only when that window has come by the mouse by legitimate means, inside a :handle-mouse operation on that window or one of the other operations invoked by it. Inferiors of the greedy window can still own the mouse when it is over them. Greediness ends when tv:window-owning-mouse is set back to nil (its normal state). Then the mouse goes back to being owned by whichever window is under it. While a window is being greedy, mouse tracking continues to use the methods of the owning window, but the way of determining the owning window is changed.

The mouse can also be *grabbed*, which means that some process has taken it away from all windows. This state is represented by tv:window-owning-mouse being t. See section 10.2.1, page 115.

Usurping the mouse is an even more drastic method of taking over control. It turns the mouse process off, so you have to do the tracking yourself. See section 10.2.2, page 118.

tv:window-owning-mouse *Variable*

If this is nil, the mouse is owned by the window under it. If this is t, the mouse is grabbed. If this is a window, the mouse is owned by that window.

tv:window-owning-mouse
Returns the window that now owns the mouse, either because it is being greedy or because the mouse is over it. If the mouse has been grabbed, the value is t.

tv:mouse-window *Variable*
The window that is currently handling the mouse. This is the window that tv:window-owning-mouse returned the last time the mouse process called it.

tv:mouse-wakeup
Informs the mouse process that the screen layout has changed. Anything which may change which window is under any point where the mouse might be should call this function.

tv:hysteretic-window-mixin *Flavor*
This mixin makes a window continue to own the mouse (by being greedy) for a small distance beyond the edges of the window. This distance is called the *hysteresis*, and you can specify it. This mixin is used by momentary menus, so that if you accidentally slip a bit outside the menu, the menu won't vanish; you have to get well away from it before it vanishes.

:hysteresis *n-pixels* *Init option for* tv:hysteretic-window-mixin
Sets the initial value of the hysteresis, in pixels. It defaults to 25. (decimal).

:hysteresis *Operation on* tv:hysteretic-window-mixin
:set-hysteresis *new-hysteresis* *Operation on* tv:hysteretic-window-mixin
Examine or set the hysteresis of the window.

10.2.1 Grabbing the Mouse

Normally mouse clicks and motion are interpreted by a window that owns the mouse. Some applications, such as Edit Screen, use the mouse for choosing a window to be operated on. Then it is necessary to make sure that control of the mouse remains with the program that is doing this (e.g. Edit Screen) rather than going to whatever window the user wants to choose. This is done by *grabbing the mouse*.

When the mouse is grabbed, the mouse process gets told that no window owns the mouse, and it changes the mouse blinker back to the default (a northeast arrow). The mouse process will continue to track the mouse, and your process can now watch the position and the buttons by using tv:mouse-x and tv:mouse-y, and the variables and functions described below.

tv:with-mouse-grabbed *Special form*
A tv:with-mouse-grabbed special form just has a body:

```
(tv:with-mouse-grabbed
  forms...)
```

The forms inside are evaluated with the mouse grabbed.

tv:mouse-last-buttons *Variable*

This variable contains a mask describing the mouse buttons, as of the last time the process handling the mouse looked at them. The numbers 1, 2, and 4 represent the left, middle, and right buttons respectively, and the value of tv:mouse-last-buttons is the sum of the numbers representing the buttons that were being held down.

tv:mouse-speed *Variable*

The speed the mouse has been moving recently, in units approximately like inches per second.

tv:mouse-wait &optional (*old-mouse-x* tv:mouse-x) (*old-mouse-y* tv:mouse-y) (*old-mouse-buttons* tv:mouse-last-buttons)

This function waits for any of the variables tv:mouse-x, tv:mouse-y, or tv:mouse-last-buttons to become different from the values passed as arguments. To avoid timing errors, your program should examine the values of the variables, use them, and then pass in the values that it examined as arguments to tv:mouse-wait when it is time to wait for the mouse to move again. It is important to do things in this order, or else you might fail to wake up if one of the variables changed while you were using the old values and before you called tv:mouse-wait.

tv:mouse-button-encode *bd*

When a mouse button has been pushed, and you want to interpret this push as a click, call this function. It watches the mouse button and figures out whether a single-click or double-click is happening. It returns nil if no button is pushed, or an encoded character describing the click (see page 49).

You should call tv:mouse-button-encode only when a button has just been pushed; that is, when you see some button down that was not down before. You have to pass in the argument, *bd*, which is a bit mask saying which buttons were pressed down: which are down now that were not down "before". The form (logand (logxor *old-buttons* -1) *new-buttons*) will compute this mask, where *old-buttons* is a mask of the buttons that were down before and *new-buttons* is a mask of the ones that are down now.

tv:merge-shift-keys *char*

Modifies *char* by setting the bits corresponding to all the shift keys currently pressed down on the keyboard. This is useful on the result returned by tv:mouse-button-encode, if you wish to record the state of the shift keys in the description of a mouse click so that the shift keys can alter the meaning of the click.

tv:who-line-mouse-grabbed-documentation *Variable*

When grabbing or usurping the mouse, you should explain what is going on in the mouse-documentation line at the bottom of the screen. with-mouse-grabbed and with-mouse-usurped bind this variable to nil, which makes the mouse-documentation line blank. Inside the body of one of these special forms, you may setq this variable to a string, which will be displayed in the mouse-documentation line. If your program has "modes" which affect how the mouse acts, each part of the program should setq this variable to its own documentation.

tv:window-under-mouse &optional *operation active-condition x y*

Returns the window that is seen at the point where the mouse is (or at (*x*,*y*) in the mouse sheet, if they are non-nil). This is the window that is partially visible at that point. If *operation* is non-nil, only windows that handle that operation are considered at all. *active-condition* is another way of filtering among windows; it can be :active or :exposed, to select among active or exposed windows.

This is used by the mouse process in deciding which window owns the mouse, and can also be used by you when you have grabbed the mouse.

tv:mouse-specify-rectangle &optional *left top right bottom* (*sheet* mouse-sheet) (*minimum-width* 0) (*minimum-height* 0) *abortable*

Grabs the mouse and asks the user to specify a rectangle by clicking at two corners. This is how the system menu Create option works. Four values are returned, the left, top, right, and bottom of the rectangle, all relative to *sheet*.

left and *top*, if non-nil, are where to position the mouse initially when asking for the upper left corner. If *right* and *bottom* are also non-nil, then when asking for the lower right corner the mouse is positioned initially so as to make a rectangle of the same size as the arguments specify. In other words, what matters about the argument *right* is how much bigger it is than *left*.

minimum-width and *minimum-height* constrain the values that may be returned.

If *abortable* is non-*nil*, the user is permitted to abort by clicking the middle button. Then the function returns nil.

It is often useful to call this function via tv:mouse-set-sheet-then-call (page 112).

tv:mouse-set-window-size *window* &optional (*move-p* t)

Grabs the mouse and asks the user for new edges for *window*, returns them, and (unless inhibited) sets the edges of *window* to them as well. *window*'s edges are set unless *move-p* is nil.

The values are the new edges, suitable for the :set-edges operation, or *nil* if the user aborted.

tv:mouse-set-window-position *window* &optional (*move-p* t)

Grabs the mouse and asks the user for a new position for *window*. The new position is returned as two values, and *window* is moved to that position unless *move-p* is nil.

The values are the new position of the upper left corner, suitable for the :set-position operation, or nil if the user aborted.

10.2.2 Usurping the Mouse

For high real-time performance, you can *usurp* the mouse. Then the mouse process steps aside and lets you do everything related to tracking the mouse until you return control of it. The variables tv:mouse-x and tv:mouse-y are not updated while the mouse is usurped. The mouse blinker disappears, and if you want any visual indication of the mouse to appear, you have to do it yourself.

tv:with-mouse-usurped *Special form*

A tv:with-mouse-usurped special form just has a body:

```
(tv:with-mouse-usurped
  forms...)
```

The forms inside are evaluated with the mouse usurped.

tv:mouse-input &optional (*wait-flag* t)

Waits until something happens with the mouse, and then returns saying what happened. Four values are returned. The first two are *delta-x* and *delta-y*, which are the distance that the mouse has moved since the last time tv:mouse-input was called. The second two are *buttons-newly-pushed* and *buttons-newly-raised*, which are bit masks (using the bit assignment used by tv:mouse-last-buttons; see above) saying what buttons have changed since the last time tv:mouse-input was called.

You may call this function only with the mouse usurped; otherwise you will get in the way of the mouse process, which calls this function itself, and mouse tracking won't work correctly.

The variables tv:mouse-x and tv:mouse-y are not maintained by this function; you must do it yourself if you want to keep track of a cumulative mouse position. tv:mouse-last-buttons is maintained.

The *buttons-newly-pushed* value is suitable for being passed as an argument to tv:mouse-buttons-encode, which can be used with the mouse usurped as well as with the mouse grabbed.

If *wait-flag* is nil, then the function will not wait; it may return with all zeroes, indicating that nothing has changed.

tv:mouse-buttons

Returns the current state of the mouse buttons, in the format used by the tv:mouse-last-buttons variable, by examining the hardware mouse registers.

10.3 How Windows Handle the Mouse

The mouse is rarely grabbed or usurped. Normally it is owned by a window (or a screen). Then, mouse handling works through various flavor operations on the owning window. There are several operations, used at various points in mouse handling, to give you convenient hooks for modifying a window's behavior.

The outermost loop of mouse handling determines the owning window and then invokes its :handle-mouse method. When this method returns, the owning window is recalculated.

:handle-mouse *Operation on windows*

This operation is invoked by the mouse process to handle the mouse while it is on this window. It should return only when the mouse moves out of the window, or if the mouse is grabbed.

The default definition is to call tv:mouse-standard-blinker followed by tv:mouse-default-handler.

tv:mouse-default-handler *window scroll-bar-p*

The guts of the :handle-mouse operation. :handle-mouse methods typically set up the desired sort of mouse blinker and then call this function. *window* is the window the mouse is being handled for, and *scroll-bar-p* is t to provide a scroll bar (see section 10.5.2, page 125), if the window implements one. Generally the :enable-scrolling-p operation is used to compute the second argument.

A second argument of :in is used for handling the scroll bar itself. Values other than nil, t and :in should be avoided.

This function invokes the :mouse-moves operation to inform the window about mouse motion, and the :mouse-buttons operation to inform it about buttons going down. They are the most convenient hooks to use for implementing simple new mouse behaviors.

:set-mouse-cursorpos *x y* *Operation on windows*

:set-mouse-position *x y* *Operation on windows*

Move the mouse instantaneously to the specified position. The effect is as if the user had moved the mouse over to that spot, without the user actually touching it. For :set-mouse-position, *x* and *y* are relative to the outside edges of the window. For :set-mouse-cursorpos, they are relative to the inside edges (as in the :set-cursorpos operation).

:mouse-moves *x y* *Operation on windows*

This operation is invoked in the mouse process every time the mouse moves either into, within or out of this window. *x* and *y* are the current position of the mouse, relative to the outside edges of this window.

:mouse-moves handlers should always call tv:mouse-set-blinker-cursorpos to make the mouse blinker move. In addition, they frequently move other blinkers or turn them on or off. This is how menus arrange to outline the item the mouse is over.

tv:mouse-default-handler is what invokes this operation.

When this window ceases to own the mouse, for whatever reason, the :mouse-moves method will always be called one final time, so that it can turn off extra blinkers, etc.

tv:mouse-set-blinker-cursorpos &rest *ignore*
Moves the current mouse blinker to the current mouse position. :mouse-moves methods typically call this function.

:mouse-buttons *mask x y* *Operation on windows*
This operation is invoked in the mouse process when a button is pressed. *mask* is a mask of the buttons pressed, and *x* and *y* are the mouse position (in the mouse sheet).

By default, this calls tv:mouse-button-encode to check for double clicks, then brings up the system menu for double-click-right; otherwise, it invokes the :mouse-click operation.

tv:mouse-default-handler is what invokes this operation.

:mouse-click *mouse-char x y* *Operation on windows*
This operation is where most handling of mouse clicks actually goes on. It is invoked in the mouse process. *mouse-char* is a character code describing the button pressed and how many times; such as, #\mouse-1-2. *x* and *y* are the position of the mouse at the beginning of the click. It is preferable to use this position rather than the current one, because the user positioned the mouse accurately before clicking and motion during the click was probably accidental.

Any window selection desired should be done in another process, using process-run-function or tv:mouse-select. It is unrobust to do something so error-prone in the mouse process.

:or method combination is used, so that all the methods are run until one of them returns non-nil. So each mixin can define a way of handling the mouse under certain circumstances, and it can decline to handle the click by returning nil. For example, tv:margin-choice-mixin defines a :mouse-click method which handles the click if the position is inside a margin choice box, and returns nil otherwise so that the window's primary way of handling clicks can be run.

tv:kbd-mouse-buttons-mixin and tv:list-mouse-buttons-mixin work by defining :mouse-click methods.

:who-line-documentation-string *Operation on windows*
This operation should return a string describing what the mouse would do if clicked on this window in its current position. For example, menus return a string describing the menu item that the mouse is over. If different buttons do different things, or if multiple clicks are in use, the string should describe all the possibilities.

tv:mouse-select *window*

Selects *window*, and safe to use in the mouse process because it creates a temporary process to do the work in that case. Used by :mouse-click methods.

tv:mouse-call-system-menu

Brings up the system menu, and designed to be safe to use in the mouse process. Used by :mouse-click methods.

10.4 Mouse Blinkers

At any time one blinker is the mouse blinker, which follows the motion of the mouse. It is not always the same blinker. Each window can set up the kind of mouse blinker it wants or change the blinker, as long as that window owns the mouse.

The mouse blinker's sheet is the mouse sheet, not the window that owns the mouse and wants this blinker to be used. This avoids problems with displaying the blinker at points near the edge of the owning window which require parts of the blinker to be outside that window.

Note that mouse blinkers are not following blinkers; the mouse cursor position is independent of the cursor position of the owning window and also independent of the cursor position of the mouse sheet.

The recommended way to make a window flavor use a special form of mouse cursor is to give the flavor a :mouse-standard-blinker method which alters the mouse blinker using tv:mouse-set-blinker or tv:mouse-set-blinker-definition (see below).

Usually there is only one form of mouse blinker used for any given window. If you want the mouse blinker's appearance to vary while the mouse remains in the same window, a good technique is to have the :mouse-standard-blinker method know how to set up whichever blinker appearance is right at the moment it is called, and then call tv:mouse-standard-blinker after every event that might necessitate changing the blinker.

tv:mouse-blinker *Variable*

The blinker now following the mouse. It should not be changed by the user directly.

tv:mouse-set-blinker *blinker* &optional *x-offset y-offset*

Makes *blinker* the new mouse blinker. If *x-offset* and *y-offset* are non-nil, *blinker*'s offsets (see below) are also set.

blinker can be a defined blinker type instead of a blinker. Then this function is equivalent to tv:mouse-set-blinker-definition with only three arguments specified (page 123).

This function is typically called from :mouse-standard-blinker methods.

tv:mouse-standard-blinker &optional (*window* (tv:window-owning-mouse))

Sets the mouse blinker to the standard kind for *window*, by invoking the :mouse-standard-blinker operation on it. This is called by the window system at appropriate times.

:mouse-standard-blinker *Operation on windows*

This should use tv:mouse-set-blinker or tv:mouse-set-blinker-definition to set up the right kind of mouse blinker to use when the mouse is on this window. By default, it is defined to pass on the message to the superior window; finally, the screen handles the operation by making the blinker an upward-left arrow.

tv:mouse-blinker-mixin *Flavor*

Not all blinkers can serve as mouse blinkers. This mixin makes a blinker suitable for use as the mouse blinker.

A mouse blinker has two offsets which relate the blinker position to the mouse position. Remember that the blinker position is where the upper left corner of the blinker is displayed. The upper left corner is not always what you want to place at the precise spot the mouse is pointing to. For example, if you are using a character blinker with the character X, probably the center of the X rather than its corner should be "the spot".

:offsets *Operation on* tv:mouse-blinker-mixin

Returns the *x* and *y* offsets of the blinker as two values. The values give the position of the mouse cursor relative to the blinker; that is, in order to locate the cursor within the area of the blinker's display, the offsets must be positive.

:set-offsets *x y* *Operation on* tv:mouse-blinker-mixin

Sets the offsets of the blinker.

tv:mouse-character-blinker *Flavor*
tv:mouse-rectangular-blinker *Flavor*
tv:mouse-hollow-rectangular-blinker *Flavor*
tv:mouse-box-blinker *Flavor*
tv:mouse-box-stay-inside-blinker *Flavor*

These are versions of popular blinker flavors described in section 9.2, page 107, which can be used as the mouse blinker. tv:mouse-box-stay-inside-blinker incorporates tv:stay-inside-blinker-mixin.

The flavors tv:bitblt-blinker and tv:magnifying-blinker are already suited to be mouse blinkers.

10.4.1 Reusable Mouse Blinker Types

Normally you do not create mouse blinkers yourself. Instead, each screen keeps a list of mouse blinkers of various sorts, and you reuse one of them. This is done by means of *mouse blinker type keywords*. A mouse blinker type keyword is given a meaning, which is a function for creating a blinker. The first time someone wants a blinker of that type on a given screen, one is created and remembered, and reused every time a blinker of that type is wanted. A blinker type keyword serves a purpose similar to that of a resource.

Predefined type keywords include :character-blinker, :rectangle-blinker, :box-blinker and :box-stay-inside-blinker.

You do not have to use this mechanism, but it saves creation of blinkers to do so.

tv:mouse-define-blinker-type *type creation-function*
Defines *type* as a mouse blinker type, with *creation-function* as a function to create one. The function will receive a screen as argument and should call *make-blinker*.

tv:mouse-get-blinker *type sheet*
Returns a blinker of type *type* whose sheet is *sheet*. The same blinker will be automatically reused for different sheets on the same screen. In fact, the blinker's sheet will be the screen, not *sheet*.

tv:mouse-set-blinker-definition *type x-offset y-offset visibility operation* &rest *args*
Sets the mouse blinker to be a blinker of type *type*, and sets its offsets and visibility as specified; then sends the blinker a message of *operation* and *args* if *operation* is non-nil. *operation* is typically used to initialize other aspects of the blinker. For example, the :set-character operation is useful with character blinkers.

This function can be used in the :mouse-standard-blinker method of a window to specify a different appearance of the mouse blinker while the mouse is in that window.

tv:mouse-blinkers *Instance variable of* tv:screen
A list of mouse blinkers, examples of various reusable mouse blinker types, created so far for this screen.

10.5 Mouse Scrolling

Some windows have the ability to *scroll*. They display only a portion of a virtual window which is (or may be) too big to be shown all at once. Scrolling means moving the actually-shown portion up or down through the entire display.

10.5.1 Scrolling Protocol

There are several ways the mouse can be used to scroll a window. Each is implemented by a mixin. They all communicate with the window using the same protocol. For the sake of this protocol, the contents of the window are considered to be divided vertically into "lines". A position for scrolling is expressed as the number of lines that are above the top of the window. These do not have to be actual lines of text, though usually they are, but they must all have the same height. Usually this common height is the window's line-height, but that is not required.

:enable-scrolling-p *Operation on scrolling windows*

The various mouse-scrolling features use this operation to decide whether they should be active at any given time. If this operation returns **nil**, the scrolling facilities do not react to the mouse.

:scroll-position *Operation on scrolling windows*

Returns four values:

top-line-num	The line-number of the line currently at the top of the window.
total-lines	The total number of lines available to scroll through.
line-height	The height (in pixels) of a line.
n-items	The number of lines that the window has room for.

:scroll-to *to* &optional (*type* ':absolute) *Operation on scrolling windows*

type is one of:

:absolute	Places the line numbered *to* at the top of the window.
:relative	Adjusts the line displayed at the top of the window by *to* lines. If *to* is positive, text moves upward on the screen.

Since *to* is not guaranteed to be legal, both types of scrolling must error check their arguments.

:new-scroll-position *Operation on windows*

This operation is used by the program managing the window to tell the mouse scrolling facilities that the contents of the window have changed under program control. It should be invoked whenever either the total number of lines to scroll through or the line number at the top of the window is changed by anything except the mouse scrolling facilities.

Mouse scrolling facilities put daemons on this operation in order to update their displays when the situation changes.

10.5.2 Scroll Bars

If you move the mouse to the left edge of an editor window from the inside, eventually the mouse cursor changes to a thick up-and-down arrow. Simultaneously, a thin vertical line appears next to and outside of the left border of the window. This is called entering the scroll bar, and the thin vertical line, which indicates the portion of the total text that is now on the screen, is the scroll bar itself.

The vertical position of the top and bottom of the thin vertical line, as proportions of the height of the window, are the same as the positions of the first and last lines of text on the screen, as proportions of the total number of lines.

While the mouse is in the scroll bar, clicks have special meanings:

single left Moves this line (the one the mouse points at) to the top of the window.

single right Moves the line at the top of the window to where the mouse points.

double left Moves this line (the one the mouse points at) to the bottom of the window.

double right Moves the line at the top of the window to where the mouse points.

middle Scrolls so that the scroll bar moves to where the mouse is. The mouse vertical position on the window thus controls where in the display to scroll to; the top of the window requests the beginning of the available display, and the bottom requests the end.

tv:basic-scroll-bar *Flavor*

This mixin gives a window the ability to have a scroll bar. It defines three instance variables:

tv:scroll-bar When the window provides margin space for a scroll bar, this is a list describing the rectangle allocated. Otherwise, it is **nil**.

tv:scroll-bar-always-displayed
If this is non-**nil**, the bar will be displayed whenever margin space is provided for it, even if the mouse is not there.

tv:scroll-bar-in This is non-**nil** when the mouse is actually in this window's scroll bar.

:scroll-bar *spec* *Init option for* tv:basic-scroll-bar

Specifies whether to have a scroll bar, how big to make it, and where. *spec* can be **nil** for no scroll bar, t for a default scroll bar, or a small positive number, which requests a scroll bar of that width. The scroll bar occupies space in the margins of the window.

:set-scroll-bar *spec* *Operation on* tv:basic-scroll-bar

Sets whether this window has a scroll bar, or how wide it is. *spec* is the same as in the :scroll-bar init option. This can change the inside size of the window, since it can change the amount of space needed in the margin.

:enable-scrolling-p *Operation on* tv:basic-scroll-bar
This mixin defines this operation to return t when the window has a scroll bar. See page 124 for a description of this operation.

:scroll-bar-always-displayed *t-or-nil* *Init option for* tv:basic-scroll-bar
Non-nil to say that the bar of the scroll bar should appear on the screen all the time, not just when the mouse is "in" it.

:scroll-bar-always-displayed *Operation on* tv:basic-scroll-bar
:set-scroll-bar-always-displayed *t-or-nil* *Operation on* tv:basic-scroll-bar
Get or set this flag in an existing window. Setting it updates the screen.

:scroll-more-above *Operation on* tv:basic-scroll-bar
:scroll-more-below *Operation on* tv:basic-scroll-bar
t if there is text to scroll up (down) to. The default definition uses the :scroll-position operation; some flavors redefine it for greater efficiency.

:mouse-buttons-scroll *mouse-char x y* *Operation on* tv:basic-scroll-bar
This operation is invoked when the mouse is clicked in the scroll bar. *mouse-char* is a character with %%kbd-mouse set, identifying the button clicked and how many times. *x* and *y* are the position at the time of the click, relative to this window's outside edges. The default definition provides the standard scrolling commands; you can redefine it.

:scroll-relative *from to* *Operation on* tv:basic-scroll-bar
Scrolls the window to move what is now at the *y*-position *from* to the *y*-position *to*. The arguments can be numeric vertical cursor positions, or the symbols :top or :bottom. The :scroll-position and :scroll-to operations are used to accomplish the scrolling.

10.5.3 Margin Scrolling

The scrolling mixins described here require that the window have tv:basic-scroll-bar as well, because they make use of operations defined by that flavor. If you do not want to have a scroll bar, you can specify nil for the :scroll-bar init option.

tv:flashy-scrolling-mixin *Flavor*
This mixin provides the ability to scroll the window a line at a time by pushing the mouse against the top or bottom edge. The mouse blinker changes to a thick up or down arrow when it is in the right place to do this.

This sort of scrolling is provided in the editor and the inspector. This flavor does *not* cause the text "*more above*" to appear, the way it does in the inspector; that is done by tv:margin-scrolling-mixin.

:flashy-scrolling-region *spec* *Init option for* tv:flashy-scrolling-mixin
spec specifies where in the window the regions should go in which the mouse can cause scrolling. It looks like this:

((*top-height top-left top-right*)
(*bottom-height bottom-left bottom-right*))

Each region always abuts the top or bottom edge of the window, overlapping the

window's margin, but possibly extending into the inside of the window. Each *height* is a number of pixels in height for the specified region. Each *left* and *right* give the sides of the region. *left* and *right* can be fixnums (positions relative to the window left edge), flonums (fractions of the width of the window, with zero at the left), or :left for the left edge or :right for the right edge.

tv:margin-scroll-mixin *Flavor*

This mixin (which requires tv:margin-region-mixin as well) provides for mouse-sensitive regions in the top and bottom margins which say "*more below*" or "*more above*" if there is something to scroll to. A mouse click on the region scrolls an entire windowfull.

:margin-scroll-regions *region-list* *Init option for* tv:margin-scroll-mixin

Each element of *region-list* describes what to do with one of the two scrolling regions. An element looks like

(*keyword at-end-message more-message font-specifier*)

keyword is :top or :bottom, and says which region this element describes. *at-end-message* is an expression evaluated to get the string to display in the region when there is no room for more scrolling in that direction. If nil or omitted, it defaults to "Top" or "Bottom". *more-message* is another expression which is supposed to evaluate to a string to print when there is room for more scrolling. "More above" and "More below" are the defaults.

Most commonly one just uses a string for the *at-end-message* and the *more-message*.

font-specifier specifies the font to use. It defaults to tr10i if it is nil or omitted.

tv:flashy-margin-scrolling-mixin *Flavor*

This mixin provides both flashy scrolling and margin scrolling, with the flashy scrolling areas overlying the margin scrolling regions. You don't need anything else except tv:basic-scroll-bar.

Here are two ways of controlling when margin scrolling regions appear or disappear:

tv:margin-scroll-region-on-and-off-with-scroll-bar-mixin *Flavor*

This mixin, when combined with tv:margin-scroll-mixin, makes the margin scroll regions disappear if the :scroll-bar init option or the :set-scroll-bar operation is used to make the scroll bar disappear, and reappear if a scroll bar is created again.

tv:scroll-stuff-on-off-mixin *Flavor*

This mixin provides a scroll bar, flashy scrolling and margin scrolling, and makes them appear or disappear according to the value returned by the :enable-scrolling-p operation.

:decide-if-scrolling-necessary *Operation on* tv:scroll-stuff-on-off-mixin

Makes the scroll bar and margin regions appear or disappear if appropriate, using the :enable-scrolling-p to decide whether they should be present. The goal is to avoid displaying scrolling features, and using up screen space for them, when there is no place to scroll to.

This operation is invoked automatically at certain times. It should be invoked also whenever the number of lines to scroll through has been changed, but before doing any associated redisplay (since the redisplay to be done may be different after this operation finishes).

If the scroll bar and margin regions must be added or removed, then either the inside size of the outside size of the window must change. The :adjustable-size-p operation is used to decide which. If it returns non-nil, the inside size is preserved and the outside size is changed; otherwise, the outside size is preserved.

Changing the inside size may affect the window's redisplay calculations, and for some windows it may cause a redisplay within this operation. You may want to invoke it inside of a tv:with-sheet-deexposed to avoid letting the user see gratuitous double redisplays, or to suppress the redisplay entirely if there is no bit-save-array.

If the outside size is to be changed, and if changing the number of displayable items changes the height of the window, that should be done before invoking this operation.

:adjustable-size-p *Operation on* tv:scroll-stuff-on-off-mixin

This operation is used to decide how to adjust the window margin size. If it returns non-nil, the inside size is preserved; otherwise, the outside size.

tv:scroll-stuff-on-off-mixin does not define this operation, but it requires users to define it.

10.6 Mouse Parameters

tv:use-kbd-buttons *Variable*

If this is non-nil, the Roman numeral keys I through III on the keyboard are treated as mouse clicks when the Mode-Lock key is down. The default is t.

tv:mouse-bounce-time *Variable*

The delay in microseconds after a change in a mouse button status before the system begins to look for another change. The default is 2000. microseconds.

tv:mouse-double-click-time *Variable*

The delay in microseconds after which the system gives up checking for an additional mouse click. The default is .2 seconds.

tv:mouse-discard-clickahead

Clears out the microcode buffer in which the mouse-tracking microcode records mouse clicks.

tv:*mouse-incrementing-keystates* *Variable*

This is a list of keys (valid arguments for tv:key-state). When the mouse is clicked, each of these keys that is held down adds one to the "number of clicks". The default value is

```
(:control :shift :hyper)
```

Thus, if you do a single click with the Control key down, it is treated as a double click.

11. Margins, Borders, and Labels

In previous sections, we have mentioned the distinction between the inside and outside parts of the window. The part of the window that is not the inside part is called the *margins*. There are four margins, one for each edge. The margins sometimes contain a *border*, which is a rectangular box drawn around the outside of the window. Borders help the user see what part of the screen is occupied by which window. The margins also sometimes contain a *label*, which is a text string. Labels help the user see what a window is for.

A label can be inside the borders or outside the borders (usually it is inside). In general, there can be lots of things in the margins; each one is called a *margin item*. Borders and labels are two kinds of margin items. In any flavor of window, one of the margin items is the innermost; it is right next to the inside part of the window. Each successive margin item is outside the previous one; the last one is just inside the edges of the window. Each margin item is created by a flavor's being mixed in. You can control which margin items your window has by which flavors you mix in, and you can control their order by the order in which you mix in the flavors. Margin item flavors closer to the front of the component flavor list are further toward the outside of the margins. The tv:window flavor has as components tv:borders-mixin and tv:label-mixin, in that order, and so the label is inside the border. The scroll bar, in windows that have one, is also a margin item (see section 10.5.2, page 125).

:margins *Operation on windows*

Returns four values: the sizes of the left, top, right, and bottom margins, respectively. Each value includes the contributions of borders, labels, and anything else, to that one margin. For a window with no margins, all four values are zero.

:left-margin-size *Operation on windows*
:top-margin-size *Operation on windows*
:right-margin-size *Operation on windows*
:bottom-margin-size *Operation on windows*

Return the size of one of the margins.

tv:left-margin-size *Instance variable of windows*
tv:top-margin-size *Instance variable of windows*
tv:right-margin-size *Instance variable of windows*
tv:bottom-margin-size *Instance variable of windows*

These hold the four values returned by the :margins operation. There are no operations to set these variables or init options to initialize them, because the margin sizes are always supposed to be computed from the labels, borders and other margin items as described below.

tv:sheet-left-margin-size *window*
tv:sheet-top-margin-size *window*
tv:sheet-right-margin-size *window*
tv:sheet-bottom-margin-size *window*

Return the value of the corresponding instance variable of *window*. These are accessor defsubsts created by the :outside-accessible-instance-variables option of defflavor.

tv:sheet-inside-left &optional (*window* self)
tv:sheet-inside-top &optional (*window* self)
tv:sheet-inside-right &optional (*window* self)
tv:sheet-inside-bottom &optional (*window* self)

Return the positions of the inside edges, relative to the top left outside corner of the window. If used with no argument, these defsubsts expand into direct references to instance variables, and therefore may be used only within methods or (declare (:self-flavor ...)) functions.

11.1 Borders

tv:borders-mixin *Flavor*

The tv:borders-mixin margin item creates the borders around windows that you often see when using the Lisp Machine. You can control the thickness of each of the four borders separately, or of all of them together. You can also specify your own function to draw the borders, if you want something more elaborate than simple lines.

The borders also include some whitespace left between the borders and the inside of the window. The thickness of this white space is called the *border margin width*. The space is there so that characters and graphics that are up against the edge of the inside of the window, or the next-innermost margin item, do not "merge" with the border.

:borders *argument* *Init option for* tv:borders-mixin

This option initializes the parameters of the borders. *argument* may have any of the following values:

nil There are no borders at all.

a symbol or a number
A specification (see below) that applies to each of the four borders.

a list (*left top right bottom*)
Specifications (see below) for each of the borders at the four edges of the window.

a list (*keyword1 spec1 keyword2 spec2...*)
Specifications (see below) for the borders at the edges selected by the keywords, which may be among :left, :top, :right, :bottom.

Each specification for a particular border may be one of the following. It specifies how thick the border is and the function to draw it.

nil This edge should not have any border.

t The border at this edge should be drawn by the default function with the default thickness.

a number The border at this edge should be drawn by the default function with the specified thickness.

a symbol The border at this edge should be drawn by the specified function with the default thickness for that function.

a cons (*function* . *thickness*)
The border at this edge should be drawn by the specified function with the specified thickness.

The default (and currently only) border function is tv:draw-rectangular-border. Its default width is 1.

To define your own border function, you should create a Lisp function that takes six arguments: the window on which to draw the label, the "alu function" (see section 8.1, page 93) with which to draw it, and the left, top, right, and bottom edges of the area that the border should occupy. The returned value is ignored. The function runs inside a tv:sheet-force-access (see page 23). You should place a tv:default-border-size property on the name of the function, whose value is the default thickness of the border; it will be used when a specification is a non-nil symbol.

Note that setting border specifications to ask for a border width of zero is not the same thing as giving nil as the argument to this option, because in the former case the space for the border margin width (see the previous page) is allocated, whereas in the latter case it is not.

:set-borders *new-borders* *Operation on* tv:borders-mixin
Redefines the borders. *new-borders* can be any of the things that can be used for the :borders init option (see above).

:border-margin-width *n-pixels* *Init option for* tv:borders-mixin
Sets the width of the white space in the margins between the borders and the inside of the window. The default is 1. If some edge does not have any border (the specification for that border was nil) then that border won't have any border margin either, regardless of the value of this option; that is the difference between border specifications of 0 and nil.

:border-margin-width *Operation on* tv:borders-mixin
:set-border-margin-width *new-width* *Operation on* tv:borders-mixin
Return or set the value of the border margin width.

tv:border-margin-width *Instance variable of* tv:borders-mixin
The current border margin width.

tv:borders *Instance variable of* tv:borders-mixin
A description of the currently specified borders. It is nil for no borders. Otherwise its format is complicated and internal in nature.

tv:full-screen-hack-mixin *Flavor*
This mixin is included in many system flavors, such as Lisp listeners, Supdup, and Zmacs frames. It offers the user the option of requesting that these windows have no borders when they occupy the full screen.

tv:flush-full-screen-borders *flush-p*

With an argument of t, eliminates the borders of all windows which are full-screen-sized and have tv:full-screen-hack-mixin.

With an argument of nil, reinstates the normal borders of all such windows.

11.2 Labels

tv:label-mixin *Flavor*

The tv:label-mixin margin item creates the labels in the corners of windows that you often see when using the Lisp Machine. You can control the text of the label, the font in which it is displayed, and whether it appears at the top of the window or the bottom.

:name *Init option for windows*

The value is the name of the window, which should be a symbol. All windows have names; note that this is an init option of tv:minimum-window. It is mentioned here because the main use of the name is as the default string for the label, if there is a label (see below).

:name *Operation on windows*

Returns the name of the window, which is a symbol. See above.

:label *specification* *Init option for* tv:label-mixin

Sets the string displayed as the label, the font in which the label is displayed, and whether the label is at the top or the bottom of the window. Anything you don't specify will default; by default, the string is the same as the name of the window, the font is the screen's standard font for the purpose :label (see page 86), and the label is at the bottom of the window.

specification may be any of:

nil	There is no label at all.
t	The label is given all the default characteristics.
:top	The label is put at the top of the window.
:bottom	The label is put at the bottom of the window.
a string	The text displayed in the label is this string.
a font	The label is displayed in the specified font.

a list (*keyword1 arg1 keyword2 ...*)

The attributes corresponding to the keywords are set; the rest of the attributes default. Some keywords take arguments and some do not. The following keywords may be given:

:top	The label is put at the top of the window.
:bottom	The label is put at the bottom of the window.
:centered	The label is printed horizontally centered, rather than starting at the left edge.

:string *string* The text displayed in the label is *string*.

:font *font-specifier*
The label is displayed in the specified font. *font-specifier* may be any font specifier (see page 85).

:vsp *vsp* If the label is multiple lines, lines will be separated by *vsp* rows of pixels.

:label-size *Operation on* tv:label-mixin
Returns the width and height of the area occupied by the label.

:set-label *specification* *Operation on* tv:label-mixin
Changes some attributes of the label. *specification* can be anything accepted by the :label init option. Any attribute that *specification* doesn't mention retains its old value.

tv:label *Instance variable of* tv:label-mixin
The value of this variable describes the label of the window. It is either nil for no label or a list of length eight, whose elements are

tv:label-left
tv:label-top
tv:label-right
tv:label-bottom
The rectangle allocated to the label. All four edges are relative to the window's outside upper left corner.

tv:label-font The font to use for the label.

tv:label-string The string to display in the label.

tv:label-vsp The separation between lines in the label.

tv:label-centered
Non-nil if the label text should be horizontally centered.

tv:top-label-mixin *Flavor*
Causes the label to appear in the top margin of the window by default instead of at the bottom. The mixin does not override an explicit specification of the label position.

tv:box-label-mixin *Flavor*
Makes the label appear to be in a box, by drawing a line just on the inside of the label. This combines with the window's borders, which surround the other three sides of the label, to make a box. The extra line is present only if the label is turned on. Menus use this mixin, so from any menu that has a label, such as the one you get from Split Screen in the system menu, you can see what it looks like.

:label-box-p *t-or-nil* *Init option for* tv:box-label-mixin
If this option is nil, the box around the label is inhibited.

tv:centered-label-mixin *Flavor*

Makes the label string appear by default horizontally centered in the width of the window.

tv:delayed-redisplay-label-mixin *Flavor*

This flavor adds the :delayed-set-label and :update-label operations to your window. You send a :delayed-set-label message to change the label in such a way that it will not actually be displayed until you send an :update-label message. This is especially useful for programs that suppress redisplay when there is typeahead; the user's commands may change the label several times, and you may want to suppress the redisplay of the changes in the label until there isn't any typeahead.

:delayed-set-label *specification* *Operation on* tv:delayed-redisplay-label-mixin

This is like the :set-label method, except that nothing actually happens until an :update-label is done.

:update-label *Operation on* tv:delayed-redisplay-label-mixin

Actually does the :set-label operation on the *specification* given by the most recent :delayed-set-label operation.

tv:label-needs-updating *Instance variable of* tv:delayed-redisplay-label-mixin

Non-nil if a :delayed-set-label has been done but not displayed yet.

11.3 Margin Regions

Margin regions are a general facility for allocating space in a window's margin for specific purposes. Each region can display text or graphics and can be mouse sensitive. Margin choices (see page 210) are implemented using margin regions.

tv:margin-region-mixin *Flavor*

This mixin gives a window the ability to have margin regions.

tv:region-list *Instance variable of* tv:margin-region-mixin

A list of margin region descriptors. Each descriptor specifies one margin region and is a list of this form:

(*function margin size left top right bottom*)

The list may be longer than seven. The meaning of the extra elements is up to you. Here is what the seven standard elements mean. We list the names of the defsubsts provided to access them.

tv:margin-region-function

A function to handle various operations on the margin region. It is called with an operation name as the first argument, so it could be a flavor instance, but no flavors are predefined for the purpose and usually the function is a defselect. The margin region descriptor itself is always one of the arguments, to identify the region being operated on.

tv:margin-region-margin

The name of the margin that this region lives in; either :left, :top, :right or :bottom.

tv:margin-region-size
: The thickness in pixels of the margin region, perpendicular to the edge it is next to. (The other dimension is controlled by the size of the window, possibly diminished by space already reserved for other margin items.)

tv:margin-region-left
tv:margin-region-top
tv:margin-region-right
tv:margin-region-bottom
: The edges of the rectangle assigned to the margin region. If positive, they are relative to the outside upper left corner of the window. If negative, they are relative to the outside lower right corner.

  You do not specify these; they are computed by the :redefine-margins operation which divides up the margin space, and recorded here so that the margin region can be displayed and found by the mouse.

The margin region descriptor may be longer than seven. Additional elements are not used by tv:margin-region-mixin itself and therefore may be used by higher-level facilities to record their own information with each margin region.

:set-region-list *new-region-list* *Operation on* tv:margin-region-mixin
: Sets the list of margin regions. The new list should be a list of margin region descriptors as described above, but only the first three elements of each descriptor need be filled in. The rest will be set up automatically.

These are the operations that the *function* of a margin region is expected to handle:

:refresh *descriptor*
: This operation should draw this region on the screen in the position specified by the margin region descriptor.

:mouse-enters-region *descriptor*
: This operation is invoked whenever the mouse moves into this region.

:mouse-leaves-region *descriptor*
: This operation is invoked whenever the mouse moves out of this region.

:mouse-moves *x y descriptor*
: This operation is invoked when the mouse moves within a region. It is also invoked, following the :mouse-enters-region operation, when the mouse moves into a region. *x* and *y* are the new mouse position, relative to the outside of the window.

:mouse-click *x y descriptor mouse-char*
: This operation is invoked when the mouse is clicked on this region, except for double click right. If the operation does nothing, the mouse click has no effect. The argument *mouse-char* is like that of the :mouse-click window operation (page 120).

:who-line-documentation-string *descriptor*
: This operation is invoked to get who line documentation to be used when the mouse is in this region. It should return a string describing the meaning of

mouse clicks on the region.

tv:margin-region-area *descriptor*
Returns the four edges of the rectangle allocated to *descriptor*'s margin region, all relative to the window's outside upper left corner. This may only be used inside of methods of the window whose margin region is being operated on.

11.3.1 Margin Region Example

This is a simplification of the function used to handle the margin regions made by tv:margin-scroll-mixin. These regions display strings such as "More above" and respond to a mouse click by scrolling a full page. The margin regions used have additional nonstandard elements beyond the seventh:

tv:margin-scroll-region-more-p
Non-nil if there is more text to scroll to past this edge.

tv:margin-scroll-region-empty-msg
The string to display when there is no more to scroll to past this edge.

tv:margin-scroll-region-more-msg
The string to display when there is more to scroll to.

tv:margin-scroll-region-msg-font
The font to display the strings in.

```
(declare-flavor-instance-variables (tv:margin-scroll-mixin)
(defselect margin-scroll-region
  (:refresh (region &optional old-valid
                    &aux more-p left top right bottom)
   (multiple-value (left top right bottom)
     (tv:margin-region-area region))
   ;; Is there anything more to scroll to past this edge?
   (setq more-p
         (send self
               (if (eq (tv:margin-region-margin region) ':top)
                   ':scroll-more-above ':scroll-more-below)))
   ;; Redisplay string in the region unless already right.
   (when (or (not old-valid)
             (neq more-p (margin-scroll-region-more-p region)))
     (setf (margin-scroll-region-more-p region) more-p)
     (tv:sheet-force-access (self)
       ;; Erase the region. Sheet has just been prepared.
       (tv:%draw-rectangle (- right left) (- bottom top)
                           left top tv:erase-aluf self)
       ;; Print the string.
       (send self ':string-out-centered-explicit
             (if more-p (margin-scroll-region-more-msg region)
               (margin-scroll-region-empty-msg region))
             left top right nil
             (margin-scroll-region-msg-font region) tv:char-aluf
             0 nil nil))))

 ((:mouse-enters-region :mouse-leaves-region :mouse-moves)
  (&rest ignore))
 (:mouse-click (ignore ignore region ignore)
  (if (margin-scroll-region-more-p region)
      (let ((from (tv:margin-region-margin region)))
        (send self ':scroll-relative
              from (if (eq from ':top) ':bottom ':top)))
      (beep)))
 (:who-line-documentation-string (ignore)
   "Any button to scroll one page.")))
```

11.4 Defining Margin Item Flavors

Let us assume that you want to define a thing called a *mumble* that goes in a window's margins, the way labels and borders do. You create a flavor mumble-margin-mixin that implements the feature.

This flavor should have certain instance variables, which will be used only by the methods of mumble-margin-mixin so their precise format is up to you.

current-mumbles
: Some sort of specification of what mumbles this window should have. It might record text to display for the mumbles, a font to use, etc.

mumble-margin-area
: Records the rectangle within the window where the mumbles should go. Everything that deals with the location of the mumbles on the screen should act based on the value of this variable.

  It is recommended to use a list of four values: the left, top, right and bottom edges of the rectangle, all relative to the upper left outside corner of the window.

Some margin mixins have just a single variable whose value is a list containing both the contents and the position of the margin item.

Example:

```
(defflavor mumble-margin-mixin
           ((current-mumbles nil) mumble-margin-area)
           ()
  (:required-flavors tv:minimum-window)
  (:inittable-instance-variables current-mumbles))

(defmethod (mumble-margin-mixin :before :init) (ignore)
  (setq current-mumbles
        (canonicalize-and-validate-mumble-spec
          current-mumbles)))
```

Now you must at the minimum create methods for two standard operations for margin computation and display, to interface mumble-margin-mixin to the rest of the system. These operations are :compute-margins and :refresh-margins.

:compute-margins *lm tm rm bm* *Operation on windows*
: :compute-margins is used by the system to find out how much space is needed in each margin of the window by borders, labels, and anything else. Each flavor that implements a kind of margin item must define a method for it. This operation uses :pass-on method combination, so that the values from one method become the arguments to the next. These arguments are interpreted as the amount of space allocated so far in each margin. Each method increments one or more of them by the amount of space needed by that mixin.

:refresh-margins *Operation on windows*

Redraws all the contents of the window's margins. Each flavor of margin item must add a daemon method to this operation. The method may assume that its own margin area is completely erased to begin with.

For example:

```
(defmethod (mumble-margin-mixin :compute-margins)
           (lm tm rm bm)
  (let ((wid (mumble-margin-width current-mumbles)))
    (setq mumble-margin-area
          (list lm tm (+ lm wid) (- tv:height bm)))
    (values (+ lm wid) tm rm bm)))
```

Here we assume that the mumbles always go in the left margin. So it is always the left margin's width that is incremented, and the others are returned just as they were passed. We also assume that mumble-margin-width is a function you have defined that computes the width of space that the mumbles need.

In addition to returning modified versions of its arguments, the method also sets up the value of mumble-margin-area. This is the only place it is necessary to set that variable. By recording the position of each margin item this way, we take into account how one margin item affects the position of the others. For example, the mumbles might come inside the borders, and then the *lm*, *tm*, *rm* and *bm* values will already contain the width of the borders. Then margin-mumble-area will describe a rectangle that is within the borders.

Usually an additional mixin-specific operation is introduced into this method, as follows:

```
(defmethod (mumble-margin-mixin :compute-margins)
           (lm tm rm bm)
  (send self ':recalculate-mumble-margins lm tm rm bm))

(defmethod (mumble-margin-mixin :recalculate-mumble-margins)
           (lm tm rm bm)
  (let ((wid (mumble-margin-width current-mumbles)))
    (setq mumble-margin-area
          (list lm tm (+ lm wid) (- tv:height bm)))
    (values (+ lm wid) tm rm bm)))
```

This way, other mixins can be defined to modify where the mumbles go by replacing the :recalculate-mumble-margins method.

The one other thing you must do is provide a method for :refresh-margins, to draw the mumbles in the rectangle recorded: You can assume that that rectangle is clear to start with.

```
(defmethod (mumble-margin-mixin :after :refresh-margins) ()
  (tv:sheet-force-access (self)
    (draw-mumbles current-mumbles mumble-margin-area)))
```

You may wish to provide the user with an operation to change the window's mumbles. This operation should use the :redefine-margins operation.

:redefine-margins *Operation on windows*

This operation recomputes how much margin space is needed for all of the margin items, by invoking the :compute-margins operation, and then actually changes the window margin sizes if necessary.

If the margin sizes have changed, then the window is erased and :refresh-margins is done; the instance variable tv:restored-bits-p (present in all windows) is left set to nil. If the margin sizes have not changed, no output whatever is done, and tv:restored-bits-p is left set to t. All this is done using the :refresh operation.

Here is an example of how to use it:

```
(defmethod (mumble-margin-mixin :set-mumbles) (new-mumbles)
  (setq current-mumbles
        (canonicalize-and-validate-mumble-spec new-mumbles))
  (send self ':redefine-margins)
  (when tv:restored-bits-p
    (tv:sheet-force-access (self)
      (erase-mumble-area mumble-margin-area)
      (draw-mumbles current-mumbles mumble-margin-area))))
```

The explicit erasure and drawing of the mumbles is done in the case where the total sizes of the margins have not changed (and therefore no screen updating has been done), in case the *contents* of the mumbles have changed.

12. Frames

A *frame* is a window that is divided into sub-windows, using the hierarchical structure of the window system (discussed in section 2.1, page 10). The sub-windows are called *panes*. The panes are the inferiors of the frame, and the frame is the superior of each pane. Several heavily-used systems programs use frames. For example, inspector windows are frames. The default inspector window has six panes: the interaction pane on top, the history pane and command menu pane below it, and three inspect panes below that. The window debugger and Zmacs also use frames. In Zmacs, each new editor window is a pane of the Zmacs Frame. ZMail uses several different frames, even frames within other frames.

From these examples, you can see some of the things that frames are good for. In general, by using a frame as a user interface to an interactive subsystem, you get a convenient way to put many different things on the screen, each in its own place. Generally you can split up the frame into areas in which you can display text or graphics, areas where you can put menus or other mouse-sensitive input areas, and areas to interact with, in which keyboard input is echoed or otherwise acknowledged.

It is usually best for a frame and its panes to be treated as a unit by the system menu Select menu and by the Terminal and System keys. The mixins tv:inferiors-not-in-select-menu-mixin (section 3.2.1, page 35) and tv:alias-for-inferiors-mixin (section 3.2.2, page 36), respectively, in the frame's flavor bring this about. Then selection of panes within the frame is done by making the chosen pane the selection substitute of the frame (section 3.3, page 37). The program managing the frame can maintain a "selected pane within the frame" this way, while letting the user decide when to select the frame as a whole.

It is also common for all of the panes to use the same input buffer so that the program can always do its input in the same fashion and collect keyboard and mouse input from all the panes. See section 5.1, page 50.

It is also possible to have frames with less coupling between their panes. For example, the frame you get from requesting a frame in the system menu Split Screen option does make its panes share an input buffer, and allows them to be individually represented in the Select menu and for Terminal and System commands. It also lets the panes be selected in their own right and not as substitutes for the frame. This is done because typically each window in the split screen frame is managed by its own process.

One kind of frame is the *constraint frame*, which adjusts the shapes of its panes automatically as its own shape is changed. These frames are described first since they are a ready-to-use facility. More basic frame flavors can be built upon to create frames which manage their panes' exposure and shapes in other ways. The editor, for example, does this.

tv:basic-frame *Flavor*

All frame flavors are built on this one. tv:frame-forwarding-mixin (see page 154) mixed with this provides a non-constraint frame to which you need only add code to decide when to expose the panes and how big to make them.

tv:basic-frame is nearly the same as tv:minimum-window; it does *not* have all the mixins that go into the tv:window flavor. In particular, it does not provide for borders or a label, and it cannot be the selected window. It also has tv:delay-notification-mixin (see page 157) as a component.

12.1 Constraint Frames

If you use Edit Screen to change the shape of an inspector or debugger window frame, the shapes of the panes are all changed so that the proportions come out looking as they are supposed to. If you play around with Edit Screen enough, you can even see the menus reformat themselves (changing their numbers of rows and columns) in order to keep all of their items visible. The way all this works is that the positions and shapes of the panes, instead of being explicitly specified in units of pixels, are specified symbolically. When the window changes shape, the symbolic description is elaborated again in light of the new shape, and the panes are reshaped appropriately.

This set of symbolic descriptions is called a set of constraints. When you make a constraint frame, you specify the configuration of panes within the frame by creating list structure to represent the layout. The format of this list structure is called the constraint language. It lets you say things like "give this pane one third of the remaining room, then give that pane 17 pixels, and then divide what remains between these two panes, evenly." The constraint language is fairly complex, and is described in full detail later. In general, a frame can have many different *configurations*. Each configuration is described in the constraint language, and each specifies one way of splitting up the frame. While the program is running, it can switch a frame from one configuration to another. Some panes may appear in more than one configuration, but other panes may be left out of one configuration, and may only be visible when the frame is switched to another configuration. For example, in ZMail, when you click on Profile, the frame changes to a new configuration whose panes include a profile editor window and another frame, the profile button frame.

12.1.1 Constraint Frame Flavors

The processing of constraints is actually implemented by a frame mixin called tv:basic-constraint-frame.

The flavor of the frame itself might be any of several flavors. The simplest thing for it to be is tv:constraint-frame.

tv:constraint-frame *Flavor*

This flavor is the basic kind of constraint frame. The rest of this section describes its behavior in detail. This flavor, like tv:basic-frame, does not provide for borders, a label, or for being selected.

tv:bordered-constraint-frame *Flavor*

This flavor is just tv:constraint-frame with tv:borders-mixin (see page 130) mixed in at the right place. It will have a border around the edge. By default (using the :default-init-plist option of the flavor system), the :border-margin-width is zero, so the panes at the edges of the frame are right next to the border itself.

Bordered constraint frames are used most often. Usually, each of the panes has borders, and the frame does too. A reason for this is that when two of the panes are right next to each other, which they usually are, their borders are side by side, and so look like a double-thick line. In order to make the edges of the panes that are at the edge of the frame (rather than up against another pane) look like they are the same thickness, the frame has a border itself.

A convenient way to make all the panes of a constraint frame use the same input buffer is to use one of the following flavors:

tv:constraint-frame-with-shared-io-buffer *Flavor*

This is like tv:constraint-frame, but all the panes of the frame share the same input buffer used by the frame itself. See section 5.1, page 50.

tv:bordered-constraint-frame-with-shared-io-buffer *Flavor*

This is just like tv:constraint-frame-with-shared-io-buffer except that it has tv:borders-mixin mixed into it at the right place, so that the frame has a border around it.

:io-buffer *io-buffer* *Init option for* tv:constraint-frame-with-shared-io-buffer

If this option is present, *io-buffer* is used as the input buffer for the frame and the panes. Otherwise, a default input buffer is created. (See section 5.4, page 56 for a discussion of I/O buffers.)

12.1.2 Examples of Specifications of Panes and Constraints

The full description of how to use constraint frames, including the full constraint language, is rather complicated. The complete specifications are given in the next section; this section gives some common examples, in order to show the general idea of how the specifications work.

The following form creates a constraint frame with two panes, one on top of the other, each of which takes up half of the frame.

```
(make-instance 'tv:constraint-frame
               ':panes
                 '((top-pane tv:window)
                   (bottom-pane tv:window))
               ':constraints
                 '((main . ((top-pane bottom-pane)
                            ((top-pane 0.5))
                            ((bottom-pane :even))))))
```

Two initialization options were given to the tv:constraint-frame flavor: the :panes option and the :constraints option. The meaning of the :panes specification is: "This frame is made of the following panes. Call the first one top-pane; its flavor is tv:window. Call the second one

bottom-pane; its flavor is tv:window". The meaning of the :constraints specification is: "There is just one configuration defined for this pane; call it main. In this configuration, the panes that appear are, in order from top to bottom, top-pane and bottom-pane. top-pane should use up 0.5 of the room. bottom-pane should use up all the rest of the room."

This example demonstrates some more features:

```
(make-instance
  'tv:bordered-constraint-frame
  ':panes
    '((graphics-pane tv:window
        :label nil :blinker-p nil)
      (message-pane tv:window
        :label "Message Pane" :blinker-p nil)
      (interaction-pane tv:window))
  ':constraints
    '((main . ((interaction-pane graphics-pane message-pane)
               ((message-pane 4 :lines))
               ((graphics-pane 400))
               ((interaction-pane :even))))))
```

This frame has a border around the edges (because of the flavor of the frame itself), and it has three panes. The panes are given some initialization options themselves. The topmost pane is interaction-pane, graphics-pane is in the middle, and message-pane is on the bottom. message-pane is four lines high, graphics-pane is 400 pixels high, and interaction-pane uses up all remaining space.

Here is a window that has two possible configurations. In the first one, there are three little windows across the top of the frame and a big window beneath them; in the second one, the same big window is at the top of the frame, and underneath it is a strip split between a menu and another window.

```
(make-instance
 'tv:bordered-constraint-frame
 ':panes
   '((huey tv:window)
     (dewey tv:window)
     (louie tv:window)
     (main-pane tv:window)
     (random-pane tv:window)
     (menu tv:command-menu
           :item-list ("Foo" "Bar" "Baz")))
   ':constraints
     '((first-config . ((top-strip main-pane)
                        ((top-strip :horizontal (.3)
                          (huey dewey louie)
                          ((huey :even)
                           (dewey :even)
                           (louie :even))))
                        ((main-pane :even))))
       (second-config . ((main-pane bottom-strip)
                         ((bottom-strip :horizontal (.2)
                           (random-pane menu)
                           ((menu :ask :pane-size))
                           ((random-pane :even))))
                         ((main-pane :even))))))
```

In this example, the frame has two different configurations. When the frame is first created, it is in the first of the configurations listed, namely first-config. In this configuration, the top three-tenths of the frame are split equally, horizontally, between three windows, and the rest of the frame is occupied by main-pane. The frame can be switched to a new configuration using the :set-configuration message (see page 153). If we switch it to second-config, then main-frame will appear on top of a strip one-fifth of the height of the window. This strip will contain a menu on the right that is just wide enough to display the strings in the menu's item list, and another pane using up the rest of the strip. When the configuration of the window is switched, main-pane must be reshaped.

Another thing to notice is that the list of items in the menu was present in the :panes option, rather than a form to be evaluated. If the list had been in a variable, it would have been necessary to write the :panes option using backquote, like this:

```
':panes
  '((huey tv:window)
    (dewey tv:window)
    (louie tv:window)
    (main-pane tv:window)
    (random-pane tv:window)
    (menu tv:command-menu
          :item-list ,the-list-of-items))
```

Menus and how to use them are explained later; see section 14.1, page 173.

In this example, the window is divided into two windows, side by side.

```
(make-instance
  'tv:bordered-constraint-frame
  ':edges '(100 100 600 600)
  ':panes
    '((left tv:window)
      (right tv:window))
  ':constraints
    '((main . ((whole-thing)
               ((whole-thing :horizontal (:even)
                             (left right)
                             ((left :even)
                              (right :even))))))))
```

This example also points out that constraint frames are windows too, and you can use init-options acceptable to tv:minimum-window with them. In this case, we give the edges of the frame as a whole, in absolute numbers. Remember that frames are *not* built out of tv:window; see page 141.

In actual practice, panes are usually made out of more interesting flavors than tv:window.

12.1.3 Specifying Panes and Constraints

This section gives the complete rules for specifying the panes of a constraint frame, and for the constraint language. It should help explain any of the above examples that were unclear, and tell you all the things you can do with the constraint language.

When you create a constraint frame, you must supply two initialization options. The :panes option specifies what panes you want the frame to have, and the :constraints option specifies the set of constraints for each of the configurations that the window may assume. For the purposes of these two options, windows are given internal names, which are Lisp symbols, used only by the flavors and methods that deal with constraint frames. These names are not used as the actual names of the windows (as in the :name message (see page 132).

:panes *pane-descriptions* *Init option for* tv:constraint-frame

This initialization option is required for all flavors of constraint frames. The argument, *pane-descriptions*, is a list of pane descriptions. Every pane description looks like this:

(*name flavor* . *options*)

name is the internal name (a symbol). *flavor* is the flavor of which the pane should be an instance. *options* is a list to be appended to the initialization plist for the pane when it is created. When the frame is first created, it will create all of its panes, using *flavor* and *options*. The frame will add some of its own options to control the position and shape of the window; you should not pass any such options in the *options* list.

:constraints *configuration-description-list* *Init option for* tv:constraint-frame
This initialization option is required for all flavors of constraint frames. The argument, *configuration-description-list*, is a list of configuration descriptions. The format of configuration descriptions is explained below.

Both init options work by initializing instance variables which are then looked at by the :init methods of constraint frames. Instead of using the init options, you can set the instance variables yourself in a :before :init method.

tv:panes *Instance variable of* tv:constraint-frame
tv:constraints *Instance variable of* tv:constraint-frame
The instance variables in which the constraint frame mechanism looks to find the lists of panes and constraints.

A configuration-description-list is a list of configuration-descriptions. There is one configuration-description in the list for each of the possible configurations that the frame can assume. Each configuration is named by a symbol, called the configuration-name. A configuration-description-list is an alist that associates the configuration-descriptions with the names. It looks like this:

(*configuration-name-1* . *configuration-description-1*)
(*configuration-name-2* . *configuration-description-2*)
...)

Each configuration-description describes the layout of the panes in a single configuration. The description has two parts. The first part specifies the order in which the windows appear, and the second part specifies how the sizes are computed. Actually, in addition to windows, there can also be *dummies* in the configuration-descriptor. A dummy is used either to hold empty space that is not used by any window, or it can reserve a region of space to be divided up by another configuration-description.

A configuration-description splits up one of the dimensions of a rectangular area into many parts. Such an area is called a *section*. Which of the two dimensions is being split up is determined by the *stacking*. If the stacking is :vertical then the section is being split up vertically; that is, the parts are stacked on top of each other. If the stacking is :horizontal then the section is being split up horizontally; that is, the parts are side-by-side. The stacking of the top-level configuration-descriptions in the :constraints option is always :vertical, but there can be more configuration-descriptions nested inside of them, and these can have either stacking.

Each part has a name, represented as a symbol. A part may hold either an actual pane, or other things; in the latter case, it is called a *dummy* part. Dummy parts can be further subdivided into more panes and dummies using another constraint-description, or their pixels can be blank or filled with some pattern.

A configuration-description looks like this:
(*ordering* . *description-groups*)

ordering is a list of names of panes and of dummies, each represented by a symbol; the order of this list is the order that the panes and dummies appear in the space being split up by the configuration-description. For vertical stacking the list goes top to bottom. For horizontal

stacking the list goes left to right. A *description-group* is a list of *descriptions*. Each description describes either exactly one pane or one dummy. A configuration-description must have one description for each element of the *ordering* list.

All of the descriptions in a description-group are processed together ("in parallel"); each of the description-groups is processed in turn, starting with the first one. By grouping the descriptions this way, you can control which constraints are elaborated together and which are elaborated at different times; when two constraints are elaborated at different times you can control which one is elaborated first. The reason that the ordering-list in the configuration-description is separate from the description-groups is so that the order in which the panes and dummies appear in the frame can be independent of the order in which their constraints are elaborated.

Each description describes one pane or one dummy. We'll get back to dummies later. A description that describes a pane looks like this:

(*pane-name* . *constraint*)

pane-name is the name of the pane being described; *constraint* is the constraint that describes the pane. We will return later to what descriptions of dummies look like. The constraint will be elaborated, and will yield a size in pixels; this size will be used for the width or height being computed.

Finally we get to constraints themselves. The basic form of a constraint is as follows:

(*key arg-1 arg-2 ...*)

key may be a fixnum, a flonum, or one of various keyword symbols. Each type of constraint may take arguments, whose meaning depends on which kind of constraint this argument is passed to.

While descriptions of panes do not have the same format as descriptions of dummies, the same kind of constraints are used in both of them. So all the formats given below may be used inside the descriptions of either panes or dummies.

Any constraint may, optionally, be preceded by a :limit clause. If a constraint has a :limit clause, the constraint looks like:

(`:limit` *limit-specification key arg-1 arg-2 ...*)

The :limit clause lets you set a minimum and a maximum value that will be applied to the size computed by the constraint. If the constraint returns a value smaller than the minimum, then the minimum value will be used; if it returns a value larger than the maximum, then the maximum value will be used. The *limit-specification* is normally a two-element list, whose elements are fixnums giving the minimum and maximum values in pixels. If the list has a third element, it should be one of the symbols :lines or :characters, and it means that the fixnums are in units of lines or characters, computed by multiplying by the line-height or char-width of the pane (see page 67). If there is a fourth element, it should be the name of a pane, and that pane's line-height or char-width is used instead of that of the pane being constrained. (If this constraint applies to a dummy instead of a pane, and the third element of the list is present, then the fourth must be present as well, since dummies do not have their own line-height nor char-width.)

The following Lisp objects may be used as values of *key* in a constraint. Note: the :funcall and :eval constraints are rarely used and you probably don't need to worry about them. The other kinds are used frequently.

fixnum This lets you specify the absolute size. The value computed by the constraint is simply this fixnum. Optionally, an argument may be given: it may be the symbol :lines or the symbol :characters, meaning that the fixnum is in units of lines or characters, and should be computed by multiplying by the line-height or char-width of the window. If a second argument is also present, it should be the name of a pane, and that pane's line-height or char-width is used instead of that of the pane being constrained. (If this constraint applies to a dummy instead of a pane, and the first argument is given, then the second must be present as well, since dummies do not have their own line-height nor char-width.)

flonum This lets you specify that a certain fraction of the remaining space should be taken up by this window. Optionally, an argument may be given: it may be :lines or :characters, and it means to round down the size of the pane to the nearest multiple of the pane's line-height or char-width. A second argument may be given; it is just like the second argument when *key* is a fixnum (see above).

The distinction between descriptors in the same group and descriptors in different groups is important when you use this kind of constraint. If you have one descriptor group with two descriptors, each of which requests .2 of the remaining space, then both panes will get the same amount of space. However, if you have the same two descriptors but put them in successive descriptor groups, then the first one will get .2 of the remaining space, and then the second one will get .2 of what remains after the first one was allocated; thus the second pane will be smaller than the first. In other words, the amount of space remaining is recomputed at the end of each descriptor group, but not at the end of each descriptor.

:even This constraint has a special restriction: you can only use it for descriptors in the last descriptor group of a configuration. Furthermore, if any of the descriptors in that group use :even, then *all* of the descriptors in the group *must* use :even. The meaning is that all of the panes in the last descriptor group evenly divide all of the remaining space.

It is usually a good idea to use :even for at least one pane in every configuration, so that the entire frame will be taken up by panes that all fit together and extend to the borders of the frame. :even is careful to choose exactly the right number of pixels to fill the frame completely, avoiding roundoff errors that might cause an unsightly line of one or a few extra pixels somewhere.

Remember that just because the :evens must be in the last descriptor group does not mean that the panes that they apply to must be at the bottom or right-hand end of the frame! The ordering of the panes in the frame is controlled by the ordering list, not by the order in which the descriptors appear.

:ask This constraint lets you ask the window how much space it would like to take up. The format of a constraint using :ask is as follows:

(:ask *operation arg-1 arg-2 ...*)

A message with operation *operation* and arguments composed of some extra arguments passed by the constraint mechanism followed by *arg-1*, *arg-2*, etc. is sent to the pane; its answer says how much space the pane should take up. Note that *arg-1*, etc., are not forms: they are the values of the arguments themselves (i.e. they are not evaluated; if you want to compute them, you must build the constraint language description at run-time. This is usually written using a backquoted list).

The arguments that are actually sent along with the message are the same as the arguments passed when you use the :funcall option except that the *constraint-node* is not passed; see below.

Various different flavors of windows accept some messages suitable for use with :ask. By convention, several kinds of windows, such as menus, accept a message called :pane-size. For example, the :pane-size method for menus figures out how much space in the dimension controlled by the :ask constraint is needed to display all the items of the menu, given the amount of space available in the other dimension. No arguments are specified in the constraint. Other useful operations, handled by all windows, are :square-pane-size (also with no additional arguments), which makes the window take up enough room to be square including its borders, and :square-pane-inside-size, which makes the window be square inside its borders.

:ask-window This constraint is a variation on :ask. Its format is:

(:ask-window *pane-name message-name arg-1 arg-2 ...*)

It works like :ask except that the message is sent to the pane named *pane-name* instead of the pane being described. This is primarily used for dummies, when the size of a dummy wants to be controlled by the needs of a pane inside it.

:funcall This constraint lets you supply a function to be called, which should compute the amount of space to use. The format is:

(:funcall *function arg-1 arg-2 ...*)

The specified *function* is called. It is first passed six arguments from inside the workings of constraint frames, and then the *arg-1*, *arg-2*, etc. values. The six arguments are:

constraint-node This is an internal data structure. [Not yet documented; you should not need to look at this anyway.]

remaining-width The amount of width remaining to be used up at the time this description is elaborated, after all of the panes in previous description groups and all of the earlier panes in this description group are allocated.

remaining-height Like *remaining-width*, but in the height direction.

total-width The amount of width remaining to be used up by all of the parts of this description group. This is the amount of room left after all of the panes in previous description groups have been allocated but none of the panes in this description group have been allocated.

total-height Like *total-width*, but in the height direction.

stacking Either :vertical or :horizontal, depending on the current stacking.

:eval This is like :funcall, but instead of providing a function and arguments, you provide a form. The format is:

```
(:eval form)
```

The six special values that are passed as arguments when the :funcall constraint-type is used can be accessed by *form* as the values of the following special variables:

```
tv:**constraint-node**
tv:**constraint-remaining-width**
tv:**constraint-remaining-height**
tv:**constraint-total-width**
tv:**constraint-total-height**
tv:**constraint-stacking**
```

This finishes the discussion of descriptions of panes. Descriptions of dummies are different; they may be in any of several formats, identified by the following keywords:

:blank This description is used if you want this part of the section to be filled up with some constant pattern. The format of the description is:

```
(dummy-name :blank pattern . constraint)
```

The *constraint* is used to figure out the size of the part of the section, in the usual way. *pattern* may be any of the following:

:white The part is filled with zeroes.

:black The part is filled with the maximum value that the pixels can hold (if the pixels are one bit wide, as on a black-and-white TV, this value is 1).

an array The part is filled with the contents of the array, using the **bitblt** function (see page 102).

a symbol The symbol should be the name of a function of six arguments. The function is expected to fill up the rectangle that has been allocated to this part of the section with some pattern. The following values are passed to the function:

constraint-node This is an internal data structure. [Not yet documented; you should not need to look at this anyway.]

x-position
y-position
width
height These four arguments tell the function the position and size of the rectangle that it should fill.

screen-array This is a two-dimensional array into which the function should write the pattern it wants to

put into the window.

a list This is similar to the case in which *pattern* is a symbol, but it lets you pass extra arguments. The first element of the list is the function to be called, and that function is passed all of the objects in the rest of the list, after the six arguments enumerated above.

:horizontal *or* :vertical

This description is used if you want to subdivide the part into more panes and dummies, using a configuration-description. If you use :vertical, it will be split up with vertical stacking, and if you use :horizontal, it will be split up with horizontal stacking. [Currently, you are required to use the opposite kind of stacking from the kind currently happening; that is, successive levels of configuration-description must use alternating kinds of stacking. This restriction may be lifted in the future.] The format is as follows:

(*dummy-name* `:horizontal` *constraint* . *configuration-description*)
or
(*dummy-name* `:vertical` *constraint* . *configuration-description*)

constraint, as usual, specifies the size of this part; it can be in any of the formats given above. Note that in this format, *constraint* appears as an element of a list rather than as the tail of a list, and so the printed representation of the list will include a pair of parentheses around the constraint. *configuration-description* tells how this part is subdivided into parts of its own.

:pane-size *Operation on windows*

remaining-width remaining-height total-width total-height stacking

This operation is invoked by constraints of the form (:ask :pane-size). It should return the size in pixels to give the pane, in the current stacking direction. The meanings of the arguments as they will be passed by the constraint manager are described above under the :funcall constraint (see page 150).

:square-pane-size *Operation on windows*

remaining-width remaining-height total-width total-height stacking

:square-pane-inside-size *Operation on windows*

remaining-width remaining-height total-width total-height stacking

These operations are invoked by constraints of the form (:ask :square-pane-size) and (:ask :square-pane-inside-size). They return the size required to make the pane square. For horizontal stacking, they returns a width equal to the specified height; for vertical stacking, they returns a height equal to the available width.

The difference between the two operations is that :square-pane-size makes the outside size of the window square, whereas :square-pane-inside-size makes the inside of the window (not including the borders) square.

12.1.4 Constraint Frame Operations

:get-pane *pane-name* *Operation on* tv:basic-constraint-frame

Returns the pane (the inferior window itself) that was named by the symbol *pane-name* in the :panes specification of this frame.

:pane-name *pane* *Operation on* tv:basic-constraint-frame

Returns the symbol that was used to name *pane* in the :panes specification of this frame. If *pane* is not one of the panes, return nil.

:create-pane *name flavor* &rest *options* *Operation on* tv:basic-constraint-frame

Creates and returns a window, to serve as a pane of this frame, made from flavor *flavor* and init options *options*. *name* is the pane name to be used. By default, it is not used here.

The panes of the frame are created from their specification using this operation, the arguments being taken from the elements of the specification. It may be useful to redefine this operation.

:send-pane *Operation on* tv:basic-constraint-frame
pane-name message &rest *arguments*

Sends the specified *message* with the specified *arguments* to the pane that was named by the symbol *pane-name* in the :panes specification of this frame.

:send-all-panes *message* &rest *arguments* *Operation on* tv:basic-constraint-frame

Sends the specified *message* with the specified *arguments* to all of the panes of this frame, including the non-exposed ones.

:send-all-exposed-panes *Operation on* tv:basic-constraint-frame
message &rest *arguments*

Sends the specified *message* with the specified *arguments* to all of the exposed panes of this frame.

:configuration *configuration-name* *Init option for* tv:basic-constraint-frame

Makes the initial configuration of the frame be the one named by the symbol *configuration-name*.

:configuration *Operation on* tv:basic-constraint-frame

Returns the symbol naming the current configuration of the frame.

:set-configuration *configuration-name* *Operation on* tv:basic-constraint-frame

Sets the configuration of the frame to the one named by the symbol *configuration-name*.

:get-configuration *configuration-name* *Operation on* tv:basic-constraint-frame

Returns the internal ("parsed") data structure that describes what is specified for configuration *configuration-name*. This describes which windows are supposed to be included, and the constraints for them.

:redefine-configuration *Operation on* tv:basic-constraint-frame
config-name new-config &optional (*parsed-p* t)

Redefines the meaning of configuration *config-name* according to *new-config*. If *parsed-p* is t, *new-config* is expected to be in parsed form, such as the value returned by the :get-configuration operation. If *parsed-p* is nil, *new-config* is treated as a configuration-description such as you would use to define the configuration when initially specifying the constraints of the frame (see page 147).

12.2 Pane-Frame Interaction

Several fundamental window operations actually ask the window's superior what to do. This has no effect for a top-level window but becomes important when the window's superior is a frame. The superior can decide whether the operations should actually go ahead as requested. These operations are :expose, :deexpose, :bury, :select and :set-edges. Here is how they are handled:

:expose, :deexpose, :bury, :select

These operations first send a message to the superior with operation :inferior-expose, :inferior-deexpose, :inferior-bury or :inferior-select. The pane itself is passed as the argument.

If the message sent to the superior returns non-nil, the operation is performed on the pane as usual. Otherwise, it is skipped.

:set-edges An :inferior-set-edges message is sent to the superior, its arguments being the pane followed by the arguments of the :set-edges message. If the operation's first value is non-nil, the pane's edges are changed as requested. Otherwise, the pane's edges are not changed, and the remaining values from the :inferior-set-edges operation are returned from the :set-edges.

Of course, the frame can change the pane's edges in some other way and then return nil.

tv:basic-frame defines only the :inferior-select operation to do anything nontrivial; it makes the pane be the frame's selection substitute and then sends a :select to the frame. The others operations do nothing but return non-nil. Thus, there is minimal interaction between the frame and its inferiors. tv:frame-forwarding-mixin defines :inferior-expose, :inferior-deexpose and :inferior-bury so that the frame and panes are all exposed together.

tv:frame-forwarding-mixin *Flavor*

Defines :inferior-expose, :inferior-deexpose, and :inferior-bury methods for a frame that normally cause :expose, :deexpose or :bury operations on panes to expose, deexpose or bury the frame rather than the pane.

An :inferior-set-edges method is also defined, for internal reasons only. Its purpose is to avoid a user-visible change in behavior rather than to provide one.

This flavor is part of tv:constraint-frame and the other standard instantiable flavors of constraint frame.

tv:basic-frame has an instance variable tv:recursion which is used to distinguish between :expose, etc. operations sent by the frame's code to its panes, and those sent by other programs. When an outside program sends a :expose, :deexpose, or :bury message to one of the panes, the :inferior-expose, etc. operation on the frame simply exposes, deexpose or buries the frame itself, and does not allow the operation on the pane to proceed. When the frame's code itself exposes a pane, it does so with tv:recursion temporarily non-nil so that when the :inferior-expose is done it will return t and let the pane be exposed.

:pane-types-alist *Operation on frames*

This should return a menu item list to be used for the handling of the Create item in the screen editor, when editing the panes of this frame. The value of the menu item should be a flavor of window to create, or a list to be evaluated to return a flavor.

The menu item's value (or the result of evaluating it) can also be t, which directs the screen editor to read a flavor name from the user.

12.2.1 The Selected Pane

A frames is normally operated with one of its inferiors as a selection substitute. The selection substitute of a frame is also called the "selected pane", as this feature used to be available only in frames. Unless you mix tv:select-mixin into your frame flavor, the frame itself cannot be the selected window. Therefore, it is important to provide a selection substitute when the frame is created. This can be done by doing :set-selection-substitute in an :after :init method:

```
(defmethod (my-frame :after :init) (ignore)
  (send self ':set-selection-substitute
        (send self ':get-pane 'interaction-pane)))
```

Explicitly selecting a pane with the :select operation actually works by setting the frame's selection substitute, by means of the forwarding mechanism described above.

In a constraint frame, or any other frame which has tv:frame-forwarding-mixin, you should not attempt to select a pane which is not already exposed, because of the effects of forwarding on the :expose operation.

:selected-pane *pane-name* *Init option for* tv:basic-constraint-frame

In a constraint frame, you can initialize the selected pane with this handy init option. Instead of fishing out the pane, just give its name.

:select-pane *inferior-window* *Operation on* tv:basic-frame

This is another, older name for the :set-selection-substitute operation, before it was generalized to apply to windows other than frames.

:selected-pane *Operation on* tv:basic-frame

This is another, older name for the :selection-substitute operation, before it was generalized to apply to windows other than frames.

tv:interaction-pane *Flavor*

(tv:preemptable-read-any-tyi-mixin tv:notification-mixin tv:autoexposing-more-mixin tv:window)

This flavor is often useful for a pane for reading and echoing multi-character commands in a system which uses a frame. This pane would typically be the selected pane.

13. Miscellaneous Features

13.1 Notifications

Notifications are asynchronous messages that come from something other than the selected window. For example, when an interactive message from another user comes in (which was sent with the qsend function), it is printed as a notification. You may have noticed that sometimes a notification is printed out immediately, while sometimes all that happens is a message in the who line. The selected window is responsible for deciding what to do with the notification.

tv:notify *window-of-interest format-string* &rest *format-args*
tv:careful-notify *window-of-interest careful-p format-string* &rest *format-args*

Make a notification. *format-string* and *format-args* are passed to format to print the text of the notification. Where this text is printed, and how, is under the control of the selected window, as described below.

window-of-interest is a window that should be selected if the user clicks the mouse on the notification window (if the notification happens to use its own window). For example, a notification about a message from another user will supply the Converse window as this argument. This window can also be selected with the Terminal 0 S command.

tv:careful-notify is different in that if *careful-p* is non-nil and the notification cannot be printed now because of windows being locked, it returns immediately. The value is non-nil if the notification was printed successfully.

:print-notification *time string window-of-interest* *Operation on windows*

The system invokes this operation on the selected window to ask it to make a notification. *time* will be a time to mention in the notification. *string* is the text to print. *window-of-interest* should be set up for the user to select in some convenient fashion, if possible.

tv:notification-mixin *Flavor*

This mixin causes a window to handle notifications which happen while it is selected by printing them out on the window itself, if the window is big enough. Lisp listeners and typeout windows of all sorts use this mixin.

:print-notification-on-self *Operation on* tv:notification-mixin
time string window-of-interest

This operation does the actual work of printing a notification on the window itself, once it has been decided definitely to do so. It is sometimes useful for window flavors incorporating tv:notification-mixin to redefine this.

tv:delay-notification-mixin *Flavor*

tv:delay-notification-mixin implements the default way of handling notifications: to make them wait. It is a component of tv:window, and also of anything that contains tv:select-mixin. tv:notification-mixin works by overriding it.

If a notification arrives while a window of this sort is selected, it is put on a list called tv:pending-notifications. All that happens immediately is a beep. But the presence of a non-nil value for this variable causes the mouse documentation line to display a message that there are notifications waiting, with blinking asterisks at each end of the line.

As soon as a window that can print the notifications is selected, they will be printed. For example, selecting a Lisp listener will do it. If you are in the editor, selecting the typeout window by typing Break will do it. There is also a command, Terminal N, which selects a window that just prints the notifications.

Alternatively, Terminal 2 N can be used to make the mouse documentation line go back to its normal function. This works by transferring everything on tv:pending-notifications onto another list, tv:deferred-notifications. These deferred notifications will still be printed if you switch to a suitable window.

Another way a window can handle a notification is to ask some other window to do so. For example, editor windows (zwei:zmacs-window-pane) ask the containing Zmacs frame to do the job, and it in turn asks the echo area window to do it. This window displays the notification itself if the notification fits.

tv:find-process-in-error

Returns a process that has got an error and is waiting, having made a notification, for a window to be selected so the debugger can be run. If no such process is waiting, returns nil. If there are several such processes, the most recent one to make its notification is returned. The window the process is waiting for selection of is returned as the second value.

tv:choose-process-in-error

Similar to tv:find-process-in-error but asks the user about each candidate process. When the user answers Y, that process is returned. If the user answers N to each candidate, the value is nil. The window the process is waiting for selection of is returned as the second value.

tv:print-notifications

Prints on standard-output all the notifications that have happened in this session.

:notice *event* *Operation on windows*

The :notice operation is used to report certain events so that flavors can redefine what to do when they happen. The argument to :notice is an event name, a keyword. Additional arguments are allowed but have no meaning for any of the events yet defined. Here are the defined events:

:input
:output — The window is being used for input (output) and is not exposed, and its deexposed input (output) action is :notify. The default action is to make a notification and wait.

:input-wait — The window is being used for input and the process is waiting because no input is available now. The default action is to adjust the vertical position at which the next **MORE** will happen.

:error The window is being used for the debugger and is not exposed. The default action is to make a notification and wait, or to get another window if this one is too small.

The :notice operation uses :or method combination: all the methods are run until one returns non-nil. Aside from that, the value returned is not meaningful.

13.2 Lisp Listeners

tv:lisp-listener *Flavor*

This flavor of window is used for the window initially selected when the system starts up, and for windows created when you ask to create a "Lisp" window with any of the system menu commands.

tv:initial-lisp-listener *Variable*

The Lisp listener window that is selected when you boot.

tv:idle-lisp-listener &optional *screen*

Returns a Lisp listener which is waiting for input at top level, and is the full size of the specified screen. The screen defaults to tv:default-screen (page 13).

tv:lisp-interactor *Flavor*

This flavor of window works just like a Lisp listener, but System L will not select this kind of window, nor will tv:idle-lisp-listener return one.

The mixin primarily responsible for making a Lisp listener behave the way it does is tv:listener-mixin-internal.

tv:listener-mixin-internal *Flavor*

This contains tv:process-mixin, and arranges by default for the process to be initialized to run the Lisp top level read-eval-print loop si:lisp-top-level1.

:package *Operation on* tv:listener-mixin-internal
:set-package *package* *Operation on* tv:listener-mixin-internal

Get or set the package being used by the read-eval-print loop. These work by interfacing with some complicated code in tv:lisp-top-level1. The value from :package can be nil. When you set the package, either a package or a package name is acceptable.

tv:listener-mixin *Flavor*

This flavor inherits its entire definition from tv:listener-mixin-internal. The only difference is that System L is defined to look for windows with this flavor, and not the other.

13.3 Editor Windows

zwei:zmacs-frame *Flavor*

This is the flavor of the window you get when you type System E. It has its own process, and can select any Zmacs buffer. Generally none of the editor-specific operations should be invoked on this window; that should be left up to the window's own process. Requests to this process, which generally ask the process to select a buffer, are passed to it as blips of the form

(:execute zwei:edit-thing *spec*)

where *spec* is anything valid as the argument to ed.

A Zmacs frame is useful for providing the user an opportunity to edit whatever he likes. Sometimes it is useful for a program to offer the user specific text to edit for its own purposes.

zwei:standalone-editor-window *Flavor*

This is a window with no panes that serves as an editor. It has a minibuffer and type-in window that pop up as its inferiors when they are needed. This window has no process of its own; use the :edit operation in any process to do editing in the window.

zwei:standalone-editor-frame *Flavor*

Another kind of standalone editor window, but this one is a frame with a permanently visible mode line and typein-window or mini buffer, just as a Zmacs frame is.

:comtab *comtab* *Init option for standalone editor windows*

Specifies the comtab to use in editing in this frame. The default is zwei:*standalone-comtab*.

:edit *Operation on standalone editor windows*

Invokes the editor command loop on this window. The End command will return.

:interval-string *Operation on editor windows*

Returns a string giving the current text in the window.

:set-interval-string *string* *Operation on editor windows*

Sets the text in the window to *string*.

:interval *Operation on editor windows*

Returns the interval which is being edited in the window. If the window is a Zmacs frame, this is the selected buffer. Standalone editor windows have their own nonshared intervals which they edit; many of the editor primitives that work on Zmacs buffers also work on these intervals.

:set-interval *interval* *Operation on editor windows*

Sets the interval that this window is displaying and editing to *interval*. On a Zmacs window, *interval* must be a Zmacs buffer; then this will actually tell the window to select the new buffer.

zwei:pop-up-standalone-editor-frame *Flavor*

A temporary window form of zwei:standalone-editor-frame.

zwei:pop-up-standalone-editor-frame *Resource*
&optional (*superior* tv:mouse-sheet)

A resource of such windows, used by the following function.

zwei:pop-up-edstring *string* &optional (*near-mode*'(:mouse)) *mode-line-list min-width min-height initial-message* (*comtab* zwei:*standalone-comtab*)

Pops up an editor window containing *string* and let the user edit it. When he types End, returns a string giving whatever he left in the editor buffer. If he types Abort, the value is nil.

near-mode specifies how to position the window before popping it up. It is passed to tv:expose-window-near.

mode-line-list is a list to be used to drive the mode line.

min-width and *min-height* are minimums for the size of the window. The window is larger than that if *string* requires more space to display.

initial-message, if non-nil, is displayed in the typein window immediately after the frame pops up.

comtab is the comtab to be used for editing.

zwei:editor-top-level *Flavor*

This is the flavor used by the Lisp (Edit) window which you can create with the system menu Create option. It is a kind of Lisp listener in which both the input and the output are recorded in an editor interval and can be edited. It is based on zwei:standalone-editor-window.

zwei:temporary-mode-line-window-with-borders-resource *Resource*
&optional (*superior* tv:mouse-sheet)

A resource of such windows, used by the following functions.

zwei:temporary-mode-line-window-with-borders *Flavor*

The temporary mode line window contains just a mode line and a mini buffer. It is a way for a program to request a small piece of input while allowing the user to edit with Zwei.

This is the flavor of window that you get in ZMail if you click right on Select in the ZMail command menu and then click on *Find File* in the Select menu.

zwei:typein-line-readline-near-window *window format-string* &rest *format-args*

Pops up a temporary mode line window near *window*, displaying its mode line by passing *format-string* and *format-args* to format, and lets the user edit. Return terminates editing. The user's input is returned as a string. *window* may be any window on the screen, or :mouse, meaning pop up near the mouse.

zwei:read-defaulted-pathname-near-window *window prompt defaults special-type*

Pops up a temporary mode line window near *window*, displaying the string *prompt* as the mode line, and lets the user edit text which (when the user types Return) is parsed into a pathname using *defaults* and *special-type*. *window* may be any window on the screen, or :mouse, meaning pop up near the mouse.

:call-mini-buffer-near-window

Operation on zwei:temporary-mode-line-window-with-borders

window function &rest *args*

Pops up this window near *window*, then uses *function* to read the input and returns the value it returns. *function* should be an editor function which invokes the mini buffer using zwei:edit-in-mini-buffer. The first argument to *function* is a stream reading from the text the user edited. *args* are passed to *function* as additional arguments.

13.4 Window Flavors for Other Programs

tv:peek-frame *Flavor*

This flavor of window is a self-contained Peek display with its own process to update it.

tv:inspect-frame *Flavor*

This flavor of window is a self-contained inspector with its own process to update it.

tv:inspect-frame-resource &optional (*superior* tv:mouse-sheet) *Resource*

A resource of inspector frames which are created in a slightly special way so that they do not have their own processes, but instead are to be invoked in some other process by the function inspect.

supdup:supdup *Flavor*

A self-contained Supdup window with its own pair of processes to transfer data to and from the network.

supdup:telnet *Flavor*

A self-contained Telnet window with its own pair of processes to transfer data to and from the network.

supdup:supdup-windows &optional (*superior* tv:mouse-sheet) *Resource*
supdup:telnet-windows &optional (*superior* tv:mouse-sheet) *Resource*

Resources of Supdup and Telnet windows, for use by the functions supdup and telnet when operating in the mode of substituting for another window.

tv:pop-up-text-window *Flavor*

A temporary window, otherwise like tv:window.

tv:pop-up-text-window &optional (*superior* tv:mouse-sheet) *Resource*

A resource of such windows.

tv:truncating-pop-up-text-window *Flavor*

A temporary window which truncates lines of output, otherwise like tv:window.

tv:truncating-pop-up-text-window-with-reset *Flavor*

Like tv:pop-up-text-window but truncates lines and resets the associated process when deexposed. This is the kind of window that Terminal F uses to print its output, and it is good for many similar applications.

tv:pop-up-finger-window &optional (*superior* tv:mouse-sheet) *Resource*

A resource of such windows.

13.5 The Who Line

The *who line* is the pair of lines at the bottom of the main Lisp Machine screen which display the current status of the machine. The first of the two lines displays documentation what mouse clicks would do at the present time, based on the actual position of the mouse. The second line displays the time, your login name, the current process's package and run state, and file or net server information. The term "who line" is sometimes used to refer to this line alone.

The window system treats the who line as a separate screen, thus preventing windows on the rest of the screen from being moved or reshaped to overlap the who line. The mouse documentation line is displayed by a window of its own, and so is each field of the second line.

The documentation displayed by the mouse documentation line is obtained by sending the window under the mouse a :who-line-documentation-string message (see page 120), or from the variable tv:who-line-mouse-grabbed-documentation when the mouse is grabbed (see page 116).

tv:who-line-documentation *t-or-nil*

Turns the who line display of mouse documentation on or off.

The package name and run state displayed in the who line describe only one process. They normally describe the process associated with the selected window, which is a different process if a new window is selected. However, the who line can be fixated on a particular process, independent of the selected window.

tv:who-line-process *Variable*

The process to describe in the who line, or nil meaning to display the one associated with the selected window. In the latter case, the :process operation on the window is used to get the process to display.

tv:last-who-line-process *Variable*

The process most recently described in the who line, regardless of why that process was chosen. May be nil if there was no process to describe (for example, if the who line was supposed to describe the selected window but there was no selected window or the window had no process).

The user can set tv:who-line-process using the Terminal W command (see "Operating the Lisp Machine").

tv:who-line-clobbered
Informs the who line that it must redisplay everything.

Recording open file streams for display:

tv:who-line-file-state-sheet
This who line window displays the status of an open stream or active network server. It can also display the idle time if there is no stream or server.

This window is also responsible for maintaining the lists of streams and servers that could be displayed. New streams and servers are reported to it with operations described here.

:add-stream *stream update-p* *Operation on* tv:who-line-file-sheet
Adds *stream* to the list of open streams recorded by the file state sheet. If *update-p* is non-nil, the who line field is updated immediately.

:delete-stream *stream* *Operation on* tv:who-line-file-sheet
Removes *stream* from the list of streams for the who line.

:delete-all-streams *Operation on* tv:who-line-file-sheet
Clears out the list of streams for the who line.

:open-streams *Operation on* tv:who-line-file-sheet
Returns the list of streams recorded for the who line.

When the who line describes an open file, the name to display for it is obtained with the :string-for-wholine pathname operation. See section 22.6 of the Lisp Machine manual.

:add-server *Operation on* tv:who-line-file-sheet
conn contact-name process function
Adds a entry to the list of active network servers recorded by the file state sheet. *conn* should be the network connection of this server, *contact-name* the contact name it responded to, *process* the process the server is running in.

:delete-server *conn* *Operation on* tv:who-line-file-sheet
Removes the entry for connection *conn* from the list of servers for the who line. Note that this happens automatically if the connection is broken or closed.

:delete-all-servers *Operation on* tv:who-line-file-sheet
Clears out the list of servers for the who line.

tv:close-all-servers &optional (*reason* "Foo on you")
Closes the connections of all network servers, giving *reason* (a string) as the reason in the CLS packet.

tv:describe-servers
Prints descriptions of all active network servers.

13.6 The Color Screen

The usual color screen on a Lisp Machine has 454. lines of 576. pixels each, and each pixel has four bits. This allows sixteen different colors to be displayed at once. There are far more than sixteen possible colors. A *color map* controls the meaning of each of the sixteen pixel values. Each of the sixteen color map slots specifies an eight-bit red intensity, an eight-bit green intensity, and an eight-bit blue intensity. Thus there are about 16 million different colors that can appear, but only sixteen can be displayed at once.

color:color-screen *Variable*
The screen object that represents the color screen. This object is always present whether the machine has a color screen or not.

color:color-exists-p
t if this machine actually has a color screen.

13.6.1 Color Map Functions

color:write-color-map *slot r g b* &optional *synchronize screen*
Writes the color map contents for *slot*, a fixnum from 0 to 17, with the three intensities *r*, *g* and *b*, all fixnums from 0 to 377 octal.

If *synchronize* is non-nil, the change is delayed until the vertical retrace time, so that it will take effect between frames. *screen* is the screen to operate on, in case you have more than one. It defaults to the normal color screen.

color:write-color-map-immediate *slot r g b* &optional *screen*
Like color:write-color-map, but faster. It performs no synchronization at all, and is intended for use when you have already waited for vertical retrace.

color:blt-color-map *array* &optional *screen*
Copies the contents of *array*, a 16 by 3 array, into the color map of *screen* (which defaults to the normal color screen). This function always waits for vertical retrace to do its work.

color:read-color-map *slot* &optional *screen*
Returns three values, the red, green and blue intensities from the color map from slot *slot*. This does not actually read the hardware color map, as there is no way to do that. Instead, color:write-color-map maintains a copy for this purpose.

color:fill-color-map *r g b* &optional (*start-slot* 1) *screen*
Writes multiple slots in the color map, starting with *start-slot* and ending with slot 17, from *r*, *g* and *b*. Note that the default omits slot 0, which is normally left as black (all three intensities zero). This function always waits for vertical retrace to do its work.

color:random-color-map &optional (*start* 1) *synchronize screen*
Sets the contents of the color map to sixteen randomly chosen colors. The slots modified are *start* through 17, by default omitting slot 0. *synchronize* is the same as in color:write-color-map.

color:spectrum-color-map
Sets the color map to a spectrum, leaving color 0 as black.

color:colorize &optional (*delay* 4)
Sets the color map (except for slot 0) randomly over and over again, waiting *delay* 60ths of a second in between.

color:colorate &optional (*delay* 4) (*steps* 1000.)
Repeatedly chooses two colors (numbers from 1 to 17) randomly and moves their color map values gradually towards and through each other, so that ultimately the two slots exchange colors. A delay of *delay* 60ths of a second elapses between exchanges.

13.6.2 Operating on Pixels

One way to draw on the color screen is to store into its screen array with as-2-reverse. The screen array of the color screen can be obtained with tv:sheet-screen-array, and it is an array of type art-4b. You can also use these functions:

color:clear
Fills the whole color screen with color 0.

color:rectangle *x y width height color* &optional *aluf screen*
Sets the contents of a rectangle on the color screen to pixel value *color*. *x* and *y* are the coordinates of the upper left corner, and *width* and *height* are the size.

aluf is an alu function to apply to each pixel, combining the specified color with the old pixel contents to get the new contents. The default is tv:alu-seta, which ignores the old contents. This alu function is used only on the pixels of the rectangle, which is different from what is done by the drawing primitives for the black and white screen; this is why tv:alu-seta does not produce incorrect results as it normally would.

color:color-draw-line *x1 x2 y1 y2* &optional (*color* 17) *aluf screen*
Sets a line from (*x1*,*y1*) to (*x2*,*y2*) on the color screen to color *color*. *aluf* is used as in color:rectangle.

color:color-draw-char *font char x y* &optional (*color* 0) *screen*
Draws character *char* in font *font* at position (*x*,*y*) in color *color*. *font* is an ordinary black-and-white font.

Color fonts can also be created. A color font is composed of four-bit pixels just like the color screen. Using a color font, characters can be drawn with the normal character drawing primitives. When this is done, each bit of the color font pixel is combined with the corresponding bit of the screen pixel using the alu function. The alu function operates bit by bit just as it does on black-and-white screens, and is applied to many pixels in the neighborhood of the character, so tv:alu-seta should not be used.

color:make-color-font *bw-font* &optional *bit-list name-suffix*
Creates a color font from black-and-white font *bw-font*. *bit-list* is a list of four numbers, zero or one, which specifies the bits of the pixels of the color font that correspond to ones in the original font. Pixels that are zero in the original font remain all zero in the color font. *bit-list* defaults to (1 1 1 1).

The name of the resulting font is color- followed by the name of the original font, followed by the value of *name-suffix*.

Windows can be created on the color screen in the ordinary manner by specifying color:color-screen as the superior. When fonts are specified for such windows, if the font specifier names a black-and-white font, a color version of it is found or created. This color font is created with bit list (1 1 1 1). This is done by the :parse-font-specifier method of the color screen.

13.7 The System Menu

This section describes how to interface with and customize the system menu which pops up when you click twice on the right mouse button.

The system menu is an instance of flavor tv:dynamic-multicolumn-momentary-window-hacking-menu (see page 187), which means that its menu items are grouped by columns, and each column's items come from the value of a corresponding variable which is examined each time the menu is popped up in case more items have been added. This is to enable you to add items to the menu and control where they go. The most common column to add to is the third one, which lists various kinds of windows to select (somewhat like the System command), so a special interface is provided for adding to it.

tv:add-to-system-menu-programs-column *name form documentation* &optional *after*
Adds an item named *name* to the third column of the system menu. *form* is what to execute if the user clicks on the item, and *documentation* is the mouse documentation string.

after is the name of an item to add after (a string), or t to add at the top, or nil to add at the bottom.

tv:*system-menu-windows-column* *Variable*

A menu item list which forms the first column of the system menu.

tv:*system-menu-this-window-column* *Variable*

A menu item list which forms the second column of the system menu. By convention this is used for things that operate on the window that the mouse was pointing at when the system menu was brought up. They are implemented with :window-op menu items.

The Select item in the system menu pops up a momentary menu with a list of windows that the user might want to select. Not all the visible windows are included; usually a team of windows belonging to a single program is represented by a single entry since selection among the team is controlled by the program rather than the user. See section 3.2.1, page 35, for full details.

The Create item in the system menu pops up a menu for the user to choose a flavor of window to create.

tv:default-window-types-item-list *Variable*

A menu item list that is used by the system menu Create option, and by Create in the screen editor when operating on a screen.

In general, the screen editor can operate on the inferiors of any window. Then, the :pane-types-alist operation on that window is used to get the item list for possible flavors to create; see page 155. On a screen, the operation returns the value of this variable.

13.8 Window Resources

A *resource* is a pool of interchangeable objects that are available to be used temporarily and then returned to the pool (see section 5.12 of the Lisp Machine manual. Read that before you continue here).

Resources whose objects are windows are often useful. For example, there is a resource of windows of the right flavor to serve as "the system menu"; when you invoke "the" system menu, a window is allocated from the resource, and it is returned to the resource's pool when it is deactivated.

Normally one defines a resource with defresource. If the objects in the resource are windows, it is better to use instead a different function, tv:defwindow-resource. Allocating windows from resources, and returning them, is just like working with any other resources, and is documented in the Lisp Machine manual.

All the names described in this manual as resources are defined in this way.

tv:defwindow-resource *name parameters* &rest *options* *Special form*

Defines a resource of windows, named *name*. *parameters* are parameters on which the object can depend. Following the parameters specified is one additional parameter that is always defined: the window's superior. When you allocate a window from the resource, this parameter defaults to tv:mouse-sheet.

options is a list of alternating keywords and values. Neither the keywords nor the values are evaluated at the time that tv:defwindow-resource is executed, but sometimes the value becomes part of an expression that will be executed later (when a window is allocated from the resource).

The allowed keywords are

:initial-copies The value is the number of windows to create in the resource when the resource is defined. The default is one. The initial copies are made inferiors of tv:default-screen. Creating an initial copy is just a way of saving time the first time a window needs to be allocated from the resource.

:constructor See the definition of defresource. If it is not specified, tv:defwindow-resource provides a default, which calls make-instance with arguments taken from the :make-window option.

:make-window
The value should be a list of a flavor name followed by keyword arguments. This list will be consed into a make-window form to get the constructor for the resource.

:reusable-when
The value should be :deexposed, :deactivated. If this keyword is not specified, then windows of the resource can be allocated to requesters if they have been explicitly returned to the pool and are not locked. :deexposed means that any window that is not exposed is considered to have been returned to the pool. :deactivated means that any window that is not active is considered to have been returned to the pool.

tv:window-resource-list *Variable*

A list of the names of all window resources defined with tv:defwindow-resource.

Example: the system menu is created thus:

```
;Resource of system menus
(defwindow-resource system-menu ()
  :make-window
  (dynamic-multicolumn-momentary-window-hacking-menu
    :column-spec-list
    '(("Windows" *system-menu-windows-column*
       :font fonts:hl12i)
      ("This window" *system-menu-this-window-column*
       :font fonts:hl12i)
      ("Programs" *system-menu-programs-column*
       :font fonts:hl12i))
    :save-bits t)
  :reusable-when :deexposed)
```

13.9 The Cold Load Stream

User programs that make use of the screen organization and standardization facilities provided by the window system are frequently in a somewhat difficult position. If that interface to the window system does not work, there seems to be no way at all to find out what is going on. Similarly, debugging code associated with switching between windows can be difficult since there may be no place to print debugging output at the time such code is executing.

One way to debug such problems is to use the *cold load stream*. This is the stream used in constructing the initial Lisp Machine environment, before the window system itself has been loaded. It has the advantage that it does not attempt to interface with the rest of the window system, or vice versa. It will never deexpose any windows or lock any locks. It types out one character at a time, by calling the microcode directly, and has very simple-minded ideas about end of line exceptions and more breaks.

tv:cold-load-stream *Variable*

The cold load stream is the value of this variable.

When the cold load stream is "waiting" for type-in, it does not actually wait; in fact, it loops until a character appears, with scheduling turned off, blinking its own special blinker by hand. The who line is not updated. Also, the chaosnet processes do not get to run. If the machine stays in this state too long, all chaosnet connections will be lost.

Whenever the system gets an error in the keyboard process, the scheduler or the mouse process, the debugger uses the cold-load-stream rather than terminal-io. You also have the option of requesting this if there is an error in a process whose terminal-io is a window that is not exposed and cannot be exposed because of locked windows. (You will be queried, using the cold load stream, to choose between this and a couple of other possibilities.)

When you exit from the debugger after it was using the cold load stream for one of these reasons, it will ask you whether to "restore the screen". Normally you should say Yes; then the screen contents will go back to what they were before the debugger was entered.

It is often preferable to use the cold load stream for debugging window problems even when the normal alternatives are available. This is because the operation of the debugger using a window for I/O may interfere with the window phenomena being debugged. Use of the cold load stream will avoid these problems. You can request use of the cold load stream by setting debug-io to the value of tv:cold-load-stream before you run your test. Once this has been done, not only errors but breakon and Meta-Break as well will use the cold load stream. To turn off use of the cold load stream for all debugger invocations, set debug-io back to nil.

You can also force trace output into the cold load stream by setting trace-output. Note that you must not set trace-output to nil when done; you must save its original value and set it back to that.

When the cold load stream is used because you have set one of the stream variables to it, you do not get the chance to restore the screen. It is not so easy to define how to do that "right" in this case; if it were done after each exit from the debugger, you would not get to see the history of multiple entries to the debugger.

The program can invoke a break loop using the cold load stream by calling tv:kbd-use-cold-load-stream. Type Resume to continue. Note that when the break is entered, the package you are typing into is typed out, because the package in the who-line is not going to be correct for this break loop.

You the user can request such a break loop by typing Terminal Call or by clicking on Emergency Break item in the system menu. You can get your program into the debugger using the cold load stream, without having made advance preparation, by getting a break loop in this fashion, setting debug-io to the cold load stream, exiting, and typing Meta-Break.

Also, it is often useful to get a cold load stream break loop and call eh on various processes or stack groups.

13.10 The Window-Based Debugger

The window-based debugger is an alternative to the usual debugger; it performs the same functions but displays graphically rather than using sequential stream I/O. You invoke the window-based debugger by typing Control-Meta-W while in the usual debugger. You can switch back and forth between the two debuggers any number of times while handling a single error.

The debugger window is divided into six panes. At the bottom is a Lisp-listener-like window, which ordinarily provides a read-eval-print loop similar to the regular keyboard debugger. More commands are available by using the mouse in the other windows as described below.

At the top is a display of the disassembled or ground code for the currently selected stack frame, depending on whether or not it is compiled. It has a scroll-bar, but is otherwise not sensitive to the mouse.

Next are the args and locals windows, side by side, displaying the names and values of the arguments to the current stack frame and its local variables; they are grayed out if there are none. They also have scroll bars. Clicking the mouse on the name of an argument will print the name and the value in the Lisp window. Clicking on just the value will print it in the Lisp

window. The mouse will highlight any relevant quantity that you are pointing to.

Next is the stack window, which displays in a pseudo-list format the functions and arguments on the stack. Clicking on a function or argument or sublists of them will cause them to be printed in the Lisp window as in the argument or local windows. Also, clicking the mouse to the left of a line containing a particular stack frame will make the debugger select that frame, changing what the above three windows show.

Below this, and above the Lisp window, is the command menu for the debugger window. The available commands are:

What error Reprints the error message for the current error, in the Lisp window.

Exit Window EH
Exits the debugger window, returning to the regular debugger.

Abort Program
Like **Abort** in the regular debugger.

Arglist Asks for the name of a function, which can be typed on the keyboard, or moused if it is on the screen. Picking an actor or a closure will ask for the message name to that actor and print the arguments to its method for that message. Picking a line of a stack frame from the stack window will try to align the printout of the arguments with what value was supplied in that position in that frame.

Edit Reads a name of a function in the same fashion as the **Arglist** command and invokes the editor on that function.

Retry Attempts to restart the current frame, like the **Control-Meta-R** command in the regular debugger.

Return a Value
Asks for the name of a value (which can be selected with the mouse) and returns it from the current frame, like **Control-R** in the regular debugger.

Proceed Proceeds from the error. Clicking left on **Proceed** is like typing **Resume** in the regular debugger. Clicking right on **Proceed** gets you a menu of available proceed types, from which you can select one. This is equivalent to using one of the available **Super** commands in the regular debugger. If proceeding asks for an object to return, you can specify it with keyboard input or by pointing to a value with the mouse.

Set arg Select an argument or local with the mouse and type or mouse a new value to be substituted in.

Search Like the **Control-S** command, except that the mouse can be used.

Throw Like **Control-T** in the regular debugger, it asks for a tag and a value and throws there. The mouse can be used to specify the tag and value.

T
NIL Ordinarily just supply those symbols as arguments or values for other commands. These can also be used to answer yes-or-no questions.

14. Choice Facilities

The window system contains several facilities to allow the user to make choices. These all work by displaying some arrangement of choices in a window. By pointing to one with the mouse the user can select it. The details (how the choices are specified, what the user interaction looks like, and what happens when a choice is selected) vary widely, which is why there are several separate facilities.

Each choice facility is implemented as a family of window flavors, providing several variations on the basic facility. For those who don't want to create their own window, each facility provides an easy-to-use function interface that temporarily pops up a window of the appropriate flavor. The function interfaces will be described first in each section. Following the function interfaces there is documentation on how to create and use a window which has the facility.

This document does not cover how to modify these facilities to provide your own specialized versions, except in the simplest ways. That is certainly a reasonable thing to want to do. In order to do it you will need to read some of the code that implements the facility in question, for instance to learn about window instance variables and about internal operations that you might want to redefine or put daemons on.

Some portions of these facilities execute in the process that calls them, while other portions execute in the mouse process. All Lisp evaluation with which the user is concerned takes place in the user's process when using the facilities described in this document, with a very few exceptions which are noted when they occur. Thus the user may freely use side-effects (both special variables and *throw) and need not worry that an error in his program will interfere with mouse tracking.

14.1 Menus

A menu is an array of choices, each identified by a word or short phrase. You can select one of the choices by moving the mouse near it, which causes it to be highlighted (a box appears around it), and then clicking any mouse button.

What happens when you select one of the choices depends on the particular type of menu. Typically the choices in a menu might be commands to some program or choices for what a command should operate upon.

The system automatically chooses the arrangement of the choices and the size and shape of the window. Naturally there are ways for the user to control this if necessary.

To see an example of a menu, click the right-hand mouse button twice, causing the system menu to appear.

14.1.1 Menu Items

A menu has a list of items; each item represents one of the choices offered. An item tells the menu what to display and what to do if the user selects (clicks on) it. "What to do" specifies both what value to return and a possible side effect.

Response to selection of an item is implemented by the :execute operation, which is always sent in the user process (rather than the mouse process). Thus side effects occur in the appropriate process. The returned value comes back to the user from tv:menu-choose, :choose, or :execute depending on how the menu is used. This will be explained in detail later.

An item can take any of the following forms:

a string or a symbol
: The string or symbol is both what is displayed and what is returned. There are no side-effects.

a cons (*name* . *atom*)
: *name* (a symbol or a string) is what to display, and *atom* is what to return. There are no side-effects.

a list (*name value*)
: *name* is a string or a symbol to display, and *value* is any arbitrary object to return. There are no side-effects.

a list (*name type arg option1 arg1 option2 arg2...*)
: This is the most general form. *name* is a string or a symbol to display. *type* is a keyword symbol specifying what to do, and *arg* is an argument to it. The *options* are keyword symbols specifying additional features desired, and the *args* following them are arguments to those options.

If nil is supplied as a menu item, it is ignored completely. It takes up no space in the menu.

A list of items is sometimes called an "item alist" since most forms of menu item look like alist elements mapping strings into what to do about them.

The possible values of *type* in the most general form of menu item are:

:value
: *arg* is what to return. There are no side-effects.

:eval
: *arg* is a form to be evaluated. Its value is returned.

:funcall
: *arg* is a function of no arguments to be called. The value it returns is returned.

:no-select
: This item cannot be selected. Moving the mouse near it will *not* cause it to be highlighted. This is useful for putting comments, headings, and blank spaces into menus. *arg* is ignored, but must be present to make the item be the form that has a *type* keyword in it.

:kbd
: *arg* is sent to the selected window via the :force-kbd-input operation. Typically it is either a character code, which is to be treated as if it were typed in from the keyboard, or a list (a blip), which is a command to the program (see section 5.2, page 52). Use of :kbd produces an effect like the effect of using a command menu (see section 14.1.5, page 184).

:menu *arg* is a new menu to choose from; it is sent a :choose message and the result is returned. Normally *arg* would be a pop-up menu. If *arg* is a symbol it gets evaluated.

:menu-choose
arg is a list (*label* . *menu-items*). The car and cdr are passed as arguments to tv:menu-choose, popping up another menu, and the result of choosing from that menu is returned. *menu-items* is another list of menu items.

:buttons *arg* is a list of three menu items. The item actually chosen (i.e. the item to be executed) is one of these three, depending on which mouse button was clicked. The order in the list is (*left middle right*). The three menu items in the list will be used only for execution, not for display, so it does not matter what they have as the string to be displayed (it can be nil), and there is no point in giving them :font or :documentation keywords. These should go in the main menu item, the one that contains the :buttons.

:window-op *arg* is a function of one argument. The argument is a list of three elements: the window the mouse was in before this menu was popped-up and the *x* and *y* coordinates of the mouse at that time. This item type is handled by the :execute-window-op menu operation, which the flavor tv:menu does not implement. The flavor tv:window-hacking-menu-mixin provides a method to implement it.

The menu item modifier keywords are:

:font This keyword is followed by a font or a symbol that is the name of a font. The item is displayed in that font instead of the menu's default font.

:documentation
This keyword is followed by a string, which briefly describes this menu item. When the mouse is pointing at this item, so that it is highlighted, the documentation string will be displayed in the documentation line at the bottom of the screen.

:bindings This keyword is followed by a list of bindings to be made, suitable for passing to the function progw (see section 3.1 of the Lisp Machine manual). These bindings are made before evaluating, funcalling, sending a message to a window, etc. If :buttons is used with :bindings, the :bindings must appear inside the menu item within the :buttons to have an effect on the final result.

Here are some examples of menu item lists:

Three items, that display as FOO, BAR and LOSE, and return the symbols foo, bar and lose when chosen.

```
(foo bar lose)
```

Another way of specifying the same thing, using more general syntax:

```
(("FOO" :value foo)
 ("BAR" :value bar)
 ("LOSE" :value lose))
```

Putting FOO in italics and adding documentation for the who line:

```
(("FOO" :value foo :font fonts:tr12i
  :documentation "Choose to FOO")
 ("BAR" :value bar
  :documentation "Request a BAR")
 ("LOSE" :value lose
  :documentation "Don't win."))
```

Some other type keywords are used here. The value of the :choose operation will be a keyword such as :read or :write, the value returned by the function read, or whatever the buffer-op-menu returns.

```
(("File" :buttons
  ((nil :value :read)
   (nil :value :write)
   (nil :menu-choose
    ("File Operation"
     ;; Item list of menu obtained for click-right on File.
     ("Read" :value ;read
      ;documentation "Read a file")
     ("Write" :value ;write
      ;documentation "Write a file")
     ("Rename" :value ;rename
      ;documentation "Rename a file")
     ("Delete" :value ;delete
      ;documentation "Delete a file"))))
  :documentation
  "L: Read file.  M: Write file.  R: Menu.")
 ;; The following makes a blank line in a one-column menu.
 ("" :no-select nil)
 ;; We assume that buffer-op-menu is a variable whose value is a menu.
 ("Buffer" :menu buffer-op-menu
  :documentation "Operate on this buffer")
 ("Read" :buttons
  ((nil :eval (read))
   (nil :eval (read)
    :bindings ((base 10.)))
   nil)
  ;documentation
  "L: Read sexp.  M: Read sexp, base ten."))
```

Here we show the use of :bindings. This expression creates a menu item which contains a host taken from the local variable host. When the menu item is chosen, the function hack-host will be called with the appropriate host as the value of the special variable host-to-hack.

```
'(("Hack This Host" :funcall hack-host
   :bindings ((host-to-hack ',host))
   :documentation "Do some hacks to this host."))
```

tv:menu-execute-mixin *Flavor*

This flavor defines the :execute operation to process a menu item according to the rules described above.

:execute *item* *Operation on* tv:menu-execute-mixin

Processes *item*, computing and returning the "value to return" according to the rules described above. Everything about the meaning of menu items, except as far as it affects displaying the menu, is determined by what the :execute operation does, so by redefining this operation you can implement new types of menu items. The overall format must be as described, however, because displaying the menu checks for the type :no-select and for the :font and :documentation modifier keywords.

:execute-no-side-effects *item* *Operation on* tv:menu-execute-mixin

Processes *item*, computing and returning the "value to return", provided that this can be done without side effects. If computing the value to return might possibly have side effects (such as for item types :eval, :funcall, :kbd, :window-op, :menu and :menu-choose), the value is not computed and nil is returned.

This operation is typically used to find the item in a given item list that would return a particular value if selected.

tv:menu-item-string *item* &optional *item-default-font menu*

Returns the string to display for *item*. The font to use is returned as the second value; it defaults to *item-default-font* if not specified by the item. *item-default-font* itself defaults to the current font of the menu as a window.

menu is the menu that *item* is for; it is used for interpreting font specifications in *item* itself.

If you are not interested in the font, you can omit the last two arguments.

tv:window-hacking-menu-mixin *Flavor*

Provides for the :window-op item type by implementing the :execute-window-op operation. This involves remembering the mouse position and the window under the mouse at the time the menu is exposed.

14.1.2 Easy Menu Interface

tv:menu-choose *item-list* &optional *label near-mode default-item superior*

Pops up a menu and allows the user to make a choice with the mouse. When the choice is made, the menu disappears and the chosen item is executed. The value of that item is returned as the first value of tv:menu-choose, and the item itself is returned as the second value.

If the user moves the mouse out of the menu and far away, the menu disappears and tv:menu-choose returns nil.

item-list is a list of items as described above.

label is a string to be displayed at the top of the menu, or nil (the default) to specify the absence of a label.

near-mode is where to put the menu. It defaults to the list (:mouse) and must be an acceptable argument to tv:expose-window-near.

default-item is the item over which the mouse should be positioned initially. This allows the user to select that item without moving the mouse. If *default-item* is nil or unspecified, the mouse is initially positioned in the center of the menu.

superior is the sheet of which the menu should be an inferior. The default is tv:mouse-sheet, which is usually a screen.

Example:

```
(tv:menu-choose '(("Read" :value foo) ("Write" :value bar))
                "Direction")
```

will return foo or bar (or nil if the user moves the mouse out of the menu).

tv:mouse-y-or-n-p *string*

Asks the user to answer Yes by clicking on a small window or No by moving the mouse out of it. The window is a menu which displays a single item, *string*.

The value is t if the user clicks on the menu, or nil if he moves the mouse out of it.

14.1.3 Geometry

The way a menu is displayed is described by six parameters that are collectively called its *geometry*. Each of these parameters may be specified as a constraint, or may be allowed to default based on the item list and the parameters that are constrained.

There are two styles of arranging the choices in the menu. They can be in an array of rows and columns, or they can be "filled", that is, each line has as many choices as will fit with a reasonable amount of white space in between. In columnar format, each line has the same number of choices: the same as the number of columns. This is not true in filled format. Filled format is specified by giving zero as the number of columns.

The geometry is represented as a list of six elements, one for each parameter.

columns	The number of columns, or 0 for filled format.
rows	The number of rows.
inside width	The inside-width of the window, in pixels.
inside height	The inside-height of the window, in pixels.
maximum width	The maximum width of the window, in pixels. This parameter is meaningful only as a constraint, since the way the menu is displayed is sufficiently

described by its actual width. If the maximum width is constrained, the system will prefer to choose a tall skinny shape rather than exceed it.

maximum height The maximum height of the window, in pixels. This parameter is meaningful only as a constraint, since the way the menu is displayed is sufficiently described by its actual height. If the maximum height is constrained, the system will prefer to choose a short fat shape rather than exceed it.

For the first four parameters, one must distinguish between the current value and the imposed constraint. The constraint values may be **nil**, meaning "do not constrain this parameter". The current values cannot be **nil**.

The last two parameters exist only as constraints, and may be **nil**.

The actual display of a menu is based on four parameters: the number of rows, the number of columns (or whether to use fill mode), the height and the width. Some of these may be specified by constraints; others may be specified on a one-time basis when the menu is displayed; the rest are chosen based on the ones already known, and on the item list.

The default geometry constraints are all **nil**, meaning that the system can choose the size and shape freely, based on the specified item list. The default shape is an upright golden rectangle, using columnar format with as many columns as fit in the width. Most small menus will have only one column.

If both the height and with are specified (either precisely or indirectly) in such a way that not all the items can fit, the menu will have a scroll bar and the user will have to scroll to see all the items.

When the item list of a menu is changed, the display of the menu is recomputed based on the new item list and the geometry.

The following init-plist options to a menu will initialize the geometry:

:geometry *list* *Init option for* **tv:menu**

Sets the complete geometry to *list*, a list of six elements. Example:

```
(make-instance 'tv:menu ':geometry (0 nil 300 nil nil 500))
```

makes a filled menu that is 300 pixels wide; when its item list is specified or changed it will become as tall as necessary to display all the items as long as that does not exceed 500 pixels. Beyond that point, it will be 500 pixels high and will require the user to scroll.

:rows *n-rows* *Init option for* **tv:menu**

Sets the number of rows.

:columns *n-columns* *Init option for* **tv:menu**

Sets the number of columns.

:fill-p *t-or-nil* *Init option for* tv:menu

Specifies whether to use filled format.

:default-font *font* *Init option for* tv:menu

Sets the default font, the font in which items which do not specify a font are displayed. If this is not specified, it defaults to the standard font for the purpose :menu on the screen the menu is on (see page 86).

The following operations manipulate the geometry of a menu:

:geometry *Operation on* tv:menu

Returns a list of six things, the menu's geometry. These are the constraints, with nil in unspecified positions; contrast :current-geometry.

:current-geometry *Operation on* tv:menu

Returns a list of six things, which are the geometry corresponding to the actual current state of the menu.

The first four elements are actually sufficient to describe the current state. These are never nil.

The last two elements returned are the constraint values for the maximum width and height, since there are no current values to return. These may be nil.

Contrast this with :geometry.

:set-geometry *Operation on* tv:menu

&optional *columns rows inside-width inside-height max-width max-height*

Sets the geometry (the constraints) from the arguments. The menu may change its shape and redisplay as a result.

Note that this takes six arguments rather than a list of six things as you might expect. This is because you frequently want to omit most of the arguments.

An explicit argument of nil means to make that aspect of the geometry unconstrained. An omitted argument or an argument of t means to leave that aspect of the geometry the way it is (if unconstrained, it remains so).

:fill-p *Operation on* tv:menu

:set-fill-p *t-or-nil* *Operation on* tv:basic-menu

Get or set the menu's fill mode, t if it displays in filled format rather than columnar format. These are special cases of the :geometry/:set-geometry operations.

:set-default-font *font* *Operation on* tv:menu

Sets the default font, the font in which items that do not specify a font are displayed. This recomputes the current display based on the constraints.

:set-edges *left top right bottom* &optional *option* *Operation on* tv:menu

This operation, in addition to setting the current position and size of the menu, also makes the specified size be a permanent constraint for the menu unless *option* is :temporary. In that case, the menu is redisplayed with the specified edges for now, but if it is redisplayed again for any reason, the permanent constraints (or lack of them) otherwise specified will re-emerge.

tv:menu-compute-geometry *draw-p* &optional *inside-width inside-height*

Computes the current display parameters from the constraints and the item list and default font. *inside-width* and *inside-height* serve as constraints for this time only, overriding any permanent constraints for those parameters.

If *draw-p* is non-nil, the menu is actually redrawn.

This function is a subroutine of various menu methods, and self must be the menu.

:minimum-width *Operation on* tv:menu

This returns the minimum width for the menu, as required to display its label. This is used in deciding how to display the menu.

Other menu flavors can redefine this operation to force the menu to be wide enough for some purpose.

14.1.4 Ordinary Menus

These are the *basic* and *mixin* flavors for the ordinary kinds of menus. They cannot be instantiated themselves but are useful to know about. Other kinds of menus are discussed in later sections.

tv:basic-menu *Flavor*

Everything else is built on this. All the operations documented here as being defined on tv:menu are really defined by this flavor.

tv:item-list *Instance variable of* tv:basic-menu

The item list of the menu.

tv:last-item *Instance variable of* tv:basic-menu

The last item actually selected with a mouse click in this menu, or nil if none has been selected yet. Used for positioning the mouse when a momentary menu pops up.

tv:current-item *Instance variable of* tv:basic-menu

The item which the mouse is pointing at, or nil.

tv:chosen-item *Instance variable of* tv:basic-menu

Set each time an item is selected, to that item. Waiting for an item to be selected is done by setting this variable to nil and waiting for it to become non-nil.

tv:geometry *Instance variable of* tv:basic-menu

The geometry (constraints) of the menu, a list of length 6.

tv:basic-momentary-menu (tv:hysteretic-window-mixin tv:basic-menu) *Flavor*

This is a kind of menu, often referred to as a "pop up" menu, which is only momentarily on the screen. A :choose operation on a menu of this flavor causes it to position itself where the mouse is. When the user selects an item in the menu, or alternatively moves the mouse far away from the menu, the menu disappears and deactivates, the mouse warps back to where it was when the menu appeared, and the :choose operation returns the chosen item or nil.

These are the interesting instantiable menu flavors:

tv:menu *Flavor*

(basic-menu borders-mixin top-box-label-mixin basic-scroll-bar minimum-window)

This is tv:basic-menu with borders and a label on top. The default is for there to be no label but you can specify one with the :label init-plist option or the :set-label operation.

tv:momentary-menu *Flavor*

This is tv:basic-momentary-menu mixed with the right other flavors. Momentary menus were described at the beginning of this section.

tv:momentary-menu &optional (*superior tv:mouse-sheet*) *Resource*

A resource of momentary menus.

tv:temporary-menu (tv:temporary-window-mixin tv:menu) *Flavor*

This is a menu that is a temporary window; that is, it saves the bits of the windows underneath it when it is exposed. It is not a momentary menu, and therefore it does not expose or deexpose itself automatically.

It is appropriate to use a temporary menu rather than a momentary menu when you want to pop a menu up and make several choices from it before popping it back down, or if you don't want to allow the user the option of choosing nothing by moving the mouse out of the window.

tv:momentary-window-hacking-menu *Flavor*

(tv:window-hacking-menu-mixin tv:momentary-menu)

A momentary menu with the window-hacking mixin. See page 177.

tv:momentary-menu &optional (*superior* tv:mouse-sheet) *Resource*

This is a resource of momentary menus. tv:menu-choose allocates a window from this resource.

The following operations are useful on any flavor of menu. Also listed are init options which are useful with any flavor of menu. Operations and init options that specifically have to do with the shape and arrangement of the menu are listed in the section on geometry (section 14.1.3, page 178).

:item-list *Operation on* tv:menu
:set-item-list *item-list* *Operation on* tv:menu
Get or set the list of items (choices). Setting the item list recomputes the geometry and redisplays the menu.

:item-list *items* *Init option for* tv:menu
The item list can be set when the menu is created.

:choose *Operation on* tv:menu
Exposes the menu if it is not already exposed, then waits for a selection to be made with the mouse. The selection is :execute'd and the resulting value is returned. A momentary menu will return nil from :choose if the mouse is moved far out of it, and in any case will pop down before returning.

:execute *item* *Operation on* tv:menu
Given an item that was selected, performs the appropriate side-effects and returns the appropriate value. For most kinds of menus, this operation is invoked automatically as part of the :choose operation, but command menus (see below) require the user program to invoke :execute explicitly if it is desired.

:move-near-window *window* *Operation on* tv:menu
Exposes the menu above or below *window*, giving it the same width.

:center-around *x y* *Operation on* tv:menu
This operation is implemented by all windows, but menus handle it a little differently. The window is positioned so that the last item chosen appears at the specified coordinates (in the superior), if possible. If this would cause the menu to stick outside of its superior, it is offset slightly to keep it inside. The actual coordinates of the center of the appropriate item are returned (you might want to put the mouse there). Momentary menus use this to put the menu in such a place that the mouse will be right over the last item chosen.

:current-item *Operation on* tv:menu
Gets the item the mouse is currently pointing at (nil if none). In most cases if you are using this operation you are doing something wrong.

:chosen-item *Operation on* tv:menu
:set-chosen-item *item* *Operation on* tv:menu
Get or set the item that has been chosen by the mouse and is being communicated back to the controlling process. In most cases if you are using these operations you are doing something wrong.

:last-item *Operation on* tv:menu
:set-last-item *item* *Operation on* tv:menu
Get or set the item that was chosen by the mouse the last time this menu was used. When a momentary menu is exposed near the mouse by the :choose operation, it will put the mouse over this item so that it easy to choose it again.

:column-row-size *Operation on* tv:menu

Returns two values: the width of a column in bits and the height of a row in bits.

:item-cursorpos *item* *Operation on* tv:menu

Returns two values, like :read-cursorpos, giving the coordinates of the center of the displayed representation of *item*. The result is nil if the item is scrolled off the display.

:item-rectangle *item* *Operation on* tv:menu

Returns four values, the coordinates of the rectangle enclosing the displayed representation of the specified item. The result is nil if the item is scrolled off the display. Note that the returned coordinates are *inside* coordinates and that they include a 1-pixel margin around the item.

:menu-draw *Operation on* tv:menu

Draws the menu's display. :menu-draw is invoked automatically by the system when required, and should not be used in application programs. However, user-defined menu flavors may redefine the operation or add daemons to it.

:mouse-buttons-on-item *buttons-down-mask* *Operation on* tv:menu

This operation is invoked by the mouse process when the mouse is clicked on an item. It is completely responsible for whatever should be done in the mouse process at that time. Its default definition is to record the chosen item and process the item type :buttons when that is used.

The instance variable tv:current-item or the :current-item operation can be used to find out which item the mouse is on.

The operations :scroll-position, :scroll-to and :scroll-bar-p are also defined for communication with the scroll bar. See section 10.5.1, page 124.

14.1.5 Command Menus

tv:command-menu-mixin *Flavor*

The menus described so far are driven by the :choose operation; that is, the program decides when it is time for the user to choose something in the menu. In some applications it should be the user who decides when to choose something from a menu. For example, in Peek, the user can select a new mode with the menu at any time, but Peek cannot spend all its time waiting for the user to do this.

The command menu is designed for such applications. When an item in a command menu is chosen, the menu puts a blip into its input buffer. The blip is a list

(:menu *item button-mask menu*)

which can read as an input character with the :any-tyi operation on any other window sharing the same input buffer. *item* is the menu item that was clicked on, *button-mask* says which mouse button was used (as in tv:mouse-last-buttons; see page 116), and *menu* is the menu that was clicked on, in case you are using more than one.

Usually a command window is part of a team of windows managed by a single process and sharing a single input buffer. Menu clicks generate input that is read in a single stream together with mouse clicks on the other windows and keyboard input. For example, Peek and the inspector both use command menus in this way. Once the controlling process reads the blip, it can do (funcall *menu* ':execute *item*) if it wishes the item to be processed in the usual way for menu items.

tv:command-menu *Flavor*

This is tv:command-menu-mixin mixed with tv:menu to make it instantiable.

:io-buffer *Operation on* tv:command-menu

:set-io-buffer *io-buffer* *Operation on* tv:command-menu

These operations get or set the I/O buffer in which a command-menu sends stores a blip when an item is selected.

:io-buffer *io-buffer* *Init option for* tv:command-menu

The input buffer to be used by a command menu is usually specified when it is created.

tv:io-buffer *Instance variable of* tv:command-menu

This is where the input buffer is recorded.

tv:command-menu-abort-on-deexpose-mixin *Flavor*

When a command menu built on this flavor is deexposed, it automatically "clicks" on its Abort item. In other words, the :deexpose method for this flavor searches the item list for an item whose displayed representation is "ABORT" (case is not significant). If such an item is found, a blip is sent to the input buffer claiming that that item was clicked on with the Left button.

14.1.6 Dynamic Item List Menus

Dynamic item list menus dynamically recompute the item list at various times. Whenever the program makes an explicit request to use the menu, the menu checks automatically to see whether its item list has changed.

tv:abstract-dynamic-item-list-mixin *Flavor*

This mixin causes a menu to invoke the :update-item-list operation at various times. This operation receives no arguments, and its value is ignored; it should update the item list if appropriate.

This mixin does not *define* the :update-item-list operation, however. Each user of the mixin must define this operation to update the item list as he desires.

:update-item-list *Operation on dynamic item list menus*

Sent by the system, this operation should be defined by the user to do a :set-item-list if the item list should change.

Note that this operation may be invoked in various processes, so your definition should use only global variables (and data structure it can find from the menu itself).

tv:dynamic-item-list-mixin *Flavor*

Provides for a form which is evaluated to get the menu's item list, kept in the tv:item-list-pointer instance variable. The :update-item-list operation is defined to evaluate the form and set the item list to the form's value.

tv:item-list-pointer *Instance variable of* tv:dynamic-item-list-mixin

This is the form evaluated to recompute the current item list.

:item-list-pointer *Operation on* tv:dynamic-item-list-mixin
:set-item-list-pointer *form* *Operation on* tv:dynamic-item-list-mixin

Get or set the form.

:item-list-pointer *form* *Init option for* tv:dynamic-item-list-mixin

Initializes the form.

These are menu flavors that are just combinations of this with other flavors:

tv:dynamic-momentary-menu *Flavor*

A momentary menu with the dynamic item-list mixin.

tv:dynamic-momentary-window-hacking-menu *Flavor*

A momentary menu with both the dynamic item-list mixin and the window-hacking mixin.

tv:dynamic-temporary-menu *Flavor*

A temporary menu with the dynamic item-list mixin.

tv:dynamic-temporary-command-menu *Flavor*

A command menu with the temporary and dynamic item-list mixins.

tv:dynamic-temporary-abort-on-deexpose-command-menu *Flavor*

A command menu with the temporary, abort-on-deexpose, and dynamic item-list mixins.

tv:dynamic-multicolumn-mixin *Flavor*

This mixin, to be used with tv:abstract-dynamic-item-list-mixin, makes a menu of several columns, in which each column's items are independently dynamically recomputed. The system menu is such a menu.

The columns are specified by the instance variable tv:column-spec-list. The value is a list; each element specifies one column of the menu, and looks like this:

(*heading item-list-form options...*)

heading is a string to be displayed (as a :no-select item) at the top of the column. *item-list-form* is a form to be evaluated to produce the list of items for the column. It should have no side effects and may be evaluated in any process. The *options* are modifier keywords and values, such as are found in menu items. These modifiers apply to the column heading only. The most useful one is the :font keyword. For example, the system menu uses this column spec list:

```
(("Windows" tv:*system-menu-windows-column*
  :font fonts:hl12i)
 ("This window" tv:*system-menu-this-window-column*
  :font fonts:hl12i)
 ("Programs" tv:*system-menu-programs-column*
  :font fonts:hl12i))
```

Each column's item list form is a symbol, the name of a special variable.

tv:column-spec-list *Instance variable of* tv:dynamic-multicolumn-mixin
This instance variable holds the column spec list.

:column-spec-list *Init option for* tv:dynamic-multicolumn-mixin
Initializes the column spec list.

:column-spec-list *Operation on* tv:dynamic-multicolumn-mixin
:set-column-spec-list *specs* *Operation on* tv:dynamic-multicolumn-mixin
Get or set the column spec list.

tv:dynamic-multicolumn-momentary-menu *Flavor*
This is an instantiable, momentary mixture of tv:dynamic-multicolumn-mixin.

tv:dynamic-multicolumn-momentary-window-hacking-menu *Flavor*
Similar to the previous, but includes tv:window-hacking-menu-mixin. The system menu is an instance of this flavor.

14.1.7 Multiple Menus

A multiple menu asks the user to select any combination of menu items rather than a single item. The menu has a "choice box" (usually named "Do it") at the bottom in addition to its menu items. Clicking on a menu item selects it or unselects it; the selected items are displayed in inverse video. Clicking on the "Do it" box specifies the set of items currently selected.

The :choose operation on a multiple menu returns as its first value a list of the values of the items selected by the user.

tv:multiple-menu-choose *item-list* &optional *label near-mode highlighted-items superior*
Pops up a menu and allows the user to choose any subset of the available items. The user finalizes his choice by clicking on the "Do It" box at the bottom of the menu. At this time, tv:multiple-menu-choose returns as its first value a list of the results of executing all the chosen menu items. The second value of tv:multiple-menu-choose is t in this case.

If the user moves the mouse out of the menu and far away, the menu disappears and tv:menu-choose returns nil for both values. The second value enables the caller to distinguish between a refusal to choose and choosing the empty set of items.

item-list is a list of menu items as described above. *highlighted-items* is a list of some of the same items; these are the items to include, initially, in the set to be chosen. The user can add items to the set or remove items from the set.

The elements of *highlighted-items* must be memq in *item-list* for proper functioning.

label is a string to be displayed at the top of the menu, or nil (the default) to specify the absence of a label.

near-mode is where to put the menu. It defaults to the list (:mouse) and must be an acceptable argument to tv:expose-window-near.

superior is the sheet of which the menu should be an inferior. The default is tv:mouse-sheet, which is usually a screen.

Example:

```
(tv:multiple-menu-choose '(rice spinach water coke)
                         "Pick some foods" nil '(water))
```

might return the list (rice spinach water) if the user clicked on the entries for rice and spinach, and did not turn off water.

```
(let ((items '(("Rice" :value rice)
               ("Spinach" :value spinach)
               ("Water" :value water)
               ("Coke" :value coke))))
  (tv:multiple-menu-choose items "Pick some foods" '(:mouse)
                           (list (assoc "Water" items))))
```

can return the same possible values, but has a prettier display.

tv:margin-multiple-menu-mixin *Flavor*

Gives a menu the ability to have multiple items selected in this manner.

tv:multiple-menu (tv:margin-multiple-menu-mixin tv:menu ...) *Flavor*

A menu that behaves as described above. This is a combination of tv:multiple-margin-menu-mixin with tv:menu.

tv:momentary-multiple-menu *Flavor*
(tv:margin-multiple-menu-mixin tv:momentary-menu ...)

A multiple menu that is also momentary.

tv:momentary-multiple-menu &optional (*superior tv:mouse-sheet*) *Resource*

A resource of momentary multiple menus, used by tv:multiple-menu-choose.

:add-item *item* *Operation on* tv:margin-multiple-menu-mixin

Adds *item* to the item list of the multiple menu, initially unhighlighted. All the existing items remain, and remain highlighted if they already were.

:set-item-list *item-list* *Operation on* tv:margin-multiple-menu-mixin

In addition to setting the item list and redisplaying the menu, all the items start out unhighlighted.

:special-choices *items* *Init option for* tv:margin-multiple-menu-mixin

This init option is equivalent to the :menu-margin-choices init option (which is provided by our component flavor tv:menu-margin-choice-mixin). It is provided for historical compatibility. *items* is a list of menu items that specify the choice boxes desired and what to do if they are clicked on.

tv:menu-highlighting-mixin *Flavor*

Provides for some of the menu items to be highlighted with inverse video. This is typically used with menus of "modes", where the modes currently in effect are highlighted. The menu items corresponding to modes will typically be set up so that when executed, they adjust the highlighting to reflect the enabling or disabling of a mode.

This flavor is used in tv:margin-multiple-menu-mixin.

tv:highlighted-items *Instance variable of* tv:menu-highlighting-mixin

The list of items currently highlighted.

:highlighted-items *Operation on* tv:menu-highlighting-mixin
:set-highlighted-items *list* *Operation on* tv:menu-highlighting-mixin

Get or set the list of highlighted items.

:highlighted-items *items* *Init option for* tv:menu-highlighting-mixin

When a menu with the menu-highlighting mixin is created, the list of items to be initially highlighted may be specified. The default is nil.

:add-highlighted-item *item* *Operation on* tv:menu-highlighting-mixin
:remove-highlighted-item *item* *Operation on* tv:menu-highlighting-mixin

Make *item* be highlighted, or make it stop being highlighted.

:highlighted-values *Operation on* tv:menu-highlighting-mixin
:set-highlighted-values *list* *Operation on* tv:menu-highlighting-mixin
:add-highlighted-value *value* *Operation on* tv:menu-highlighting-mixin
:remove-highlighted-value *value* *Operation on* tv:menu-highlighting-mixin

These operations are similar to the preceding four, except that instead of referring to items directly you refer to their values, i.e. the result of executing them. For instance if your item list is an association list, with elements (*string* . *symbol*), these operations use *symbol*. This only works for menu items that can be executed without side-effects, not for item types :eval, :funcall, etc.

tv:menu-margin-choice-mixin (tv:margin-choice-mixin) *Flavor*

This mixin gives a menu the ability to have choice boxes in the margin. It is used in multiple menus.

Choice boxes appear in a single line in the bottom margin of the menu. Each one consists of a name followed by a little square or box. Clicking on the box activates the choice.

This flavor adapts tv:margin-choice-mixin (see page 211) for use in menus.

:menu-margin-choices *Operation on* tv:menu-margin-choice-mixin
:set-menu-margin-choices *items* *Operation on* tv:menu-margin-choice-mixin
Get or set the list of choice box items. The items look and work just like menu items, and clicking on one has the same effect. The difference is only in how and where they display.

:menu-margin-choices *items* *Init option for* tv:menu-margin-choice-mixin
Initializes the list of choice box items. The default value is

```
(("Do It"
  :eval (values (funcall-self ':highlighted-values)
                t)))
```

which provides a single choice box and implements the values returned by :tv:multiple-menu-choose.

tv:margin-choice-menu *Flavor*
An instantiable menu flavor that also allows margin choices.

tv:momentary-margin-choice-menu *Flavor*
A instantiable momentary menu flavor that also allows margin choices.

14.2 Multiple Choice Facility

The *multiple choice* facility provides a window containing a bunch of items, one per text line, and several choices about each item. To see an example of its use, invoke the editor command Meta-X Kill Or Save Buffers.

For each item, there can be several yes/no choices for the user to make. There is the same set of choices for each item (though some items may omit some choices). For example, in Kill Or Save Buffers, there is an item (a line) for each buffer, and each line offers choices "Save", "Kill" and "Unmod". The choices of the same kind for different items form a column, with a heading at the top saying what that choice is for. The leftmost column contains the text naming each item. The remaining columns contain small boxes (called *choice boxes*). A "no" box has a blank center, while a "yes" box contains an "X". Pointing the mouse at a choice box and clicking the left button turns it on or off. Each choice can be initialized by the program to "yes" or "no" as appropriate for a default.

There can be constraints among the choices for an item. For example, if you want the choices to be mutually exclusive, you can set up constraints so that clicking one choice box to "yes" will automatically set the other choice boxes on the same line to "no".

A multiple choice window may have more lines of choices to offer than the window has lines. In this case, the user can scroll, as the multiple choice window is a kind of text scroll window (see chapter 16, page 219).

There are several parameters associated with a multiple-choice window:

The *item-name* is a string, the column heading for items. In the editor example, it is "Buffers".

The *item-list* is a list of representations of items. Each element is a list, (*item name choices*). *item* is any arbitrary object, such as an editor buffer. *name* is a string which names that object; it will be displayed on the left on the line of the display devoted to this item. *choices* is a list of keywords representing the choices the user can make for this item. Each element of *choices* is either a symbol, *keyword*, or a list, (*keyword default*). If *default* is present and non-nil, the choice is initially "yes"; otherwise it is initially "no". This is how the editor initializes the "Save" choice to be "yes" for a modified buffer.

The *keyword-alist* is a list defining all the choice keywords allowed. Each element takes the form (*keyword name*). *keyword* is a symbol, the same as in the *choices* field of an *item-list* element. *name* is a string used to name that keyword. *name* is used as the column heading for the associated column of choice boxes.

An element of *keyword-alist* can have up to four additional list elements, called *implications*. These control what happens to other choices for the same item when this choice is selected by the user. Each implication can be nil, meaning no implication, a list of choice keywords, or t meaning all other choices. The first implication is *on-positive*; it specifies what other choices are also set to "yes" when the user sets this one to "yes". The second implication is *on-negative*; it specifies what other choices are set to "no" when the user sets this one to "yes". The third and fourth implications are *off-positive* and *off-negative*; they take effect when the user sets this choice to "no". The default implications are nil t nil nil, respectively. In other words the default is for the choices to be mutually exclusive.

If a *keyword-alist* element does not contain implications, the default implications are rplacd'ed into it.

Kill Or Save Buffers specifies the implications as

```
((:save "Save" nil (:not-modified) nil nil)
 (:kill "Kill" nil (:not-modified) nil nil)
 (:not-modified "UnMod" nil (:save :kill) nil nil)
 (:compile "QC-FILE" nil nil nil nil))
```

so that "Unmod" cannot be chosen together with either "Save" or "Kill".

The *finishing-choices* are the choices to go in the bottom margin. When the user clicks on one of these he is done. The variable tv:default-finishing-choices contains a reasonable default for this, providing Do It and Abort choices.

14.2.1 Functional Interface

This is the easy interface to the multiple choice facility:

tv:multiple-choose *item-name item-list keyword-alist* &optional *near-mode maxlines*

Pops up a multiple-choice window and allows the user to make choices with the mouse. The dimensions of the window are automatically chosen for the best presentation of the specified choices. If there are too many choices, more than *maxlines*, scrolling of the window is enabled.

item-name, *item-list*, and *keyword-alist* are as described above. *finishing-choices* cannot be specified and is always the default.

When the user clicks on one of the two finishing choices in the bottom margin (Do It and Abort) the window disappears and tv:multiple-choose returns. If the user finishes by choosing Abort the returned value is nil, and the second returned value is :abort. If the user chooses Do It, the returned value is a list with one element for each item. Each element is a list whose car is the *item* (that arbitrary object which the user passed in in the *item-list* argument) and whose cdr is a list of the keywords for the "yes" choices selected for that item.

near-mode tells the window where to pop up. It is a suitable argument for tv:expose-window-near. The default is the list (:mouse). *maxlines*, which defaults to twenty, is the maximum number of choices allowed before scrolling is used.

Here is an example:

```
(tv:multiple-choose "Word"
   '((:eat "Eat" (:add :make-permanent))
     (:drink "Drink" (:forget :make-permanent)))
   '((:add "Add" nil nil nil (:make-permanent))
     (:forget "Forget" nil (:make-permanent) nil nil)
     (:make-permanent "Make Permanent" (:add) (:forget) nil nil)))
```

offers the possibilities of :add or :make-permanent for :eat and the possibilities of :forget or :make-permanent for :drink. Presumably this would be done because :drink has already been "added" and :eat has not been.

The implications say that making permanent is incompatible with forgetting when forgetting is possible, and requires adding when adding is possible.

The value returned might be

```
((:eat :add :make-permanent)
 (:drink))
```

In this example, the items are keywords (symbols), but that is not significant. The system never looks inside them; it just compares them with eq and puts them in the returned value.

14.2.2 Flavors and Operations

These are the grubby details:

tv:basic-multiple-choice *Flavor*

(tv:displayed-items-text-scroll-window tv:margin-choice-mixin ...)

This is the *basic* flavor that makes a window implement the multiple-choice facility. Like most basic mixins, it is not itself instantiable but it does commit any window that incorporates it to being a multiple-choice rather than any different sort of window.

tv:item-name *Instance variable of* tv:basic-multiple-choice

The window's item name.

tv:choice-types *Instance variable of* tv:basic-multiple-choice

The window's keyword alist.

tv:multiple-choice *Flavor*

(tv:basic-multiple-choice tv:top-box-label-mixin tv:window)

This is a reasonable window with the multiple-choice facility in it. It has borders and a label area on top which is used for the column headings.

tv:temporary-multiple-choice-window *Flavor*

(tv:temporary-window-mixin tv:multiple-choice)

This is a multiple-choice window which is equipped to pop up temporarily.

tv:temporary-multiple-choice-window *Resource*

&optional (*superior tv:mouse-sheet*)

This is a resource of temporary multiple-choice windows, used by the tv:multiple-choose function.

The following operations are provided by multiple choice windows.

:item-list *item-list* *Init option for* tv:basic-multiple-choice

Initializes the window's item list to *item-list*.

:setup *Operation on* tv:multiple-choice

item-name keyword-alist finishing-choices item-list &optional *maxlines*

This operation sets up all the various parameters of the window. Usually it is used while the window is deexposed. The window decides what size it should be and whether all the items will fit or scrolling is required, then draws the display into its bit-array. Thus when the window is exposed the display will appear instantaneously.

maxlines is the maximum number of lines the window may have; if there are more items than this only some of them will be displayed and scrolling will be enabled. *maxlines* defaults to 20.

The *finishing-choices* are a list of choices for tv:margin-choice-mixin (see page 211). When one of these finishing choices is clicked on, it should set the instance variable tv:choice-value of self to either a symbol (for an abnormal exit) or a list for the :choose operation to return.

tv:choice-value *Instance variable of* tv:basic-multiple-choice

When the mouse process sets this non-nil, the :choose operation returns.

:choose &optional *near-mode* *Operation on* tv:multiple-choice

Moves the window to the place specified by *near-mode*, which defaults to the list (:mouse), and exposes it. Then waits for the user to make a finishing choice and returns the window to its original activate/expose status before the :choose. This operation returns the same value as the function tv:multiple-choose.

14.3 Choose-Variable-Values Facility

This facility presents the user with a display of a bunch of Lisp variables and their values. The user may change the value of some of the variables. When the values are to his liking he may indicate that he is done.

The choose-variable-values window is a kind of text scroll window, so each line of the display corresponds to one variable. The name of the variable, a colon, and the value of the variable are displayed. Pointing the mouse at the value causes a box to appear around it. Clicking the left mouse button at that point allows the value to be changed.

For an example of a choose-variable-values window, try the Frame option of the Split Screen item in the system menu. ZMail profile mode is also a good example.

14.3.1 Specifying the Variables

When you use a choose-variable-values window, you must specify one or more variables with a list of specifiers. You pass the list as an argument to tv:choose-variable-values.

Each variable has a *type* which controls what values it may take on, the way the value is displayed and the way the user enters a new value. The type mechanism is extensible and is described in detail later. The types fall into two categories, those with a small number of legal values and those with a large or infinite number of legal values. The first kind of type displays all the choices, with the one which is the current value of the variable in bold-face. Pointing at a choice and clicking the mouse sets the variable to that value. Those types with a large number of legal values display the current value. Pointing at the value and clicking the mouse allows a new value to be entered from the keyboard. Rubbing out more characters than typed in restores the original value instead of changing it.

The variables themselves can be either symbols, which are effectively examined and set as special variables in the calling program's process, or locatives, whose contents are examined and set. The syntax for input and output is controlled by the binding of base, ibase, *nopoint, prinlevel, prinlength, package, and readtable as usual.

Each line of the display is specified by an *item*, which can be one of the following:

a string — The string is simply displayed. This is useful for putting headings and blank separating lines into the display.

a symbol — The symbol is a variable whose type is :sexp; that is, its value may be any Lisp object. The name of the variable on the display is simply its print-name, and the value is stored as the value of the symbol.

a list (*variable name type args...*)

This is the general form. *variable* is the variable whose value is being chosen. It is either a symbol or a locative. If *name* is supplied it can be a string, which is displayed as the name of the variable, or it can be nil, meaning that this line should have no variable name, but only a value. *name* is optional; if it is omitted it defaults to the print-name of *variable*, or to nil if *variable* is a locative.

type is an optional keyword giving the type of variable; if omitted it defaults to :sexp. *args* are possible additional specifications dependent on *type*.

It is possible to omit *name* and supply *type* since one is always a string or nil and the other is always a non-nil symbol.

For clarification of this, refer to the examples on page 198.

14.3.2 Predefined Variable Types

The following are the types of variables supported by default, along with any *args* that may be put in the item after the *type* keyword:

:sexp

:any — The value is any Lisp expression (sometimes called an S-expression), printed with prin1, read with read.

:princ — Same as :sexp except that the value is printed with princ rather than prin1.

:string — The value is a string, printed with princ, read with readline.

:number — The value is any type of number. It is printed with prin1 and read with read, but only a number is accepted as input.

:number-or-nil
The value may be either a number or nil.

:date — The value is a universal date-time. It is printed with time:print-universal-time and read with readline-trim and time:parse-universal-time.

:date-or-never
The value is either a universal date-time or nil. nil is printed as "never", and a number is printed using time:print-universal-time. Input is read with readline-trim; if the string is not "never" it is passed to time:parse-universal-time.

:interval-or-never
The value is either nil or a number of seconds. It is printed with time:print-interval-or-never and new values are read using time:read-interval-or-never.

:character — The value is a character code. It is printed as the character name (using the ~:@C format operator), and is read as a single keystroke.

:character-or-nil
Like :character but nil is also allowed as the value. nil displays as "none" and can be input via the Clear Input key.

:string-list — The value is a list of strings, whose printed representation for input and output consists of the strings separated by commas and spaces.

:pathname
a list (:pathname *defaults*)
The value is a pathname (see chapter 22 of the Lisp Machine manual). It is printed with princ and read with readline, fs:parse-pathname, and fs:merge-pathname-defaults. If *defaults* is provided, it is a pathname or a defaults-alist to pass to fs:merge-pathname-defaults. It can also be a symbol whose value

should be used. If it is the same variable this item is setting, then each typed-in value is merged with the previous setting.

:pathname-or-nil
Like :pathname but nil is also allowed as a value. It is read and printed as a blank line.

:pathname-list
The value is a list of pathnames. In the printed representation they are separated by commas.

:choose *values-list print-function*
The value of the variable must be one of the elements of the list *values-list*. Comparison is by equal rather than eq. All the choices are displayed, with the current value in boldface. A new value is input by pointing to it with the mouse and clicking. *print-function* is the function to print a value; it is optional and defaults to princ.

:assoc *values-list print-function*
Like :choose but car of each element of *values-list* is what to display, while cdr is the value that goes in the variable.

:menu-alist *item-list*
Like :choose, but instead of a list of values there is *item-list*, which is a list of menu items (see section 14.1, page 173). The usual menu mechanisms for specifying the string to display, the value to return, and the mouse documentation work with this.

:boolean The value of the variable is either t or nil. The choices are displayed as yes and no.

:documentation *doc type args...*
This is not really a variable type, but goes in the place where a type would normally be expected. The real type is *type*; it and its *args* are optional as usual. *doc* is a string which is displayed in the mouse documentation line when the mouse is pointing at this item. The default if no documentation is supplied in this way depends on the type, and generally is something like "Click left to input a new value from the keyboard."

14.3.3 Functional Interface

tv:choose-variable-values *variables* &rest *options*
This is the easy-to-use function interface to the choose-variable-values facility. It pops up a window displaying the values of the specified variables and permits the user to alter them. One or more choice boxes (as in the multiple-choice facility) appear in the bottom margin of the window. When the user clicks on the *Exit* choice box the window disappears and this function returns. The value returned is not meaningful; the result is expressed in the values of the variables.

The system chooses the dimensions of the window, and enables scrolling if there are too many variables to fit in the chosen height.

variables is a list whose elements can be special variables or the more general items described above. See the examples below.

options is the usual list of alternating option keywords and argument values. The following option keywords are allowed:

:label — The argument is a string that is the label to be displayed at the top of the window. The default is "Choose Variable Values".

:function — The function to be called if the user changes the value of a variable. The default is nil (no function). The use of this function is described below (page 197).

:near-mode — Where to position the window. This is a suitable argument for tv:expose-window-near. The default is the list (:mouse).

:width — Specifies how wide to make the window. This can be a number of characters, or a string (it is made just wide enough to display that string). The default is to make it wide enough to display the current values of all the variables, provided that isn't too wide to fit in the superior.

:extra-width — When :width is not specified, this specifies the amount of extra space to leave after the current value of each variable of the kind that displays its current value (rather than a menu of all possible values). This extra space allows for changing the value to something bigger. The extra space is specified as either a number of characters or a character string. The default is ten characters. If :width is specified, then :extra-width is ignored.

:margin-choices — The argument is a list of specifications for choice boxes to appear in the bottom margin. Each element can be a string, which is the label for the box which means "done", or a cons of a label string and a form to be evaluated if that choice box is clicked upon. Since this form is evaluated in the user process it can do such things as alter the values of variables or *throw out. The default for :margin-choices is ("Exit").

:superior — The argument is the window to which the pop-up choose-variable-values window should be inferior. The default is the value of tv:mouse-sheet, or the superior of *w* if *near-mode* is (:window *w*).

:reverse-video-p — The argument is used to control whether the window displays white-on-black or black-on-white. It is used as the argument of the :set-reverse-video-p operation.

A choose-variable-values window optionally may have an *associated function*, which is called whenever the user commands the window to change the value of one of the variables.

This function can implement constraints among the variables. It is called with arguments *window*, *variable*, *old-value*, and *new-value*. The function should return nil if just the original variable needs to be redisplayed, or t if no redisplay is required; in this case it would usually setq several of the variables, then perform a :refresh operation on the window.

Here are some examples of how to call tv:choose-variable-values. The simplest sort of thing you can do is:

```
(tv:choose-variable-values '(base ibase *nopoint)
                           ':label "Number format parameters")
```

which displays the three variables' names and values and lets the user change them. The same example can be done with nicer formatting with:

```
(tv:choose-variable-values
        '((base "Output Base" :number)
          (ibase "Input Base" :number)
          (*nopoint "Decimal Point"
                    :assoc (("Yes" . nil)
                            ("No" . t))))
        ':label "Number format parameters")
```

The entry for *nopoint would have been simply

```
(*nopoint "No Decimal Point" :boolean)
```

except that we wanted to reverse the sense of t and nil. We might even have used

```
(*nopoint :boolean)
```

if we wanted to use the name of the variable as the label rather than spelling it out.

For a hokier example, consider a grocery store. Suppose we have variables *cuts-of-beef*, *cuts-of-pork*, *cuts-of-lamb*, and *lettuce-types*, which contain lists of strings indicating what is available, *squash-type*, which indicates whether we stock summer squash or winter squash, and *milk-price*, which contains a floating-point number that is the current price of a gallon of milk. Then the following expression would display the inventory and allow it to be modified, using several different kinds of items:

```
(tv:choose-variable-values
        '("Meat Department"
          (*cuts-of-beef* "Beef" :string-list)
          (*cuts-of-pork* "Pork" :string-list)
          (*cuts-of-lamb* "Lamb" :string-list)
          ""
          "Produce"
          (*lettuce-types* "Lettuce" :string-list)
          (*squash-type* "Squash" :choose ("Summer" "Winter"))
          ""
          "Dairy"
          (*milk-price* "Milk"
                :documentation
                    "Click left to raise the price of milk"
                :number)))
```

Note the use of strings to provide labels for the sections, and null strings to separate the sections with blank lines.

14.3.4 Defining Your Own Variable Type

:decode-variable-type *kwd-and-args* *Operation on* tv:basic-choose-variable-values

The system uses this operation on a choose-variable-values window when it needs to understand an item. *kwd-and-args* is a list whose car is the item's type keyword and whose remaining elements, if any, are the arguments to that keyword. Six values are returned; these values are described below. The default method for :decode-variable-type looks for two properties on the keyword's property list:

tv:choose-variable-values-keyword
: The value of this property is a list of the six values described below. Unnecessary values of nil may be omitted at the end.

tv:choose-variable-values-keyword-function
: The value of this property is a function that is called with one argument, *kwd-and-args*. The function must return the six values.

You may add a new variable type to the standard set by putting one of the above properties on the keyword. You may define your own flavor of choose-variable-values window and give it a :decode-variable-type method to make it not use the standard variable types. This method must take care of implementing the :documentation keyword, which can appear in an item where a variable type would normally appear.

The six magic values are:

print-function A function of two arguments, object and stream, to be used to print the value. prin1 is acceptable.

read-function A function of one argument, the stream, to be used to read a new value. read is acceptable. If nil is specified, there is no read-function and instead new values are specified by pointing at one choice from a list. If the *read-function* is a symbol, it is called inside a rubout-handler, and over-rubout will automatically leave the variable with its original value. If *read-function* is a list, its car is the function, and it will be called directly rather than inside a rubout-handler.

choices A list of the choices to be printed, or nil if just the current value is to be printed. The choices are printed using the *print-function*, just as the current value is.

print-translate If there are choices, and this function is supplied non-nil, it is given an element of the choice list and must return the value to be printed using the *print-function*.

value-translate If there are choices, and this function is supplied non-nil, it is given an element of the choice list and must return the value to be stored in the variable.

documentation A string to display in the mouse documentation line when the mouse is pointing at this item. This string should tell the user that clicking the mouse will change the value of this variable and give any special information (e.g. that the value must be a number).

Alternatively, this can be a symbol that is the name of a function. It will be called with one argument, which is the current element of *choices* or the current value of the variable if *choices* is nil. It should return a documentation string or nil if the default documentation is desired. This can be useful when you want to document the meaning of a particular choice, rather than simply saying that clicking the mouse on this choice will select it. Note that the function should return a constant string, rather than building one with format or other string operations, because it will be called over and over as long as the mouse is pointing at an item of this type. The function is called by the who-line updating in the scheduler, not in the user process.

For example, :boolean is defined thus:

```
(defprop :boolean
         (choose-variable-values-boolean-print nil (t nil))
         choose-variable-values-keyword)
(defun choose-variable-values-boolean-print (value stream)
  (funcall stream ':string-out (if value "Yes" "No")))
```

The type :any is defined with

```
(defprop :any (prin1 read) tv:choose-variable-values-keyword)
```

14.3.5 Making Your Own Window

The function tv:choose-variable-values may not be adequate if you wish to keep the window permanently exposed or if you wish to alter its behavior. Then you must create a window yourself. Here are the pertinent flavors.

tv:basic-choose-variable-values *Flavor*

(tv:mouse-sensitive-text-scroll-window-without-click)

This is the *basic* flavor which makes a window implement the choose-variable-values facility. It is not instantiable.

tv:choose-variable-values-window *Flavor*

(tv:basic-choose-variable-values tv:window ...)

This is a choose-variable-values window with a reasonable set of features, including borders, a label at the top, stream I/O, the ability to be scrolled if there are too many variables to fit in the window, and the ability to have choice boxes in the bottom margin.

tv:choose-variable-values-pane (tv:choose-variable-values-window) *Flavor*

A tv:choose-variable-values-window designed to be a pane of a constraint frame. It redefines the :adjustable-size-p operation to return nil always, on the assumption that the window's size has been specified by the frame and cannot be changed except by the frame.

tv:temporary-choose-variable-values-window *Flavor*
(tv:choose-variable-values-window tv:temporary-window-mixin)
A tv:choose-variable-values-window that is equipped to pop up temporarily.

tv:temporary-choose-variable-values-window *Resource*
&optional (*superior tv:mouse-sheet*)
This is a resource of such windows, from which tv:choose-variable-values gets a window to use.

There are two main styles of use: to create a window giving all of the parameters in the init-plist, or to create a window without specifying the parameters, and then use the :setup operation (see below) to set the parameters before using the window. But in any case, you must specify the list of variable-specifiers (see section 14.3.1, page 194) and the stack group to evaluate variables in before you can use the window.

The following init options are available:

:variables *specifier-list* *Init option for* tv:basic-choose-variable-values
Initializes the list of variable-specifiers, telling the window which variables to display and how to read and print the values.

:function *fcn* *Init option for* tv:basic-choose-variable-values
Initializes the *associated function* (see page 197), the function called when the window changes the value of one of the variables it displays. The default is nil (no function).

tv:function *Instance variable of* tv:basic-choose-variable-values
The window's associated function.

:stack-group *sg* *Init option for* tv:basic-choose-variable-values
The stack group in which the variables whose values are to be chosen are bound. The window needs to know this so that it can get the values while running in another process, for instance the mouse process, in order to update the window display when it is refreshed or scrolled. If you do not specify the stack group at this time, you must specify it with the :setup operation, before you can use the window.

tv:stack-group *Instance variable of* tv:basic-choose-variable-values
The stack group in which variables' values should be evaluated.

:name-font *font* *Init option for* tv:basic-choose-variable-values
The font in which names of variables are displayed. The default is the system default font.

:value-font *font* *Init option for* tv:basic-choose-variable-values
The font in which values of variables are displayed. The default is the system default font.

:string-font *font* *Init option for* tv:basic-choose-variable-values

The font in which items that are just strings (typically heading lines) are displayed. The default is the system default font.

:unselected-choice-font *font* *Init option for* tv:basic-choose-variable-values

The font in which choices for a value, other than the current value, are displayed. The default is a small distinctive font.

:selected-choice-font *font* *Init option for* tv:basic-choose-variable-values

The font in which the current value of a variable is displayed, when there is a finite set of choices. This should be a bold-face version of the preceding font. The default is the bold-face version of the default unselected-choice font.

:margin-choices *choice-list* *Init option for* tv:choose-variable-values-window

The default is a single choice box, labeled "Done". See page 210 for the details of what you can put here. Note that specifying nil for this option will suppress the margin-choices entirely.

If no dimensions are specified in the init-plist, the width and height will be automatically chosen according to the other init-plist parameters. The height is dictated by the number of variables to be displayed. Specifying a height in the init-plist, using any of the standard dimension-specifying init-plist options, overrides the automatic choice of height.

Choose-variable-values windows provide these operations:

:setup *Operation on* tv:choose-variable-values-window

items label function margin-choices

Changes the list of items (variables), the window label, the constraint function, and the choices in the bottom margin, and sets up the display. Also remembers the current stack-group as the stack-group in which the variables are bound. If the window is not exposed (more generally, if the :adjustable-size-p operation on the window returns non-nil), this reshapes the window to a good size based on the specified items.

:set-variables *Operation on* tv:choose-variable-values-window

item-list &optional *dont-set-height*

Sets the list of variable-specifiers which controls the variables displayed in the window, then redisplays the window.

Unless *dont-set-height* is supplied non-nil, the height of the window will be adjusted according to the number of lines required. If more than 25. lines would be required, 25. lines will be used and scrolling will be enabled. The :setup operation uses :set-variables to do part of its work.

:adjustable-size-p *Operation on* tv:choose-variable-values-window

If this returns non-nil, :setup will reshape the window. By default, this operation returns non-nil when the window is deexposed.

:appropriate-width *Operation on* tv:choose-variable-values-window &optional *extra-space*

Returns the inside-width appropriate for this window to accommodate the current set of variables and their current values. Use this operation after a :setup and before a :expose, and use the result to do a :set-inside-size. The returned width will not be larger than the maximum that will fit inside the superior.

If *extra-space* is supplied, it specifies the amount of extra space to leave after the current value of each variable that displays its current value (rather than a menu of all possible values). This extra space allows for changing the value to something bigger. The extra space is specified as either a number of characters or a character string. The default is to leave no extra space.

:redisplay-variable *variable* *Operation on* tv:choose-variable-values-window

Redisplays just the value of that variable.

In the simplest mode of operation, you call the tv:choose-variable-values function, which takes care of creating the window and all necessary communication with it. When you make your own choose-variable-values window, you need to handle the communication yourself, using the information given below. An example of a situation in which this is necessary is when you have a frame, some panes of which are choose-variable-values windows.

A choose-variable-values window handles mouse clicks by putting blips (lists) in its input buffer. These blips are generated by the mouse process and are supposed to be read in the controlling process. There are two types of blip, both used for specific purposes, and your program must be able to take the appropriate actions when it reads them. The easy way for you to do this is to call the function tv:choose-variable-values-process-message, which is provided just for this purpose.

:io-buffer *io-buffer* *Init option for* tv:choose-variable-values-window

The I/O buffer to be used for blips and for ordinary input from the window.

The following forms of list are inserted as blips into the input buffer:

(:variable-choice *window item value line-no*)

Indicates that the user clicked on the value of a variable, expressing the desire to change it. The controlling process should read keyboard input as necessary and set the variable.

(:choice-box *window box*)

Indicates that the user clicked on one of the choice boxes in the bottom margin. The controlling process may wish to deexpose the window if the box was the "Done" box.

tv:choose-variable-values-process-message *window blip*

This function implements the proper response to the above blips. It should be called in the process and stack-group in which the variables being chosen are bound. *window* should be the choose-variable-values window and *blip* should be the object read as input.

This function returns nil except in the case where *blip* indicates a click on a "Done" choice box.

If *blip* says that the user clicked on a variable, this function reads user input from the window as necessary and sets the variable.

If *blip* is a :choice-box blip, the action depends on the *box* in it. If the sixth element of *box* is nil, which is normally the case for the "Done" box, this function returns t. Otherwise, the sixth element of *box* is evaluated, but this function returns nil.

If *blip* is actually a character rather than a blip, it is ignored unless it is a Clear-screen, in which case the choose-variable-values window is refreshed. Therefore, it is reasonable to use this function with a loop like this:

```
(do ()
    ((tv:choose-variable-values-process-message
       c-v-v-window
       (progn
         (process-wait "Choose" c-v-v-window ':listen)
         (send c-v-v-window ':any-tyi)))))
```

14.3.6 User Option Facility

There is a facility, based on the choose-variable-values facility, for keeping track of options to a program of the sort that a user would specify once and keep in his init file. Special forms are provided for defining options, and there are functions for putting all the options into a choose-values window so that the user can alter them, for writing the current state of the options into an init file, and for resetting all the options to their default initial values.

define-user-option-alist *name constructor documentation* *Special form*

Defines *name* a special variable whose value is a "user option alist", something which may be used by the other functions below. This alist will keep track of all of the option variables for a particular program.

The simplest usage is (define-user-option-alist *name*), which just defines *name*.

(define-user-option-alist *name constructor*) specifies in addition the name of a constructor macro to be defined, which provides a slightly different way of defining an option variable from defvar-user-option. The form (*constructor option default name type args...*) will define an option in this user-option-alist. The arguments are the same as the similarly-named arguments to defvar-user-option.

A third argument may be used to specify a documentation string for the variable *name*. To specify a documentation string and no constructor, give nil for the constructor.

defvar-user-option *Special form*

Defines an option and adds it to a user option list.

```
(defvar-user-option option default documentation
  alist name type args...)
```

defines the special variable *option* to be an option in the *alist*, which must have been previously defined with define-user-option-alist. The variable is declared and initialized via (defvar *option default documentation*). The value of the form *default* is remembered so that the variable can be reset back to it later.

type is the type of the variable for purposes of the choose-variable-values facility. It is optional and defaults to :sexp. *args*, which are evaluated (at the time the definition is done), are the arguments for the type keyword used.

name is the name of the variable to be displayed in the choose-variable-values window. If it is omitted or nil, the default is (string-capitalize-words (get-pname *option*)); except that when the first and last characters of the print-name are asterisks, they are removed. E.g. the default name for sowq:*sunny-side-up* would be "Sunny Side Up".

Example:

```
(defvar-user-option preferred-radix 8
  "Radix to use for files that don't specify one."
  my-program-option-alist "Preferred radix"
  :assoc '(("8" 8) ("10" 10.)))
```

defvar-site-user-option *Special form*

This is like defvar-user-option, except that instead of an initial value a site option keyword is specified. Instead of a default value, you specify the name of a site option (a keyword). The actual default value is the value of that site option in the current site table. Loading a new site table resets the option.

defvar-site-alist-user-option *Special form*

Defines a user option whose possible values are controlled by site options.

```
(defvar-site-alist-user-option option default documentation
  alist name menu-alist)
```

defines *option* as a user option on *alist*, like *defvar-site-user-option*. The *type* for tv:choose-variable-values is always :menu-alist, and the list of menu items to be used is determined from the site table according to *menu-alist*.

menu-alist is a symbol whose value is a menu alist, a list of menu items. These items are the alternatives offered to the user, as in the :menu-alist type of variable. However, each menu item specifies a site option keyword, and that alternative is available to the user only if that site option currently has a non-nil value.

The menu item can specify the controlling site keyword using the modifier keyword :site-keyword, as in

```
("Foo" :value :foo :site-keyword :foo-present)
```

If this is not done, the menu item's value-to-return is also the site keyword.

default is the name of a site keyword whose value specifies the default. This site option's value is matched against each menu item, comparing it against the value of the modifier keyword :default-site-keyword, or, if that is not present, against the menu item's site keyword name. The first match is the default alternative. Thus "Foo" will be the default alternative if the *default* site option's value is :foo-present.

If *default* is nil, then the first available menu alist item is also the default.

choose-user-options *alist* &rest *options*
Displays the values of the option variables in *alist* to the user and allows them to be altered. The *options* are passed along to tv:choose-variable-values. Note that *alist* is an actual alist, not a symbol whose value is an alist.

reset-user-options *alist*
Each of the option variables in *alist* is reset to its default initial value.

tv:restrict-user-option *option restriction-type site-options...* *Macro*
Specifies that the user option variable *option* is significant only if the site tables for your site do (or, if they do not) contain one of the specified *site-options*.

restriction-type is either :if or :unless. If it is :if, the option should be mentioned in the choose-variable-values window only if one of the specified site options is present in the currently loaded site table. :unless means that the option should be offered only if none of the specified site options is loaded.

Each option may have an :if restriction and an :unless restriction.

Elimination of options from an alist according to their restrictions is done by tv:prune-user-option-alist, calling which is up to you.

restriction-type may also be :never. Then the option is never offered to the user to change, but it will still be reset and written with the other options.

tv:prune-user-option-alist *alist*
Returns an alist containing only some of the elements of *alist*, lacking those that are suppressed by restrictions, or that offer only a single alternative. (The latter is likely to happen with a site-menu-alist user option if a given site allows only one of the possible alternatives.)

write-user-options *alist stream*
For each option variable in *alist* whose current value is not equal to its default initial value, a form is printed to *stream* that will set the variable to its current value. The form uses login-setq so it is appropriate for putting into an init file.

14.4 Mouse-Sensitive Type Out

The mouse-sensitive items facility is a feature somewhat related to the choice facilities described above. It is similar in its appearance to the user, but quite different in the way it is interfaced to by a program. Mixing tv:basic-mouse-sensitive-items into a window flavor equips the window with mouse-handling according to the paradigm described in this section. Mouse-sensitive items are something you use when defining your own window, rather than a complete, stand-alone facility, and consequently do not have an "easy to use" functional interface.

For an example of mouse-sensitive items, try the C-X C-B (List Buffers) command in the editor. Try moving the mouse over the list of buffers and clicking the right-hand button.

The word "typeout" appears here and there in the mouse-sensitive items facility for historical reasons. Often mouse-sensitive items are typed out on top of some other display, such as an editor buffer. However, the mouse-sensitive-item facility has nothing to do with the typeout-window facility. At this point it would be a fairly big incompatible change to fix this.

tv:basic-mouse-sensitive-items *Flavor*

Mixing this flavor into a window provides for areas of the screen which are sensitive to the mouse. Moving the mouse into such an area highlights the area by drawing a box around it. At this point clicking the mouse performs a user-defined operation. This flavor is called *basic* because it fixes the handling of the mouse by the window; it will not work to mix it with another flavor that expects to define some other kind of mouse handling. However it is less basic than many basic flavors in that it does not do anything special with the displayed image of the window.

A mouse-sensitive item has a *type*, which is a keyword which controls what you can do to it, an *item*, which is an arbitrary Lisp object associated with it, and a rectangular area of the window. Typically something is displayed in that area at the same time as a mouse-sensitive item is created, using normal stream output to the window. Unlike things such as menu items, these mouse-sensitive items are not a permanent property of the window; they are just as ephemeral as the displayed text and go away if you clear the window or if typeout wraps around and types over them. Of course, if you don't type out more items and text than fit in the window, and never clear the window, then they will be permanent.

Associated with each type is a set of operations that are legal to perform on items of that type. One of these operations is selected as the default. The tv:item-type-alist instance variable is an alist that defines these. This alist is composed of elements of the following form:

```
(type left-button-alternative
 documentation
 (string . alternative)          ;A menu item
 (string :value alternative)     ;Another menu item
 menu-item...)                        ;More of them
```

documentation is the string to be displayed in the who line while the mouse is pointing at an item of this type. The menu items may also have documentation strings in them. *documentation* may also be a list of the form

(*doc-function label-function*)

where *doc-function* is a function that, when applied to a mouse-sensitive item, returns a documentation string, and *label-function* is a similar function that returns a string to use as the

menu label, to identify the item that the menu is going to apply to.

Here is part of the item type alist used in typeout windows of editor windows:

```
((zwei:directory zwei:directory-edit-1
  "Left: Run DIRED on this directory.  Right: menu of View, Edit."
  ("View" :value zwei:view-directory
   :documentation "View this directory")
  ("Edit" :value zwei:directory-edit-1
   :documentation
   "Run DIRED on this directory."))
 (zwei:file zwei:find-defaulted-file
  "Left: Find file this file.  Right: menu of Load, Find, Compare."
  ("Load" :value zwei:load-defaulted-file
   :documentation "LOAD this file.")
  ("Find" :value zwei:find-defaulted-file
   :documentation "Find file this file.")
  ("Compare" :value zwei:srccom-file
   :documentation
   "Compare this file with the newest version."))
 (zwei:flavor-name zwei:edit-definition-for-mouse
  "Left: Edit definition.  Right: menu of Describe, Edit."
  ("Describe" :value zwei:describe-flavor-internal
   :documentation "Describe this flavor.")
  ("Edit" :value zwei:edit-definition-for-mouse
   :documentation "Edit definition.")))
```

When an item is clicked on with the mouse, a blip which is a list of the form

```
(:typeout-execute alternative item)
```

is placed in the window's input buffer. *item* is the datum supplied when the item was constructed, whose purpose is to identify which item was clicked on, and *alternative* is obtained by looking up the *type* of the item in the window's item-type-alist.

If the item is clicked on with the left mouse button, the *left-button-alternative* is used in the :typeout-execute blip. If the item is clicked on with the right button, the menu items are put into a menu, and the user chooses one. The value returned by the :choose operation is used as the *alternative* in the :typeout-execute blip. Clicking on an item of a type that is not one of the alternatives in the item-type-alist just beeps.

For the Load alternative on a file item in the editor, the blip might be

```
(:typeout-execute zwei:load-defaulted-file
 #⊂fs:logical-pathname "SYS: SYS; QFCTNS LISP"⊃)
```

:item-type-alist *Operation on* tv:basic-mouse-sensitive-items

:set-item-type-alist *Operation on* tv:basic-mouse-sensitive-items
new-item-type-alist

Return or set the item type alist of the window.

:item-type-alist *alist* *Init option for* tv:basic-mouse-sensitive-items
Initializes the item type alist of the window.

tv:add-typeout-item-type *Special form*
The special form

(tv:add-typeout-item-type *alist type name function default-p documentation*)

is used to declare information about a mouse-sensitive item type by adding an entry to an alist kept in a special variable. This alist can then be put into the item-type alist of a mouse-sensitive window, for instance using the :item-type-alist init-plist option. Note that each possible alternative for a particular mouse-sensitive item type is defined with a separate tv:add-typeout-item-type form; this allows each alternative to be defined at the place in the program where it is implemented, rather than collecting all the alternatives into a separate table. It also allows new alternatives to be added in a modular fashion.

alist is the special variable containing the alist. You should defvar it to nil before defining the first item type. Each program that uses mouse-sensitive items has its own alist of item types, so that there is no conflict in the names of the types. *type* is the keyword symbol for the type being defined. *name* is the string that names the operation and *alternative* is the representation of the alternative (the object to be put in the second element of the :typeout-execute blip). *default-p* is optional; if it is supplied and non-nil, it means that this operation is the default performed when you click the left button on an item of this type. *documentation* is optional but highly recommended; it is a string that documents what *function* does. When the user points the mouse at an item of this type, the documentation line at the bottom of the screen will give the documentation for the default function (reachable by the left button) and a list of the functions in the menu (reachable by the right button). If the user clicks right, calling for a menu, then the documentation for whichever function in the menu he points the mouse at will be displayed.

alist, *type*, and *function* are not evaluated. *name*, *default-p*, and *documentation* are evaluated.

In the editor, *alternative* is interpreted (when a :typeout-execute blip is read) as a function to be called, and the tv:add-typeout-item-type form is typically placed right before the function definition of *alternative*.

These are the operations used to print items on a window.

:item *type item* &rest *format-args* *Operation on* tv:basic-mouse-sensitive-items
A new item *item* of type *type* is printed, either by calling format with *format-args*, or by princ'ing *item* if *format-args* is nil.

The mouse-sensitive area of the item is whatever space is used up by printing it, as judged by the motion of the cursor.

The arguments *item* and *type* is not necessarily used in printing the item, but they are used in handling a click on the item. *type* is used to look up a function in the item type alist, and *item* is placed directly into the :typeout-execute blip.

Example:

```
(send standard-output ':item 'zwei:file pathname)
```

in the editor, where standard-output is a window that supports mouse-sensitive items, will princ the value of pathname and make an item of type zwei:file whose datum is that pathname.

:primitive-item *Operation on* tv:basic-mouse-sensitive-items
type item left top right bottom

:primitive-item-outside *Operation on* tv:basic-mouse-sensitive-items
type item left top right bottom

This operation is used to define a mouse-sensitive item without printing it. (Presumably you print it yourself, either before or after.) The *type* and *item* are used as in the :item operation. The remaining arguments are coordinates that describe the four edges of the mouse-sensitive rectangle.

In :primitive-item, the four coordinates are relative to the inside top left corner of the window (that is, they are cursor positions such as :read-cursorpos would return). In :primitive-item-outside, they are relative to the outside corner of the window (like values of the instance variables tv:cursor-x and tv:cursor-y).

:item-list *type list* *Operation on* tv:basic-mouse-sensitive-items

Several items are printed, arranged neatly in columns, one for each element of *list*. An element of *list* can be either a string or a list (*name* . *item*). In the latter case, *name* (typically a string) is printed with *princ*, and *item* is used as the datum for the item. If the element is an atom, that atom serves both to be princ'd and used as the datum. All the items are of type *type*.

:mouse-sensitive-item *x y* *Operation on* tv:basic-mouse-sensitive-items

Returns a list describing the mouse-sensitive item found at cursor position *x*, *y* in the window, or nil if there is none there.

The list looks like this:

(*type item left top right bottom*)

The *type* and *item* are as specified in the :item operation and the coordinates are cursor positions (that is, relative to the outside top left corner of the window).

14.5 Margin Choices

A window can be augmented with choice boxes (see page 190) in its bottom margin using the flavor tv:margin-choice-mixin. These give the user a few labeled mouse-sensitive points that are independent of anything else in the window.

Margin choices are not a complete, stand-alone choice facility and consequently do not have an "easy to use" functional interface.

For an example of a window with margin choices (as well as choice boxes in its interior), try the editor command Meta-X Kill or Save Buffers.

tv:margin-choice-mixin *Flavor*

Puts choice boxes in the bottom margin, according to a list of choice-box descriptors which can be specified with the :margin-choices init-plist option or the :set-margin-choices operation. A choice-box descriptor is a list, (*name state function x1 x2*). It is legal to use a longer list as a choice-box descriptor and store your own data in the additional elements.

name is a string that labels the box. *state* is t if the box has an "X" in it, nil if it is empty. *x1* and *x2* are used internally to remember where the choices boxes are; they are always spread out evenly in the available width.

function is a function that is called in a separate process if the user clicks on the choice box. It receives three arguments: the choice-box descriptor for the choice box, the "margin region" that contains the choice boxes, and the *y*-position of the mouse relative to this window. You probably want to ignore the last two arguments. When *function* is called, self is bound to the window, so *function* may use (declare (:self-flavor *flavor*)) to access the window's instance variables. The structure access functions tv:choice-box-name and tv:choice-box-state may be of use inside *function* (they are just more specific names for car and cadr). If *function* changes the state of the choice box, it will need to refresh the choice boxes by doing

```
(funcall (tv:margin-region-function region) ':refresh region)
```

where *region* is its second argument, which is why that argument is passed.

tv:margin-choice-mixin contains tv:margin-region-mixin as an included flavor; this means approximately that tv:margin-region-mixin will appear in any combination right after tv:margin-choice-mixin if it is not explicitly specified to appear somewhere else. The position of tv:margin-region-mixin controls where the choice boxes appear in relation to the other margin items (borders, labels. etc). See chapter 11, page 129.

:margin-choices *choices* *Init option for* tv:margin-choice-mixin

choices is a list of choice-box descriptors, described above. A line of choice-boxes will appear in the bottom margin of the window. If *choices* is nil, there will be no choice boxes and no space for them in the bottom margin; however, the window will still be capable of accepting the :set-margin-choices operation to create a line of choice boxes later.

:set-margin-choices *choices* *Operation on* tv:margin-choice-mixin

Changes the set of margin choices according to *choices*, which is nil to turn them off or a list of choice-box descriptors, described above. If the choice boxes are turned on or off, the size of the window's bottom margin will change accordingly.

tv:margin-choices *Instance variable of* tv:margin-choice-mixin

A list of margin choices, or nil.

To get a menu with margin choices, it is best to use tv:menu-margin-choice-mixin (page 189), which goes to a little extra trouble to interface the margin choices to the menu.

15. Typeout Windows

Typeout windows are a facility provided to make it easier for a program that normally displays a single updating picture to print a stream of unrelated output from time to time.

For example, Zmacs windows normally present a continuously updated display of an editor buffer. But some editor commands are designed to print output, such as a directory listing from Control-X Control-D or a list of buffers from Control-X Control-B. This output cannot conveniently be printed on the editor window itself, since that window is set up to maintain its standard display of an editor buffer and is no longer suitable for displaying anything else. Instead, the output is printed on a special kind of window called a typeout window, which exists as an inferior of the editor window. Other programs that maintain updating displays, such as the inspector and Peek, also use typeout windows for this purpose.

A typeout window is an inferior of another window such as the editor or Peek display window, and "grows" over its superior as output is done on it. The output starts at the top of the typeout window, which is also the top of its superior, and proceeds downward. The typeout window always keeps track of how far down output has proceeded, so that the superior window can eventually find out how much of its permanent display has been clobbered by the typeout window and therefore needs to be redisplayed. A horizontal line or "window shade" appears just below the point of lowest output, to enable the user to separate the typeout from the remains of the permanent display. If output to the typeout window proceeds far enough, it wraps around to the top of the screen. Then the typeout window records that the entire superior has been clobbered and no longer displays any horizontal line.

tv:basic-typeout-window *Flavor*

This is the base flavor for all kinds of typeout windows. It is actually just a mixin, not instantiable by itself.

tv:typeout-window *Flavor*

(tv:basic-typeout-window tv:notification-mixin tv:window)

This is the flavor normally used for actual typeout windows.

tv:typeout-window-with-mouse-sensitive-items *Flavor*

(tv:basic-mouse-sensitive-items tv:typeout-window)

This flavor of typeout window also provides the :item operation, for including mouse-sensitive rectangles among the typeout. See page 207.

:bottom-reached *Operation on* tv:basic-typeout-window

Returns the greatest *y*-position clobbered by the typeout window. This is a cursor position, relative to the typeout window. The horizontal line (typeout window border), when enabled, appears at this position, provided it is not zero or equal to the inside bottom of the window.

The value is nil when the typeout window is not active.

The typeout window has an instance variable tv:bottom-reached, but this method does not simply return the value of the instance variable.

tv:*enable-typeout-window-borders* *Variable*

When this variable is non-nil, a horizontal line is used to indicate the bottom of the area used by the typeout window. No line appears when the typeout window has used its entire area (if it has wrapped around or done a :clear-screen). When this variable is nil, the horizontal line does not appear. The default value is t.

15.1 Activation and Deactivation

A typeout window is deactivated when not in use. Any attempt to output to it automatically activates and exposes it because its deexposed-typeout-action is (:expose-for-typeout).

:expose-for-typeout *Operation on* tv:basic-typeout-window

Sent in order to prepare the typeout window to be typed out on. The typeout window marks itself "exposed" while leaving the bits of its superior on the screen. It initializes itself as "empty" and its bottom-reached as zero. It also finds a suitable ancestor and makes itself that ancestor's selection substitute. In normal use, this typically causes the typeout window to become selected.

:active-p *Operation on* tv:basic-typeout-window

Returns non-nil if the typeout window is active, which is the case if and only if typeout is currently visible in it.

Exposing the typeout window automatically causes it to become the selection substitute of one of its ancestors (see section 3.3, page 37). Just which ancestor is determined according to the situation; it is the nearest ancestor in the existing path of selection substitutes. This is the nearest ancestor that can be used for the purpose and actually make the typeout window be selected. It is the typeout window's direct superior only if that superior is selected. For example, if you type Meta-X in Zmacs and then type Help, the help message will print on the main editor window's typeout window, but that editor window is not selected (the minibuffer is). So the typeout window will substitute for the editor frame rather than for the nonselected editor window immediately above it.

When the program wants to make the typeout go away and put back its standard display, it must first deactivate the typeout window with the :deactivate operation.

When the typeout window is deactivated, it sends a :remove-selection-substitute message to whichever ancestor it had decided to substitute for. As a result, if the typeout window is still that ancestor's selection substitute, the substitute is set back to what it had been before the typeout window was exposed. If the ancestor's substitute has been changed since then, it is left alone.

The purpose of making the typeout window a selection substitute is primarily to make its cursor blinker blink. A typeout window by default shares the input buffer of its superior, so which of them is selected has no effect on reading keyboard input. A separate feature of typeout windows turns the superior's blinkers off completely while the typeout is exposed.

15.2 Superiors of Typeout Windows

To make a window possess an inferior typeout window, include the flavor tv:essential-window-with-typeout-mixin in it. This causes a typeout window to be created and provides the methods to handle communication with the typeout window.

tv:essential-window-with-typeout-mixin *Flavor*

This is the basic mixin that gives a window the ability to manage a typeout window as its inferior.

tv:window-with-typeout-mixin *Flavor*

(tv:no-screen-managing-mixin tv:essential-window-with-typeout-mixin)

This is what you typically use, rather than tv:essential-window-with-typeout-mixin, because it prevents screen management of this window's inferiors from getting in the way of the operation of the typeout window.

tv:typeout-window *Instance variable of* tv:essential-window-with-typeout-mixin

This window's typeout window.

:typeout-window *Operation on* tv:essential-window-with-typeout-mixin

Returns the value of the instance variable tv:typeout-window, which is the typeout window associated with this window.

:typeout-window *Init option for* tv:essential-window-with-typeout-mixin

(flavor-name options...)

This init option specifies what kind of typeout window to create. The car of the value is the name of flavor of typeout window to use, and the cdr is a list of alternating options and values to pass to make-instance.

If the option is not specified, or is nil, no typeout window is actually created.

The tv:basic-typeout-window flavor provides for daemons and wrappers that cause the :mouse-moves and :mouse-buttons messages to get passed either to the typeout window or to its superior, depending on whether the typeout window has grown down to where the mouse is.

:turn-on-blinkers-for-typeout

Operation on tv:essential-window-with-typeout-mixin

Sent to the superior of a typeout window when the mouse moves into an area that the typeout window is not using, this operation should make visible any blinkers that are associated with the use of the mouse. The definition actually provided by the flavor tv:essential-window-with-typeout-mixin does nothing; this operation exists so that you can add daemons to it.

:turn-off-blinkers-for-typeout

Operation on tv:essential-window-with-typeout-mixin

Sent to the superior of a typeout window when the mouse moves into the area used by the typeout window, this operation should turn off any blinkers that were turned on by

:turn-on-blinkers-for-typeout. The definition actually provided by the flavor tv:essential-window-with-typeout-mixin does nothing; this operation exists so that you can add daemons to it.

A typeout window does **MORE** processing if and only if that is enabled for its superior. The usual motivation for using a typeout window is that the superior is to be used for something other than sequential output; therefore, **MORE** processing is usually not desired on the superior. However, it is not desirable to simply disable **MORE** processing for the superior because this disables it for the typeout window as well and because the user could reenable it for both windows with Terminal M.

:more-p *Operation on* tv:basic-typeout-window

:set-more-p *new-more-p* *Operation on* tv:basic-typeout-window

These operations are passed along to the superior, so that the user who types the Terminal M command need not be aware of the distinction between the typeout window and its superior.

tv:intrinsic-no-more-mixin *Flavor*

This mixin, intended for use in superiors of typeout windows, prevents **MORE** processing unconditionally without saying that it is "disabled". Programs and the user can think they can enable and disable **MORE** processing for the window using the the :more-p and :set-more-p operations, and the Terminal M command, but only the typeout window is affected.

An alternative way to accomplish this is as follows:

```
(defmethod (my-display-window-with-typeout-window
             :more-exception)
           ()
  (setf (tv:sheet-more-flag) 0))
```

15.3 Delaying Redisplay After Typeout

The typeout window superior must know how to check before redisplaying to find out whether part of its last display has been overwritten by the typeout window and therefore must be redisplayed. To find out how much screen height the typeout window has used, use the :bottom-reached operation on it. The typeout window must also be deactivated so that more typeout, happening after the redisplay, will work properly.

Here is an example which is how general scroll windows do this:

```
(defmethod (tv:scroll-window-with-typeout-mixin
             :before :redisplay)
           (&rest ignore)
  (when (funcall tv:typeout-window ':active-p)
    (let ((br (min tv:screen-lines
                   (1+ (truncate (send tv:typeout-window
                                       ':bottom-reached)
                                 tv:line-height)))))
      ;; br is the number of lines of our display
      ;; that were clobbered by typeout.
      (funcall tv:typeout-window ':deactivate)
      (dotimes (l br)
        ;; Mark lines as clobbered
        (aset nil tv:screen-image l 0)
        (aset -1 tv:screen-image l 1)
        (aset -1 tv:screen-image l 2))
      ;; Erase the clobbered area.
      (send self ':draw-rectangle
            (tv:sheet-inside-width)
            (* br tv:line-height)
            0 0
            tv:alu-andca))))
```

The editor normally updates its display after each command. But after a command that prints typeout, it is important not to update the permanent display right away, because that would make the typeout disappear almost as soon as it appeared. The same consideration applies to other programs that use typeout windows.

The convention in this situation is that after a command that has produced typeout, redisplay should be delayed until the user types another input character. If that character is a space, it is discarded. Otherwise, it is interpreted as a command.

The way the program should decide whether to wait before redisplaying is to invoke the :incomplete-p operation on typeout window. This reads a flag that is set whenever output is done on the typeout window and can be cleared by the program's command loop between commands. Thus, the flag indicates whether the typeout window was used during the last command.

Here is a sample piece of code that illustrates this technique:

```
(let ((standard-output typeout-window))
  (do-forever
    ;; Clear the flag.
    (send standard-output ':make-complete)

    ;; Read and execute one command.
    (process-command (send standard-input ':tyi))

    (when (send standard-output ':incomplete-p)
      ;; If this command printed some typeout,
      ;; delay redisplay by waiting for next input char.
      (let ((ch (send standard-input ':tyi)))
        (unless (eq ch #\sp)
          ;; Anything but Space, execute as a command.
          ;; Since Space is not untyi'd, it allows
          ;; immediate redisplay.
          (send standard-input ':untyi ch))))

    ;; Here is where we redisplay after each command.
    (unless (send standard-input ':listen)
      ;; Normal redisplay must deactivate the typeout window;
      ;; see the previous example.
      (redisplay-normal-display))))
```

Note that this command loop follows the editor's practice of not redisplaying when there is input available. As a result, when the character read is not a Space, the :untyi causes redisplay to be prevented by the presence of input. Then the same character is read again at the top of the loop and processed as a command. If this command too prints typeout, its typeout will add on to that already on the typeout window. If this command does not print typeout, the old typeout will be erased after it is done.

:incomplete-p *Operation on* tv:basic-typeout-window

Returns the window's incomplete-flag: t if the command loop should wait for the next character before deactivating the typeout window.

tv:incomplete-p *Instance variable of* tv:basic-typeout-window

The window's incomplete-flag: t if the command loop should wait for the next character before deactivating the typeout window.

:make-complete *Operation on* tv:basic-typeout-window

Clears the incomplete-flag. The command loop can use this to clear the flag after examining it.

Certain functions such as fquery perform this operation on the I/O stream to tell the program not to wait before redisplaying, as it normally would do. The idea is that the fquery question is not worth preserving on the screen once the user has answered it.

:make-incomplete *Operation on* tv:basic-typeout-window

Sets the incomplete-flag. All the standard output stream operations also do this.

16. Text Scroll Windows

Text scroll windows provide a simple means of maintaining a display of a number of lines of the same type with scrolling. For example, they are used by the inspector to display the slots of a structure. (See chapter 17, page 228 for a more general kind of scroll window.)

tv:text-scroll-window *Flavor*

This is the base flavor for all kinds of text scroll windows. It is not instantiable by itself.

A text scroll window updates its display based on a sequence of *items*. Each item generates one line of display. An item can be any Lisp object, and how it displays is controlled by how you define the :print-item operation. For example, you could define this operation to do a :string-out; then the items would have to be strings. By default, :print-item uses the function prin1, so each item is a Lisp object to be printed.

:print-item *item line-no index* *Operation on* tv:text-scroll-window

Displays *item*, which should be the *index*th item of those currently displayed, at the current cursor position in the window, which should be on line number *line-no* of the window.

This operation is the primitive used by all other text scroll window operations to do output of items. As defined by tv:text-scroll-window, it just does prin1 of *item*, ignoring the other arguments. Other flavors built on tv:text-scroll-window are expected to redefine this operation.

In any case, no item may print out as more than one line. This is enforced by truncating output at the margin.

16.1 Specifying the Item List

In simple use, you specify an array of items to be displayed, or a list of items (which is converted into an array). Items are referred to sometimes by their indices in the array. A more sophisticated technique is to specify an *item generator*, which is a function that simulates the effect of a possibly very large array of items without requiring you to actually create the array.

tv:items *Instance variable of* tv:text-scroll-window

The array whose elements are the items to be scrolled through. The index of an item in this array is called the index of the item. This array contains the entire set of items to be scrolled through, not just those that are on the screen at any time.

tv:top-item *Instance variable of* tv:text-scroll-window

The index of the first item currently being displayed (on the first line of the window). This is how the current scroll position is remembered.

The flavor tv:text-scroll-window provides these operations:

:items *Operation on* tv:text-scroll-window

Returns the window's array of items.

:set-items *new-items* *Operation on* tv:text-scroll-window

Sets a new array of items. *new-items* may be a suitable array (it should have a fill pointer), or a list of items (an array is made from it), or a number of items (the array is made that long, but initially empty).

The item-generator of the window is set to nil, turning off that feature, so that the array of items will actually be used.

:top-item *Operation on* tv:text-scroll-window
:set-top-item *new-top-item* *Operation on* tv:text-scroll-window

The top-item is the index of the item to be displayed on the first line of the window.

:number-of-items *Operation on* tv:text-scroll-window

Returns the number of items this window is currently scrolling through.

:number-of-item *item* *Operation on* tv:text-scroll-window

Returns the item number (index) of *item*.

:item-of-number *index* *Operation on* tv:text-scroll-window

Returns the item at index *index*.

:last-item *Operation on* tv:text-scroll-window

Returns the value of the last item to be scrolled through (that is, the one whose index is one less than the number of items).

:put-item-in-window *item* *Operation on* tv:text-scroll-window
:put-last-item-in-window *Operation on* tv:text-scroll-window

Scroll the window so that the specified item, or the last item, appears on the screen. The argument *item* is an item value, not an index.

:delete-item *index* *Operation on* tv:text-scroll-window

Modifies the list of displayable items, removing the item at *index*, and updates the screen if that index is within the portion currently displayed.

:insert-item *index item* *Operation on* tv:text-scroll-window
:append-item *item* *Operation on* tv:text-scroll-window

Add a new item *item* to the list of items to be displayed, either at index *index* (before the item currently at that index) or at the end.

The following auxiliary operations are also defined.

:redisplay *start end* *Operation on* tv:text-scroll-window
This is the internal function that causes a :print-item message to get sent for each line in the range *start* to *end*, which are screen line indices. It should not be redefined, but daemons may be placed on it to note changes in the screen layout.

:scroll-redisplay *new-top delta* *Operation on* tv:text-scroll-window
This is the internal scrolling function that causes partial redisplay with bitblting and then sends a :redisplay message for the rest. *new-top* is the new tv:top-item, and *delta* the number of lines actually to be scrolled. This operation should not be redefined, but daemons may be placed on it.

The operations :scroll-bar-p, :scroll-position, :scroll-to, and :new-scroll-position are also defined for interface with the scroll bar. Other scrolling commands can also use them.

16.2 Bells and Whistles

Function text scroll windows provide for you to change dynamically the function used to display items. These windows have an instance variable which holds the function to be used. The inspector uses this feature so that each data type you can inspect can be handled in an independent manner, with its own conventions for what an item means.

tv:function-text-scroll-window (tv:text-scroll-window) *Flavor*
An instantiable function text scroll window.

tv:print-function *Instance variable of* tv:function-text-scroll-window
This is the function to be called to display an item. See page 224 for an example of a print function, taken from the inspector.

tv:print-function-arg *Instance variable of* tv:function-text-scroll-window
This is an additional argument to be passed to the print function. The print-function's complete list of arguments are the item itself, the value of tv:print-function-arg, the window, and the item number.

:print-function *function* *Init option for* tv:function-text-scroll-window
:print-function-arg *arg* *Init option for* tv:function-text-scroll-window
Initialize the corresponding instance variable.

:print-function *Operation on* tv:function-text-scroll-window
:print-function-arg *Operation on* tv:function-text-scroll-window
:set-print-function *function* *Operation on* tv:function-text-scroll-window
:set-print-function-arg *arg* *Operation on* tv:function-text-scroll-window
Get or set the corresponding instance variable.

:setup *list* *Operation on* tv:function-text-scroll-window
list is a list of the form

(*print-function print-function-arg*
(*item...*)
top-item-number
label
item-generator)

As you can see, it specifies everything relevant to telling the window what items to display and how to display them. *label* is passed to the :set-label operation.

It is not useful to specify both a list of items and a non-nil *item-generator*, since the list of items is not used if the item-generator is non-nil.

The display is updated by this operation.

Since a text scroll window updates a display according to a fixed pattern, it is often useful for it to have an inferior which is a typeout window, for the sake of occasional output that is not part of the standard display (such as, the output for Help in the inspector).

tv:text-scroll-window-typeout-mixin (tv:window-with-typeout-mixin) *Flavor*

This can be added to a flavor containing tv:text-scroll-window and provides a typeout window. It also arranges for proper interaction with the typeout window and partial redisplay over the area it clobbers.

:flush-typeout *Operation on* tv:text-scroll-window-typeout-mixin

If the typeout window is active, this deexposes it, and makes sure that redisplay knows that the lines have been clobbered.

tv:text-scroll-window-empty-gray-hack *Flavor*

This is a mixin that goes with tv:text-scroll-window. When windows of this type have an empty array for tv:items, or an item generator that says the number of items is zero, the interior of the window becomes gray.

This is used in some panes of the window-based debugger frame.

16.3 Item Generators

The item generator feature is how the inspector can scroll through the elements of a large array without having to cons up another equally large array of items.

tv:item-generator *Instance variable of* tv:text-scroll-window

The item generator function, or nil if no item generator is in use. The item generator is a function which simulates the effect of an array of items. It overrides any explicit array of items; the value of tv:items will still be an array, but it will not affect the display.

:item-generator *Operation on* tv:text-scroll-window
:set-item-generator *new-item-generator* *Operation on* tv:text-scroll-window

Get or set the window's item-generator.

The :set-items operation sets the item generator to nil, since if you want to use an explicit list of items, you must not want the item generator to cause them to be ignored.

The item generator function should expect its first argument to be an item generator operation keyword. These are the keywords defined:

:number-of-items
: Returns the number of items to scroll through (the equivalent of the fill pointer in an actual array of items).

:number-of-item *item*
: Returns the index of the specified item. If an actual array were in use, this would be the index in the array where *item* is found.

:item-of-number *index*
: Returns the item at index *index*. If an actual array were being used, this would be the *index*'th element of the array.

:insert-item *index item*
: Insert a new item *item*, before the one at index *index*. If an actual array were in use, this would be done by moving the following elements down. The item generator need support this only if you wish to use the :insert-item or :append-item operation on the window.

:delete-item *index*
: Delete the item at index *index*. The following items move to lesser indices. The item generator need support this only if you wish to use the :delete-item operation on the window.

The inspector uses an item generator to display the elements of an array, so that it does not have to create another array of items as big as the array being displayed. If l is the length of the array's leader, then item numbers 0 through $l - 1$ correspond to the leader, and item number $l + i$ corresponds to array element i (multidimensional arrays being treated as one-dimensional).

The value of the item at item number n is just n. In other words, the virtual array of items that the item generator simulates is an array of consecutive integers, independent of the data being displayed. This may seem to be a weird way of doing things, but consider this: we do not want the line for the i th element to print out as simply that element. We want it to contain the number i as well. So the item value is simply $l + i$, and the :print-item operation is redefined to "print" such a number by printing i followed by the ith array element.

Here is a simplified version of the item generator used by the inspector. Note that the array whose elements are being displayed is found as (car print-function-arg), and (cadr print-function-arg) is non-nil if the leader should be displayed. tv:print-function-arg is an instance variable from the flavor tv:function-text-scroll-window; see page 221.

```
(defselect inspect-array-item-generator
  (:number-of-items ()
    (declare (:self-flavor tv:basic-inspect))
    (+ (if (cadr tv:print-function-arg)
           (or (array-leader-length (car tv:print-function-arg)) 0)
       0)
       (array-length (car tv:print-function-arg))))
  (:number-of-item (item)
    item)        ;; The item's number is the item!
  (:item-of-number (number)
    number))     ;; The number's item is the number!
```

:insert-item and :delete-item are not supported, since the inspector does not try to insert or delete items.

The inspector uses a tv:function-text-scroll-window (see page 221) so :print-object is handled by calling a dynamically changeable *print-function*. Here is a simplified version of the *print-function* used by the inspector when displaying an array.

```
(defun inspect-array-printer
       (item arg window
        &aux (array (car arg))
        (leader-length-to-mention
          (or (and (cadr arg) (array-leader-length array)) 0)))
  ;; arg is the value of tv:print-function-arg.
  ;; (car arg) is the array.
  ;; (cadr arg) is t to display the leader.
  ;; item is a number, as described above.
  (cond ((< item leader-length-to-mention)
         (format window "Leader ~D" item)
         (format window ":~12T ")
         (tv:print-item-concisely
           (array-leader array item) window))
        (t
         (let ((item (- item leader-length-to-mention))
               (rank (array-rank array))
               indices)
           (or (= rank 1)
               (setq indices
                     (array-indices-from-index array item)))
           (format window "Elt ~D"
                   (if (= rank 1) item indices))
           (format window ":~9T ")
           (tv:print-item-concisely
             (ar-1-force obj item) window)))))
```

16.4 Mouse Sensitive Text Scroll Windows

tv:mouse-sensitive-text-scroll-window *Flavor*

Windows of this flavor allow the lines to contain mouse-sensitive items just like those of tv:basic-mouse-sensitive-items (see page 207) though the implementation is different.

Note that the word "item" in "mouse-sensitive item" is completely unrelated in meaning to the items of the text scroll window itself.

:item *Operation on* tv:mouse-sensitive-text-scroll-window
type item &rest *format-args*

All output to text scroll windows is done with the :print-item operation, which is responsible for printing a single item. This operation can include mouse-sensitive items in the output by using the :item operation, which is compatible with that of tv:basic-mouse-sensitive-items (see page 209).

Note that the *item* argument here is the datum to identify the mouse-sensitive item, not the text scroll window item being displayed on this line.

The :item-list and :primitive-item operations are not provided, since in this context they are not really useful.

:item1 *Operation on* tv:mouse-sensitive-text-scroll-window
item type print-function &rest *args*

This is another way of outputting a mouse-sensitive item. *item* and *type* have the same meanings as for the :item operation, but the output is done by calling *print-function* with *item*, the window, and the elements of *args* as arguments.

The :item operation used to do this, but it was changed for compatibility, and the old functionality renamed to :item1.

In a typical tv:basic-mouse-sensitive-items window, mouse-sensitive items are output on specific occasions, and only because they are supposed to be present and mouse-sensitive at that time. In a text scroll window, typically a single display is maintained at all times, but the parts that should be sensitive to the mouse may need to depend on other things. For example, in the inspector, normally the values of slots are sensitive, but when you are specifying a slot to store into, the names of the slots are sensitive instead.

tv:sensitive-item-types *Instance variable of* tv:mouse-sensitive-text-scroll-window

The list of sensitive item types. A mouse sensitive item is sensitive to the mouse if its *type* (as specified in the :item operation) is a member of this list.

t can also be used instead of a list; then all mouse sensitive items actually are sensitive. t is the default value, so that this feature does not get in the way if you do not use it.

:sensitive-item-types *Operation on* tv:mouse-sensitive-text-scroll-window
:set-sensitive-item-types *Operation on* tv:mouse-sensitive-text-scroll-window
new-item-types
Get or set the list of sensitive item types.

:sensitive-item-types *Init option for* tv:mouse-sensitive-text-scroll-window
item-types
Initializes the set of sensitive item types.

The inspector's print function shown in the previous section really does its output using the :item1 operation so that the output becomes mouse-sensitive. Here is the real code for the cond-clause that handles leader elements:

```
((< item leader-length-to-mention)
 (funcall window ':item1 item 'leader-slot
          #'(lambda (item window)
              (format window "Leader ~D" item)))
 (format window ":~12T ")
 (funcall window ':item1 (array-leader array item)
          ':value #'tv:print-item-concisely))
```

leader-slot and :value are item types which the inspector makes mouse sensitive at various times.

When the mouse is clicked on a mouse sensitive item, a blip is placed in the window's input buffer. The blip looks like

(*type item window mouse-character*)

type is the item type, such as leader-slot or :value, and *item* is the actual item value specified in the :item or :item1 operation. *window* is the text scroll window itself. (This is how the inspector can tell which inspect pane you click on.) *mouse-character* is a character whose %%kbd-mouse bit is 1. It tells the program which button was clicked.

tv:line-area-text-scroll-mixin *Flavor*
This mixin, when added to tv:text-scroll-window, creates a "line area" near the left edge where the mouse cursor changes to a rightward arrow and a click means something different. The line area is an additional part of the left margin and does not overlap the space used for displaying the items.

You must also include the flavor tv:margin-region-mixin in the flavor combination you instantiate.

A mouse click in the line area puts a blip into the input buffer that looks like this:

(:line-area *item window button-mask*)

button-mask is a mask of bits corresponding to mouse buttons; see tv:mouse-last-buttons, page 116, for how to interpret it.

:line-area-width *number* *Init option for* tv:line-area-text-scroll-mixin
Specifies the width of the line area in pixels as *number*.

:line-area-mouse-documentation *Operation on* tv:line-area-text-scroll-mixin
This operation should return a string to display in the mouse documentation line while the cursor is in the line area.

tv:line-area-mouse-sensitive-text-scroll-mixin *Flavor*
This flavor should be used instead of tv:line-area-text-scroll-mixin if tv:mouse-sensitive-text-scroll-window is in use.

tv:current-item-mixin *Flavor*
This flavor, when added to tv:line-area-text-scroll-mixin, identifies one of the items with an arrow in the line-area.

tv:current-item *Instance variable of* tv:current-item-mixin
The item to be marked with an arrow, or nil if none. An arrow will mark this item if it is on the screen, no matter where it scrolls to.

:current-item *Operation on* tv:current-item-mixin
:set-current-item *item* *Operation on* tv:current-item-mixin
Get or set the value of tv:current-item.

These flavors are part of the implementation of tv:mouse-sensitive-text-scroll-window.

tv:mouse-sensitive-text-scroll-window-without-click *Flavor*
This is a component of tv:mouse-sensitive-text-scroll-window that provides everything but the :mouse-click method. Since this operation uses :or method-combination, it is not possible to override a method once it is present.

tv:displayed-items-text-scroll-window *Flavor*
This flavor records additional information about the items that are actually displayed. It provides an instance variable, tv:displayed-items, which is an array indexed by line number. In this array, the :print-item operation can store any relevant information about what was displayed on the line.

The meaning of elements of the array is not defined by this flavor. The :print-item operation is responsible for storing whatever information is useful into the appropriate slot of the array. However, this flavor does move elements of the array when scrolling is done, and set them to nil when parts of the window are cleared, or when they are about to be redisplayed.

This flavor is essentially a subroutine of tv:mouse-sensitive-text-scroll-window, which uses each element of tv:displayed-items to hold information on the mouse-sensitive items for the line.

tv:displayed-items *Instance variable of* tv:displayed-items-text-scroll-window
The array of information about lines on the screen.

Despite all this hair, no window yet devised is as mouse-sensitive as my mother.

17. General Scroll Windows

General scroll windows are used to put up a continuously-maintained display of items, each of which can vary in size. They are used by Peek. General scroll windows (from now on called simply scroll windows) are not a generalization or a building block of text scroll windows, but rather an independent facility.

The scroll window's display is made up of items. These items are not the same as items in text scroll windows; the same term is used because they fit in a similar place in the scheme of things.

An item in a scroll window always occupies an entire line or several entire lines. An item can be composed of sub-items which are juxtaposed vertically, each sub-item occupying and filling up some number of lines. The sub-items can in turn be composed of more items. New sub-items can be dynamically added or deleted at any level, and the display is updated automatically to match by moving lines around on the screen.

Eventually this process of subdivision must come to an end, with *lowest-level* items made up of *entries*, which are juxtaposed in a horizontal sequence.

An entry displays a single string or quantity, updating its display if the value changes. The entry must record how to obtain a value to display, how to tell when the value has changed since the screen was updated, and how to output the new value. A single entry can wrap around at the right margin just like ordinary output. Entries can be added to and removed from an item dynamically.

In Peek's Active Processes display, there is a single item that displays the entire set of processes. It is composed of sub-items, one for each process. If a new process appears, a new sub-item is created to display it. The sub-item for a single process is a lowest-level item. Each of the things displayed about a process—its name, its run state, its priority, its percentage use of the cpu—is displayed by a single entry in that item.

The line of column headings at the top of the display is also a lowest-level item; its entries display constant strings.

Every character displayed on a scroll window comes from an entry. The items serve only to group entries, and to control the automatic insertion and deletion of entries.

Entries can be either fixed or variable width. A variable width entry takes up as much space as is needed to print its data; this can change when the window is redisplayed. When that happens, the remaining entries in the item all have to move left or right. A fixed width entry specifies an amount of horizontal space and always occupies that much space. As a result, it can be redisplayed without redisplaying the rest of the item afterward. The entries used in the Active Processes display are all fixed-width so that they will line up in columns

Note: if the entry specifies a fixed width and the printing of its contents goes past that width, the window redisplay algorithm will be confused.

The data structure that represents an item is either a list or an array. If it is a list, its cdr is a list of component items, and its car contains information on how to update the list (add or remove component items). Then the item is displayed simply as the concatenation of its components. If it is an array, then it is a lowest-level single-line item, and the elements of the array represent entries on the line. The array also has leader slots whose meanings are described below.

17.1 Specifying Items and Entries

You do not generally create an array item or an entry yourself. They are made by calling the function tv:scroll-parse-item, which is given a descriptive data structure made out of lists. Examples of its use are at the end of this section.

The arguments to tv:scroll-parse-item are *entry descriptors*, each of which specifies how to create one entry. The entries thus specified all go together into one item.

Here are the possible kinds of entry descriptors:

a string *string*
a list (:string *string* [*width*])

The entry is displayed by printing *string*. A string entry never varies, since it always displays precisely the specified string, and is always fixed width. The width can be specified as a means of controlling the position of the following entry; otherwise, the actual width needed to print the string is the width of the item.

Example: either (:string "Foobar" 10.) or "Foobar " specifies an entry that prints as `Foobar` followed by 4 spaces.

a list (:symeval *symbol* [*width-or-nil*] [*format-string*])

The entry is displayed by printing the value of *symbol*, by passing it to format together with *format-string*. If *format-string* is omitted, the value is printed with princ. This type of entry is automatically updated when the value of *symbol* changes.

width-or-nil may be a number of pixels, to specify a fixed-width entry, or nil to specify a variable-width entry.

Example:

```
(:symeval base nil " ~D. ")
```

specifies an entry that prints the value of base in decimal with a following period and a space in front and in back. It is variable-width so the space it takes up is three plus however many digits are needed to print the value of base.

a list (:function *function list-of-args* [*width-or-nil*] [*format-string*])

The value to display is obtained by applying *function* to *list-of-args*. If this value has changed since the last time it was checked, it is displayed by passing it to format together with *format-string*. If *format-string* is nil, the value is simply princ'd.

width-or-nil may be a number of pixels, to specify a fixed-width entry, or **nil** to specify a variable-width entry.

Example:

```
'(:function si:process-quantum-remaining
  (,process) 5. ("~4D//"))
```

is an expression that creates an entry descriptor which specifies an entry that will call si:process-quantum-remaining on some process and print the result in decimal, followed by a slash, in a field 5 characters wide.

an interpreted function (lambda ...)
an interpreted function (named-lambda ...)
a compiled function

An entry descriptor which is either a compiled function (a FEF) or a list starting with lambda or named-lambda is considered a function. It is treated as an abbreviation for (:function *function*), which specifies no arguments, variable width, and no format string (the value is printed with princ).

a list (:value *index* [*width-or-nil*] [*format-string*])

The value to be displayed is found at *index* in the window's *value-array*.

Two other keywords can be used in an entry descriptor to make the entry mouse sensitive. They can be used only in scroll windows which have tv:essential-scroll-mouse-mixin (see section 17.6, page 238). To use these keywords, first you construct an entry descriptor to specify how the entry should print, according to the preceding table. Then you add one of these keywords and a value to go with it at the front of the list. The mouse keyword gives the entry mouse sensitivity but has no effect on how the entry appears on the screen.

:mouse The keyword :mouse is used in an entry descriptor that looks like

(:mouse *mouse-data* . *another-entry-descriptor*)

Such an entry descriptor is handled by creating an entry from *another-entry-descriptor*, and then modifying it by recording *mouse-data* as the mouse sensitivity of the entry. The resulting entry will print according to *another-entry-descriptor* but will be mouse sensitive as well.

:mouse-item The keyword :mouse is used in an entry descriptor that looks like

(:mouse-item *mouse-data* . *another-entry-descriptor*)

:mouse-item is like :mouse except that the symbol tv:item is replaced throughout *mouse-data* with *item*, the item this entry is going to become part of. *mouse-data* better be a list.

There is no way to cause the entry itself to be inserted into its own mouse sensitivity datum because this is not useful when scroll windows are used in the intended manner.

tv:scroll-parse-item &rest *keyword-args-and-entry-descriptors*

Creates and returns an array item containing entries constructed according to *keyword-args-and-entry-descriptors*.

keyword-args-and-entry-descriptors begins optionally with some alternating keywords and values. They are followed by entry descriptors, one for each entry you want in the item. The keywords and values at the beginning specify information that applies to the item as a whole. Keywords and entry descriptors are distinguished by the fact that an entry descriptor is never a symbol.

The keywords defined are

:mouse — The value is stored as the mouse-sensitivity of the entire item. This is meaningful only if the window flavor includes tv:essential-scroll-mouse-mixin (see section 17.6, page 238).

:mouse-self — The value is stored as the mouse-sensitivity of the entire item, but first the symbol self is replaced wherever it appears by the item itself (the array that this function is constructing). This is meaningful only if the window flavor includes tv:essential-scroll-mouse-mixin (see section 17.6, page 238).

:leader — This keyword requests extra slots to be allocated in the array leader of the item array. It is either a number, the number of extra slots desired, or a list, whose length is the number of extra slots and whose contents are used to initialize them.

tv:scroll-interpret-entry *entry-descriptor item*

Creates and returns an entry according to *entry-descriptor* for use in the array item *item*. You do not normally call this function yourself; it is used as a subroutine of tv:scroll-parse-item.

tv:scroll-string-item-with-embedded-newlines *string*

Returns an item that will display the string *string*. This item is composed of one item for each line making up *string*.

Here is an example taken from Peek; it makes the item for a process (the value of process) in Active Processes mode. The entries that use the process as a function work because the process is a flavor object; the argument given to the process is a flavor operation. Note that tv:peek-process-menu is a function in Peek which asks for a choice with a momentary menu.

```
(tv:scroll-parse-item
  ;; The first entry is mouse-sensitive.
  '(:mouse-item
     (nil :eval (peek-process-menu ',process 'item 0)
          :documentation
          "Menu of useful things to do to this process.")
     :string ,(process-name process) 30.)
  '(:function ,#'peek-whostate ,(ncons process) 25.)
  '(:function ,process (:priority) 5. ("~D."))
  '(:function ,process (:quantum-remaining) 5. ("~4D//"))
  more entries...)
```

17.2 Using a Scroll Window

tv:basic-scroll-window *Flavor*

All flavors of scroll window are built on this flavor, which provides all the facilities specific to scroll windows. It is not instantiable by itself.

tv:scroll-window *Flavor*

(tv:flashy-scrolling-mixin tv:basic-scroll-window tv:borders-mixin tv:basic-scroll-bar tv:window)

This is an instantiable scroll window flavor. It provides for a scroll bar and margin scrolling, and for borders and labels.

In addition to being able to create a tree of items and entries, you must tell the scroll window to display them. At the highest level, the entire display is grouped into a single item, the *root item*. Switching modes in Peek works by switching to a new root item.

tv:display-item *Instance variable of* tv:basic-scroll-window

The root item of the window. The window's display is precisely whatever comes from this item, and nothing more. Usually the root item contains some number of subitems which do the real work.

:display-item *Operation on* tv:basic-scroll-window
:set-display-item *item* *Operation on* tv:basic-scroll-window

Get or set the root item of the window. Setting the root item redisplays the window.

:display-item *item* *Init option for* tv:basic-scroll-window

Initializes the root item.

tv:truncation *Instance variable of* tv:basic-scroll-window

If this is nil, entries can wrap around at the right margin. Otherwise, each item can occupy only one line.

:truncation *Operation on* tv:basic-scroll-window
:set-truncation *flag* *Operation on* tv:basic-scroll-window

Get or set the truncation flag. Setting the flag redisplays the window.

:truncation *flag* *Init option for* tv:basic-scroll-window

Initializes the truncation flag.

A scroll window has a *value array* whose elements may be used to hold arbitrary data to be displayed by entries using the keyword :value. Such an entry specifies the index of a slot in the value array whose contents are the data to display. Putting appropriate data in the value array is up to you. One technique is to have an automatically updating item whose update function stores data into the value array, and have entries in the item look in those slots. There can be many such items, all using the same value array slots. See section 17.4, page 234.

tv:value-array *Instance variable of* tv:basic-scroll-window
The window's value array.

:value-array *Operation on* tv:basic-scroll-window
Returns the window's value array.

:value-array *array-or-length* *Init option for* tv:basic-scroll-window
Initializes the window's value array, or specify how long to make it.

The :redisplay operation updates the display based on the current root item, automatically reprinting the entries whose contents have changed. :redisplay will be done automatically by the window system at certain times (such as when the window size is changed, or the screen is refreshed), but if you want it to happen simply because some of the displayed data has changed, you must send a :redisplay message yourself.

:redisplay-selected-items is another way to request display updating, which allows you to control which items will be checked.

:redisplay &optional *full-p force-p* *Operation on* tv:basic-scroll-window
Redisplays the contents of the scroll window. If *full-p* is nil, the window assumes that its screen bits contain the result of the last redisplay that was done, and only items and entries whose contents are different from last time are actually output. If *full-p* is non-nil, everything that is supposed to be on the screen is redrawn.

force-p non-nil means update the contents of the window even if it is not exposed. Normally, this operation will wait if the window is not exposed.

:redisplay-selected-items *list-of-items* *Operation on* tv:basic-scroll-window
Redisplays the items in *list-of-items*, if they are present on the screen. Other items in the current item hierarchy are not even considered for redisplay.

Since a scroll window shows a constantly updated display, it is often useful to have a typeout window in it for occasional output that is not part of the display that is usually shown.

tv:scroll-window-with-typeout-mixin *Flavor*
This mixin should be used in addition to tv:window-with-typeout-mixin on any scroll window that is to have a typeout window. It handles interfacing between typeout window output and redisplay of the scroll window.

tv:scroll-window-with-typeout *Flavor*
A scroll window that has an inferior typeout window. See chapter 15, page 212.

17.3 Inserting and Deleting Items

Scroll windows provide operations for replacing, inserting and deleting items explicitly. Since the items form a multilevel hierarchy, the position at which to replace, insert, or delete the item must be specified as a list of numbers. For example, (1 3 0) as a position means item number 0 within item number 3 within item number 1 (within the root item, tv:display-item). nil as a position refers to the root item itself.

:get-item *position* *Operation on* tv:basic-scroll-window
Returns the item at *position*.

:set-item *position item* *Operation on* tv:basic-scroll-window
Stores *item* into the hierarchy at *position*.

:insert-item *position item* *Operation on* tv:basic-scroll-window
Inserts *item* at *position*, before the item that was at *position*.

:delete-item *position* *Operation on* tv:basic-scroll-window
Deletes the item at *position*, so that the following item moves to that position.

These operations also update the window on the screen as necessary.

17.4 Automatically Updating Items

Just as an entry automatically updates the value it displays, sometimes one wants an item to update automatically the list of items it contains. For example, the Active Processes display contains one item that displays a list of all active processes. This item contains a list of component items, one item per process. Just before the displayed entries for each process are updated if necessary, additional items should be created and inserted in the list if there are any newly active processes, and items should be removed if processes have become inactive.

The first element of an item that is a list is used to store the data of a property list for the item. Two properties are given standard meanings:

:pre-process-function
The value of this property is a function to be called whenever it is time to display this item. Its sole argument is the item itself. The function can modify the item. The value it returns is ignored.

:function The value of this property is a function to update an individual component of this item. This function is called each time any component item is about to be displayed or otherwise thought about.

The arguments given to the function are the component item, the reverse of the *position* of that item (a list of integers), and the location of the property list of the containing item, the same property list on which this :function property appears (this can be passed directly to get). To repeat, the second argument is the reverse of the position as would be passed to the :get-item operation or related operations. This is because it is easier to implement that way without consing.

The function should return an updated component item, perhaps the same one as it was passed, perhaps a new one.

Other properties can be used for any purpose. Some of the commonly used pre-process functions use other properties for their internal state information and additional parameters.

tv:scroll-maintain-list *init-fun item-fun* &optional *per-elt-fun stepper compact-p pre-proc-fun*

Returns an item which maintains a list of component items, one for each element of a driving list. The item updates automatically so that component items appear and disappear as elements of the driving list do.

init-fun should be a function of no arguments that returns the current value of the driving list. *item-fun* should be a function that, given an element of the list, returns a component item to use to display for that element. *item-fun* is called each time a new element appears in the driving list. The item created starts out with no component items. The appropriate set of component items is created by adding them one by one in this way, the first time the item is updated.

This item works because it is given a suitable pre-process function. The other arguments to tv:scroll-maintain-list are also stored on the property list of the item created. In particular, *per-elt-fun* becomes the :function property. (That is all *per-elt-fun* is used for.)

Normally the value from *init-fun* is a list, and the objects that the items are made from are the elements of this list, but it is possible to extract the objects in other ways. If *stepper* is not nil, it should be a function to step through a "kind of list". *stepper* is called with one argument, a "kind of list", and returns three values:

the first element extracted from it
a "kind of list" of the remaining elements
non-nil to say there are no more elements
nil as the "kind of list" is always recognized as being empty, regardless of the third value.

stepper is first called with the value returned by *init-fun*. The first value goes (if it is new) to the *item-fun*; the second is fed back to *stepper* unless either it is nil or the third value is non-nil.

A *stepper* function that could step through the properties in a property list might be:

```
(defun plist-stepper (plist-tail)
  (values (car plist-tail) (cddr plist-tail)))
```

compact-p non-nil says to recopy the list each time an element is inserted or deleted, so that the list remains compact and localized.

Here is how Peek, in Window Hierarchy mode, recursively creates a tree of automatically updating items:

```
;; Make an item to describe the entire window hierarchy.
(defun peek-window-hierarchy (ignore)
  (tv:scroll-maintain-list
    ;; The init-fun. When called, it returns a current list of screens.
    #'(lambda () tv:all-the-screens)
    ;; The item-fun, which makes an item for a screen.
    #'(lambda (screen)
        (list ()
              (tv:scroll-parse-item
                (format nil "Screen ~A" screen))
              (peek-window-inferiors screen 2)
              (tv:scroll-parse-item "")))))
;; No per-elt-fun is needed. Also, the default stepper works
;; because our "list" really is a list.

;; Make an item to describe window and its inferiors.
;; indent is an indentation to print with.
(defun peek-window-inferiors (window indent)
  (declare (special window indent))
  (tv:scroll-maintain-list
    (closure '(window) #'(lambda () (tv:sheet-inferiors window)))
    (closure '(indent)
      #'(lambda (sheet)
          ;; Make an item with two subitems
          (list ()
                ;; One for this window,
                (tv:scroll-parse-item
                  (format nil "~VX" indent)
                  '(:mouse
                    (nil :eval (peek-window-menu ',sheet)
                     :documentation
                     "Menu of useful things to do to this window.")
                    :string ,(send sheet ':name)))
                ;; and one with subitems for its inferiors.
                (peek-window-inferiors sheet (+ indent 4)))))))
```

And here is how it makes the item that displays a chaosnet connection's packets.

```
(tv:scroll-maintain-list
  '(lambda () (chaos:read-pkts ',conn))
  '(lambda (x)
     (peek-chaos-packet-item x ,(+ indent 2)))
  nil
  #'(lambda (state)
      (values state (chaos:pkt-link state)
              (null (chaos:pkt-link state)))))
```

Note that instead of a list of packets there is a chain, with each packet pointing to the next one. Therefore, an explicit *stepper* is required. chaos:pkt-link is the function which, given one packet, returns the next one in the chain (or nil at the end).

tv:scroll-maintain-list-unordered *init-fun item-fun* &optional *per-elt-fun stepper*

Returns an item which maintains an unordered list of component items, one for each element of a driving list. The item updates automatically so that component items appear and disappear as elements of the list do.

This function is very much like tv:scroll-maintain-list. The difference is that new component items are always added at the front of the combined item, no matter where they appear in the driving list. Changes in the order of that list have no effect at all. This is why this function is called "unordered".

tv:scroll-maintain-list-update-states *elements window* &optional *item*

Redisplays some of the component items of *item*, an item of the sort created by tv:scroll-maintain-list or tv:scroll-maintain-list-unordered.

elements is a list that specifies which component items to update. If the element of the driving list from which a component item was made is memq of *elements*, then the component item is updated.

17.5 Representation of Items

An item is either a list or an array. A list item contains other items, while an array item contains entries.

List items have these accessor functions:

tv:scroll-item-component-items

Returns the list of component items of this item.

tv:scroll-item-plist

Returns the contents of the property list of this item.

Array items have these accessor functions, which refer to array leader slots. (The array elements themselves hold the entries in the item.)

tv:scroll-item-size

Returns the number of entries in the item.

tv:scroll-item-mouse-items
Returns a list of mouse-sensitive areas of entries in this item.

tv:scroll-item-line-sensitivity
Returns what was specified for mouse sensitivity of the item as a whole (using the :mouse or :mouse-self keyword in tv:scroll-parse-item).

tv:scroll-item-leader-offset *Variable*
The number of standardly-defined slots in an item's array leader. The slot with this number and beyond can be used by applications for their own purposes.

Entries are also arrays. They have a lot of components, all managed internally, and users should probably not access them directly. Peek never needs to do so.

17.6 Mouse Sensitive Scroll Windows

tv:essential-scroll-mouse-mixin *Flavor*
This mixin gives a scroll window the ability to make either items or entries mouse sensitive.

tv:scroll-mouse-mixin *Flavor*
This mixin in addition defines the :execute operation to be the same as on menus.

tv:scroll-parse-item provides syntax, described above, for associating a mouse sensitivity to any item or entry. The mouse sensitivity is a list whose purpose is to identify which mouse-sensitive area was clicked on, and also specify what to do when that happens.

If the car of the mouse sensitivity is nil, then the mouse sensitivity is interpreted as a menu item. When the sensitive area is clicked on, the menu item is executed by means of the :execute operation—but this is done in the mouse process. Unfortunately, there is no way to avoid this, since mouse clicks on scroll windows are expected to be able to happen "at any time", and no other process has expressed its willingness to handle them with a :choose operation.

If the car of the mouse sensitivity is non-nil, a click is handled by putting a blip into the scroll window's input buffer. The blip has the form
(*blip-type sensitivity window mouse-character*)
sensitivity is the mouse sensitivity list. *blip-type* is the car of that list. *window* is the scroll window itself, and *mouse-character* is a character such as #\mouse-1-1 which indicates which button was clicked.

The reason that the *blip-type* is extracted and put at the front is that programs that use scroll windows may need to handle blips from many sources. By specifying the car of each mouse sensitivity, the program can arrange to distinguish these blips from blips coming from menus, typeout windows, etc. and process each one in the correct fashion.

Often a scroll window displays many similar items that describe different data objects. These items will all have the same patterns of mouse sensitivity. One way for the program to tell which item the user clicked on is to set up the mouse sensitivity using the :mouse-self keyword (for an item) or :mouse-item (for an entry). This inserts the item itself into the sensitivity in place of

the symbol **self** or **tv:item**, respectively.

Concept Index

active window . . . 11
alu function . . . 66
ancestors of a window . . . 10
autoexposure . . . 26
autoselection . . . 26

baseline . . . 87
bit-save array . . . 15
black on white . . . 14
blinker . . . 103
blinker height . . . 88
blinker width . . . 88
blip . . . 52
blip types . . . 160, 174, 184, 203, 208, 226, 238
border margin width . . . 130
borders . . . 130
burying . . . 29

char-aluf . . . 66
char-exists table . . . 88
character height . . . 87
character width . . . 67, 88
choice boxes . . . 210
choose-variable-values windows . . . 194
clicks, mouse, encoding of . . . 113
clipping . . . 93
color screen . . . 165
constraint frames . . . 142
continuation of lines . . . 73
current font . . . 67, 83

deexposed typein action . . . 32
deexposed typeout action . . . 21
delaying screen management . . . 30
descendants of a window . . . 10

encoding of mouse clicks . . . 113
entries, in scroll windows . . . 228
exposable window . . . 17
exposed window . . . 17

filled menus . . . 178
fixed-width font . . . 88
flavor . . . 6
following blinker . . . 103
font format . . . 89
font indexing table . . . 90
font map . . . 67, 83
font purposes . . . 86
font specifier . . . 85
forcing keyboard input . . . 50
frame . . . 141
fully visible . . . 10

geometry (or menus) . . . 178
global asynchronous characters . . . 63
grabbing the mouse . . . 115

half-period of a blinker . . . 103
hierarchy of windows . . . 10
holding output . . . 21
horizontal wraparound . . . 73

i/o buffers . . . 56
inferior windows . . . 10
input buffer . . . 50
input buffers, sharing . . . 51
inside . . . 129
item generators . . . 222
items, in scroll windows . . . 228
items, in text scroll windows . . . 219

keyboard input . . . 49

label . . . 132
left kern . . . 88
line height . . . 67

margin choices . . . 210
margin item . . . 129
margin regions . . . 134
margins . . . 129
menu geometry . . . 178
menu items . . . 174
menus . . . 173
mouse . . . 112
mouse process . . . 113
multiple choice windows . . . 190

negative priorities . . . 29
notifications . . . 157

opening blinkers . . . 103
output hold . . . 21
output hold flag . . . 21
output of text . . . 66
overlapping windows . . . 10
overstriking . . . 66
owning the mouse . . . 113, 114

pane . . . 10, 141
partially visible . . . 10, 26
pixel . . . 14
position of window . . . 43
priority . . . 28
processes . . . 40

raster height . . . 89

raster width 89

screen array 17
screen manager 11, 26
screens 13
scroll bar 125
scroll windows 228
sections, in constraint frames. 147
select menu 35
selected pane. 155
selected window 31
selection. 31
selection substitutes 37
sharing input buffers. 51
size of window 43
sorting priority 28
stacking, in constraint frames. 147
superior window 10
System key. 63

teams of windows 34
temp-locking 23
temporary window. 24
Terminal key 63
text scroll windows 219
tracking the mouse 113
truncation of lines 73
typeout windows 212

usurping the mouse 118

value array of a scroll window 232
variable-width font 88
vertical spacing (vsp) 67
visibility of a blinker 103
visible 10, 17
vsp 67

warping the mouse 112
white on black 14
who line 163
wide fonts. 89
window 3
window hierarchy. 10
wraparound, horizontal 73

Operation Index

:activate
(on windows) 11
:active-p
(on windows and screens). 12
(on tv:basic-typeout-window). 213
:add-asynchronous-character
(on tv:stream-mixin) 62
:add-highlighted-item
(on tv:menu-highlighting-mixin) 189
:add-highlighted-value
(on tv:menu-highlighting-mixin) 189
:add-item
(on tv:margin-multiple-menu-mixin) 188
:add-server
(on tv:who-line-file-sheet). 164
:add-stream
(on tv:who-line-file-sheet). 164
:adjustable-size-p
(on tv:scroll-stuff-on-off-mixin) 128
(on tv:choose-variable-values-window). 202
:alias-for-inferiors
(on windows) 36
:alias-for-selected-windows
(on windows) 36
:any-tyi
(on tv:stream-mixin) 52
:any-tyi-no-hang
(on tv:stream-mixin) 53
:append-item
(on tv:text-scroll-window) 220
:appropriate-width
(on tv:choose-variable-values-window). 203
:array
(on tv:bitblt-blinker) 110
:arrest
(on tv:select-mixin) 42
:asynchronous-character-p
(on tv:stream-mixin) 62
:await-exposure
(on windows) 23
:backward-char
(on windows) 75
:baseline
(on windows) 84
:beep
(on windows) 69
:bit-array
(on windows) 16
:bitblt
(on tv:stream-mixin) 94
:bitblt-from-sheet
(on tv:stream-mixin) 95
:bitblt-within-sheet
(on tv:stream-mixin) 95
:blink
(on blinkers) 107
:blinker-list
(on windows and screens) 104
:border-margin-width
(on tv:borders-mixin). 131
:bottom-margin-size
(on windows) 129
:bottom-reached
(on tv:basic-typeout-window) 212
:bury
(on windows) 29
:call
(on tv:select-mixin) 42
:call-mini-buffer-near-window
(on zwei:temporary-mode-line-window-with-borders) 162
:center-around
(on windows) 46
(on tv:menu) 183
:change-of-default-font
(on windows) 87
:change-of-size-or-margins
(on windows) 47
:character
(on tv:character-blinker) 109
:character-width
(on windows) 77
:choose
(on tv:multiple-choice) 193
(on tv:menu) 183
:chosen-item
(on tv:menu) 183
:clear-between-cursorposes
(on windows) 76
:clear-char
(on windows) 75
:clear-eof
(on windows) 75
:clear-eol
(on windows) 75
:clear-input
(on tv:stream-mixin) 54
:clear-screen
(on windows) 76
:clear-string
(on windows) 75
:column-row-size
(on tv:menu) 184
:column-spec-list
(on tv:dynamic-multicolumn-mixin). 187
:compute-margins
(on windows) 138
:compute-motion
(on windows) 77

:configuration
(on tv:basic-constraint-frame) 153
:create-pane
(on tv:basic-constraint-frame) 153
:current-font
(on windows) 84
:current-geometry
(on tv:menu) 180
:current-item
(on tv:menu) 183
(on tv:current-item-mixin). 227
:deactivate
(on windows) 11
:decide-if-scrolling-necessary
(on tv:scroll-stuff-on-off-mixin) 127
:decode-variable-type
(on tv:basic-choose-variable-values). 199
:deexpose
(on windows and screens) 20
:deexposed-typein-action
(on windows) 32
:deexposed-typeout-action
(on windows) 22
:defer-reappearance
(on tv:blinker). 106
:delayed-set-label
(on tv:delayed-redisplay-label-mixin) 134
:delete-all-servers
(on tv:who-line-file-sheet) 164
:delete-all-streams
(on tv:who-line-file-sheet) 164
:delete-char
(on windows) 76
:delete-item
(on tv:text-scroll-window). 220
(on tv:basic-scroll-window) 234
:delete-line
(on windows) 76
:delete-server
(on tv:who-line-file-sheet) 164
:delete-stream
(on tv:who-line-file-sheet) 164
:delete-string
(on windows) 76
:deselect
(on windows) 33
:deselected-visibility
(on tv:blinker). 106
:display-item
(on tv:basic-scroll-window) 232
:display-lozenged-string
(on windows) 70
:draw-char
(on tv:graphics-mixin) 95
:draw-circle
(on tv:graphics-mixin) 96
:draw-circular-arc
(on tv:graphics-mixin) 96
:draw-cubic-spline
(on tv:graphics-mixin) 97
:draw-curve
(on tv:graphics-mixin) 96
:draw-dashed-line
(on tv:graphics-mixin) 95
:draw-filled-in-circle
(on tv:graphics-mixin) 97
:draw-filled-in-sector
(on tv:graphics-mixin) 97
:draw-line
(on tv:graphics-mixin) 95
:draw-lines
(on tv:graphics-mixin) 95
:draw-point
(on tv:graphics-mixin) 94
:draw-rectangle
(on tv:stream-mixin). 96
:draw-regular-polygon
(on tv:graphics-mixin) 97
:draw-triangle
(on tv:graphics-mixin) 96
:draw-wide-curve
(on tv:graphics-mixin) 96
:edges
(on windows) 46
:edit
(on standalone editor windows). 160
:enable-scrolling-p
(on tv:basic-scroll-bar). 126
(on scrolling windows) 124
:end-of-line-exception
(on windows) 73
:end-of-page-exception
(on windows) 71
:execute
(on tv:menu-execute-mixin). 177
(on tv:menu) 183
:execute-no-side-effects
(on tv:menu-execute-mixin). 177
:exposable-p
(on windows and screens) 20
:expose
(on windows and screens) 19
:expose-for-typeout
(on tv:basic-typeout-window) 213
:expose-near
(on windows) 46
:exposed-inferiors
(on windows and screens) 21
:exposed-p
(on windows and screens) 20
:fat-string-out
(on windows) 69
:fill-p
(on tv:menu) 180

:flush-typeout
(on tv:text-scroll-window-typeout-mixin). 222
:font-map
(on windows) 83
:force-kbd-input
(on tv:stream-mixin) 53
:forward-char
(on windows) 74
:fresh-line
(on windows) 69
:geometry
(on tv:menu). 180
:get-configuration
(on tv:basic-constraint-frame). 153
:get-item
(on tv:basic-scroll-window) 234
:get-pane
(on tv:basic-constraint-frame). 153
:half-period
(on tv:blinker) 106
:handle-asynchronous-character
(on tv:stream-mixin) 62
:handle-exceptions
(on windows) 71
:handle-mouse
(on windows) 119
:height
(on windows) 45
:highlighted-items
(on tv:menu-highlighting-mixin) 189
:highlighted-values
(on tv:menu-highlighting-mixin) 189
:home-cursor
(on windows) 74
:home-down
(on windows) 74
:hysteresis
(on tv:hysteretic-window-mixin) 115
:incomplete-p
(on tv:basic-typeout-window). 217
:increment-cursorpos
(on windows) 74
:inferiors
(on windows and screens). 12
:insert-char
(on windows) 77
:insert-item
(on tv:text-scroll-window) 220
(on tv:basic-scroll-window) 234
:insert-line
(on windows) 77
:insert-string
(on windows) 77
:inside-edges
(on windows) 46
:inside-height
(on windows) 45
:inside-size
(on windows) 45
:inside-width
(on windows) 45
:interval
(on editor windows) 160
:interval-string
(on editor windows) 160
:io-buffer
(on tv:stream-mixin) 52
(on tv:command-menu). 185
:item
(on tv:mouse-sensitive-text-scroll-window). 225
(on tv:basic-mouse-sensitive-items) 209
:item-cursorpos
(on tv:menu) 184
:item-generator
(on tv:text-scroll-window). 222
:item-list
(on tv:menu) 183
(on tv:basic-mouse-sensitive-items) 210
:item-list-pointer
(on tv:dynamic-item-list-mixin) 186
:item-of-number
(on tv:text-scroll-window). 220
:item-rectangle
(on tv:menu) 184
:item-type-alist
(on tv:basic-mouse-sensitive-items) 208
:item1
(on tv:mouse-sensitive-text-scroll-window). 225
:items
(on tv:text-scroll-window). 220
:kill
(on windows) 11
:label-size
(on tv:label-mixin). 133
:last-item
(on tv:text-scroll-window). 220
(on tv:menu) 183
:left-margin-size
(on windows) 129
:line-area-mouse-documentation
(on tv:line-area-text-scroll-mixin) 227
:line-out
(on windows) 69
:list-tyi
(on tv:stream-mixin) 53
:listen
(on tv:stream-mixin) 53
:magnification
(on tv:magnifying-blinker). 110
:make-complete
(on tv:basic-typeout-window) 217
:make-incomplete
(on tv:basic-typeout-window) 218
:margins

(on windows) 129
:menu-draw
(on tv:menu) 184
:menu-margin-choices
(on tv:menu-margin-choice-mixin) 190
:minimum-width
(on tv:menu) 181
:more-exception
(on windows) 72
:more-p
(on windows) 80
(on tv:basic-typeout-window) 215
:more-vpos
(on windows) 72
:mouse-buttons
(on windows) 120
:mouse-buttons-on-item
(on tv:menu) 184
:mouse-buttons-scroll
(on tv:basic-scroll-bar) 126
:mouse-click
(on windows) 120
:mouse-moves
(on windows) 119
:mouse-or-kbd-tyi
(on tv:stream-mixin) 53
:mouse-or-kbd-tyi-no-hang
(on tv:stream-mixin) 53
:mouse-select
(on windows) 33
:mouse-sensitive-item
(on tv:basic-mouse-sensitive-items) 210
:mouse-standard-blinker
(on windows) 122
:move-near-window
(on tv:menu) 183
:name
(on windows) 132
:name-for-selection
(on windows) 35
:new-scroll-position
(on windows) 124
:notice
(on windows) 158
:number-of-item
(on tv:text-scroll-window) 220
:number-of-items
(on tv:text-scroll-window) 220
:offsets
(on tv:mouse-blinker-mixin) 122
:open-streams
(on tv:who-line-file-sheet) 164
:order-inferiors
(on windows and screens) 28
:output-hold-exception
(on windows) 71
:package
(on tv:listener-mixin-internal) 159
:pane-name
(on tv:basic-constraint-frame) 153
:pane-size
(on windows) 152
:pane-types-alist
(on frames) 155
:parse-font-name
(on tv:screen) 86
:parse-font-specifier
(on tv:screen) 86
:phase
(on tv:blinker) 107
:playback
(on tv:stream-mixin) 54
:point
(on tv:graphics-mixin) 94
:position
(on windows) 45
:preemptable-read
(on tv:preemptable-read-any-tyi-mixin) 55
:primitive-item
(on tv:basic-mouse-sensitive-items) 210
:primitive-item-outside
(on tv:basic-mouse-sensitive-items) 210
:print-function
(on tv:function-text-scroll-window) 221
:print-function-arg
(on tv:function-text-scroll-window) 221
:print-item
(on tv:text-scroll-window) 219
:print-notification
(on windows) 157
:print-notification-on-self
(on tv:notification-mixin) 157
:priority
(on windows) 29
:process
(on tv:select-mixin) 42
(on tv:process-mixin) 41
:processes
(on windows) 41
:put-item-in-window
(on tv:text-scroll-window) 220
:put-last-item-in-window
(on tv:text-scroll-window) 220
:read-cursorpos
(on windows) 74
(on tv:blinker) 105
:redefine-configuration
(on tv:basic-constraint-frame) 154
:redefine-margins
(on windows) 140
:redisplay
(on tv:text-scroll-window) 221
(on tv:basic-scroll-window) 233
:redisplay-selected-items

(on tv:basic-scroll-window) 233
:redisplay-variable
(on tv:choose-variable-values-window) 203
:refresh
(on windows) 16
:refresh-margins
(on windows) 139
:refresh-rubout-handler
(on tv:stream-mixin) 55
:remove-asynchronous-character
(on tv:stream-mixin) 62
:remove-highlighted-item
(on tv:menu-highlighting-mixin) 189
:remove-highlighted-value
(on tv:menu-highlighting-mixin) 189
:remove-selection-substitute
(on windows) 38
:restore-rubout-handler-buffer
(on tv:stream-mixin) 55
:reverse-video-p
(on windows) 81
:right-margin-size
(on windows) 129
:rubout-handler
(on tv:stream-mixin) 54
:save-bits
(on windows) 16
:save-rubout-handler-buffer
(on tv:stream-mixin) 55
:screen-array
(on windows and screens) 21
:screen-manage
(on windows and screens) 26
:screen-manage-autoexpose-inferiors
(on windows and screens) 27
:screen-manage-deexposed-visibility
(on windows) 27
:scroll-bar-always-displayed
(on tv:basic-scroll-bar) 126
:scroll-more-above
(on tv:basic-scroll-bar) 126
:scroll-more-below
(on tv:basic-scroll-bar) 126
:scroll-position
(on scrolling windows) 124
:scroll-redisplay
(on tv:text-scroll-window) 221
:scroll-relative
(on tv:basic-scroll-bar) 126
:scroll-to
(on scrolling windows) 124
:select
(on windows) 32
:select-pane
(on tv:basic-frame) 155
:selectable-windows
(on windows) 35
:selected-p
(on windows) 33
:selected-pane
(on tv:basic-frame) 155
:selection-substitute
(on windows) 38
:self-or-substitute-selected-p
(on windows) 38
:send-all-exposed-panes
(on tv:basic-constraint-frame) 153
:send-all-panes
(on tv:basic-constraint-frame) 153
:send-pane
(on tv:basic-constraint-frame) 153
:sensitive-item-types
(on tv:mouse-sensitive-text-scroll-window) 226
:set-array
(on tv:bitblt-blinker) 110
:set-border-margin-width
(on tv:borders-mixin) 131
:set-borders
(on tv:borders-mixin) 131
:set-character
(on tv:character-blinker) 109
:set-chosen-item
(on tv:menu) 183
:set-column-spec-list
(on tv:dynamic-multicolumn-mixin) 187
:set-configuration
(on tv:basic-constraint-frame) 153
:set-current-font
(on windows) 84
:set-current-item
(on tv:current-item-mixin) 227
:set-cursorpos
(on windows) 74
(on tv:blinker) 105
:set-deexposed-typein-action
(on windows) 32
:set-deexposed-typeout-action
(on windows) 22
:set-default-font
(on tv:menu) 180
:set-deselected-visibility
(on tv:blinker) 106
:set-display-item
(on tv:basic-scroll-window) 232
:set-edges
(on windows) 46
(on tv:menu) 181
:set-fill-p
(on tv:basic-menu) 180
:set-follow-p
(on tv:blinker) 105
:set-font-map
(on windows) 84
:set-geometry

(on tv:menu) 180
:set-half-period
(on tv:blinker) 106
:set-highlighted-items
(on tv:menu-highlighting-mixin) 189
:set-highlighted-values
(on tv:menu-highlighting-mixin) 189
:set-hysteresis
(on tv:hysteretic-window-mixin) 115
:set-inside-size
(on windows) 45
:set-interval
(on editor windows) 160
:set-interval-string
(on editor windows) 160
:set-io-buffer
(on tv:stream-mixin) 52
(on tv:command-menu) 185
:set-item
(on tv:basic-scroll-window) 234
:set-item-generator
(on tv:text-scroll-window) 222
:set-item-list
(on tv:menu) 183
(on tv:margin-multiple-menu-mixin) 188
:set-item-list-pointer
(on tv:dynamic-item-list-mixin) 186
:set-item-type-alist
(on tv:basic-mouse-sensitive-items) 208
:set-items
(on tv:text-scroll-window) 220
:set-label
(on tv:label-mixin) 133
:set-last-item
(on tv:menu) 183
:set-magnification
(on tv:magnifying-blinker) 110
:set-margin-choices
(on tv:margin-choice-mixin) 211
:set-menu-margin-choices
(on tv:menu-margin-choice-mixin) 190
:set-more-p
(on windows) 80
(on tv:basic-typeout-window) 215
:set-mouse-cursorpos
(on windows) 119
:set-mouse-position
(on windows) 119
:set-offsets
(on tv:mouse-blinker-mixin) 122
:set-package
(on tv:listener-mixin-internal) 159
:set-position
(on windows) 46
:set-print-function
(on tv:function-text-scroll-window) 221
:set-print-function-arg
(on tv:function-text-scroll-window) 221
:set-priority
(on windows) 29
:set-process
(on tv:select-mixin) 42
(on tv:process-mixin) 41
:set-region-list
(on tv:margin-region-mixin) 135
:set-reverse-video-p
(on windows) 81
:set-save-bits
(on windows) 16
:set-scroll-bar
(on tv:basic-scroll-bar) 125
:set-scroll-bar-always-displayed
(on tv:basic-scroll-bar) 126
:set-selection-substitute
(on windows) 38
:set-sensitive-item-types
(on tv:mouse-sensitive-text-scroll-window) 226
:set-sheet
(on tv:blinker) 106
:set-size
(on windows) 45
(on tv:rectangular-blinker) 108
(on tv:blinker) 105
(on tv:bitblt-blinker) 109
:set-size-and-cursorpos
(on tv:rectangular-blinker) 108
:set-size-in-characters
(on windows) 75
:set-status
(on windows) 39
:set-superior
(on windows) 12
:set-top-item
(on tv:text-scroll-window) 220
:set-truncation
(on tv:basic-scroll-window) 232
:set-variables
(on tv:choose-variable-values-window) 202
:set-visibility
(on tv:blinker) 105
:set-vsp
(on windows) 80
:setup
(on tv:multiple-choice) 193
(on tv:function-text-scroll-window) 221
(on tv:choose-variable-values-window) 202
:sheet
(on tv:blinker) 106
:size
(on windows) 45
(on tv:blinker) 105
(on tv:bitblt-blinker) 109
:size-in-characters
(on windows) 75

:square-pane-inside-size
(on windows) 152
:square-pane-size
(on windows) 152
:status
(on windows) 39
:string-length
(on windows) 78
:string-out
(on windows) 69
:string-out-centered
(on windows) 69
:string-out-centered-explicit
(on windows) 80
:string-out-explicit
(on windows) 79
:string-out-x-y-centered-explicit
(on windows) 80
:superior
(on windows and screens). 12
:temporary-bit-array
(on windows) 25
:top-item
(on tv:text-scroll-window) 220
:top-margin-size
(on windows) 129
:truncation
(on tv:basic-scroll-window) 232
:turn-off-blinkers-for-typeout
(on tv:essential-window-with-typeout-mixin) 214
:turn-on-blinkers-for-typeout
(on tv:essential-window-with-typeout-mixin) 214
:tyi
(on tv:stream-mixin) 53
:tyi-no-hang
(on tv:stream-mixin) 53
:tyo
(on windows) 68
:tyo-right-margin-character
(on windows) 73
:typeout-window
(on tv:essential-window-with-typeout-mixin). . . . 214
:ultimate-selection-substitute
(on windows) 38
:un-arrest
(on tv:select-mixin) 42
:untyi
(on tv:stream-mixin) 53
:update-item-list
(on dynamic item list menus) 185
:update-label
(on tv:delayed-redisplay-label-mixin) 134
:value-array
(on tv:basic-scroll-window) 233
:visibility
(on tv:blinker). 105
:vsp
(on windows) 80
:wait-for-input-with-timeout
(on tv:stream-mixin) 54
:who-line-documentation-string
(on windows) 120
:width
(on windows) 45

Keyword Index

:absolute
(for tv:mouse-set-blinker-definition) 124
:activate-p
(for windows) 11
:any
(for choose variable values) 195
:array
(for tv:bitblt-blinker) 109
:ask
(for constraint frames) 149
:ask-window
(for constraint frames) 150
:assoc
(for choose variable values) 196
:asynchronous-characters
(for tv:stream-mixin) 61
(for input buffer plist) 59
:backspace-not-overprinting-flag
(for windows) 81
:beginning
(for :deselect) 33
:bindings
(for menu item type) 175
:black
(for constraint frames) 151
:blank
(for constraint frames) 151
:blink
(for blinker visibility) 103
:blinker-deselected-visibility
(for windows) 104
:blinker-flavor
(for windows) 104
:blinker-p
(for windows) 104
:boolean
(for choose variable values) 196
:border-margin-width
(for tv:borders-mixin) 131
:borders
(for tv:borders-mixin) 130
:bottom
(for windows) 43
(for labels) 132
(for borders) 130
:buttons
(for menu item type) 175
:centered
(for labels) 132
:character
(for tv:character-blinker) 109
(for choose variable values) 195
:character-height
(for windows) 44
:character-or-nil
(for choose variable values) 195
:character-width
(for windows) 44
:character-x-offset
(for tv:reverse-character-blinker) 111
:character-y-offset
(for tv:reverse-character-blinker) 111
:choice-box
(for blip type) 203
:choose
(for choose variable values) 196
:clean
(for :expose) 19
:column-spec-list
(for tv:dynamic-multicolumn-mixin) 187
:columns
(for tv:menu) 179
:complete-redisplay
(for tv:sheet-bit-array) 16
:comtab
(for standalone editor windows) 160
:configuration
(for tv:basic-constraint-frame) 153
:constraints
(for tv:constraint-frame) 147
:cr-not-newline-flag
(for windows) 81
:date
(for choose variable values) 195
:date-or-never
(for choose variable values) 195
:deactivated
(for tv:preserve-substitute-status) 39
:deexposed
(for tv:preserve-substitute-status) 39
:deexposed-typein-action
(for windows) 32
:deexposed-typeout-action
(for windows) 22
:default
(for font purpose) 86
(for :deexpose) 20
:default-font
(for tv:menu) 180
:delayed
(for :save-bits) 16
:delete-item
(for item-generator) 223
:deselected-visibility
(for tv:blinker) 106
:display-item
(for tv:basic-scroll-window) 232
:documentation

(for menu item type). 175
(for choose variable values). 196
:dont-save
(for tv:without-screen-management) 33
:dont-upcase-control-characters
(for input buffer plist). 59
:edges
(for windows) 44
:edges-from
(for windows) 44
:end
(for tv:without-screen-management) 33
:error
(for deexposed typeout action). 22
(for :notice) 159
:eval
(for menu item type). 174
(for constraint frames) 151
:even
(for constraint frames) 149
:execute
(for blip type) 160
:expose
(for deexposed typeout action). 22
:expose-p
(for windows) 20
:exposed
(for tv:preserve-substitute-status) 39
:exposed-in-superior
(for tv:preserve-substitute-status) 39
:extra-width
(for tv:choose-variable-values) 197
:fill-p
(for tv:menu) 180
:first
(for tv:without-screen-management) 33
:flashy-scrolling-region
(for tv:flashy-scrolling-mixin). 126
:follow-p
(for tv:blinker) 105
:font
(for tv:character-blinker). 109
(for menu item type). 175
(for labels) 133

tv:font-map
(for windows) 84

:force
(for :deexpose) 20
:full-rubout
(for :rubout-handler). 54
:funcall
(for menu item type). 174
(for constraint frames) 150
:function
(for tv:choose-variable-values) 197
(for tv:basic-choose-variable-values) 201
(for scroll items) 234
(for scroll window entries). 229
:geometry
(for tv:menu) 179
:half-period
(for tv:blinker) 106
:height
(for windows) 43
(for tv:rectangular-blinker) 108
(for tv:ibeam-blinker) 108
(for tv:bitblt-blinker) 109
:highlighted-items
(for tv:menu-highlighting-mixin) 189
:horizontal
(for constraint frames) 152
:hysteresis
(for tv:hysteretic-window-mixin) 115
:if
(for tv:restrict-user-option) 206
:initial-input
(for :rubout-handler) 54
:input
(for :notice). 158
:input-wait
(for :notice). 158
:insert-item
(for item-generator) 223
:inside-height
(for windows) 43
:inside-size
(for windows) 43
:inside-width
(for windows) 43
:integral-p
(for windows) 44
:interval-or-never
(for choose variable values). 195
:io-buffer
(for tv:stream-mixin) 52
(for tv:constraint-frame-with-shared-io-buffer) . . . 143
(for tv:command-menu). 185
(for tv:choose-variable-values-window) 203
:item-list
(for tv:menu) 183
(for tv:basic-multiple-choice) 193
:item-list-pointer
(for tv:dynamic-item-list-mixin) 186
:item-of-number
(for item-generator) 223
:item-type-alist
(for tv:basic-mouse-sensitive-items). 209
:kbd
(for menu item type) 174
:label
(for tv:label-mixin). 132
(for tv:choose-variable-values) 197

(for font purpose) 86
:label-box-p
(for tv:box-label-mixin). 133
:last
(for :deselect). 33
:leader
(for tv:scroll-parse-item) 231
:left
(for windows) 43
(for borders) 130
:limit
(for constraint frames) 148
:line-area
(for blip type). 226
:line-area-width
(for tv:line-area-text-scroll-mixin) 226
:magnification
(for tv:magnifying-blinker). 110
:margin-choice
(for font purpose) 86
:margin-choices
(for tv:margin-choice-mixin). 211
(for tv:choose-variable-values-window) 202
(for tv:choose-variable-values) 197
:margin-scroll-regions
(for tv:margin-scroll-mixin) 127
:margins-only
(for tv:sheet-bit-array) 17
:menu
(for menu item type) 175
(for font purpose) 86
(for blip type). 184
:menu-alist
(for choose variable values) 196
:menu-choose
(for menu item type) 175
:menu-margin-choices
(for tv:menu-margin-choice-mixin) 190
:menu-standout
(for font purpose) 86
:minimum-height
(for windows) 44
:minimum-width
(for windows) 44
:more-p
(for windows) 80
:mouse
(for tv:scroll-parse-item) 231
(for scroll window entries). 230
(for :expose-near). 46
(for :edges-from) 44
:mouse-button
(for blip type). 113
:mouse-click
(for margin region functions). 135
:mouse-enters-region
(for margin region functions). 135
:mouse-item
(for scroll window entries) 230
:mouse-leaves-region
(for margin region functions) 135
:mouse-moves
(for margin region functions) 135
:mouse-self
(for tv:scroll-parse-item) 231
:name
(for windows). 9, 132
:name-font
(for tv:basic-choose-variable-values) 201
:near-mode
(for tv:choose-variable-values). 197
:no-select
(for menu item type) 174
:noop
(for :expose). 19
(for :deexpose). 20
:normal
(for deexposed typeout action) 22
(for deexposed typein action) 32
:notify
(for deexposed typeout action) 22
(for deexposed typein action) 32
:number
(for choose variable values) 195
:number-of-item
(for item-generator). 223
:number-of-items
(for item-generator) 223
:number-or-nil
(for choose variable values) 195
:off
(for blinker visibility) 103
:on
(for blinker visibility) 103
:output
(for :notice) 158
:panes
(for tv:constraint-frame) 146
:pass-through
(for :rubout-handler) 54
:pathname
(for choose variable values) 195
:pathname-list
(for choose variable values) 196
:pathname-or-nil
(for choose variable values) 196
:permit
(for deexposed typeout action) 22
:point
(for :expose-near) 46
:position
(for windows). 43
:pre-process-function
(for scroll items). 234

:princ
(for choose variable values) 195
:print-function
(for tv:function-text-scroll-window) 221
:print-function-arg
(for tv:function-text-scroll-window) 221
:priority
(for windows) 29
:process
(for tv:process-mixin) 41
:prompt
(for :rubout-handler) 54
:raw
(for input buffer plist) 59
:refresh
(for margin region functions) 135
:relative
(for tv:mouse-set-blinker-definition) 124
:reprompt
(for :rubout-handler) 54
:restore
(for :expose) 19
:reverse-video-p
(for windows) 81
(for tv:choose-variable-values) 197
:right
(for windows) 43
(for borders) 130
:right-margin-character-flag
(for windows) 81
:rows
(for tv:menu) 179
:save-bits
(for windows) 16
:scroll-bar
(for tv:basic-scroll-bar) 125
:scroll-bar-always-displayed
(for tv:basic-scroll-bar) 126
:selected
(for tv:preserve-substitute-status) 39
:selected-choice-font
(for tv:basic-choose-variable-values) 202
:selected-pane
(for tv:basic-constraint-frame) 155
:sensitive-item-types
(for tv:mouse-sensitive-text-scroll-window) 226
:sexp
(for choose variable values) 195
:size
(for windows) 43
:size-changed
(for tv:sheet-bit-array) 16
:special-choices
(for tv:margin-multiple-menu-mixin) 189
:stack-group
(for tv:basic-choose-variable-values) 201
:string
(for choose variable values) 195
(for scroll window entries) 229
(for labels) 132
:string-font
(for tv:basic-choose-variable-values) 202
:string-list
(for choose variable values) 195
:superior
(for windows) 12
(for tv:choose-variable-values) 197
:symeval
(for scroll window entries) 229
:tab-nchars
(for windows) 81
:top
(for windows) 43
(for labels) 132
(for borders) 130
:truncate-line-out-flag
(for tv:line-truncating-mixin) 73
:truncation
(for tv:basic-scroll-window) 232
:typeout-execute
(for blip type) 208
:typeout-window
(for tv:essential-window-with-typeout-mixin) 214
:unless
(for tv:restrict-user-option) 206
:unselected-choice-font
(for tv:basic-choose-variable-values) 202
:use-old-bits
(for tv:sheet-bit-array) 16
:value
(for menu item type) 174
(for scroll window entries) 230
:value-array
(for tv:basic-scroll-window) 233
:value-font
(for tv:basic-choose-variable-values) 201
:variable-choice
(for blip type) 203
:variables
(for tv:basic-choose-variable-values) 201
:vertical
(for constraint frames) 152
:visibility
(for tv:blinker) 106
:vsp
(for windows) 80
(for labels) 133
:white
(for constraint frames) 151
:who-line-documentation-string
(for margin region functions) 135
:width
(for windows) 43
(for tv:rectangular-blinker) 108

(for tv:choose-variable-values) 197
(for tv:bitblt-blinker) 109
:window
(for :expose-near) 46
:window-op
(for menu item type) 175
:x
(for windows) 43
:x-pos
(for tv:blinker) 105
:y
(for windows) 43
:y-pos
(for tv:blinker) 105

Flavor and Resource Index

tv:abstract-dynamic-item-list-mixin 185
tv:alias-for-inferiors-mixin 36
tv:autoexposing-more-mixin 73

tv:basic-choose-variable-values 200
tv:basic-frame 141
tv:basic-menu 181
tv:basic-momentary-menu 182
tv:basic-mouse-sensitive-items 207
tv:basic-multiple-choice 192
tv:basic-scroll-bar 125
tv:basic-scroll-window 232
tv:basic-typeout-window 212
tv:bitblt-blinker 109
tv:blinker 103
tv:bordered-constraint-frame 143
tv:bordered-constraint-frame-with-shared-io-buffer . 143
tv:borders-mixin 130
tv:box-blinker 108
tv:box-label-mixin 133

tv:centered-label-mixin 134
tv:character-blinker 109
tv:choose-variable-values-pane 200
tv:choose-variable-values-window 200
tv:command-menu 185
tv:command-menu-abort-on-deexpose-mixin 185
tv:command-menu-mixin 184
tv:constraint-frame 142
tv:constraint-frame-with-shared-io-buffer 143
tv:current-item-mixin 227

tv:delay-notification-mixin 157
tv:delayed-redisplay-label-mixin 134
tv:displayed-items-text-scroll-window 227
tv:dynamic-item-list-mixin 186
tv:dynamic-momentary-menu 186
tv:dynamic-momentary-window-hacking-menu . . . 186
tv:dynamic-multicolumn-mixin 186
tv:dynamic-multicolumn-momentary-menu 187
tv:dynamic-multicolumn-momentary-window-hacking-menu 187
tv:dynamic-temporary-abort-on-deexpose-command-menu 186
tv:dynamic-temporary-command-menu 186
tv:dynamic-temporary-menu 186

zwei:editor-top-level 161
tv:essential-scroll-mouse-mixin 238
tv:essential-window-with-typeout-mixin 214

tv:flashy-margin-scrolling-mixin 127
tv:flashy-scrolling-mixin 126
tv:frame-forwarding-mixin 154
tv:full-screen-hack-mixin 131
tv:function-text-scroll-window 221

tv:graphics-mixin 93
tv:gray-deexposed-right-mixin 27
tv:gray-deexposed-wrong-mixin 27

tv:hollow-rectangular-blinker 108
tv:hysteretic-window-mixin 115

tv:ibeam-blinker 108
tv:inferiors-not-in-select-menu-mixin 35
tv:initially-invisible-mixin 27
tv:inspect-frame 162
tv:inspect-frame-resource 162
tv:interaction-pane 156
tv:intrinsic-no-more-mixin 215

tv:kbd-mouse-buttons-mixin 113

tv:label-mixin 132
tv:line-area-mouse-sensitive-text-scroll-mixin . . . 227
tv:line-area-text-scroll-mixin 226
tv:line-truncating-mixin 73
tv:lisp-interactor 159
tv:lisp-listener 159
tv:list-mouse-buttons-mixin 113
tv:listener-mixin 159
tv:listener-mixin-internal 159

tv:magnifying-blinker 110
tv:margin-choice-menu 190
tv:margin-choice-mixin 211
tv:margin-multiple-menu-mixin 188
tv:margin-region-mixin 134
tv:margin-scroll-mixin 127
tv:margin-scroll-region-on-and-off-with-scroll-bar-mixin 127
tv:menu 182
tv:menu-execute-mixin 177
tv:menu-highlighting-mixin 189
tv:menu-margin-choice-mixin 189
tv:minimum-window 6
tv:momentary-margin-choice-menu 190
tv:momentary-menu 182
tv:momentary-multiple-menu 188
tv:momentary-window-hacking-menu 182
tv:mouse-blinker-mixin 122
tv:mouse-box-blinker 122
tv:mouse-box-stay-inside-blinker 122
tv:mouse-character-blinker 122
tv:mouse-hollow-rectangular-blinker 122
tv:mouse-rectangular-blinker 122
tv:mouse-sensitive-text-scroll-window 225
tv:mouse-sensitive-text-scroll-window-without-click . 227
tv:multiple-choice 193
tv:multiple-menu 188

tv:no-screen-managing-mixin. 27
tv:not-externally-selectable-mixin 37
tv:notification-mixin. 157

tv:peek-frame 162
tv:pop-up-finger-window 163
zwei:pop-up-standalone-editor-frame 161
tv:pop-up-text-window 162
tv:preemptable-read-any-tyi-mixin 55
tv:process-mixin 40

tv:rectangular-blinker 108
tv:reset-on-output-hold-mixin 42
tv:reverse-character-blinker. 110

tv:screen 13
tv:scroll-mouse-mixin 238
tv:scroll-stuff-on-off-mixin 127
tv:scroll-window 232
tv:scroll-window-with-typeout 233
tv:scroll-window-with-typeout-mixin 233
tv:select-mixin 32
tv:sheet 7
tv:show-partially-visible-mixin 27
zwei:standalone-editor-frame 160
zwei:standalone-editor-window 160
tv:stay-inside-blinker-mixin 108
tv:stream-mixin 49
supdup:supdup. 162
supdup:supdup-windows. 162
supdup:supdup. 162
supdup:supdup-windows. 162

supdup:telnet 162
supdup:telnet-windows 162

supdup:telnet 162
supdup:telnet-windows 162
tv:temporary-choose-variable-values-window 201
tv:temporary-menu. 182
zwei:temporary-mode-line-window-with-borders . . .161
zwei:temporary-mode-line-window-with-borders-resource 161
tv:temporary-multiple-choice-window 193
tv:temporary-window-mixin 25
tv:text-scroll-window. 219
tv:text-scroll-window-empty-gray-hack 222
tv:text-scroll-window-typeout-mixin 222
tv:top-label-mixin 133
tv:truncating-pop-up-text-window 163
tv:truncating-pop-up-text-window-with-reset . . 42, 163
tv:truncating-window 74
tv:typeout-window 212
tv:typeout-window-with-mouse-sensitive-items 212

tv:window 7
tv:window-hacking-menu-mixin 177
tv:window-with-typeout-mixin. 214

zwei:zmacs-frame 160
zwei:editor-top-level 161
zwei:pop-up-standalone-editor-frame 161
zwei:standalone-editor-frame 160
zwei:standalone-editor-window. 160
zwei:temporary-mode-line-window-with-borders . . .161
zwei:temporary-mode-line-window-with-borders-resource 161
zwei:zmacs-frame 160

Variable Index

%%kbd-char 49
%%kbd-control 49
%%kbd-control-meta 49
%%kbd-hyper 49
%%kbd-meta 49
%%kbd-mouse 49
%%kbd-mouse-button 49
%%kbd-mouse-n-clicks 49
%%kbd-super. 49

tv:**constraint-node** 151
tv:**constraint-remaining-height** 151
tv:**constraint-remaining-width** 151
tv:**constraint-stacking** 151
tv:**constraint-total-height** 151
tv:**constraint-total-width** 151
tv:*enable-typeout-window-borders* 213
tv:*escape-keys* 63
tv:*mouse-incrementing-keystates* 128
tv:*system-keys* 64
tv:*system-menu-this-window-column* 168
tv:*system-menu-windows-column* 168

tv:12%-gray 27

tv:25%-gray 27

tv:33%-gray 27

tv:50%-gray 27

tv:75%-gray 27

tv:all-the-screens 13
tv:alu-and 94
tv:alu-andca 94
tv:alu-ior 93
tv:alu-seta 94
tv:alu-xor 94

tv:cold-load-stream 170
color:color-screen 13, 165

tv:default-screen 13
tv:default-window-types-item-list 168

tv:initial-lisp-listener 159

tv:kbd-global-intercepted-characters 63
tv:kbd-intercepted-characters 60
tv:kbd-last-activity-time 50
tv:kbd-standard-intercepted-characters 60
tv:kbd-tyi-hook 61

tv:last-who-line-process 163

tv:main-screen 13
tv:more-processing-global-enable 72
tv:mouse-blinker 121
tv:mouse-bounce-time 128
tv:mouse-double-click-time 128
tv:mouse-last-buttons 116
tv:mouse-sheet 112
tv:mouse-speed 116
tv:mouse-window. 115
tv:mouse-x 112
tv:mouse-y 112

tv:previously-selected-windows 36

tv:screen-manage-update-permitted-windows 28
tv:scroll-item-leader-offset. 238
tv:selected-window 31
tv:sheet-area. 9

tv:array
(of tv:bitblt-blinker) 110
tv:baseline
(of windows) 85
tv:bit-array
(of windows) 16
tv:bits-per-pixel
(of tv:screen) 15
tv:blinker-list
(of windows and screens). 104
tv:border-margin-width
(of tv:borders-mixin) 131
tv:borders
(of tv:borders-mixin) 131
tv:bottom-margin-size
(of windows) 129
tv:buffer
(of tv:screen) 15
tv:buffer-halfword-array
(of tv:screen) 15
tv:char-aluf
(of windows) 67
tv:char-width
(of windows) 67
tv:character
(of tv:character-blinker). 109
tv:choice-types
(of tv:basic-multiple-choice). 193
tv:choice-value
(of tv:basic-multiple-choice). 193
tv:chosen-item
(of tv:basic-menu) 181
tv:column-spec-list

(of tv:dynamic-multicolumn-mixin) 187
tv:constraints
(of tv:constraint-frame) 147
tv:control-address
(of tv:screen) 15
tv:current-font
(of windows) 84
tv:current-item
(of tv:current-item-mixin) 227
(of tv:basic-menu) 181
tv:cursor-x
(of windows) 66
tv:cursor-y
(of windows) 66
tv:deexposed-typeout-action
(of windows) 22
tv:deselected-visibility
(of tv:blinker) 106
tv:display-item
(of tv:basic-scroll-window) 232
tv:displayed-items
(of tv:displayed-items-text-scroll-window) 227
tv:erase-aluf
(of windows) 67
tv:exposed-inferiors
(of windows and screens). 21
tv:exposed-p
(of windows and screens). 21
tv:font
(of tv:character-blinker) 109
tv:font-map
(of windows) 84
tv:function
(of tv:basic-choose-variable-values) 201
tv:geometry
(of tv:basic-menu) 182
tv:half-period
(of tv:blinker) 106
tv:height
(of windows) 47
(of tv:bitblt-blinker) 110
tv:highlighted-items
(of tv:menu-highlighting-mixin) 189
tv:incomplete-p
(of tv:basic-typeout-window). 217
tv:inferiors
(of windows and screens). 12
tv:io-buffer
(of tv:stream-mixin) 51
(of tv:command-menu) 185
tv:item-generator
(of tv:text-scroll-window) 222
tv:item-list
(of tv:basic-menu) 181
tv:item-list-pointer
(of tv:dynamic-item-list-mixin) 186
tv:item-name
(of tv:basic-multiple-choice) 193
tv:items
(of tv:text-scroll-window). 219
tv:label
(of tv:label-mixin). 133
tv:label-needs-updating
(of tv:delayed-redisplay-label-mixin) 134
tv:last-item
(of tv:basic-menu). 181
tv:left-margin-size
(of windows) 129
tv:line-height
(of windows) 67
tv:lock
(of windows and screens) 24
tv:lock-count
(of windows and screens) 24
tv:magnification
(of tv:magnifying-blinker). 110
tv:margin-choices
(of tv:margin-choice-mixin). 211
tv:more-vpos
(of windows) 72
tv:mouse-blinkers
(of tv:screen) 123
tv:name
(of windows) 9
tv:panes
(of tv:constraint-frame). 147
tv:phase
(of tv:blinker). 107
tv:print-function
(of tv:function-text-scroll-window). 221
tv:print-function-arg
(of tv:function-text-scroll-window). 221
tv:priority
(of windows) 29
tv:process
(of tv:process-mixin). 41
tv:region-list
(of tv:margin-region-mixin). 134
tv:restored-bits-p
(of windows) 17
tv:right-margin-size
(of windows) 129
tv:screen-array
(of windows and screens) 21
tv:selection-substitute
(of windows) 37
tv:sensitive-item-types
(of tv:mouse-sensitive-text-scroll-window). 225
tv:sheet
(of tv:blinker). 106
tv:stack-group
(of tv:basic-choose-variable-values) 201
tv:superior
(of windows and screens) 12

tv:time-until-blink
(of tv:blinker) 106
tv:top-item
(of tv:text-scroll-window) 219
tv:top-margin-size
(of windows) 129
tv:truncation
(of tv:basic-scroll-window) 232
tv:typeout-window
(of tv:essential-window-with-typeout-mixin) 214
tv:value-array
(of tv:basic-scroll-window) 233
tv:visibility
(of tv:blinker) 105
tv:width
(of windows) 47
(of tv:bitblt-blinker) 110
tv:x-offset
(of windows) 47
tv:x-pos
(of tv:blinker) 105
tv:y-offset
(of windows) 47
tv:y-pos
(of tv:blinker) 105

Function Index

sys:%color-transform 102
sys:%draw-char. 101
sys:%draw-line 101
sys:%draw-rectangle. 101
tv:%draw-rectangle-clipped. 101
sys:%draw-triangle 101

tv:add-escape-key 64
tv:add-system-key 64
tv:add-to-system-menu-programs-column 167
tv:add-typeout-item-type 209
tv:await-window-exposure 23

beep 69
bitblt 102
tv:black-on-white 14
color:blt-color-map 165

tv:careful-notify 157
tv:choose-process-in-error 158
choose-user-options. 206
tv:choose-variable-values 196
tv:choose-variable-values-process-message 203
color:clear 166
tv:close-all-servers 164
color:color-draw-char 167
color:color-draw-line 166
color:color-exists-p 165
color:blt-color-map 165
color:clear 166
color:color-draw-char 167
color:color-draw-line 166
color:color-exists-p 165
color:colorate. 166
color:colorize. 166
color:fill-color-map 166
color:make-color-font 92, 167
color:random-color-map 166
color:read-color-map 165
color:rectangle 166
color:spectrum-color-map 166
color:write-color-map 165
color:write-color-map-immediate 165
color:colorate. 166
color:colorize. 166
tv:complement-bow-mode 15

define-user-option-alist 204
defvar-site-alist-user-option 205
defvar-site-user-option 205
defvar-user-option 205
tv:defwindow-resource. 169
tv:delaying-screen-management 30
tv:describe-servers 165

tv:deselect-and-maybe-bury-window 33
tv:draw-char 101
tv:draw-rectangle-inside-clipped 101

color:fill-color-map 166
tv:find-process-in-error. 158
tv:find-window-of-flavor 65
tv:flush-full-screen-borders 132
tv:font-baseline 90
tv:font-blinker-height 91
tv:font-blinker-width. 91
tv:font-char-height. 90
tv:font-char-width 90
tv:font-char-width-table 91
tv:font-chars-exist-table 91
tv:font-evaluate 87
tv:font-indexing-table 91
tv:font-left-kern-table 91
tv:font-name 90
tv:font-raster-height 91
tv:font-raster-width 91
tv:font-rasters-per-word 91
tv:font-words-per-char 91

tv:idle-lisp-listener. 159
tv:io-buffer-clear 58
tv:io-buffer-empty-p. 57
tv:io-buffer-full-p 57
tv:io-buffer-get 57
tv:io-buffer-input-function 56
tv:io-buffer-input-pointer. 56
tv:io-buffer-last-input-process 57
tv:io-buffer-last-output-process 57
tv:io-buffer-output-function. 56
tv:io-buffer-output-pointer 56
tv:io-buffer-plist. 57
tv:io-buffer-push 57
tv:io-buffer-put 57
tv:io-buffer-record. 57
tv:io-buffer-record-pointer 57
tv:io-buffer-size 56
tv:io-buffer-state 56
tv:io-buffer-unget 57

tv:kbd-asynchronous-intercept-character 62
kbd-char-available 56
tv:kbd-char-typed-p 59
tv:kbd-default-output-function. 58
tv:kbd-intercept-abort 60
tv:kbd-intercept-abort-all 60
tv:kbd-intercept-break 60
tv:kbd-intercept-error-break. 60
tv:kbd-io-buffer-get 58
tv:kbd-snarf-input 59

kbd-tyi 55
kbd-tyi-no-hang 56
tv:kbd-wait-for-input-or-deexposure. 59
tv:kbd-wait-for-input-with-timeout 59
tv:key-state 65

tv:label-bottom 133
tv:label-centered 133
tv:label-font 133
tv:label-left 133
tv:label-right 133
tv:label-string 133
tv:label-top 133
tv:label-vsp 133
tv:lock-sheet 24

tv:make-blinker. 104
color:make-color-font 92, 167
tv:make-default-io-buffer 58
make-instance 9
tv:make-io-buffer. 57
tv:make-sheet-bit-array 102
tv:make-window 9
tv:map-over-exposed-sheet. 12
tv:map-over-exposed-sheets 12
tv:map-over-sheet. 12
tv:map-over-sheets 12
tv:margin-region-area 136
tv:margin-region-bottom 135
tv:margin-region-function 134
tv:margin-region-left 135
tv:margin-region-margin 134
tv:margin-region-right 135
tv:margin-region-size 135
tv:margin-region-top 135
tv:menu-choose. 177
tv:menu-compute-geometry. 181
tv:menu-item-string 177
tv:merge-shift-keys 116
tv:mouse-button-encode 116
tv:mouse-buttons 118
tv:mouse-call-system-menu. 121
tv:mouse-default-handler. 119
tv:mouse-define-blinker-type 123
tv:mouse-discard-clickahead 128
tv:mouse-get-blinker 123
tv:mouse-input 118
tv:mouse-select 121
tv:mouse-set-blinker. 121
tv:mouse-set-blinker-cursorpos 120
tv:mouse-set-blinker-definition 123
tv:mouse-set-sheet 112
tv:mouse-set-sheet-then-call 112
tv:mouse-set-window-position. 117
tv:mouse-set-window-size 117
tv:mouse-specify-rectangle 117
tv:mouse-standard-blinker 122
tv:mouse-wait 116
tv:mouse-wakeup. 115
tv:mouse-warp. 113
tv:mouse-y-or-n-p. 178
tv:multiple-choose 191
tv:multiple-menu-choose 187

tv:notify 157

tv:open-blinker 107

zwei:pop-up-edstring 161
tv:prepare-sheet 99
tv:preserve-substitute-status 38
tv:print-notifications 158
tv:process-typeahead 58
tv:prune-user-option-alist 206

color:random-color-map. 166
color:read-color-map 165
zwei:read-defaulted-pathname-near-window 162
color:rectangle 166
tv:remove-escape-key 64
tv:remove-system-key. 65
reset-user-options 206
tv:restrict-user-option 206

tv:scroll-interpret-entry 231
tv:scroll-item-component-items. 237
tv:scroll-item-line-sensitivity 238
tv:scroll-item-mouse-items. 238
tv:scroll-item-plist 237
tv:scroll-item-size 237
tv:scroll-maintain-list 235
tv:scroll-maintain-list-unordered 237
tv:scroll-maintain-list-update-states 237
tv:scroll-parse-item 230
tv:scroll-string-item-with-embedded-newlines. . . . 231
tv:select-or-create-window-of-flavor 65
tv:set-default-font 87
tv:set-screen-standard-font. 87
tv:set-standard-font. 87
tv:set-tv-speed. 14
tv:sheet-backspace-not-overprinting-flag 81
tv:sheet-baseline 85
tv:sheet-bit-array. 16
tv:sheet-blinker-list. 104
tv:sheet-bottom-margin-size 129
tv:sheet-bounds-within-sheet-p. 48
tv:sheet-calculate-offsets. 48
tv:sheet-can-get-lock 24
tv:sheet-char-aluf 67
tv:sheet-char-width. 67
tv:sheet-clear-locks 24
tv:sheet-contains-sheet-point-p. 48
tv:sheet-cr-not-newline-flag 81
tv:sheet-current-font 85

tv:sheet-cursor-x 67
tv:sheet-cursor-y 67
tv:sheet-deexposed-typeout-action. 22
tv:sheet-end-page-flag 71
tv:sheet-erase-aluf 67
tv:sheet-exposed-inferiors 21
tv:sheet-exposed-p 21
tv:sheet-following-blinker 107
tv:sheet-font-map 85
tv:sheet-force-access 23
tv:sheet-get-screen 13
tv:sheet-height 47
tv:sheet-inferiors 12
tv:sheet-inside-bottom. 130
tv:sheet-inside-height 47
tv:sheet-inside-left 130
tv:sheet-inside-right. 130
tv:sheet-inside-top 130
tv:sheet-inside-width 47
tv:sheet-left-margin-size. 129
tv:sheet-line-height. 67
tv:sheet-line-out 70
tv:sheet-lock. 24
tv:sheet-me-or-my-kid-p 12
tv:sheet-more-flag 71
tv:sheet-more-handler. 72
tv:sheet-more-vpos 72
tv:sheet-number-of-inside-lines. 48
tv:sheet-output-hold-flag 21, 71
tv:sheet-overlaps-edges-p 48
tv:sheet-overlaps-p 48
tv:sheet-overlaps-sheet-p 48
tv:sheet-right-margin-character-flag 81
tv:sheet-right-margin-size 129
tv:sheet-screen-array 21
tv:sheet-superior 12
tv:sheet-tab-nchars 81
tv:sheet-tab-width 82
tv:sheet-top-margin-size. 129
tv:sheet-truncate-line-out-flag 73
tv:sheet-width 47
tv:sheet-within-p 48
tv:sheet-within-sheet-p 48
tv:sheet-x-offset 47
tv:sheet-y-offset 47
color:spectrum-color-map 166
tv:spline 98
sys:%color-transform 102
sys:%draw-char 101
sys:%draw-line 101
sys:%draw-rectangle 101
sys:%draw-triangle 101

tv:turn-off-sheet-blinkers 107
zwei:typein-line-readline-near-window 161

tv:white-on-black 15
tv:who-line-clobbered 164
tv:who-line-documentation 163
tv:who-line-file-state-sheet 164
tv:window-call 34
tv:window-mouse-call 34
tv:window-owning-mouse 115
tv:window-under-mouse 117
tv:with-blinker-ready. 107
tv:with-mouse-grabbed 115
tv:with-mouse-usurped 118
tv:with-selection-substitute 38
tv:with-sheet-deexposed 21
tv:without-screen-management 30
color:write-color-map 165
color:write-color-map-immediate. 165
write-user-options 206

zwei:pop-up-edstring. 161
zwei:read-defaulted-pathname-near-window 162
zwei:typein-line-readline-near-window 161