Introduction to Using the Window System

Daniel Weinreb
David A. Moon

This document is a draft copy of a portion of the Lisp Machine window system manual. It is being published in this form now to make it available, since the complete window system manual is unlikely to be finished in the near future. The information in this document is accurate as of system 67, but is not guaranteed to remain 100% accurate. To understand some portions of this document may depend on background information which is not contained in any published documentation.

This paper is a portion of a document which will explain how a programmer may make use of and extend the facilities in the Lisp Machine known collectively as the Window System.

Printed by Symbolics, Inc. under license from the Massachusetts Institute of Technology.

Table of Contents

1. Concepts

1.1 Purpose of this Document

This document 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. This document does not explain how to extend the window system by defining your own flavors; that will require a further and more extensive document, of which this document will be a prerequisite.

It is assumed that you have a working familiarity with Lisp Machine Lisp, 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. Furthermore, you must understand something about flavors. While you need not be familiar with how methods are defined and combined, 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. (Chapter 20 of the Lisp Machine Manual provides more than enough information on flavors.)

1.2 Purpose of the Window System

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. (At MIT, all 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 co-exist, and even run simultaneously, without getting in each other's way, sharing the screens according to a set of established rules.

The current implementation of the window system is usually called the "New Window System" (or "NWS" for short; at MIT, bugs in it should be reported to BUG-NWS at MIT-AI). The NWS implementation is built on flavors. The NWS replaces a previous implementation which was built on classes, which, in turn, replaced the previous version (the one documented in the preliminary editions of the Lisp Machine Manual) which didn't use message passing at all. The NWS was designed and implemented primarily by Howard Cannon and Mike McMahon during 1980.

1.3 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 (you can even have many of each of these) 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 blinker (the thing on the screen that tracks the mouse) is in a region associated with a certain program, this can be interpreted as meaning that the mouse is talking to that program. Allowing many programs to share the input and output 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 commands, 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.

Sometimes windows overlap; two windows may occupy some of the same space. While the Split Screen command will never do this, you can make it happen by creating two windows and simply placing them so that they partially overlap, by using Edit Screen. 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. If both of the windows were allowed to use it, then they would get in each other's way. Of these two windows, only one can be *visible* at a time; the other one has to be not fully visible, but either partially visible or not visible at all. Only the visible window has an area of the screen to use.

If you play around with this, you will see that it looks as if one window is on top of the other, as if they were two overlapping pieces of paper on a desk and one were on top. Create two Lisp Listeners using the Create command of the System Menu or the Edit Screen 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. We will return to the concepts of visible and not-fully-visible windows later in more detail.

From the point of view of the Lisp world, each window is a Lisp object. A window is an instance of some flavor of window. There are many different window flavors available; some of them are described in this document.

Windows can function as streams by accepting all the messages 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 21.5.4 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.

Windows have internal state, contained in instance variables, that indicate which screen the window is on, where on the screen it is, where its cursor is, what blinkers it has, how it fits into the window hierarchy, and much more. You can get windows to do things by sending them messages; they accept a wide variety of messages, telling them to do such things as changing their position and size, writing characters and graphics, changing their labels and borders, changing status in various ways, redrawing themselves, and much more. The main business of this document is to explain the meaning of the internal state of windows, and to explain what messages you can send and what those messages do.

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.

1.4 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 error handler and Zmail also use elaborate windows with panes. These panes are not exactly the same as the other windows we have discussed, because instead of serving to split up the screen, they serve to split up the program's window itself. Sometimes you don't see this, because often the program's window is taking up the whole screen itself. Try going into the Edit Screen system and reshaping a whole Inspector or Zmail window. You will see that the panes serve to divide this window up into smaller areas.

In fact, the same window system functionality is used to split up a paned window into panes as is used to split up a screen into windows. Each pane is, in fact, a window in its own right. 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).

The position of a window is remembered in terms of its relative position with respect to the its superior; that is, what we remember about each window is where it is within 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.

One effect of the hierarchical arrangement is that you can use Edit Screen to edit the configuration of panes in a frame as well as to edit the configuration of windows on the screen, by clicking right on Edit Screen. If you have ever clicked right on Edit Screen while the mouse was on top of a window with inferiors, such as an editor, you will have noticed that you get a

menu asking which of these two things you want to do. In fact, that menu can have more than two items; the number of items grows as the height of the hierarchy.

So, what Edit Screen really does is to manipulate a set of inferiors of some specific superior window, 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*. Windows can be activated and deactivated. 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 uncovered. 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 brought to the top. 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.

Each superior window keeps track of all of its active inferiors, and each inferior window keeps track of its superior, in internal state variables. 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.

1.5 Pixels and Bit-Save Arrays

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. On regular black-and-white screens, each pixel can have only two values: lit up, and not lit up. The way the display of pixels is produced is that inside the Lisp Machine, there is a special memory associated with each physical screen that has some number of bits assigned to each pixel of the screen; those bits say, for each pixel, what brightness and color it should display. For regular black-and-white screens, since a pixel can have only two values, only a single bit is stored for each pixel. If the bit is a one, the pixel is not lit up; if it is a zero, the pixel is lit up. (Actually, this sense can be inverted if you want; see <not-yet-written>.) Everything you see on the screen, including borders, graphics, characters, and blinkers, is made up out of pixels.

When a window is fully visible, its *contents* are displayed on a screen so that they can be seen. What happens to the contents when the window ceases to be fully visible? There are two possibilities. A window may have a *bit-save array*. A bit-save array is a Lisp array in which the contents of the window can be saved away when the window loses its visibility; if a window has a bit-save array, then the window system will copy its contents out of the screen and into the bit-save array when the window ceases to be fully visible. If the window does not have a bit-save array, then there is no place to put the bits, and they are lost. When the window becomes visible again, if there is a bit-save array, the window system will copy the contents out of the bit-save array and back onto the screen. If there is no bit-save array, the window will try to redraw its contents; that is, to regenerate the contents from some state information in the window. Some windows can do this; for example, editor windows can regenerate their contents by looking at the editor buffer they are displaying. Lisp listener windows cannot regenerate their contents, since they do not remember what has been typed on them. In lieu of regenerating their contents, such

windows just leave their contents blank, except for the decorations in the margins of the window (see below), which they are able to 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 uses up storage in the Lisp world, and since it can be pretty big, it may need to be paged in from the disk in order to be referenced (depending on how hard the virtual memory system is being strained). If the paging overhead for the bit-save array is very high, it might have been faster not to have one in the first place (although the system goes through some special trouble to try to keep the bit-array out of main memory when it is not being used).

The other important use of bit-save arrays is for windows that have inferiors. If the superior window is not visible, the inferiors can use the bit-save array of the superior as if it were a screen, and they can draw on it and become exposed on it. This tricky concept will be examined in much greater detail shortly.

An additional benefit of having a bit-save array is that the screen manager can do useful things for partially-visible windows when those windows have bit-save arrays; at certain times it can copy some of the pixels from the bit-save array onto the part of the screen in which the window is partially visible, so that when a window is only partially visible, you can see whatever part is visible. We will discuss this more later.

1.6 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 this interacts with the desire of a program or of the user to have some windows visible and other windows not visible at a particular time. These are complex concepts, which you don't have to understand completely to make use of the window system. You probably *do* need to understand these ideas thoroughly only if you plan to make advanced use of the window system, such as creating your own frame or customizing very basic aspects of the system's behavior. Therefore, if this is your first time through this document, read this section over briefly but don't worry if it doesn't completely make sense; you can come back to it later.

The following discussion attempts to explain what it means for a window to be *exposed*. It will be necessary for us to refer to the concept of a window being exposed before we explain exactly what that means. For the time being, the approximate meaning of "exposed" is that a window is exposed if it has somewhere for its typeout to go. A window that is fully visible on a screen is exposed, because its typeout can go on the screen. A window might be exposed even if it is not fully visible, because its typeout might be able to go to a bit-save array somewhere.

Each window has in it a set of all those inferiors that are "ready to be exposed". This set is a subset of the set of active inferiors, discussed above. When you send a window an :expose message, it becomes "ready to be exposed" and is added to the set; when you send a window a :deexpose message, it ceases being ready to be exposed and is removed from the set. These are the only ways anything ever gets into or out of the set. The meaning of "ready" to be exposed

will be cleared up soon; for the time being, we will just say that either all the windows on that list are, in fact, exposed, or else none of them are exposed but they are all still "ready" to become exposed.

Each window has an internal state variable called its *screen-array*. The value of the screen-array variable is where output to the window should go; if a program draws a character "on a window" or draws a triangle "on a window", that means it is changing the values of pixels in the window's screen-array. The value of the screen-array variable is used in figuring out whether a window is exposed.

The screen-array of a screen (remember, a screen is a window itself) is the special memory that gets displayed on the physical screen. For any other window, if the window is exposed, then its screen-array is an indirect array that points into a section of the superior's screen-array; namely, it points into the area of the superior's screen-array where the inferior gets displayed on the superior. For example, consider a window whose superior is a screen, which is exposed, and whose upper-left-hand corner is at location (100,100) in the screen. Then 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.

What happens to the screen-array variable if the window is not exposed? That depends on whether the window has a bit-save array or not. If there is a bit-save array, then the screen-array becomes the bit-save array. If there is no bit-save array, the screen-array becomes nil.

The most important thing to understand about the value of screen-array is that it is defined recursively, in terms of the superior's screen-array. Consider a window which is exposed, and all of whose ancestors are exposed: the superior is exposed, the superior's superior is exposed, and so on all the way back to the screen. Then each window has a screen-array that points into the middle of its superior's screen-array, all the way up the hierarchy, through the window whose screen-array points into the middle of the screen. When typeout is done on the window, it will appear on the screen, offset by the combined offsets of all the superiors, so that it will appear in the correct absolute position on the screen.

Now, suppose one of those ancestors becomes deexposed. There are two alternative things that might happen. First, consider the case in which that ancestor (the one that got deexposed) has a bit-save array. That ancestor's screen-array will no longer point to its own superior; its screen-array will be its bit-save array. That means that our window's screen-array will be pointing, perhaps through several levels of indirection, into that ancestor's bit-save array. The ancestor window is not exposed, but our window *is* still exposed. If typeout is done on our window, it will appear on the bit-save array of the ancestor. This won't actually be visible to the user, since it is only a bit-save array and not an actual screen, but the typeout can proceed and the bits can be drawn into the bit-save array. Later, if and when the ancestor is exposed again, the window system will copy the bit-save array onto the screen, and the drawing that had been done will become visible.

There is another case: suppose the ancestor is deexposed, and it does not have a bit-save array. Then the ancestor's screen-array becomes nil. Well, now we have a problem. The ancestor's inferior is exposed, and so its screen-array is supposed to point into the screen-array of its superior. However, there is no way to point into the middle of a nil. There just isn't

anywhere for the screen-array to point to; the window doesn't have anywhere to type out. Since it has nowhere to type out, it gets deexposed too. In general: when a window is deexposed, and it has no bit-save array, all of its inferiors that are ready to be exposed (all of which are, in fact, exposed) become deexposed. They continue to be "ready to be exposed", though.

In fact, this is the distinction between "ready to be exposed" and actually being exposed. The rule is: a window is exposed when and only when it is "ready to be exposed" *and* its superior has a screen-array. That is what "exposed" means.

When a window is sent an :expose message, it always becomes "ready to be exposed". If the superior has a screen-array, then it immediately becomes exposed. If the superior does not have a screen array, then the window just stays "ready", and when the window's superior finally gets its screen array, the window itself is exposed. If a window is "ready to be exposed" but is not exposed yet, then it is waiting for its superior to acquire a screen-array; when the superior gets one, the window becomes exposed. The usual way that the superior gets a screen array is for it to get exposed itself; when this happens, the inferiors that are "ready to be exposed" will all get exposed.

Also, if the superior has no screen-array then obviously it has no bit-save array; it can be given one by the :set-save-bits message, which can change a window that doesn't have a bit-save array into a window that does have a bit-save array. You can dynamically change which windows have and don't have bit-save arrays, and windows that are affected will be exposed and deexposed accordingly. This is much less common, though; usually whether a window has a bit-save array or not is specified when the window is created, and it doesn't change.

So, the important point is that when a window is sent an :expose message, it may not become exposed then and there. If the superior has a screen-array, then the window will be exposed immediately. But if the superior does not have a screen array, then making the window exposed is delayed until the superior acquires a screen array. When the superior gets its screen array, then the window itself becomes exposed. So what the :expose always does is to add the window to the set of windows that are "ready to be exposed"; a window is exposed precisely when it is "ready to be exposed" and the window's superior has a screen-array. The :deexpose message always removes a window from the set of windows "ready to be exposed", and therefore is always stops the window from being exposed.

Note well that "exposed" does not mean "visible". A window can be exposed by virtue of being able to type out on a bit-save array, and not be visible at all. A window is fully visible if and only if all its ancestors are exposed, and the top level ancestor is a screen.

(A detail: if a window is top-level (if it has no superior) then it is as if "its superior has a screen array"; sending a top-level window an :expose message always exposes it immediately. You usually don't deexpose top-level windows anyway.)

(Another detail: 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 all its inferiors). 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.)

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 9).

1.7 Ability to Output

The main reason for worrying about whether a window is exposed or not is in order to figure out whether it should be allowed to type out. If a window is not exposed, either its superior has no screen-array (so there is no place for its output to go), or it is not ready to be exposed at all (so it is supposed to be hidden). Normally, when a process tries to do output to a window that is not exposed, by sending stream messages (such as :tyo and :string-out), the process waits in a state called Output Hold; the process continues to wait until the window becomes exposed again, at which time it proceeds with its typeout. The term "typeout" refers not only to character output, but to any form of modification of the window's contents, including drawing of graphics.

This is the normal case that you run into most of the time. However, there are some exceptions to this rule. You don't have to worry about the exceptions the first time you read this document; you may skip over the remainder of this section and the following section.

A process trying to output to a window does not actually decide to wait in the Output Hold state based on whether or not the window is exposed. There is actually a flag in each window, called the "output hold flag", that is really being checked to see whether output can go ahead. The "output hold flag" is cleared when the window is exposed and set when the window is deexposed, and output is held when this flag is set. The complexity comes from other things besides exposing that clear this flag.

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; it indicates an action that should be taken when there is an attempt to type out to a deexposed window. After the 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".

By default, the "deexposed typeout action" is :normal, which means that no special action should be taken; hence the process will wait for the window to become exposed.

If the "deexposed typeout action" is :expose, however, then the action will be 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 be cleared and typeout will be able to proceed immediately. If the superior is the screen, the :expose option provides a very different user interface from the :normal option.

If the "deexposed typeout action" is :permit, that means that typeout should be permitted 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.

The "deexposed typeout action" may also be :notify, which 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 <not-yet-written>). The default response to this is to notify the user that the window wants to type out and to make the window "interesting" (see <not-yet-written>) so that Terminal-0-S can select it. supdup and telnet windows have :notify deexposed typeout action by default.

Another permissible value is :error, which means that an error should be signalled.

If the "deexposed typeout action" is not any of these keywords, then it should be a list; the action will be to send the message specified by the first element of the list to the window, passing the rest of the elements of the list as arguments.

There is another exception to the rule that you can only type out on exposed windows: the 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. Note that the screen array must be this window's bit-save array (since the window is not exposed). What tv:sheet-force-access does is to temporarily turn off the "output hold flag" while executing its body. This is useful for drawing things on a window while the window is not visible on the screen. It is better to do it this way than to use a "deexposed typeout action" of :permit, in most cases, since the effect of tv:sheet-force-access is local to the program, while the deexposed typeout action affects anything that types out on the window. If the window does not have a screen-array, tv:sheet-force-access doesn't do anything at all; it just returns *without* evaluating its body.

Another way that typeout can be held up is if the window is *locked*. Locking is independent of the "output hold flag" and is not affected by the deexposed typeout action nor by tv:sheet-force-access. There are two ways that a window can be locked. The normal form of locking is a mutual exclusion that guarantees that only one process at a time operates on the window's contents and attributes. If one process is working on the window and another tries to do so, the second process will wait until the first one is finished. In the absence of program bugs, this wait is for a very short time and should not be noticeable.

The other form of locking is called "temp-locking". If a window is temp-locked, then any attempt to type out on it will wait, regardless of everything else. Temp-locking has to do with temporary windows, which are explained in the next section.

1.8 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 out of *temporary windows*. In general, when you create a window, you can specify that you want it to be a temporary window. 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.

There would be some problems if temporary windows were this simple. Suppose there is a normal window, and a temporary window has appeared over it; some of the contents of the normal window are being saved in an array inside the temporary window. Now, if the normal window is moved somewhere else, and possibly becomes deexposed or is overlapped by other windows or something, and then the temporary window is deexposed, the temporary window will dump back its saved bits where the normal window used to be, even though the normal window isn't there any more, and so some innocent bystander will be clobbered. Furthermore, suppose typeout were done on the normal window; we have not deexposed it, so nothing would prevent the typeout from overwriting the temporary window, nor prevent the typeout from being overwritten in return when the temporary window is deexposed. 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) are *temp-locked*. When 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.

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. Temporary windows should only be used when you want the user to see something for a little while and then have the window disappear. The temp-locking is undone when the temporary window is deexposed.

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.

1.9 The Screen Manager

The *screen manager* is a subsystem of the window system that does various background jobs involved with keeping things straight in the window system. It has several responsibilities. One job of the screen manager is to find any window that is active and deexposed, but not covered up by any windows. There is no reason for such a window not to be exposed, so the screen manager exposes it. This is called *autoexposure*.

Another job of the screen manager is to manage those parts of the screen that are not currently part of any exposed window. When you first start using the Lisp Machine, the entire screen is covered by a big Lisp Listener window, and the initially-created windows for Zmacs, Zmail, and so on, are all as large as the entire screen, so this issue does not arise. Similarly, if you use Split Screen to divide the screen up into windows, the windows will use up all of the area of the screen. However, if you use the Create or Edit Screen commands, you can make windows of arbitrary shapes and sizes, and you can leave parts of the screen where there is no exposed window.

When the screen manager sees that there is such an area of the screen, it considers all of the active windows that aren't exposed. If it finds such a window, and that windows has a bit-save array, then the screen manager displays the contents of the bit-save array for the corresponding portion of the screen. This gives the visial impression of overlapping pieces of paper on a desktop; the de-exposed window is partially covered up by the exposed windows, but you can still see those parts that aren't covered.

If there is more than one active de-exposed window that might be displayed in a given area of the screen, then the screen manager uses its priority ordering (discussed below) to decide which one to display.

Usually the screen manager only displays partially visible windows that have bit-save arrays. But if you want to make a window that doesn't have a bit-save array and you want the screen manager to try to display it when it is only partially exposed, use the following mixin:

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. The screen manager cannot display the contents of the window, since there is no bit-save array to hold them, but it does give the window a screen array temporarily, tells it to refresh itself, and then shows whatever the window displays. Often this means that you will see the label and borders of the window, but no contents.

The screen manager does not only manage screens; it can manage any window that has inferiors. Windows with panes are split up into windows just the same way screens are split up into windows, and so the screen manager can do the same thing to panes of paned windows that it does with windows directly on screens. The action of the screen manager on the inferiors of a window is controlled by that window's response to the :screen-manage message; the default is to do screen management in the same way as it is done on a screen.

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

Prevents the screen manager from dealing with the inferiors of a window.

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 might be more than one deexposed window that might be displayed in a given area of the screen. Somehow the screen manager must decide which window to display.

The way it decides is 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.

The screen manager has a somewhat complicated algorithm for keeping track of this ordering. Don't worry if you don't understand it; you don't need to. (If you skip over this part, resume reading where we discuss delaying of screen management; that is important for you to know.) Part of the algorithm involves a value kept for 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. If a window has priority nil, then its priority is less than that of any window with numerical priority; that is, nil acts like the lowest possible number. The default value for priority is nil.

The ordering itself is not based on just the priorities. Instead, the way it works is that the ordering is remembered, and at various times, the windows are resorted according to the following set of rules. First, exposed windows go in front of non-exposed windows. Secondly, if two windows are both exposed or both have the same value of priority, their order is not changed by the sorting. Finally, if two non-exposed windows have different values of priority, then the one with the higher value goes in front of the one with the lower value. So not only the priority values make a difference; the relative positions of windows before the resorting matters too.

The resorting happens whenever some event occurs that might change the ordering. For example, when a window is exposed or deexposed, or when a window's priority changes, the ordering it is on must be resorted. Note that the sort is *stable*; that is, if we don't have any preference for one window over another then they keep their previous ordering. Since most of the time numerical priorities are not used anyway (the priorities of most windows are nil), this is generally what determines the ordering. When a window is exposed, it gets pulled up to the front of the ordering, and then as other windows later get exposed on top of it, it sinks back down. More recently exposed windows will be closer to the front.

There is also an operation called *burying* a window, which first deexposes the window, then moves it to the end of the ordering, and finally (since something interesting has happened) causes the ordering to be resorted. So burying a window essentially makes it be the farthest from the front of the ordering of all windows with the same priority as it. A program usually 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.

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 not become exposed unless sent an explicit :expose message.

(Another minor point: Windows whose area of the screen does not lie within the boundaries of their superior cannot be exposed at all, and so the screen manager does not try to autoexpose such windows. However, they can be partially visible.)

You may have noticed a problem that screen management can cause. Suppose you send a :deexpose message to an exposed window. The window is no longer exposed, but since it is closer to the front of the ordering, and especially if numerical priorities are not being used much, then it may end up being the foremost window in the ordering that occupies its area of the superior, and so autoexposure is likely to expose it again immediately! If you want to do a series of deexposing and exposing operations, they can get messed up this way by the screen manager. In order to prevent this from happening, you can use the tv:delaying-screen-management special form (see page 13) to delay the actions of the screen manager until all of your operations have been done. In simple applications, you should not need to send your own :deexpose messages anyway (most deexposure is done automatically when new windows are exposed), and you should not need tv:delaying-screen-management; explicit deexposure and delaying of screen management is mostly used in advanced applications, and if you use these for something simple then you are probably doing something wrong.

While screen management is delayed, notes to the screen manager telling what areas of the screen have been played with are put on a queue. When the tv:delaying-screen-management form is returned from, all of the entries on the queue are examined, and the screen manager figures out all the things that need to be done and does them all at once. So, by delaying screen management, you prevent the screen manager from seeing various intermediate states and doing unnecessary work, which would consume computation time and make the windows on the screen visibly undergo unnecessary contortions.

When a tv:delaying-screen-management form is exited, normally or abnormally (i.e. thrown through), the screen manager tries to run and empty the queue, using an unwind-protect. However, under some circumstances it cannot do screen management at this time. In these cases, it leaves the requests on the queue. There is a background process that runs all the time, called Screen Manager Background, that wakes up to do the screen management that these queue entries specify, when screen management stops being delayed. So the screen management does eventually happen, when the special form is exited and the background process wakes up. 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 *Special Form*

The tv:delaying-screen-management special form just has a body:

```
(tv:delaying-screen-management
  form-1
  form-2
  ...)
```

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

This background process has another useful function, which is optional. Recall that if a window has its "deexposed typeout action" set to :permit, processes can type out on the window, but the typeout goes to the bit-save array rather than to the screen. The screen manager background process can be told to find any such windows on which some typeout has happened, and copy their partially-visible parts to the screen so that they can be seen. This way, you get to see the typeout that happens on the part of the window that isn't being covered by any other windows.

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

This 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.

The screen manager also has one other job. At the same time that it does autoexposing, it can also select a window if there isn't any selected window at the time. Since selection is being changed, this will not be discussed further at this time.

1.10 The Selected Window

[Selection is a very important concept. However, it is currently being completely redesigned. This section of the document will be supplied later.]

[The general idea is that when you type characters on the keyboard, they have to get sent to some specific window. The selected window is the one that they get sent to.]

If a process tries to do input from a window that does not have any characters in its input buffer, what happens depends on the window's "deexposed typein action". It may be either :normal or :notify. If the "deexposed typein action" is :normal, and/or the window is exposed,

then the process just waits until something appears in the input buffer. If the "deexposed typein action" is :notify and the window is not exposed, then the user is notified (see <not-yet-written>) with a message like "Process X wants typein", and the window is "made interesting" so that Terminal-O-S can select it.

2. Flavors and Messages

2.1 Overview

In the previous chapter, we discussed the concepts behind windows: how they relate to each other, what state they have, and what sorts of things they do. In this chapter we will present the actual messages that can be sent to windows to examine and alter their state and to get them to do things. Just how a window reacts to a message depends on what flavor it is an instance of, and so we will also explain the various flavors that exist. This chapter also explains how to create new windows, and how to compose new flavors of windows by mixing together existing flavors.

Windows have a wide variety of functions, and can respond to any of a large set of messages. To help you find your way around among all the messages, this chapter groups together messages that deal with the same facet of the functionality of windows. Here is a summary of the various groups of messages that are documented.

First of all, a window can be used as if it were the screen of a display computer terminal. 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 sending stream messages to the window. This way, windows can act as output streams, and any function that takes a stream for its argument (such as print or format) can be passed a window. Characters can be drawn in any of a large set of *fonts* (typefaces), and you can switch from one to another within a single window. Windows do useful things when you try to run the cursor off the right or bottom edges; they also have a facility called *more processing* to stop characters from coming out faster than you can read them.

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

A window can also be used for reading in characters from the keyboard; you do this by sending it stream input messages (such as :tyi and :listen). This way, windows can act as input streams, and any function that takes a stream for its argument (such as read or readline) can be passed a window. Each window has an *input buffer* holding characters that have been typed at the window but not read yet, and there are messages that deal with these buffered characters. 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 which another process is reading characters from.

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) and they 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; a blinker is visible to the user, but while programs are examining and altering the contents of a window the blinkers all go away.

This means that blinkers do not affect the contents of the window as seen from programs; whenever a program looks at a window, the blinkers are all turned off. The reason for this is so that you can draw characters and graphics on the window without having to worry whether the flashing blinker will overwrite them. If you have anything that should appear to the user but not be visible to the program, then it should be a blinker. The window system provides a few kinds of blinkers, and you can define your own kinds. Blinkers are instances of flavors, too, and have their own set of messages that they understand.

Any program can use the mouse as an input device. The window system provides many ways for you to get at the mouse. Some of them are very easy to use, but don't have all the power you might want; others are somewhat more difficult to use but give you a great deal of control. The window system also takes responsibility for figuring out which of many programs have control over the mouse at any time.

There are a large number of messages for manipulating the size and position of a window. 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.

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 drawing and graphics drawing all occur on the inside part of the window. You have a great deal of control over what goes in the margins of a window. Control can be exercised either by mixing in different flavors that put different things in the margins or by specifying parameters such as the width of the borders or the text to appear in the label.

You can create windows with several panes (inferior windows). These are called *frames*, and there are messages that deal specifically with frames, their configuration, and their inferiors.

Sometimes a background process wants to tell the user something, but it does not have any window on which to display the information, and it does not want to pop one up just for one little message. A facility is provided wherein the process can send such *notification* messages to the selected window, and it will find some way to get the message to the user. Different windows do different things when someone tries to use them for notification.

Screens are windows themselves; they also have extra functions that windows don't have, since they do not have superiors and since they correspond to actual pieces 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*, that specifies which values of the pixels display which colors.

There are also messages for changing the status of windows: whether they are active, exposed, or selected. There are several options to exactly how exposure and deexposure should affect the screen. You can also ask windows to refresh their contents, kill them, and so on. There are also ways to deal with the screen manager, including messages to examine and alter priorities, and other functions and variables and flavors for affecting what the screen manager

does.

You can define your own fonts, and/or convert fonts from other formats to the Lisp Machine's format. Font characters have various attributes such as their height, baseline, left kern, and so on.

The who-line at the bottom of the screen shows the user something about the state of the Lisp Machine. There are several functions for controlling just what it does and for getting things to be displayed in it.

The window system provides a facility called *I/O buffers*. An I/O buffer is a general purpose first-in first-out ring buffer, with various useful features. I/O buffers are what the input buffers of windows are made out of. Programs can use I/O buffers for anything else, too; it need not even have anything to do with the window system.

There are some interrelationships between windows and processes. Exactly how processes and windows relate depends on the flavor of the window, and, as usual, there are several messages to manipulate the connections.

2.2 Getting a window to use

Many programs never need to create any new windows. Often, all you are interested in doing is sending messages to standard-output and standard-input and performing the extended stream operations offered by windows to read and type characters, position the cursor (and other things that you do on display terminals), and draw graphics. Other programs want to create their own windows for various reasons; a common way to organize an interactive system on the Lisp Machine is to create a process that runs the command loop of the system, and have it use its own window or suite of windows to communicate with the user. This kind of system is what the editor and ZMail use, and it is very convenient to deal with.

Whichever of these you use, it is important for you to know what flavor of window you are getting. Some flavors accept certain messages that are not handled by others. The details of different flavors' responses to the same message may vary in accordance with what those flavors are supposed to be for. The following is a discussion of window flavors.

The most primitive flavor of window is called tv:minimum-window; it is the basic flavor on which all other window flavors are built, and it contains the absolute minimum amount of functionality that a window must have to work. There is another flavor called tv:window, which is built on tv:minimum-window and has about six mixins that do a variety of useful things. When you cold-boot a Lisp Machine, the window you are talking to is of flavor tv:lisp-listener, which is built on tv:window and has three more mixins.

In the following sections we will learn which mixins they use and what each of those mixins does. However, if you don't want to have to read through all that, the basic rule to follow is that tv:window has all the stuff you need to do the normal things that are done with windows; tv:minimum-window is missing messages for character output and input, selection, borders, labels, and graphics, and so there isn't much you can do with it. Anything built on tv:window, including Lisp Listeners, will be able to accept all the basic messages.

Some programs may benefit from more carefully tailored mixings of flavors. For the benefit of programmers who want to do this, we specify below, with each message and init-option, which flavor actually handles it. If you are just using tv:window then you don't really care exactly what mixin specific features are in; you just need to know which ones are in tv:window. With the discussion of each flavor or group of messages, we will say which relevant flavors are in tv:window and which are not. For reference, tv:window is defined (ignoring defflavor options) as follows:

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

So if you use tv:window then you have all the above mixins, and can take advantage of their features.

If you want to create your own window, you use the tv:make-window function. Never try to instantiate a window flavor yourself with make-instance or instantiate-flavor; always use tv:make-window which takes care of a number of internal system issues.

tv:make-window *flavor-name* &rest *init-options*

Create, initialize, and return a new window of the specified flavor. The *init-options* argument is the init-plist (it is just like the &rest argument of make-instance (see section 20.8 of the Lisp Machine manual)). The allowed initialization options depend on what flavor of window you are making. Each window flavor handles some init-options; the options and what they mean are documented with the documentation of the flavor.

Example:

```
(tv:make-window '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.

:activate-p *t-or-nil* (Init Option for tv:minimum-window)

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

:expose-p *t-or-nil* (Init Option for tv:minimum-window)

If this option is specified non-nil, the window is exposed 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 message (the *turn-on-blinkers* option).

2.3 Typing Characters

As we said above, a window can be used as if it were the screen of a display computer terminal, and it can act as an output stream. The flavor tv:stream-mixin implements the messages of the Lisp Machine output stream protocol (see section 21.5.2 of the Lisp Machine manual). It implements a large number of optional messages of that protocol, such as :insert-line. The tv:stream-mixin flavor is a component of the tv:window flavor. 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 messages 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.

In the messages below, the cursor position is always expressed in "inside" coordinates (see page 17); that is, its coordinates are always relative to the top-left corner of the inside part of the window, and 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!)

To draw a character "at" the cursor position basically means that the top-left corner of the character will appear at the cursor position; so if the cursor position is at position *(0,0)* and you draw a character, it will appear at the top-left corner of the window. (Things can actually get more complicated when fonts with left-kerns are used; see page 40.)

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 on page 27 in the section on graphics. 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. 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.

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 messages that type out characters always type in the current font. Details of fonts and the font map are gone into later in detail (see section 2.6, page 37). For now, it is only important to understand fonts in order to understand what the *character-width* and *line-height* of the window are; these two units are used by many of the messages 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 also you can add vertical spacing between lines by controlling the *vsp* of the window. (Messages for controlling the *vsp* are documented on page 25.)

Every window has a *current font*, which the messages 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*. In a variable-width font, expressing horizontal positions in numbers of characters is not

meaningful, 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.

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 messages.

First of all, as was explained earlier, before doing any typeout the process must wait until it has the ability to output (see section 1.7, page 8). The "output hold flag" must be off and the window must not be temp-locked.

Before actually typing anything, various exceptional conditions are checked for. If an exceptional condition is discovered, a message is sent to the window; the message keyword is the name of the condition. Different flavors handle the various exceptions different ways; you can control how exceptions are handled by what flavors your window is made of. First, if the *y*-position of the cursor is less than one line-height above the inside bottom edge of the window, an :end-of-page-exception happens. The handler for this exception in the tv:minimum-window flavor moves the cursor position to the upper-left-hand corner of the window and erases the first line, doing the equivalent of a :clear-eol operation.

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 handler in the tv:minimum-window flavor does a :clear-eol operation, types out **MORE**, waits for any character to be typed, 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.

Note that the *more flag* is only set 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.

Finally, if there is not enough room left in the line for the character to be typed out, an :end-of-line-exception happens. The handler for this exception in the tv:minimum-window flavor advances the cursor to the next line just as typing a #\return character does normally (see below). This 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 26), 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.

The way the cursor position goes to the next line when it reaches the right edge of the window is called *horizontal wraparound*. You can make windows that truncate lines instead of wrapping them around by using tv:line-truncating-mixin; see page 26.

After checking for all these exceptions, 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 26).

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 22). 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.

:tyo *ch* (to tv:stream-mixin)
Type *ch* on the window, as described above. Basically, type the character *ch* in the current font at the cursor position, and advance the cursor position.

:string-out *string* &optional (*start* 0) (*end* nil) (to tv:stream-mixin)
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 sent to the window with a :tyo message, but it is much faster.

:line-out *string* &optional (*start* 0) (*end* nil) (to tv:stream-mixin)
Do the same thing as :string-out, and then advance to the next line (like typing a #\return character). The main reason that this message exists is so that the stream-copy-until-eof function (see <not-yet-written>) 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 26).

:fresh-line (to tv:stream-mixin)
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 26).

:read-cursorpos &optional (*units* ':pixel) (to tv:stream-mixin)
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.)

:set-cursorpos *x y* &optional (*units* ':pixel) (to tv:stream-mixin)
Move 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 (to tv:stream-mixin)
Move the cursor to the upper left corner of the window.

:home-down (to tv:stream-mixin)
Move the cursor to the lower left corner of the window.

:size-in-characters (to tv:stream-mixin)
Return two values: the dimensions of the window in character-widths and line-heights. (Note that character-widths don't mean much when you are using variable-width fonts.)

:clear-char &optional *char* (to tv:stream-mixin)
Erase 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-eol (to tv:stream-mixin)
Erase from the current cursor position to the end of the current line; that is, erase 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 (to tv:stream-mixin)
Erase from the current cursor position to the bottom of the window. In more detail, first do a :clear-eol, and then clear all of the window past the current line.

:clear-screen (to tv:minimum-window)
Erase the whole window and move the cursor position to the upper left corner of the window.

:delete-char &optional (*n 1*) (to tv:stream-mixin)
Without an argument, delete the character at the current cursor position. Otherwise, delete *n* characters, starting with that character. 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. (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) (to tv:stream-mixin)
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 number, it is considered to be a character code. Excise a region exactly as wide as that character at the current cursor position, and move 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 string, excise a region exactly as wide as that string, or a substring specified by *start* and *end*, and close the gap as in the single-character case.

:delete-line &optional (*n 1*) (to tv:stream-mixin)
Without an argument, delete the line that the cursor is on. Otherwise delete *n* lines, starting with the one the cursor is on. Move up the display below the deleted section to close the resulting gap.

:insert-char &optional (*n 1*) (to tv:stream-mixin)
Open up a space the width of *n* characters in the current line at the current cursor position. Shift 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. (This assumes all characters are one character-width wide, and so will not do anything useful with variable-width fonts.)

:insert-string *string* &optional (*start 0*) (*end* nil) (*type-too t*) (to tv:stream-mixin)
Insert 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, suppress the actual display of the string, and the space that was opened is left blank.

:insert-line &optional (*n l*) (to tv:stream-mixin)
Take the line containing the cursor and all the lines below it, and move 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, open up *n* blank lines. Lines pushed off the bottom of the window are lost.

The following messages are not part of the stream protocol, but their functions are relevant to the typing out of characters.

:character-width *char* &optional (*font* current-font) (to tv:stream-mixin)
Return 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 *string* &optional (*start* 0) (*end* nil) (*x* cursor-x) (*y* cursor-y)
(*cr-at-end-p* nil) (*stop-x* 0) *stop-y* (to tv:stream-mixin)
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.

The method does a triple-value return of the *x* and *y* coordinates you ended up at and an indication of how far down the string you got. This indication is nil if the whole string (or the part specified by *start* and *end*) was exhausted, or the index of the next character to be processed when the stopping point (end of window) was reached, or t if the stopping point was reached only because of an extra carriage return due to *cr-at-end-p* being t.

All coordinates for this message are in pixels.

:string-length *string* &optional (*start* 0) (*end* nil) *stop-x* (*font* current-font) (*start-x* 0)
(to tv:stream-mixin)
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 *font* can be specified.

:string-length returns three values: where the imaginary cursor ended up, 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*), and the maximum x-coordinate reached by the cursor (this is the same as the first value unless there are #\return characters in the string).

The following messages and initialization options initialize, get, and set various window attributes which are relevant to the typing out of characters. (See also the messages to manipulate the current font, on page 38.)

:more-p *t-or-nil* (Init Option for tv:minimum-window)
Initialize whether the window should have more processing. It defaults to t.

:more-p (to tv:minimum-window)
Return t if more processing (see page 21) is enabled; otherwise, return nil.

:set-more-p *more-p* (to tv:minimum-window)
If *more-p* is nil, turn off more processing (see page 21); otherwise turn it on.

:vsp *n-pixels* (Init Option for tv:minimum-window)
Initialize the window's *vsp*. It defaults to 2.

:vsp (to tv:minimum-window)
Return the value of *vsp* for this window (see page 20).

:set-vsp *new-vsp* (to tv:minimum-window)
Set the value of *vsp* for this window (see page 20) to *new-vsp*.

:reverse-video-p (to tv:minimum-window)
Return 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 (set by Terminal-C).

:set-reverse-video-p *t-or-nil* (to tv:minimum-window)
Enable or disable reverse-video display. Changing this mode inverts all of the bits in the window.

:deexposed-typeout-action *action* (Init Option for tv:minimum-window)
Initialize the "deexposed typeout action" (see page 8) of the window to *action*. It defaults to :normal.

:deexposed-typeout-action (to tv:minimum-window)
Return the "deexposed typeout action" (see page 8) of the window.

:set-deexposed-typeout-action *action* (to tv:minimum-window)
Set the "deexposed typeout action" (see page 8) of the window to *action*.

:deexposed-typein-action *action* (Init Option for tv:minimum-window)
Initialize the "deexposed typein action" (see page 14) of the window to *action*. It defaults to :normal.

:deexposed-typein-action (to tv:minimum-window)
Return the "deexposed typein action" (see page 14) of the window.

:set-deexposed-typein-action *action* (to tv:minimum-window)
Set the "deexposed typein action" (see page 14) of the window to *action*.

:right-margin-character-flag *x* (Init Option for tv:minimum-window)
If *x* is 1, print an exclamation point in the right margin when :end-of-line-exception happens; if *x* is 0, don't. It defaults to 0. See page 22.

:backspace-not-overprinting-flag *x* (Init Option for tv:minimum-window)
If *x* is 0, typing #\backspace will move the cursor position backward; if it is 1, typing #\backspace will display "overstrike" in a lozenge (that is, #\backspace will be just like other special characters). It defaults to 0. See page 22.

:cr-not-newline-flag *x* (Init Option for tv:minimum-window)
If *x* is 0, typing #\return will move the cursor position to the beginning of the next line and clear that line; if it is 1, typing #\return will display "return" in a lozenge (that is, #\return will be just like other special characters). It defaults to 0. This flag does not affect the behavior of the :line-out nor the :fresh-line messages.

:tab-nchars *n* (Init Option for tv:minimum-window)
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.

The following flavors are relevant:

tv:line-truncating-mixin *Flavor*
If you mix in this flavor, 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 (as opposed to causing wraparound; see page 22). In other words, the text will be truncated at the end of the window.

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.

tv:autoexposing-more-mixin *Flavor*
If you mix in this flavor, when a :more-exception happens, the window will be exposed (a :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 8), so that a process can type out on a deexposed window and then have the window expose itself when a **MORE** break happens.

2.4 Drawing Graphics

A window can be used to draw graphics (pictures). There is a set of messages 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 messages documented below will work on windows of flavor (or flavors built on) tv:window.

There are also some messages in this section that are in tv:stream-mixin 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 messages are :draw-rectangle, and the :bitblt message 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 messages; the messages 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 1.7, page 8). 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.

Most of these messages 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 20), 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 messages 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*

Set all bits in the affected region. 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.

There are a few simple microcoded primitives for drawing graphics. They can be used for drawing pictures into Lisp arrays, and they are documented at the end of this section. However, when drawing on windows you should send the messages below rather than directly calling the microcode primitives because these messages provide several essential services which are too complex for the microcode, such as protecting blinkers from being affected from drawing, and locking out other processes. If you think these messages are too slow, complain and we will make them faster or provide a better way to do what you need.

:point *x y* (to tv:graphics-mixin)

Return 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* (to tv:graphics-mixin)

Draw *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 message will have no effect if the coordinates are outside the window. *value* defaults to -1, i.e. a number with as many 1's as the number of bits in a pixel.

:bitblt *alu width height from-array from-x from-y to-x to-y* (to tv:stream-mixin)

Copy 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 (in the Lisp Machine Manual) for complete details. Note that *to-array* is constrained as described in the the description of bitblt in the Lisp Machine Manual. See the tv:make-sheet-bit-array function below (page 31).

:bitblt-from-sheet *alu width height from-x from-y to-array to-x to-y*
(to tv:stream-mixin)
Copy a rectangle of bits from the window to *to-array*. All the other arguments have the same significance as in :bitblt. Note that *to-array* is constrained as described in the the description of bitblt in the Lisp Machine Manual. See the tv:make-sheet-bit-array function below (page 31).

:bitblt-within-sheet *alu width height from-x from-y to-x to-y* (to tv:stream-mixin)
Copy 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.

[The bitblt messages are going to be changed in the future, since they don't clip at window boundaries, and don't allow for negative width and height.]

:draw-char *font char x y* &optional *alu* (to tv:graphics-mixin)
Display 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 *x1 y1 x2 y2* &optional *alu* (*draw-end-point* t) (to tv:graphics-mixin)
Draw a line on the window with endpoints (*x1,y1*) and (*x2,y2*). If *draw-end-point* is specified as nil, do not draw the last point. This is useful in cases such as xoring a polygon made up of several connected line segments.

:draw-lines *alu x0 y0 x1 y1 ... xn yn* (to tv:graphics-mixin)
Draw *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-curve *x-array y-array* &optional *end alu* (to tv:graphics-mixin)
Draw 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.

:draw-wide-curve *x-array y-array width* &optional *end alu* (to tv:graphics-mixin)
Like :draw-curve but *width* is how wide to make the lines.

:draw-rectangle *width height x y* &optional *alu* (to tv:stream-mixin)
Draw 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* (to tv:graphics-mixin)
Draw a filled-in triangle with its corners at *(x1,y1)*, *(x2,y2)*, and *(x3,y3)*.

:draw-circle *center-x center-y radius* &optional *alu* (to tv:graphics-mixin)
Draw the outline of a circle specified by its center and radius.

:draw-filled-in-circle *center-x center-y radius* &optional *alu* (to tv:graphics-mixin)
Draw a filled-in circle specified by its center and radius.

:draw-filled-in-sector *center-x center-y radius theta-1 theta-2* &optional *alu* (to tv:graphics-mixin)
Draw 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 *x1 y1 x2 y2 n* &optional *alu* (to tv:graphics-mixin)
Draw 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 *px py z* &optional *curve-width alu c1 c2 p1-prime-x p1-prime-y pn-prime-x pn-prime-y* (to tv:graphics-mixin)
Draw 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 message, using *z* points for each such curve. If *curve-width* is provided, the :draw-wide-curve message 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 — Make the derivative zero at this end.

:clamped — Allow 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 — Make 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 Make the derivative at the starting point be the negative of the derivative at the ending point. If *c1* is :anticyclic then *c2* is ignored.

The following functions are primitives for drawing pictures onto arrays. You should only use them on arrays and not directly on windows.

sys:%draw-rectangle *width height x-bitpos y-bitpos alu-function array*
This is analogous to the :draw-rectangle message to tv:graphics-mixin (see above).

sys:%draw-line *x0 y0 x y alu draw-end-point-p array*
This is analogous to the :draw-line message to tv:graphics-mixin (see above).

sys:%draw-triangle *x1 y1 x2 y2 x3 y3 alu sheet*
This is analogous to the :draw-triangle message to tv:graphics-mixin (see above).

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* it 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.

See also the description of the bitblt function in the Lisp Machine manual. The following function is useful for creating arrays that are bitblt'ed into and out of windows.

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.

2.5 Input

A window can be used as if it were the keyboard of a computer terminal, and it can act as an input stream. The flavor tv:stream-mixin implements the messages of the Lisp Machine input stream protocol (see section 21.5.2 of the Lisp Machine manual). The tv:stream-mixin flavor is a component of the tv:window flavor. The reason you do input from windows rather than just from the keyboard is so that many programs can share the keyboard without getting in each other's way. If two processes try to read from the keyboard at the same time, they can do it by going through windows. Characters from the keyboard will only go to the selected window, and not to any of the others; this way, you can control which process you are typing at, by selecting the window you are interested in.

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 CTRL, META, SUPER, and HYPER keys. The format of such fixnums and the symbolic names of the bit fields are documented in the Lisp Machine manual in section 21.1 of the Lisp Machine manual.

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.

Every window (that has tv:stream-mixin as a component) has 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 this buffer, and stays there until a program tries to read characters from this window. There are some messages below that deal with the input buffer, letting you clear it and ask whether there is anything in it.

There are several characters that are specially intercepted by the window system. Some are intercepted as soon as they are typed; others are intercepted when some user process is about to read the character from a window. In the first category, the TERMINAL and SYSTEM keys are always intercepted as soon as they are typed; they cause the Keyboard process to take special action to handle the command that the user is giving. You can add your own TERMINAL and SYSTEM commands; see page 35. CALL, CTRL/ABORT, CTRL/META/ABORT, CTRL/BREAK, and CTRL/META/BREAK are also intercepted as soon as they are typed; you can prevent them from being intercepted by having a non-nil value of the :super-image property on the property list of the input buffer (see below). The effect of these keys is explained in the document "Operating the Lisp Machine".

In the second category, the io-buffer-output-function of the input buffer (which you can change (see <not-yet-written>) defaults to tv:kbd-default-output-function, which intercepts certain characters when they are read. The value of the variable tv:kbd-intercepted-characters is a list of characters which are intercepted and not returned as input from the window. These characters, which default to ABORT, META/ABORT, BREAK, and META/BREAK, are processed by the function tv:kbd-intercept-character which does the standard action, throwing to the tag sys:command-level, resetting the current process, calling the break function, or breaking to the error handler, respectively.

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 35.

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 or by supplying your own io-buffer-output-function, either so that ABORT goes to your own intercept routine instead of tv:kbd-intercept-character, or so that ABORT is read as an input character from the stream like any other and then is handled by your program. If you replace the io-buffer-output-function, you will need to look carefully at the default one and make sure that yours incorporates all desired features.

Normally, fixnums get into the input buffer because characters were typed on the keyboard. However, you can also get any Lisp object into a window's input buffer under program control, by sending a :force-kbd-input message to the window. One common use of this feature is for the mouse process (see page 47) to tell a user process about activity on the mouse buttons. That is how characters with the %%kbd-mouse bit can get read from the window. It is possible to put Lisp objects other than fixnums into an input buffer; by convention, such objects are usually lists whose first element is a symbol saying what kind of a "message" this object is. (Such lists are sometimes called *blips*.) You can also get the mouse to send blips instead of fixnums, in order to find out the mouse position at the time of the click. Using the mouse is explained later on.

Input buffers are made out 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 messages. One thing you can do is to make several windows use the same input buffer; this is often used to make panes of a paned window all share the same input buffer, a practice that is discussed further on page 62. 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 <not-yet-written>.

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.

:tyi &optional *eof-action* (to tv:stream-mixin)
: Read and return 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 only exists because it is part of the input stream protocol.

:tyi-no-hang &optional *eof-action* (to tv:stream-mixin)
: Do the same things as :tyi, except if no input is immediately available, return nil instead of waiting for input. 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.

:untyi *ch* (to tv:stream-mixin)
: Put *ch* back into the window's input buffer so that it will be the next character returned by :tyi. Note that *ch* must be exactly the last character that was :tyi'ed 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.

:listen (to tv:stream-mixin)

Return 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.

:clear-input (to tv:stream-mixin)

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

:rubout-handler *options function* &rest *args* (to tv:stream-mixin)

Apply *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, it returns instead. 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.

:reprompt *function*

The same as :prompt except that the function is not called the first time through.

tv:any-tyi-mixin *Flavor*

This flavor makes it easier for some programs to deal with an input stream that contains both fixnums and blips. Mixing this flavor into a window changes some messages and adds some new messages. In the following, a "blip" is any list in the input stream.

:tyi &optional *eof-action* (to tv:any-tyi-mixin)
:tyi-no-hang &optional *eof-action* (to tv:any-tyi-mixin)
These messages are changed so that they only return fixnums and never return blips. If they encounter any blips in the input stream, they discard them entirely (they are removed from the input buffer and the program never sees them).

:mouse-or-kbd-tyi (to tv:any-tyi-mixin)
:mouse-or-kbd-tyi-no-hang (to tv:any-tyi-mixin)
These are like the :tyi and :tyi-no-hang messages, except that if they see a blip whose car is the symbol :mouse, they return the third element of the blip as the first returned value, and the blip as the second returned value. Blips whose first element is :mouse are sometimes produced by the user's clicking on the mouse; see page 47. These messages only return fixnums.

:list-tyi (to tv:any-tyi-mixin)
This is like :tyi except that it only returns blips and never returns fixnums. If it encounters any fixnums in the input stream, it discards them entirely (they are removed from the input buffer and the program never sees them).

:any-tyi &optional *eof-action* (to tv:any-tyi-mixin)
:any-tyi-no-hang &optional *eof-action* (to tv:any-tyi-mixin)
These do what the :tyi and :tyi-no-hang messages normally do (when you aren't using this mixin).

tv:preemptable-read-any-tyi-mixin *Flavor*
This flavor includes tv:any-tyi-mixin. It also defines the :preemptable-read message.

:preemptable-read *options function* &rest *arguments*
(to tv:preemptable-read-any-tyi-mixin)
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 message takes the same arguments as the normal :rubout-handler message, 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 message 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 message is sent, so the user can keep typing his expression in.

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 messages; 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.

tv:*escape-keys* *Variable*

The value of this variable is an alist, each entry of which describes a subcommand of the TERMINAL key. You can add your own by pushing an entry onto this list. Entries on the list are of the form:

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

char is the character (a fixnum) 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 lowercase 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, which is nil if there was no numeric argument, or else the numeric argument (a fixnum). *documentation* should be a one-line string giving documentation, or a form that gets evaluated and returns either a string or a list of strings which are successive lines of the documentation (usually you use a Lisp form that looks like '("line 1" "line 2" ...)). *documentation* will be printed by TERMINAL-HELP, except that if it is a form which evaluates to nil this key will be omitted 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 io-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:*system-keys* *Variable*

[Note: the upcoming change in selection will probably completely change the way this variable works.]

The value of this variable is an alist, each entry of which describes a subcommand of the SYSTEM key. You can add your own by pushing an entry onto this list. Entries are of the form:

(*char flavor documentation create*)

char is the character (a fixnum) 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 lowercase characters. *documentation* should be a string giving the colloquial name of this flavor; it will be printed by SYSTEM-HELP.

If *flavor* is an instance of a flavor, then it should be a window, and the SYSTEM command will select that particular window. However, normally *flavor* 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. If *create* is nil, it beeps. If *create-p* is t, it creates a new window of that flavor (calling tv:make-window with no options) and selects it. 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 CTRL 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)
```

2.6 Fonts

2.6.1 Using 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 typeface 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.

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 by sending the appropriate messages.

Before we go into the details of these messages, there is a little issue to clear up. 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 you write a program that 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 that the fonts you are using has been 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 descriptor*, because it describes a font rather than actually being a font.

In the messages below, where the message expects to be passed a font descriptor, you normally pass a symbol as explained above. You may also pass in a font, in which case the symbol that names that font will be used as the font descriptor. In other words, if you pass in a font explicitly, that font itself might not be used; if you pass in a black-and-white font to a window on a color screen, the name of the black-and-white font will be used as a font descriptor and a color version of the font will be found.

The functions that understand font descriptors 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 MIT systems load fonts from host AI, device DSK, directory LMFONT.) 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 43).

Every screen has a default font. When a window is created, by default, all elements of its font map are this default font, and the current font is this font.

:font-map (to tv:minimum-window)
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. 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 message.

:set-font-map *new-map* (to tv:minimum-window)
Set the font map to contain the fonts given in *new-map*. Return 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 descriptors, in which case this array is installed as the new internal array of the window, and the font descriptors are replaced by fonts. *new-map* may also be a list of font descriptors, 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).

tv:font-map *new-map* (Init Option for tv:minimum-window)
This option lets you initialize the font map. *new-map* is interpreted the same way it is interpreted by the :set-font-map message.

:current-font (to tv:minimum-window)
Returns the current font, as a font object.

:set-current-font *new-font* (to tv:minimum-window)
Set 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 descriptor, in which case the font that the descriptor describes is used, unless that font is not in the font map, in which case an error is signalled. You may only select a font that is already in the font map.

:baseline (to tv:minimum-window)
Returns the maximum baseline of all the fonts in the font map. The bases of all characters will be so aligned as to be this many pixels below the 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.

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 **tr10**, used for selected items in Choose Variable Values windows.

2.6.2 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 20), 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 lowercase "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, and draws the character in a position adjusted vertically to make the bases of the characters all 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 only affects where the single character is drawn and does not have any cumulative effect on the cursor position.

A font which 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 only used for making blinkers. Using a fixed width blinker for a variable-width font is not the nicest-looking thing to do; 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 blinker width as the width of the 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.

2.6.3 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, and 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.

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 "physical" characters side by side. For the time being, we will discuss regular fonts, in which logical characters and physical characters correspond one-for-one.

Each physical 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 physical 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. Then the raster width of the font would be seven, and the raster height would be six. Each row of a character would be seven bits, and so four of them would fit into a thirty-two bit word, with four bits wasted. The remaining two rows would go into the next word, and the rest of that word would be unused, because the number of words per character must be an integer. So for this font there would be four rows per word, and two words per character. To find the bits for character three of the font, you multiply the physical 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 the type of 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 thirty-two bits, then even a single row of the font will not fit into a word. For such fonts, there is a table called the *font indexing table* that, for each logical character, refers to several physical characters. To draw the single logical character, you draw all of the physical characters next to each other. The raster width of the font means the width of each physical character, and each physical character is of this same width. Logical character n is formed from a contiguous sequence of physical characters, which are drawn side-by-side; the first physical character number is found from the nth element of the indexing table, and the last physical character is one less than the contents of the $n+1$st element of the indexing table. For example, suppose you wanted to find the bits for character

seven in a wide font. You find that the contents of the seventh and eigth elements of the indexing table are eleven and fourteen, respectively. This means that physical characters eleven, twelve, and thirteen, drawn side by side, give the drawing of character seven in the 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 binding is this 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 would typically be 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 array of non-negative fixnums, one for each logical character of the font, giving the character width for that character.

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 physical character; 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 with one element for each logical character plus one more at the end (to show where the last character stops) containing physical character numbers.

2.6.4 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 descriptor 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 a four-bit color display (the only type presently supported), for example, 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* "")
Create and return 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 descriptor 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 wha you want, you should make sure that the background is made of all zeroes before drawing characters onto a color window.

2.7 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 messages.

Also, 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 typein will go if you type on the keyboard. The way this behavior is obtained is that selection and deselection of a window have an effect on the visibility of the window's blinkers.

When the window is selected, any of its blinkers whose visibility is :on or :off have their visibility set to :blink. Blinkers whose visibility is t or nil are unaffected (that is the difference between t and :on, and between nil and :off); blinkers whose visibility is :blink continue to blink.

Each blinker has a *deselected visibility*, which should be one of the symbols above; when a window is deselected, the visibilities of all blinkers that are blinking (whose visibility is currently :blink) are set to the deselected visibility.

Most often, blinkers have visibility :on when their window is not selected, and visibility :blink when their window is selected. In this case, the deselected visibility is :on.

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. 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 29).

Every blinker is associated with a particular window. A blinker cannot leave the area described by its window; its position is expressed relative to the window. When characters are output or graphics are drawn on a window, only the blinkers of that window and its ancestors are opened (since blinkers of other windows 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 of the screen.

The window system provides a few kinds of blinkers. Blinkers are implemented as instances of flavors, too, and have their own set of messages that they understand, which is distinct from the set that windows understand.

Positions of blinkers are always expressed in pixels, relative to the inside of the window (that is, the part of the window that doesn't include the margins).

tv:make-blinker *window* &optional (*flavor* 'tv:rectangular-blinker) &rest *options*
Create and return 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.

: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 (to 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* (to tv:blinker)
Set the position of the blinker within the inside of the window. If the blinker had been following the cursor, it stops doing so, and stays where you put it.

:follow-p *t-or-nil* (Init Option for tv:blinker)
Set 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* (to tv:blinker)
Set 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.

:visibility *symbol* (Init Option for tv:blinker)
Set the initial visibility of the blinker. This defaults to :blink.

:set-visibility *new-visibility* (to tv:blinker)
Set the visibility of the blinker. *new-visibility* should be one of :on, nil, :off, t, or :blink; their meanings are described above.

:deselected-visibility *symbol* (Init Option for tv:blinker)
Set the initial deselected visibility. By default, it is :on.

:deselected-visibility (to tv:blinker)
:set-deselected-visibility *new-visibility* (to tv:blinker)
Examine or change the deselected visibility of the blinker.

:half-period *n-60ths* (Init Option for tv:blinker)
Set the initial value of the half-period of the blinker. This defaults to 15.

:half-period (to tv:blinker)
:set-half-period *new-half-period* (to tv:blinker)
Examine or change the half-period of the blinker.

:set-sheet *new-window* (to tv:blinker)
Set the window associated with the blinker to be *new-window*. If the old window is an ancestor or descendant of *new-window*, adjust the (relative) position of the blinker so that it does not move. Otherwise, move it to the point *(0,0)*.

tv:sheet-following-blinker *window*
Take a *window* and return a blinker that follows the window's cursor. If there isn't any, it returns nil. 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*
Set the visibility of all blinkers on *window* to :off.

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 44) of the zeroth font of the window associated with the blinker.

:set-size *new-width new-height* (to tv:rectangular-blinker)
Set the width and height of the blinker, in pixels.

tv:hollow-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 messages work on this too.

tv:box-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 messages work on this too.

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)
Set the initial height of the blinker. It defaults to the *line-height* (see page 20) 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)
Set the font in which to find the character to display. This may be anything acceptable to the :parse-font-descriptor message (see <not-yet-written>) of the window's screen. You must provide this.

:char *ch* (Init Option for tv:character-blinker)
Set the character of the font to display. You must provide this.

:set-character *new-character* &optional *new-font* (to tv:character-blinker)
Set 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-descriptor message (see <not-yet-written>) of the window's screen.

2.8 The Mouse

Along with the keyboard, the mouse can be used by any program 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 extend the window system by writing your own methods, which is beyond the scope of this manual. Of course, you can invoke the built-in choice facilities, such as menus and multiple-choice windows and so on; these high-level facilities are described in their own volume of documentation.

The window system includes a process called **Mouse** which normally *tracks* the mouse. To track the mouse means to examine the hardware mouse interface, noting how the mouse is moving, and adjust Lisp variables and the mouse blinker to follow the position being indicated by the user. 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 window that owns the mouse is the window that is under the mouse; but since the 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.

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. For you to do it yourself, you must extend the window system, creating your own methods to be run in the mouse process; that is beyond the scope of this document. The flavor of the window owning the mouse is also what usually controls the effect of clicking the mouse buttons.

Below, we explain three ways for you to use the mouse without writing your own methods. First, you can mix in flavors to your window to tell the mouse process to let you know when the mouse is clicked. Secondly, you can watch the mouse moving and watch the buttons, letting the mouse process do the tracking. Finally, you can turn off the mouse process and do your own tracking. You have to choose one of these three ways to use the mouse; you can't mix them. Note that you can also use various high-level facilities to get certain specific mouse behavior: you can create windows with mouse-sensitive items (like the List Buffers command in the Editor), menus, multiple-choice windows, and more. This is explained on <not-yet-written>.

Clicks on the mouse are sometimes *encoded* into a fixnum. Such fixnums are normally forced into input buffers of windows (see <not-yet-written>), and so they are distinguished from regular keyboard characters by having the %%kbd-mouse bit turned on. If this bit is set in a fixnum in an input buffer, it is interpreted as a mouse click. The %%kbd-mouse-button field tells you

which button was clicked; 0, 1, and 2 mean the left, middle, and right buttons, respectively. The value in the %%kbd-mouse-n-clicks field is one less than the number of times the mouse was clicked. These characters can be typed in symbolically as #\mouse-*b*-*n*, where *b* a letter for which button (l, m, or r) and *n* is one greater than the %%kbd-mouse-n-clicks field. For example, #\mouse-r-2 means a double-click on the right-hand button.

The first way to use the mouse is to get mouse clicks sent to your input buffer. This is the easiest thing to do, though it is insufficient if your application requires that you know more than just when the mouse is clicked.

tv:kbd-mouse-buttons-mixin *Flavor*

If you mix in this flavor, then when the mouse is clicked on the window, it will be handled as follows: if it is a double-click on the right button, the system menu will be called forth. Otherwise, the encoded fixnum representation of the click will be forced into the input buffer of the window. Furthermore, if it is a single-click on the left button, the window will be selected. This is the usual easy way to interpret mouse clicks; by mixing in this flavor, the mouse process will put the clicks into the input stream.

tv:list-mouse-buttons-mixin *Flavor*

This is just like tv:kbd-mouse-buttons-mixin except that instead of forcing the fixnum into the input buffer, it forces in a blip of the form:

(: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 primarily useful if several windows are sharing the same input buffer).

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 charcter from the keyboard. [This may be fixed in the future.] See <not-yet-written> for further discussion of these issues.

The second way to use the mouse is to *grab* it. 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 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
  form1
  form2)
```

The forms inside are evaluated with the mouse grabbed.

[Explain here how to grab the mouse and confine it to your particular window.]

tv:mouse-x *Variable*
tv:mouse-y *Variable*

These variables give the position of the mouse, in pixels, measured from the upper-left corner of the screen the mouse is on (tv:mouse-sheet). These variables are maintained by the process handling the mouse, normally the mouse process. They are both in outside coordinates, since the mouse might be in the margins somewhere.

tv:mouse-last-buttons *Variable*

This variable contains the last setting of the mouse pushbuttons noticed by the process handling the mouse, which is normally the mouse process. 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-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 done using the values and wants to wait for them to change 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 fixnum giving the click in the usual way.

You only call tv:mouse-button-encode 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 (boole 2 *old-buttons new-buttons*) will compute this mask.

The third way to use the mouse is to tell the mouse process to not do anything, and track the mouse in your own process. This is called *usurping* the mouse. 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
  form1
  form2)
```

The forms inside are evaluated with the mouse usurped.

tv:mouse-input &optional (*wait-flag* t)

Wait until something happens with the mouse, and then return 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 only call this function with the mouse usurped; otherwise you will get in the way of the mouse process, which calls it 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

Return the current state of the mouse buttons, in the format used by the tv:mouse-last-buttons variable. This function has no state or anything; it just goes straight to the hardware and reads the current state.

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. The 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.

Here is some more stuff relevant to the mouse.

tv:hysteretic-window-mixin *Flavor*

By mixing this flavor into your window, you control the mouse for a small area outside the window as well as the area inside the window. You can control the hysteresis, which is the number of pixels away from the window that the mouse has to get before this window ceases to own 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 for it before it vanishes.

:hysteresis *n-pixels* (Init Option for tv:hysteretic-window-mixin)
Set the initial value of the hysteresis, in pixels. It defaults to 25. (decimal).

:hysteresis (to tv:hysteretic-window-mixin)
:set-hysteresis *new-hysteresis* (to tv:hysteretic-window-mixin)
Examine or set the hysteresis of the window.

2.9 The Keyboard

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. This only works on new-style keyboards (the ones with the Super and Hyper keys); the old-style keyboards have no hardware capability to do this.

One application for checking the state of keys is in user interfaces where the action of mouse clicks is modified by the shift keys on the keyboard; you can have one hand on the mouse and the other on the keyboard. [When facilities for this are installed into the standard system they will be documented in this section.]

tv:key-state *key-name*
Returns t if the keyboard key named *key-name* is currently depressed, nil if it is not. On "Knight" keyboards, this function always returns nil.

key-name may be the symbolic name of a shift key, from the table below, or the number 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		

2.10 Sizes and Positions

The messages and init-options in this section are used to examine and set the sizes and positions of windows. There are many different messages, 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 20.

As has been mentioned before, 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 messages 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.

:left *left-edge* (Init Option for tv:minimum-window)
:x *left-edge* (Init Option for tv:minimum-window)
:top *top-edge* (Init Option for tv:minimum-window)
:y *top-edge* (Init Option for tv:minimum-window)
:position (*left-edge top-edge*) (Init Option for tv:minimum-window)
:right *right-edge* (Init Option for tv:minimum-window)
:bottom *bottom-edge* (Init Option for tv:minimum-window)
:width *outside-width* (Init Option for tv:minimum-window)
:height *outside-height* (Init Option for tv:minimum-window)
:size (*outside-width outside-height*) (Init Option for tv:minimum-window)
:inside-width *inside-width* (Init Option for tv:minimum-window)
:inside-height *inside-height* (Init Option for tv:minimum-window)
:inside-size (*inside-width inside-height*) (Init Option for tv:minimum-window)
:edges (*left-edge top-edge right-edge bottom-edge*) (Init Option for tv:minimum-window)

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

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 tv:minimum-window)
:minimum-height *n-pixels* (Init Option for tv:minimum-window)
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 messages below is used to examine or change the size or position of a window. Many messages 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. Another reason a new setting of the edges may be invalid is that if the window is exposed, it is not valid to change its edges in such a way that it is not enclosed inside its superior. In all of the messages that take the *option* argument, *option* may be either nil or :verify. If it is nil, that means that you really want to set the edges, and if the new edges are

not valid, an error should be signalled. If it is :verify, that 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 message will return t; otherwise it will return 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.)

:size (to tv:minimum-window)
Return two values: the outside width and outside height.

:set-size *new-width new-height* &optional *option* (to tv:minimum-window)
Set 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 (to tv:minimum-window)
Return two values: the inside width and the inside height.

:set-inside-size *new-inside-width new-inside-height* &optional *option*
(to tv:minimum-window)
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.

:size-in-characters (to tv:minimum-window)
Return two values: the inside size in characters, and the inside height in lines.

:set-size-in-characters *width-spec height-spec* &optional *option* (to tv:minimum-window)
Set 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.

:position (to tv:minimum-window)
Return two values: the *x* and *y* positions of the upper-left corner of the window, in pixels, relative to the superior window, respectively.

:set-position *new-x new-y* &optional *option* (to tv:minimum-window)
Set the *x* and *y* position of the upper-left corner of the window, in pixels, relative to the superior window, respectively.

:edges (to tv:minimum-window)
Return four values: the left, top, right, and bottom edges, in pixels, relative to the superior window, respectively.

:set-edges *new-left new-top new-right new-bottom* &optional *option* (to tv:minimum-window)
Set the edges of the window to *new-left*, *new-top*, *new-right*, and *new-bottom*, in pixels, relative to the superior window, respectively.

:margins (to tv:minimum-window)
Return four values: the sizes of the left, top, right, and bottom margins, respectively.

:inside-edges (to tv:minimum-window)
Return 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 message is *not* analogous to the :edges message, which returns the outside edges relative to the superior window.

:center-around *x y* (to tv:minimum-window)
Without changing the size of the window, position the window so that its center is as close to the point *(x,y)*, in pixels, relative to the superior window, as is possible without hanging off an edge.

:expose-near *mode* &optional (*warp-mouse-p* t) (to tv:minimum-window)
If the window is not exposed, change its position according to *mode* and expose it (with the :expose message; see <not-yet-written>). If it is already exposed, do 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 close to the point *(x,y)*, in pixels, relative to the superior window, as is 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 <not-yet-written>) to the center of the window. (The mouse only moves 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 <not-yet-written>) 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 <not-yet-written>) to the center of the window.

2.11 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 outside in 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.

This section lists the margin item flavors that you can mix in, and explains some messages and init-options that you can use to control what the margin items do.

You can ask for the size of the margins with the :margins message (see page 58).

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) which 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 page 28) 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 <not-yet-written>). 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* (to tv:borders-mixin)
Redefine 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)
Set 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 (to tv:borders-mixin)
:set-border-margin-width *new-width* (to tv:borders-mixin)
Return or set the value of the border margin width.

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 tv:minimum-window)

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 (to tv:minimum-window)

Return the name of the window, which is a symbol. See above.

:label *specification* (Init Option for tv:label-mixin)

Set 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 (see <not-yet-written>), the font is the default font for the screen, 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.

:string *string* — The text displayed in the label is *string*.

:font *font-descriptor*

The label is displayed in the specified font. *font-descriptor* may be any font descriptor (see page 39).

:label-size (to tv:label-mixin)
Return the width and height of the area occupied by the label.

:set-label *specification* (to tv:label-mixin)
Change 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:top-label-mixin *Flavor*
The tv:top-label-mixin margin item is just like tv:label-mixin except that the label is placed at the top of the window by default, instead of the bottom.

tv:top-box-label-mixin *Flavor*
The tv:top-box-label-mixin is just like tv:top-label-mixin except that in addition to the label in the top margin, it also draws a line below the label in the top margin. If you surround the label with borders, then the label will appear inside a box. You have probably seen windows like this appear as momentary menus, with a prompt at the top in a box.

tv:changeable-name-mixin *Flavor*
Mixing in this flavor defines a :set-name method, so that you can change the name of the window, redrawing the label if appropriate (see below). This flavor includes tv:label-mixin, so one of the above kinds of label must be in the margins of the window.

:set-name *new-name* (to tv:changeable-name-mixin)
Set the name of the window to *new-name*, which should be a string. If the window is currently displaying the old name of the window as the label, then redraw the label using the new name as the text to be displayed.

tv:delayed-redisplay-label-mixin *Flavor*
This flavor adds the :delayed-set-label and :update-label messages 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* (to tv:delayed-redisplay-label-mixin)
This is like the :set-label method, except that nothing actually happens until an :update-label message is sent.

:update-label (to tv:delayed-redisplay-label-mixin)
Actually do the :set-label operation on the *specification* given by the most recent :delayed-set-label message.

2.12 Frames

A *frame* is a window that is divided into sub-windows, using the hierarchical structure of the window system (discussed in section 1.4, page 3). 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 Error Handler and Zmacs also use frames. In Zmacs, each new editor window is a pane of the Zmacs Frame. Zmail uses frames heavily.

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.

If you use Edit Screen to change the shape of an Inspector or Window Error Handler 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, and the kind of frame that implements the constraint mechanism is a flavor called tv:basic-constraint-frame. While there are other, more basic frame flavors, you cannot use them alone; you must write a new flavor that includes the more basic frame flavors in its components, and has new methods. Since writing new methods is beyond the scope of this document, we will simply explain how to use constraint frames.

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 mouse the Mail command, the frame changes to a new configuration showing the Headers and Mail panes.

2.12.1 Flavors for Panes and Frames

To have a frame with panes, you must have a frame, which is a window, and you must have panes, each of which is a window. The flavor of each pane of a frame must have, as one of its components, the flavor tv:pane-mixin. Some system facilities (to be described later) provide flavors for you that already have this flavor mixed in. For example, the flavor tv:command-menu-pane is a flavor that consists of tv:command-menu and tv:pane-mixin. (This is the kind of menu most often used in frames; menus are a higher-level facility, described later.) In general, you can take any flavor of window that you might want to use in a pane, and make a new flavor suitable to actually be a pane simply by mixing in tv:pane-mixin.

tv:pane-mixin *Flavor*

The flavor of any window used as a pane of a frame must have tv:pane-mixin as one of its components. For example, to have a pane of a frame that understands everything that tv:window does, you could define the following:

```
(defflavor simple-pane () (tv:pane-mixin tv:window))
```

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. A frame of this flavor is built out of almost the same facilities as is tv:minimum-window; the frame does *not* have all the mixins that go into the tv:window flavor. In particular, it will not have any borders, nor a label. It also has tv:pop-up-notification-mixin (see <not-yet-written>) as a component.

tv:bordered-constraint-frame *Flavor*

This flavor is just tv:constraint-frame with tv:borders-mixin (see page 59) 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.

It is common in frame-oriented interactive subsystems for all of the panes to use the same input buffer (input buffers are discussed on page 33). The reason for this is that such subsystems are usually organized as a single process that reads commands and executes them. But with a many-paned frame, there may be many windows (each pane is a window) at which characters might be typed or mouse-clicks might be clicked. When the process is waiting for its next command, it would be inconvenient for it to have to wait for the complex condition that any of these windows has input available in its input buffer. Instead, since the command stream is only one serial stream of commands anyway, it is common to have all the panes of a frame share the same input buffer.

What happens when many windows share an input buffer is that any characters typed at any of them, or any mouse-clicks that generate forced keyboard input, are all put into the same input buffer, in the chronological order in which they are generated. The process then does successive :tyi stream operations from any pane of the frame or from the frame itself, and it receives anything that has been typed at any pane. When the input buffer is shared like this, it doesn't matter which pane is selected: all the characters go to the same place anyway, and the information as to which pane was typed at is lost. However, the forced keyboard input generated by mouse clicks at a facility that is designed to be used as a pane of a frame (tv:command-menu-pane for instance) will return all useful and relevant information to the sender of the :tyi message, including which pane the mouse was pointing at when it was clicked.

To have all of the panes share the same input buffer, 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.

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 io-buffer is created. (See <not-yet-written> for a discussion of io-buffers.)

2.12.2 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.

```
(tv:make-window 'tv:constraint-frame
                ':panes
                   '((top-pane tv:window-pane)
                     (bottom-pane tv:window-pane))
                ':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:

```
(tv:make-window
  'tv:bordered-constraint-frame
  ':panes
   '((graphics-pane tv:window-pane
       :label nil :blinker-p nil)
     (message-pane tv:window-pane
       :label "Message Pane" :blinker-p nil)
     (interaction-pane tv:window-pane))
  ':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.

```
(tv:make-window
 'tv:bordered-constraint-frame
 ':panes
   '((huey tv:window-pane)
     (dewey tv:window-pane)
     (louie tv:window-pane)
     (main-pane tv:window-pane)
     (random-pane tv:window-pane)
     (menu tv:command-menu-pane
           :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 will be 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 71). 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-pane)
    (dewey tv:window-pane)
    (louie tv:window-pane)
    (main-pane tv:window-pane)
    (random-pane tv:window-pane)
    (menu tv:command-menu-pane
          :item-list ,the-list-of-items))
```

Menus and how to use them are explained later; see <not-yet-written>.

In this example, the window is divided into two windows, side by side.

```
(tv:make-window
  'tv:bordered-constraint-frame
  ':edges '(100 100 600 600)
  ':panes
    '((left tv:window-pane)
      (right tv:window-pane))
  ':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 64.

In actual practice, panes are usually made out of more interesting flavors than tv:window-pane. Many useful flavors of windows exist; they will be described later.

2.12.3 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 61).

: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 the *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.

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 either hold an actual pane, or it may hold something else; if it holds something else, 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 preceeded 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 20). 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, both 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 pane. 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 *message-name arg-1 arg-2 ...*)

A message whose name is *message-name* and whose arguments are 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, which 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. You don't have to worry about these unless you want to define your own methods to be used by :ask constraints, and definition of new methods is generally beyond the scope of this document anyway.

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. Another useful message, handled by tv:pane-mixin (and therefore by *all* panes) is :square-window-size (also with no arguments), which makes the window take up enough room to be square.

:ask-window This constraint is a variation on :ask. Its format is:

(:ask *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 section 8.7 of the Lisp Machine manual).

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 must only 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.

2.12.4 Messages to Frames

:get-pane *pane-name* (to tv:basic-constraint-frame)
Return the pane (the inferior window itself) that was named by the symbol *pane-name* in the :panes specification of this frame.

:pane-name *pane* (to tv:basic-constraint-frame)
Return 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.

:send-pane *pane-name message* &rest *arguments* (to tv:basic-constraint-frame)
Send 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* (to tv:basic-constraint-frame)
Send the specified *message* with the specified *arguments* to all of the panes of this frame, including the non-exposed ones.

:send-all-exposed-panes *message* &rest *arguments* (to tv:basic-constraint-frame)
Send 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)
Make the initial configuration of the frame be the one named by the symbol *configuration-name*.

:configuration (to tv:basic-constraint-frame)
Return the symbol naming the current configuration of the frame.

:set-configuration *configuration-name* (to tv:basic-constraint-frame)
Set the configuration of the frame to the one named by the symbol *configuration-name*.

Concept Index

Flavor Index

Function Index

Message Index

Variable Index

Window Creation Options