7B Programming the User Interface

symbolics

7B Programming the User Interface

symbolics™

Cambridge, Massachusetts

Programming the User Interface, Volume B
999029

August 1986

This document corresponds to Genera 7.0 and later releases.

Symbolics, Inc.
4 New England Tech Center
555 Virginia Road
Concord, MA 01742

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.
Text masters produced on Symbolics 3600-family computers and printed on Symbolics LGP2 Laser Graphics Printers.
Cover Design: Schafer/LaCasse
Printer: CSA Press
Printed in the United States of America.

Printing year and number: 88 87 86 9 8 7 6 5 4 3 2 1

Table of Contents

List of Figures

PART I.

Interactive Streams

1. Introduction to Interactive Streams

An interactive stream is a bidirectional stream designed for interaction with human users. It supports input editing, which lets the user edit input before a function that reads from the stream sees it. Interactive streams are built on the flavor **si:interactive-stream**.

si:interactive-stream *Flavor*

A stream that includes this flavor is interactive, or designed for interaction with a human user. In order to be useful, **si:interactive-stream** must, in turn, include one of the following mixins: **si:display-input-editor**, **si:printing-input-editor**, or **si:halfduplex-input-editor**.

To find out whether or not a stream is interactive, send the stream an **:interactive** message.

:interactive *Message*

The **:interactive** message to a stream returns **t** if the stream is interactive and **nil** if it is not. Interactive streams, built on **si:interactive-stream**, are streams designed for interaction with human users. They support input editing. Use the **:interactive** message to find out whether a stream supports the **:input-editor** message.

Interactive streams are generally connected to a terminal of some kind. Windows built on **tv:stream-mixin** are one kind of interactive stream: See the section "Input From Windows", page 147. Remote terminals are another: See the section "Remote Login" in *Networks*.

Some reading functions can be used to get input from both interactive and noninteractive streams; others are designed to read only from interactive streams. See the section "Input Functions for Interactive Streams", page 5.

Interactive streams support general operations on input and output streams. For more information on these operations: See the section "Stream Operations" in *Reference Guide to Streams, Files, and I/O*. Interactive streams also have specialized input operations, mainly to handle interactions with the input editor: See the section "Messages for Input From Interactive Streams", page 13. They also intercept some characters when read and maintain a list of characters to be handled asynchronously: See the section "Intercepted Characters", page 17. See the section "Interactive-Stream Operations for Asynchronous Characters", page 19. (Remote terminals do not handle asynchronous characters.)

Some interactive streams can display mouse-sensitive items. See the section "Interactive Streams and Mouse-Sensitive Items", page 21.

For information on the program interface to the input editor: See the section "The Input Editor Program Interface", page 23.

The command processor is a utility that reads commands from an interactive stream. For more information: See the section "Communicating with Genera" in *User's Guide to Symbolics Computers*. See the section "The Command Processor Program Interface", page 45.

One common use for interactive streams is to ask a question of the user: See the section "Querying the User", page 63.

2. Input Functions for Interactive Streams

The general reading functions like **zl:read**, **zl:readline**, and **zl:read-delimited-string** can be used to read from either interactive or noninteractive streams. See the section "Input Functions" in *Reference Guide to Streams, Files, and I/O.* The functions described here are designed to read only from interactive streams. The functions that read Command Processor commands, **cp:read-command** and **cp:read-command-or-form**, are described elsewhere: See the section "The Command Processor Reader", page 45. Also: See the section "Overview of Basic Command Facilities" in *Programming the User Interface, Volume A.*

sys:read-character &optional *stream* &key (*fresh-line* **t**) (*any-tyi* **nil**) (*eof* **nil**) (*notification* **t**) (*prompt* **nil**) (*help* **nil**) (*refresh* **t**) (*suspend* **t**) (*abort* **t**) (*status* **nil**) *Function*

Reads and returns a single character from *stream.* This function displays notifications and help messages and reprompts at appropriate times. It is used by **fquery** and the **:character** option for **prompt-and-read.**

stream must be interactive. It defaults to **zl:query-io**.

Following are the permissible keywords:

:fresh-line If not **nil**, the function sends the stream a **:fresh-line** message before displaying the prompt. If **nil**, it does not send a **:fresh-line** message. The default is **t**.

:any-tyi If not **nil**, the function returns blips. If **nil**, blips are treated as the **:tyi** message to an interactive stream treats them. The default is **nil**.

:eof If not **nil** and the function encounters end-of-file, it returns **nil**. If **nil** and the function encounters end-of-file, it beeps and waits for more input. The default is **nil**.

:notification If not **nil** and a notification is received, the function displays the notification and reprompts. If **nil** and a notification is received, the notification is ignored. The default is **t**.

:prompt If **nil**, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor", page 36. The default is **nil**.

:help	If not **nil**, the value should be a help option. See the section "Displaying Help Messages in the Input Editor", page 37. Then, when the user presses HELP, the function displays the help option and reprompts. If **nil** and the user presses HELP, the function just returns **#\help**. The default is **nil**.
:refresh	If not **nil** and the user presses REFRESH, the function sends the stream a **:clear-window** message and reprompts. If **nil** and the user presses REFRESH, the function just returns **#\refresh**. The default is **t**.
:suspend	If not **nil** and the user types one of the **sys:kbd-standard-suspend-characters**, a **zl:break** loop is entered. If **nil** and the user types a suspend character, the function just returns the character. The default is **t**.
:abort	If not **nil** and the user types one of the **sys:kbd-standard-abort-characters**, **sys:abort** is signalled. If **nil** and the user types an abort character, the function just returns the character. The default is **t**.
:status	This option takes effect only if the stream is a window. If the value is **:selected** and the window is no longer selected, the function returns **:status**. If the value is **:exposed** and the window is no longer exposed or selected, the function returns **:status**. If the value is **nil**, the function continues to wait for input when the window is deexposed or deselected. The default is **nil**.

zl:read-expression &optional *stream* &key (*completion-alist* **nil**) (*completion-delimiters* **nil**) *Function*

Like **sys:read-for-top-level** except that if it encounters a top-level end-of-file, it just beeps and waits for more input. This function is used by the **:expression** option for **prompt-and-read**.

stream defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

If *completion-alist* is not **nil**, this function also sets up COMPLETE and c-? as input editor commands. When the user presses COMPLETE, the input editor tries to complete the current symbol over the set of possibilities defined by *completion-alist*. When the user presses c-?, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. *completion-alist* can be **nil**, an alist, an **sys:art-q-list** array, or a keyword:

nil	No completion is offered.
alist	The car of each alist element is a string representing one possible completion.
array	Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.
keyword	If the symbol is **:zmacs**, completion is offered over the definitions in Zmacs buffers. If the symbol is **:flavors**, completion is offered over all flavor names. If the symbol is **:documentation**, completion is offered over all documentation topics available to the Document Examiner.

The default for *completion-alist* is **nil.**

completion-delimiters is **nil** or a list of characters that delimit "chunks" for completion. As in Zwei, completion works by matching initial substrings of "chunks" of text. If *completion-delimiters* is **nil**, the entire text of the current symbol is a single "chunk". The default is **nil.**

zl:read-form &optional *stream* &key (*edit-trivial-errors-p* **zl:*read-form-edit-trivial-errors-p***) (*completion-alist* **zl:*read-form-completion-alist***) (*completion-delimiters* **zl:*read-form-completion-delimiters***) *Function*

Like **zl:read-expression** except that it assumes that the returned value will be given immediately to **eval.** This function is used by the Lisp command loop and by the **:eval-form** and **:eval-form-or-end** options for **prompt-and-read.**

stream defaults to **zl:standard-input.** This function is intended to read only from interactive streams.

If *edit-trivial-errors-p* is not **nil**, the function checks for two kinds of errors. If a symbol is read, it checks whether the symbol is bound. If a list whose first element is a symbol is read, it checks whether the symbol has a function definition. If it finds an unbound symbol or undefined function, it offers to use a lookalike symbol in another package or calls **zl:parse-ferror** to let the user correct the input. *edit-trivial-errors-p* defaults to the value of **zl:*read-form-edit-trivial-errors-p*.** The default value is **t.**

If *completion-alist* is not **nil,** this function also sets up `COMPLETE` and `c-?` as input editor commands. When the user presses `COMPLETE`, the input editor tries to complete the current symbol over the set of possibilities defined by *completion-alist*. When the user presses c-?, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei. *completion-alist* can be **nil**, an alist, an **sys:art-q-list** array, or a keyword:

nil	No completion is offered.
alist	The car of each alist element is a string representing one possible completion.
array	Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.
keyword	If the symbol is **:zmacs**, completion is offered over the definitions in Zmacs buffers. If the symbol is **:flavors**, completion is offered over all flavor names. If the symbol is **:documentation**, completion is offered over all documentation topics available to the Document Examiner.

The default for *completion-alist* is the value of **zl:*read-form-completion-alist***. The default value is **:zmacs**.

completion-delimiters is **nil** or a list of characters that delimit "chunks" for completion. As in Zwei, completion works by matching initial substrings of "chunks" of text. If *completion-delimiters* is **nil**, the entire text of the current symbol is a single "chunk". The default is the value of **zl:*read-form-completion-delimiters***. The default value is **(#/- #/: #\space)**.

zl:*read-form-edit-trivial-errors-p* *Variable*

If not **nil**, **zl:read-form** checks for two kinds of errors. If a symbol is read, it checks whether the symbol is bound. If a list whose first element is a symbol is read, it checks whether the symbol has a function definition. If it finds an unbound symbol or undefined function, it offers to use a lookalike symbol in another package or calls **zl:parse-ferror** to let the user correct the input. The default is **t**.

zl:*read-form-completion-alist* *Variable*

If not **nil**, **zl:read-form** sets up COMPLETE and c-? as input editor commands. When the user presses COMPLETE, the input editor tries to complete the current symbol over the set of possibilities defined by *completion-alist*. When the user presses c-?, the input editor displays the possible completions of the current symbol.

The style of completion is the same as that offered by Zwei.

zl:*read-form-completion-alist* can be **nil**, an alist, an **sys:art-q-list** array, or a keyword:

nil	No completion is offered.
alist	The car of each alist element is a string representing one possible completion.
array	Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.
keyword	If the symbol is **:zmacs**, completion is offered over the definitions in Zmacs buffers. If the symbol is **:flavors**, completion is offered over all flavor names. If the symbol is **:documentation**, completion is offered over all documentation topics available to the Document Examiner.

The default value is **:zmacs**.

zl:*read-form-completion-delimiters* *Variable*

The value is **nil** or a list of characters that delimit "chunks" for completion in **zl:read-form**. As in Zwei, completion works by matching initial substrings of "chunks" of text. If **zl:*read-form-completion-delimiters*** is **nil**, the entire text of the current symbol is a single "chunk". The default value is **(#/- #/: #\space)**.

read-or-end &optional (*stream* **zl:standard-input**) *reader* *Function*

Like **zl:read-expression** except that if it is reading from an interactive stream and the user presses END as the first character or the first character after only whitespace characters, it returns two values, **nil** and **:end**. If it encounters any nonwhitespace characters, it calls the *reader* function with an argument of *stream* to read the input. *reader* defaults to **zl:read-expression**. *stream* defaults to **zl:standard-input**.

The **:expression-or-end** and **:eval-form-or-end** options for **prompt-and-read** invoke **read-or-end**.

This function is intended to read only from interactive streams.

zl:read-or-character &optional *delimiters stream reader* *Function*

Like **zl:read-expression**, except that if it is reading from an interactive stream and the user types one of the *delimiters* as the first character or the first character after only whitespace characters, it returns four values: **nil**, **:character**, the character code of the delimiter, and any numeric argument to the delimiter. If it encounters any nonwhitespace characters, it calls the *reader* function with an argument of *stream* to read the input.

delimiters is a character, a list of characters, or **nil**. The default is **nil**. *reader* defaults to **zl:read-expression**. *stream* defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

zl:read-and-eval &optional *stream* (*catch-errors* **t**) *Function*

Calls **zl:read-expression** to read a form, without completion. It then evaluates the form and returns the result. If *catch-errors* is not **nil**, it calls **zl:parse-ferror** if an error occurs during the evaluation (but not the reading) so that the input editor catches the error.

stream defaults to **zl:standard-input**. This function is intended to read only from interactive streams.

zl:readline-no-echo &optional *stream* &key (*terminators* '(#**return** #**line** #**end**)) (*full-rubout* **nil**) (*notification* **t**) (*prompt* **nil**) (*help* **nil**) *Function*

Reads a line of input from *stream* without echoing the input, and returns the input as a string, without the terminating character. This function is used to read passwords and encryption keys. It does not use the input editor but does allow input to be edited using `RUBOUT`.

stream must be interactive. It defaults to **zl:query-io**.

Following are the permissible keywords:

:terminators A list of characters that terminate the input. If the user types #**return**, #**line**, or #**end** as a terminator, the function echoes a `NEWLINE`. If the user types any other character as a terminator, the function echoes that character. The default is (#**return** #**line** #**end**).

:full-rubout If not **nil** and the user rubs out all characters on the line, the function returns **nil**. If **nil** and the user rubs out all characters on the line, the function waits for more input. The default is **nil**.

:notification If not **nil** and a notification is received, the function displays the notification and reprompts. If **nil** and a notification is received, the notification is ignored. The default is **t**.

:prompt If **nil**, no prompt is displayed. Otherwise, the value should be a prompt option to be displayed at appropriate times. See the section "Displaying Prompts in the Input Editor", page 36. The default is **nil**.

:help If not **nil**, the value should be a help option. See the section "Displaying Help Messages in the Input Editor", page 37. Then, when the user presses HELP, the function displays the help option and reprompts. If **nil** and the user presses HELP, the function just returns #**help**. The default is **nil**.

3. Messages for Input From Interactive Streams

All interactive streams support these input operations. Some streams have specialized versions of some operations, partly because different kinds of streams have different sources of input when input is to come from the stream instead of the input buffer. Windows, for example, take input from an I/O buffer. See the section "Messages for Input From Windows", page 149.

:any-tyi &optional *eof-action* of **si:interactive-stream** *Method*

Read and return the next character of input from the stream, waiting if there is none. Where the character comes from depends on the value of the variable **sys:rubout-handler**. Following is a summary of actions for each possible value of **sys:rubout-handler**:

nil	If the input buffer contains unscanned input, take the next character from there. Otherwise, take the next character from the stream.
:read	If the input buffer contains unscanned input, take the next character from there. Otherwise, if an activation blip or character is present, return that. Otherwise, enter the input editor.
:tyi	Take the next character from the stream.

If *eof-action* is not **nil**, an error is signalled when an end-of-file is encountered. Otherwise, the method returns **nil** when an end-of-file is encountered. The default for *eof-action* is **nil**.

:any-tyi-no-hang &optional *eof-action* of **si:interactive-stream** *Method*

Return the next character from the stream if it is immediately available. If no characters are immediately available, return **nil**. It is an error to call this method from inside the input editor (that is, if the value of **sys:rubout-handler** is not **nil**). *eof-action* is ignored. This is used by programs that continuously do something until a key is typed, then look at the key and decide what to do next.

:tyi &optional *eof-action* of **si:interactive-stream** *Method*

If called from outside the input editor, this is the same as **:any-tyi**, except that only integers and **nil** can be returned. Blips are discarded, unless the first element of the blip is **:mouse-button** and the second element is **#\mouse-r-1**; in this case, the method pops up a system menu. If called from inside the input editor with **:full-rubout** specified and if an activation blip is read when the input buffer is empty, the method causes control to be returned from the input editor.

:tyi-no-hang &optional *eof-action* of **si:interactive-stream** *Method*

This is like **:any-tyi-no-hang**, except that only integers and **nil** can be returned. Blips are discarded, unless the first element of the blip is **:mouse-button** and the second element is #**mouse-r-1**; in this case, the method pops up a system menu.

:list-tyi of **si:interactive-stream** *Method*

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

:untyi *ch* of **si:interactive-stream** *Method*

Return *ch* to the input buffer or the stream so that it will be the next character returned by **:any-tyi** or **:tyi**. *ch* must be the last character that was **:tyi**'ed, and it is illegal to do two **:untyi**'s in a row. Where *ch* is put depends on the value of the variable **sys:rubout-handler**. Following is a summary of actions for each possible value of **sys:rubout-handler**:

nil	If the input buffer contains scanned input, decrement the scan pointer. Otherwise, give *ch* back to the stream.
:read	Decrement the input editor scan pointer.
:tyi	Give *ch* back to the stream.

This method is used by parsers that look ahead one character, such as **zl:read**.

:listen of **si:interactive-stream** *Method*

Return **t** if there are any characters available to **:any-tyi** or **: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 of **si:interactive-stream** *Method*

Clear the input buffer and any input buffered by the stream. This flushes all the characters that have been typed at this stream, but have not yet been read.

:line-in &optional *leader* of **si:interactive-stream** *Method*

Reads characters from the stream and returns them as a string. If called from outside the input editor, reads characters until a #**return**, #**line**, or #**end** activation character is encountered. If called from inside the input editor, reads characters until a #**return** delimiter is encountered. The activation or delimiter character is not part of the returned string.

The method returns two values: the string and an *eof* flag. If the stream reaches end-of-file while reading characters, it returns the characters read as a string and returns a second value of **t**. Otherwise, the second returned value is **nil**.

If *leader* is an integer, the returned string has an array leader of length *leader*, and the fill pointer is set to the location in the string following the last one read. Otherwise, the string has no array leader.

Example:

This feature is useful for debugging programs that read from noninteractive streams. For example, the following function reads a single line-oriented record, in which the first line is a decimal number saying how many lines are in the rest of the record.

```
(defun read-record (&optional (stream standard-input))
  (loop repeat (parse-number (send stream :line-in) 0 nil 10.)
        collect (send stream :line-in)))
```

If this function is invoked on an interactive stream, the input editor is enabled automatically each time the **:line-in** message is sent, but it is not possible to edit across line boundaries. For example, once the number of lines in the record is typed, it isn't possible to change it.

```
(defun read-record (&optional (stream standard-input))
  (with-input-editing (stream)
    (loop repeat (parse-number (send stream :line-in) 0 nil 10.)
          collect (send stream :line-in))))
```

Wrapping a **with-input-editing** form around the body establishes a single input editing context for each record. **with-input-editing** has no effect when **stream** is a noninteractive stream, so this same function may used for reading from a file or reading from an interactive stream.

:string-in *eof string* &optional (*start* **0**) *end* of **si:interactive-stream** *Method*

Reads characters from the stream into *string*, using the substring delimited by *start* and *end*. *start* defaults to **0**, and *end* defaults to the length of the string.

eof specifies stopping actions:

Value	*Action*
nil	Reading characters into the string stops either when it has transferred the specified character count or when end-of-file is reached, whichever comes first. For a string with a fill pointer, sets the fill pointer to the location one greater than the last location into which a character was stored.

not **nil**	If end-of-file is encountered while trying to transfer a specific number of characters, signals **sys:end-of-file**, with the value of *eof* as the report string. If *eof* is **t**, a default report string is used.

The method returns two values. The first is the location in the string that is one greater than the last one into which a character was stored. The second value is **t** if end-of-file was reached, **nil** otherwise.

:string-line-in *eof string* &optional (*start* **0**) *end* of **si:interactive-stream** *Method*

:string-line-in is a combination of **:string-in** and **:line-in**. It reads a line of characters from the stream into *string*, using the substring delimited by *start* and *end*. *start* defaults to **0** and *end* to the length of *string*. If called from outside the input editor, reads characters until a #**return**, #**line**, or #**end** activation character is encountered. If called from inside the input editor, reads characters until a #**return** delimiter is encountered. The activation or delimiter character is not stored into *string*.

eof specifies stopping actions:

Value	*Action*
nil	Reading characters into the string stops when a delimiter is encountered, when the string is full, or when end-of-file is reached, whichever comes first. For a string with a fill pointer, sets the fill pointer to the location one greater than the last location into which a character was stored.
not **nil**	If end-of-file is encountered, signals **sys:end-of-file**, with the value of *eof* as the report string. If *eof* is **t**, a default report string is used.

The method returns three values:

- The location in *string* that is one greater than the last location into which a character was stored.
- **t** if end-of-file was reached, **nil** otherwise.
- **nil** if the entire contents of the line fit into the string or end-of-file was reached, otherwise **t**. If this value is **t**, as much of the line as possible was stored into the string and more is waiting to be read.

If the second and third values are both **nil**, a delimiter was read. If either is **t**, no delimiter was read.

4. Intercepted Characters

Interactive streams specially intercept some characters. Some are intercepted when some user process is about to read the character from a stream; others are intercepted as soon as they are typed. This section describes the first kind of interception. For information on asynchronously intercepted characters: See the section "Asynchronous Characters", page 154. See the section "Interactive-Stream Operations for Asynchronous Characters", page 19.

The value of the variable **sys:kbd-intercepted-characters** is a list of characters that are intercepted and not returned as input from the stream. These characters default to #**abort**, #**m-abort**, #**suspend**, and #**m-suspend**. Following are the standard actions to be taken when these characters are intercepted:

#\abort	Signal **sys:abort**
#\m-abort	Reset the current process
#\suspend	Call the **zl:break** function
#\m-suspend	Break to the Debugger

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 is intercepted automatically. Most interactive programs just set up restart handlers for **sys:abort**. But some programs may want to do something specific when the user presses ABORT (or SUSPEND).

You can replace the system default action by binding the variable **sys:kbd-intercepted-characters**. By default, this variable is bound to the value of **sys:kbd-standard-intercepted-characters**. If you want the system to intercept only the standard abort characters, you can bind this variable to the value of **sys:kbd-standard-abort-characters**. If you want the system to intercept only the standard break characters, you can bind this variable to the value of **sys:kbd-standard-suspend-characters**.

sys:kbd-intercepted-characters *Variable*

The value is a list of characters that are intercepted when they are read from an interactive stream.

Bind this variable when you want to change the characters that the system intercepts. The default value is the value of
sys:kbd-standard-intercepted-characters:
(#\abort #\m-abort #\suspend #\m-suspend).
sys:kbd-intercepted-characters is reset to this value on warm booting.
You can bind **sys:kbd-intercepted-characters** to any subset of the default

value, but you cannot include any characters that are not members of the default value. If you want the system to intercept only the standard abort characters, bind **sys:kbd-intercepted-characters** to the value of **sys:kbd-standard-abort-characters**. If you want the system to intercept only the standard break characters, bind **sys:kbd-intercepted-characters** to the value of **sys:kbd-standard-suspend-characters**.

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

The value is a list of characters that is the default value of **sys:kbd-intercepted-characters**. The default value is **(#\abort #\m-abort #\suspend #\m-suspend)**. This is a constant. If you want to change the characters that the system intercepts, bind **sys:kbd-intercepted-characters**, not **sys:kbd-standard-intercepted-characters**.

sys:kbd-standard-abort-characters *Variable*

The value is a list of characters that are the default abort characters intercepted by the system. The default value is **(#\abort #\m-abort)**. This is a constant. If you want the system to intercept only the standard abort characters, bind **sys:kbd-intercepted-characters** to the value of **sys:kbd-standard-abort-characters**.

sys:kbd-standard-suspend-characters *Variable*

The value is a list of characters that are the default suspend characters intercepted by the system. The default value is **(#\suspend #\m-suspend)**. This is a constant. If you want the system to intercept only the standard suspend characters, bind **sys:kbd-intercepted-characters** to the value of **sys:kbd-standard-suspend-characters**.

5. Interactive-Stream Operations for Asynchronous Characters

The keyboard process intercepts some characters as soon as they are typed: See the section "Asynchronous Characters", page 154. All interactive streams maintain a list of characters to be handled asynchronously. Remote terminals, however, do not handle asynchronous characters.

You can set up your own handling of asynchronous characters by using the **:asynchronous-character-p**, **:handle-asynchronous-character**, **:add-asynchronous-character**, and **:remove-asynchronous-character** messages and the **:asynchronous-characters** init option for **si:interactive-stream.**

:asynchronous-characters *spec-list* (for **si:interactive-stream)** *Init Option*

Specifies the asynchronous characters for the stream. *spec-plist* is a list of specs, each of which is a list containing a character name and a function spec. The following default asynchronous characters are defined for **si:interactive-stream**:

```
(:default-init-plist
 :asynchronous-characters
 '((#\c-abort tv:kbd-asynchronous-intercept-character)
   (#\c-m-abort tv:kbd-asynchronous-intercept-character)
   (#\c-suspend tv:kbd-asynchronous-intercept-character)
   (#\c-m-suspend tv:kbd-asynchronous-intercept-character)))
```

:asynchronous-character-p *character* of **si:interactive-stream** *Method*

Returns non-null when *character* is an asynchronous character for this stream.

:handle-asynchronous-character *character* of **si:interactive-stream** *Method*

Finds the function associated with *character* in the asynchronous characters list. It calls the function with two arguments, *character* and **self.** This is mainly for use by the Keyboard Process although user processes can use it also.

:add-asynchronous-character *character handler* of **si:interactive-stream** *Method*

Defines a new asynchronous character for the stream. *character* is the character to be treated asynchronously and *handler* is the function to be called (with two arguments, *character* and **self**). It checks the types of the arguments.

:remove-asynchronous-character *character* of **si:interactive-stream** *Method*

Removes an asynchronous character from the list for the stream.

6. Interactive Streams and Mouse-Sensitive Items

Some windows support mouse sensitivity. They can display representations of items in such a way that moving the mouse onto the item causes it to be highlighted, and clicking the mouse on the item does something with the item. One example is the basic mouse-sensitive items facility: See the section "The Mouse-Sensitive Items Facility", page 323. (Note that the referenced section and the facilities discussed here are based on static windows. Mouse sensitivity is a built-in feature of Dynamic Windows and the SemantiCue input system. See the section "Introduction to Basic User Input Facilities" in *Programming the User Interface, Volume A*.)

The fundamental message that creates and displays a mouse-sensitive item is **:item**. All interactive streams support this message, whether or not they support mouse sensitivity. If they do not support mouse sensitivity, they just display a printed representation of the item.

Any interactive stream can also display an ordered list of items, using the function **si:display-item-list**. This displays each item by sending an **:item** message to the stream.

:item *type item* &rest *format-args* of **si:interactive-stream** *Method*

Creates and displays a (possibly mouse-sensitive) item of type *type* on the stream. If the stream does not support mouse-sensitivity, this just ignores *type* and displays *item* on the stream. If *format-args* are supplied, they are a **zl:format** control string and control args to be used to display the item. Otherwise, the item is displayed by calling **princ** with a first argument of *item*.

si:display-item-list *stream type list* &optional *item-string* (*order-columnwise* **t**) *Function*

Displays a list of items on *stream* in evenly spaced columns. *stream* must be interactive. If it supports mouse sensitivity, the items displayed are also made mouse sensitive.

list is a list of items to be displayed. Each item in the list is displayed by sending the stream an **:item** message with *type* as the first argument. If the item is not itself a list, the item is the second argument to the **:item** message.

If the item to be displayed is a list, the arguments to the **:item** message depend on *item-string*. If *item-string* is not **nil**, the second argument to the **:item** message is the first element of the item. If *item-string* is **nil**, the item should be an alist whose car is a string to be displayed and whose cdr is the item itself. In this case, the second argument to the **:item** message is the cdr of the item, the third argument is "~a", and the fourth argument is the car of the item. The default for *item-string* is **nil**.

If *order-columnwise* is not **nil**, the items are ordered down columns. If *order-columnwise* is **nil**, the items are ordered across rows. The default is **t**.

7. The Input Editor Program Interface

7.1 How the Input Editor Works

The input editor is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to let you edit minor mistakes in typein. At the same time, it is not supposed to get in the way; Lisp is to see the input as soon as you have typed a syntactically complete form. The definition of "syntactically complete form" depends on the function that is reading from the stream; for **zl:read**, it is a Lisp expression. This section describes the general protocol used for communication between the input editor and reading functions such as **zl:read** and **zl:readline.**

By *reading function* we mean a function that reads a number of characters from a stream and translates them into an object. For example, **zl:read** reads a Lisp expression and returns an object. **zl:readline** reads a line of characters and returns a string as its first value. Reading functions do not include the more primitive **:tyi** and **:any-tyi** stream operations, which take and return one character or blip from the stream.

The tricky thing about the input editor is the need for it to figure out when you are all done. The idea of an input editor is that as you type in characters, the input editor saves them up in an *input buffer* so that if you change your mind, you can edit them and replace them with different characters. However, at some point the input editor has to decide that the time has come to stop putting characters into the input buffer and let the reading function start processing the characters. This is called "activating".

The right time to activate depends on the function calling the input editor, and determining it may be very complicated. If the function is **zl:read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently defined reader macros do, and so on. The input editor should not have to know how to parse the characters in the input buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The input editor interface is organized so that the calling function can do all the parsing, while the input editor does all the handling of editing commands, and the two are kept completely separate.

Following is a summary of how the input editor works. The input editor used to be called the rubout handler, and some operations and variables still have "rubout-handler" in their names.

When a reading function is called to read from a stream that supports the **:input-editor** operation, that function "enters" the input editor. It then goes ahead **:tyi**'ing characters from the stream. Because control is inside the input editor, the stream echoes these characters so the user can see the input. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the input editor it is also handing it the responsibility for echoing). The input editor is also saving all these characters in the input buffer, for reasons disclosed in the following paragraph. When the reading function decides it has enough input, it returns and control "leaves" the input editor. That was the easy case.

If you press RUBOUT or a keystroke that represents another editing command, the input editor processes the command and lets you insert characters before the last one in the line. The input editor modifies the input buffer and the screen accordingly. Then, when you type the next nonediting character at the end of the line, a **throw** is done, out of all recursive levels of **zl:read**, reader macros, and so forth, back to the point where the input editor was entered. Now the **zl:read** is tried over again, rereading all the characters you had typed and not rubbed out, but not echoing them this time. When the saved characters have been exhausted, additional input is read from you in the usual fashion.

The input editor has options that can cause the **throw** to occur at other times as well. With the **:activation** option, when you type an activation character a **throw** occurs, a rescan is done if necessary, and a final blip is returned to the reading function. With the **:preemptable** and **:command** options, a blip or special character in the input stream causes control to be returned from the input editor immediately, without a rescan. These options let you process mouse clicks or special keystroke commands as soon as they are read.

The effect of all this is a complete separation of the functions of input editing and parsing, while at the same time mingling the execution of these two functions in such a way that input is always "activated" at just the right time. It does mean that the parsing function (in the usual case, **zl:read** and all macro-character definitions) must be prepared to be thrown through at any time and should not have nontrivial side-effects, since it may be called multiple times.

If an error occurs while inside the input editor, the error message is printed and then additional characters are read. When you press RUBOUT, it rubs out the error message as well as the last character. You can then proceed to type the corrected expression; the input is reparsed from the beginning in the usual fashion.

7.2 Invoking the Input Editor

The variable **sys:rubout-handler** indicates the current state of input editing. This variable is not **nil** if the current process is already inside the input editor.

sys:rubout-handler *Variable*

Indicates the status of input editing within a process.

This variable is used internally by the **:input-editor** method and the input editor. It should not be necessary for user programs to examine its value since the **with-input-editing** special form is provided for this purpose.

The possible values for this variable are:

Value	*Meaning*
nil	The process is outside the input editor.
:read	The process is inside the **:input-editor** method.
:tyi	The process is inside the editing portion of the **:tyi** method.

The input editor is invoked on a stream when the stream receives an **:input-editor** message. The **:input-editor** and **:tyi** methods of **si:interactive-stream** contain the code of the input editor. The **:input-editor** method initializes the input editor, establishes its **catch**, and then calls back to the reading function with **sys:rubout-handler** bound to **:read**. When the reading function sends the **:tyi** or **:any-tyi** message, input is taken from the input buffer. If no input is available, the editing or **:tyi** portion of the input editor is invoked, and **sys:rubout-handler** is bound to **:tyi**.

The first argument to the **:input-editor** message is the function that the input editor should call to do the reading, and the rest of the arguments are passed to that function. If the reading function returns normally, the values returned by the **:input-editor** message are just those returned by the reading function. If the input editor returns by throwing out of the reading function, the return values depend on which option caused the input editor to throw: See the option :full-rubout, page 30. See the option **:preemptable**, page 34. See the option **:command**, page 34.

The input editor can take a series of options. These are specified dynamically by the special forms **with-input-editing-options** and **with-input-editing-options-if**. For a description of the options: See the section "Input Editor Options", page 30.

with-input-editing-options *options* &body *body* *Special Form*

Specifies input editing options and executes *body* with those options in effect. The scope of the option specifications is dynamic.

options is a list of input editor option specifications. Each element is a list whose car is an option-name specification and whose cdr is a list of forms to be evaluated to yield "arguments" for the option. The option-name specification is a keyword symbol or a list whose car is a keyword symbol. The symbol is the name of the option.

If the option-name specification is a list and if the symbol **:override** is an element of the cdr of the list, this option specification overrides any higher-level specifications for this option. Otherwise, the specification for each option that is dynamically outermost (that is, the specification from the highest-level caller) is in effect during the execution of *body*.

with-input-editing-options returns whatever values *body* returns.

In the following example, the user is prompted for a Lisp expression. Two input editor options are specified. The first says that the caller is also willing to receive mouse or menu blips. The second specifies a prompt.

```
(with-input-editing-options ((:preemptable :blip)
                             (:prompt "Form: "))
  (read))
```

In the following example, the user is prompted for a line of text. The text may be activated by any of the characters RETURN, END, or TRIANGLE. This might be useful if activating with TRIANGLE meant something different from activating with RETURN. This example also demonstrates the use of **:override** to make this **:activation** specification override any higher-level **:activation** specifications.

```
(with-input-editing-options
    (((:activation :override) 'memq '(#\return #\end #\triangle)))
  (prompt-and-read :string "Name: "))
```

For a list of input editor options: See the section "Input Editor Options", page 30. See the special form **with-input-editing-options-if**, page 26.

with-input-editing-options-if *cond options* &body *body* *Special Form*

Executes *body*, possibly with specified input editing options in effect. The scope of the option specifications is dynamic.

cond is a form to be evaluated at run-time. If *cond* returns non-**nil**, the specified input editor options are in effect during the execution of *body*.

options is a list of input editor option specifications. Each element is a list whose car is an option-name specification and whose cdr is a list of forms to be evaluated to yield "arguments" for the option. The option-name specification is a keyword symbol or a list whose car is a keyword symbol. The symbol is the name of the option.

If the option-name specification is a list and if the symbol **:override** is an element of the cdr of the list, this option specification overrides any higher-level specifications for this option. Otherwise, the specification for each option that is dynamically outermost (that is, the specification from the highest-level caller) is in effect during the execution of *body*.

with-input-editing-options-if returns whatever values *body* returns.

For a list of input editor options: See the section "Input Editor Options", page 30. See the special form **with-input-editing-options**, page 25.

This example illustrates the use of the **:command**, **:preemptable**, and **:prompt** input editor options. It is a simple command loop that reads different kinds of commands -- typed Lisp expressions, single-keystroke commands, and mouse clicks. The Lisp expressions are read using the **read-or-end** function. You can provide four kinds of input:

Input	*Action*
END	Exit the command loop
Lisp form	Print form on next line
Mouse click	Display type of click and mouse coordinates
Single-key command	Display keystroke

The predicate for detecting a single-keystroke command simply checks for the Super bit. In a more complex program, it might look up the character in a command table.

```
(defun command-char-p (c) (char-bit c :super))
```

```
(defun command-loop ()
  (loop
    do (multiple-value-bind (value flag)
           (with-input-editing-options
               ((:command 'command-char-p)
                (:preemptable :blip)
                (:prompt "Command loop input: "))
             (read-or-end))
         (selectq flag
           (:end
            (format t "Done")
            (return t))
           (:blip
            (selectq (car value)
              (:mouse-button
               (destructuring-bind (click nil x y) (cdr value)
                 (format t "~C click at ~D, ~D" click x y)))
              (otherwise (format t "Random blip -- ~S" value))))
           (:command
            (format t "Execute ~:C command" (second value)))
           (otherwise
            (format t "~&Value is ~S" value))))))
```

To write a reading function that invokes the input editor, you should use the **with-input-editing** special form instead of sending the **:input-editor** message directly. Such functions as **zl:read** and **zl:readline** use this special form to provide input editing.

with-input-editing (&optional *stream keyword*) &body *body* *Special Form*

Provides a convenient way of invoking the input editor for use by a reading function. It establishes a context in which input editing should be provided. Use **with-input-editing** instead of sending an **:input-editor** message directly.

Both "arguments" are optional. *stream* is the stream from which characters are read; if *stream* is not provided or is **nil**, ***standard-input*** is used.

keyword determines the activation characters for the input editor:

Value	*Activation characters*
nil	None (unless specified at a higher level). This is the default.

:end-activation	**#\end**
:line-activation	**#\end**, **#\return**, and **#\line**
:line	**#\end**, **#\return**, and **#\line**. In addition, a Newline is echoed after the reading function returns.

To supply other input editor options: See the special form with-input-editing-options, page 25. See the special form with-input-editing-options-if, page 26.

with-input-editing defines an internal lexical closure with *body* as its body. When the **with-input-editing** form is evaluated from outside the input editor, the stream is sent an **:input-editor** message if it handles it. The argument to the **:input-editor** message is the lexical closure, except that if the **:line** keyword is supplied, **with-input-editing** also arranges to echo a Newline after the lexical closure returns. If the **with-input-editing** form is evaluated from inside the input editor or if the stream does not handle the **:input-editor** message, the lexical closure is called instead.

with-input-editing returns whatever values *body* returns.

The following example defines a simple sentence parser.

```
(defun read-sentence (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream)
      (loop named sentence
            with sentence = nil
            for word = (make-array 20. :type art-string :fill-pointer 0)
            do (loop for char = (send stream :tyi)
                     do
                 (cond ((memq char '(#\space #\return #/. #/? #/,))
                        (if (not (equal word ""))
                            (push word sentence))
                        (selectq char
                          ((#\space #\return #/,)
                           (return))
                          (#\.
                           (push :period sentence)
                           (return-from sentence (nreverse sentence)))
                          (#\?
                           (push :question-mark sentence)
                           (return-from sentence (nreverse sentence)))))
                       (t (array-push-extend word char))))))))
```

7.3 Input Editor Options

The input editor can take a series of options, specified by the special forms **with-input-editing-options** and **with-input-editing-options-if**. Following are descriptions of the options.

:full-rubout *token* *Option*

If the user rubs out all the characters that were typed, control is returned from the input editor immediately. Two values are returned: **nil** and *token*. If the user does not rub out all the characters, the input editor propagates multiple values back from the function that it calls, as usual. In the absence of this option, the input editor simply waits for more characters to be typed and ignores any additional rubouts.

:pass-through &rest *characters* *Option*

The characters in *characters* are not to be treated as special by the input editor. This option is used to pass format effectors (such as HELP or CLEAR INPUT) through to the reading function instead of interpreting them as input editor commands. **:pass-through** is allowed only for characters with no modifier bits set, that is, for character codes 0 through 377 (octal). For characters that have modifier bits set and must be visible to the reading function, use **:do-not-echo** or **:activation**.

:prompt &rest *prompt-option* *Option*

When it is time for the user to be prompted, the input editor displays *prompt-option*. *prompt-option* can have one element, which can be **nil**, a string, a function, or a symbol other than **nil**; or it can have more than one element: See the section "Displaying Prompts in the Input Editor", page 36.

The difference between **:prompt** and **:reprompt** is that the latter does not display the prompt when the input editor is first entered, but only when the input is redisplayed (for example, after a screen clear). If both options are specified, **:reprompt** overrides **:prompt** except when the input editor is first entered.

:reprompt &rest *prompt-option* *Option*

When it is time for the user to be reprompted, the input editor displays *prompt-option*. *prompt-option* can have one element, which can be **nil**, a string, a function, or a symbol other than **nil**; or it can have more than one element: See the section "Displaying Prompts in the Input Editor", page 36.

Unlike **:prompt**, **:reprompt** displays the prompt only when input is redisplayed (for example, after a screen clear), not when the input editor is first entered. If both **:prompt** and **:reprompt** are specified, **:reprompt** overrides **:prompt** except when the input editor is first entered.

:complete-help &rest *help-option* *Option*

When the user presses HELP, the input editor types out a message determined by *help-option.* None of the standard input editor help is displayed. If a **:brief-help** option has been specified, it overrides **:complete-help**. **:complete-help** overrides **:merged-help** and **:partial-help**.

help-option can have one element, which can be a string, a function, or a symbol; or it can have more than one element. For an explanation: See the section "Displaying Help Messages in the Input Editor", page 37.

This option is intended for programs that supply their own input editor help messages.

:partial-help &rest *help-option* *Option*

When the user presses HELP, the input editor first types out a message determined by *help-option.* It then types out a message describing how to invoke input editor commands and other information about the stream. If a **:brief-help, :complete-help,** or **:merged-help** option has been specified, it overrides **:partial-help.**

help-option can have one element, which can be a string, a function, or a symbol; or it can have more than one element. For an explanation: See the section "Displaying Help Messages in the Input Editor", page 37.

This option is intended for use when inexperienced users might be typing to the input editor. Often *help-option* gives some information about the program to which the user is typing and what the user can do to exit from it.

:merged-help *function* &rest *arguments* *Option*

When the user presses HELP, the input editor types out a message determined by the arguments. *function* is a function that takes at least two arguments. The input editor calls the function to print the help message. The first argument is the stream. The second argument is a continuation (a list) to print a standard message describing how to invoke input editor commands and other information about the stream. When the function wants to print this message, it should apply the car of the continuation to the cdr. If any *arguments* are supplied, they are the remaining arguments to the function.

If a **:brief-help** or **:complete-help** option has been specified, it overrides **:merged-help. :merged-help** overrides **:partial-help.**

This option is intended for programs that want to decide when and where to display their own help messages and the standard help message.

:brief-help &rest *help-option* *Option*

When the user presses HELP, the input editor displays a message determined by *help-option* on the same line as the typein. The message is displayed in the default typeout font, and none of the usual conventions about input editor typeout apply. **:brief-help** overrides **:complete-help**, **:merged-help**, and **:partial-help**.

help-option can have one element, which can be a string, a function, or a symbol; or it can have more than one element. For an explanation: See the section "Displaying Help Messages in the Input Editor", page 37.

This option is intended for programs like **fquery** that need to supply only a brief help message, usually about expected typein.

:initial-input *string* &optional *begin end cursor-position* *Option*

When the input editor is entered, *string* is inserted into the input buffer as if the user had typed it. The user can edit the string before activating. *begin* and *end* are indices into *string* and mark the portion of the string to be copied into the input buffer. *begin* defaults to **0**; *end* defaults to **(zl:array-active-length** *string***)**. *cursor-position* is an index into the string where the cursor should initially be placed. The default is to place the cursor at the end of the portion of the string copied into the input buffer. *string* can be **nil**, which is the same as not specifying the option.

In the following example, the user is prompted for a line of text. The input buffer initially contains the name of the user, and the cursor is placed at the beginning of the input buffer.

```
(with-input-editing-options
    ((:initial-input fs:user-personal-name nil nil 0))
  (prompt-and-read :string "Full name: "))
```

Placing a string in the input buffer is one style of input defaulting. Another style leaves the input buffer empty but allows a default to be yanked with c-m-Y. See the option **:input-history-default**, page 32.

:input-history-default *string* *Option*

Specifies *string* as the default to be yanked by c-m-Y. *string* is temporarily placed at the head of the input history. If the user types c-m-Y m-Y, the true first element of the input history is yanked. c-m-0 c-m-Y shows *string* at the head of the input history, and the entries in the input history are shifted down by one.

In the following example, the user is prompted for a line of text. The input buffer is initially empty, but the c-m-Y command yanks a default, which is the name of the user.

```
(with-input-editing-options
    ((:input-history-default fs:user-personal-name))
  (prompt-and-read :string "Full name: "))
```

This option is used by the **:pathname** option for **prompt-and-read**.

:blip-handler *function* *Option*

Specifies a function to handle blips received while inside the input editor. *function* must be a function of two arguments. The first argument is the blip; the second argument is the stream that received the blip. The handler is invoked when the input editor receives a blip. If the handler returns non-**nil**, no further action is taken. If it returns **nil** and a **:preemptable** option is in effect, the actions specified by that option are taken. Otherwise, the default blip handler is invoked.

In the following example, the user is prompted for a line of text. While entering this text, the user may also click the left or middle mouse buttons. If the left mouse button is clicked, the coordinates of the mouse with respect to the window are inserted into the input buffer. If the middle button is clicked, the name of the window is inserted.

```
(defun example-blip-handler (blip ignore)
  (destructuring-bind (type click window x y) blip
    (and (eq type :mouse-button)
         (selectq click
           (#\mouse-l-1
            (si:ie-insert-string (format nil " ~D ~D" x y))
            t)
           (#\mouse-m-1
            (si:ie-insert-string (format nil " ~A" window))
            t))))))

(with-input-editing-options ((:blip-handler 'example-blip-handler))
  (prompt-and-read :string "Blip handler test: "))
```

si:ie-insert-string is an internal function for inserting a string into the input buffer. Since the language for writing input editor commands has not been formalized, this example might not work in a later release.

:do-not-echo &rest *characters* *Option*

The characters in *characters* are interpreted as activation characters and are not echoed. The comparison is done with **char=**, not **char-equal**, so that the control and meta bits are not masked off. The characters are not inserted into the input buffer and are not interpreted as input editor commands. When one of these characters is typed, the final **:tyi** value returned is the character, not a blip.

This option exists only for compatibility with earlier releases. New programs should use the **:activation** option.

:activation *function* &rest *arguments* *Option*

For each character typed, the input editor invokes *function* with the character as the first argument and *arguments* as the remaining arguments. If the function returns **nil**, the input editor processes the character as it normally would. Otherwise, the cursor is moved to the end of the input buffer, a rescan of the input is forced (if one is pending), and the blip (**:activation** *character numeric-arg*) is returned by the final sending of the **:any-tyi** message to the stream. Activation characters are not inserted into the input buffer, nor are they echoed by the input editor. It is the responsibility of the reading function to do any echoing. For instance, **zl:readline**, not the input editor, types a Newline at the end of the input buffer when RETURN, END, or LINE is pressed.

:preemptable *token* *Option*

A blip in the input stream causes control to be returned from the input editor immediately. Two values are returned: the blip and *token*, which is usually a keyword symbol. Any unscanned input typed before the blip remains in the input buffer, available to the next read operation from the stream.

:no-input-save *Option*

The input editor does not save the scanned contents of the input buffer on the input history when returning from the reading function. This is intended for use by functions such as **fquery** that use the input editor to ask simple questions whose responses are not worth saving. **zl:yes-or-no-p** uses **:no-input-save** by default.

:command *function* &rest *arguments* *Option*

This option is used to implement nonediting single-keystroke commands. For each character typed, the input editor invokes *function* with the character as the first argument and *arguments* as the remaining arguments. If the function returns **nil**, the input editor processes the character as it normally would. Otherwise, control is returned from the input editor immediately. Two values are returned: a blip of the form (**:command** *character numeric-arg*) and the keyword **:command**. Any unscanned input typed before the command character remains in the input buffer, available to the next read operation from the stream.

:editor-command &rest *command-alist* *Option*

This option lets you specify your own input editor editing commands. Each element of *command-alist* is a cons whose car is a character and whose cdr is a symbol or a list. If the cdr is a symbol, it is a function to be called with no arguments when the user types the associated character. If the cdr is a list, the car of the list is a function to be applied to the cdr of the list when the user types the associated character. The function can examine the internal special variables that describe the state of the input editor.

If **:editor-command** specifies a command that is invoked by the same character as one of the standard input editor editing commands, the command specified by **:editor-command** overrides the standard command.

:input-wait &optional *whostate function* &rest *arguments* *Option*

When the input editor waits for input, it sends the stream an **:input-wait** message with the arguments to the **:input-wait** option as arguments. In addition, unless the **:suppress-notifications** option has been specified, **:input-wait** returns when a notification is received. See the message **:input-wait** in *Reference Guide to Streams, Files, and I/O.*

:input-wait-handler *function* &rest *arguments* *Option*

When the input editor is waiting for input it sends the stream an **:input-wait** message. After **:input-wait** returns, the input editor applies *function* to *arguments.* The input editor does not process the input or display the notification until *function* returns.

:suppress-notifications *flag* *Option*

If a notification is received while in the input editor, and *flag* is supplied as **nil**, the input editor itself handles the notification, regardless of any other way you have specified that notifications should be handled. If *flag* is **t**, notifications are handled in the input editor the same way they would be handled if you were not in the input editor. That is, the input editor does not handle the notification itself.

:notification-handler *function* &rest *arguments* *Option*

If a notification is received while in the input editor, *function* is called to handle it. *function* should take at least one argument, the notification (as returned by the **:receive-notification** message to the stream). *arguments* are the remaining arguments to *function. function* can do anything it wants with the notification. To display the notification, *function* would usually call **sys:display-notification.**

If this option is not specified, notifications appear one after the other using **:insert**-style typeout.

Following are two simple examples of notification handlers. The first handler assumes that you want each notification to overwrite the previous one. The second handler assumes that you want them to appear one after another. ***window*** should be bound to a window and ***stream*** to a stream where you want the notifications to appear.

```
(defun my-notification-handler-1 (notification)
  (send *window* :clear-window)
  (sys:display-notification *window* notification :window))

(defun my-notification-handler-2 (notification)
  (sys:display-notification *stream* notification :stream))
```

7.4 Displaying Prompts in the Input Editor

The input editor options **:prompt** and **:reprompt** and the functions **zl:readline-no-echo** and **sys:read-character** take *prompt* arguments that let you specify an input editor prompt. *prompt* can be **nil**, a string, a function, a symbol other than **nil**, or a list (for the input editor options, the list is an **&rest** argument):

nil — No prompt is displayed.

string — A **zl:format** control string to be passed to **zl:format** with one argument, the stream on which the prompt is displayed.

function or symbol other than **nil** — A function to display the prompt. The function should take two arguments: the first is the stream on which the prompt is displayed, and the second is a keyword that indicates the origin of the function call.

list — If the first element is **nil**, no prompt is displayed. If the first element is a string, it is a **zl:format** control string to be passed to **zl:format** with the remaining elements of the list as arguments. If the first element is a function or a symbol other than **nil**, it is a function to display the prompt. The first argument to the function is the stream on which the prompt is displayed. The second argument is a keyword that indicates the origin of the function call. The remaining arguments are the remaining elements of the list.

When a function is called to display the prompt, the second argument to the function is a keyword that indicates the origin of the function call:

Keyword	*Function called from*
:prompt	**:input-editor** method of **si:interactive-stream**, when the input editor is entered

:restore	**:restore-input-buffer** method of **si:interactive-stream**
:finish-typeout	**:finish-typeout** method of **si:interactive-stream**
:refresh	Body of the input editor, when the user presses REFRESH
:erase-typeout	Body of the input editor, when the user presses PAGE

7.5 Displaying Help Messages in the Input Editor

The input editor options **:brief-help**, **:partial-help**, and **:complete-help** and the functions **zl:readline-no-echo** and **sys:read-character** take *help* arguments that let you specify input editor help messages. *help* can be a string, a function, a symbol, or a list (for the input editor options, the list is an **&rest** argument):

string
: A **zl:format** control string to be passed to **zl:format** with one argument, the stream on which the help message is displayed.

function or symbol
: A function to display the help message. The function should take one argument, the stream on which the help message is displayed.

list
: If the first element is a string, it is a **zl:format** control string to be passed to **zl:format** with the remaining elements of the list as arguments. If the first element is a function or a symbol, it is a function to display the help message. The first argument to the function is the stream on which the help message is displayed, and the remaining arguments are the remaining elements of the list.

7.6 Examples of Use of the Input Editor

This series of examples shows several different ways of using the input editor, gradually increasing in complexity. The examples are also available in the file sys: examples; interaction.lisp.

We refer to functions whose names begin with "read-" as "reading functions" or "readers", since they read individual characters and construct a Lisp object as a returned value. Examples of readers the Lisp system provides are **read**, **readline**, and **read-delimited-string**. **read** returns Lisp objects of many types. **readline** and **read-delimited-string** return strings.

read-two-lines-1 reads two lines of input from the console. You type each line in its own editing context. After you enter the first line by pressing RETURN, LINE, or END, you can no longer rub out or otherwise edit any of the characters in the first line. You can type and edit only the second line at that point.

```
(defun read-two-lines-1 () (list (readline) (readline)))
```

read-two-lines-2 lets you edit both lines in a single context by using the **with-input-editing** special form. Even after entering the first line you can edit it. For example, the m-< input editor command moves the cursor to the first character of the first line. **read-two-lines-2** also adds a stream parameter so that you can read from different streams without having to bind ***standard-input***. You can also use this function for reading from noninteractive streams, such as file streams.

```
(defun read-two-lines-2 (&optional (stream *standard-input*))
  (with-input-editing (stream) (list (readline stream) (readline stream))))
```

read-two-lines-3 demonstrates the use of the **:prompt** input editor option and the **:end-activation** option for **with-input-editing**. When you invoke this function on an interactive stream you receive a prompt. This prompt is redisplayed if typeout to the stream occurs. This might happen if you press HELP or the window receives a notification.

The **:end-activation** option defines **#\end** as an activation character. This lets you activate previous input to **read-two-lines-3**, after yanking and editing it, by pressing END. The **:prompt** and **:end-activation** options have no effect on the behavior of the function for noninteractive streams.

```
(defun read-two-lines-3 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type two lines: "))
    (with-input-editing (stream :end-activation)
      (list (readline stream) (readline stream)))))
```

read-n-lines is like **read-two-lines** except that you specify the number of lines to be read using the **n-lines** argument. It also uses a prompt function instead of a string to generate the prompt.

```
(defun read-n-lines-prompt (stream ignore n-lines)
  (format stream "Type ~R line~:P:~%" n-lines))

(defun read-n-lines (n-lines &optional (stream *standard-input*))
  (with-input-editing-options ((:prompt 'read-n-lines-prompt n-lines))
    (with-input-editing (stream :end-activation)
      (loop repeat n-lines collect (readline stream)))))
```

Next is an example of a simple sentence parser. It builds a list of strings and symbols that represent the words and punctuation marks of the sentence. A sentence may be any number of lines long. It is delimited by a period or a question mark. Words are delimited by a space, newline, or punctuation mark.

This is also an example of a reading function written entirely in terms of **:tyi** as the primitive input operation.

```
(defun read-sentence-1 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream)
      (loop named sentence
            with sentence = nil
            for word = (make-array 20. :type art-string :fill-pointer 0)
            do (loop for char = (send stream :tyi)
                     do
                 (cond ((memq char '(#\space #\return #/. #/? #/,))
                        (if (not (equal word ""))
                            (push word sentence))
                        (selectq char
                          ((#\space #\return #/,)
                           (return))
                          (#\.
                           (push :period sentence)
                           (return-from sentence (nreverse sentence)))
                          (#\?
                           (push :question-mark sentence)
                           (return-from sentence (nreverse sentence)))))
                       (t (array-push-extend word char))))))))
```

Following is a different sentence parser that calls **read-delimited-string** to accumulate characters into a string. It uses the **:end-activation** option for **with-input-editing** so that previous input to **read-sentence-2** can be yanked, edited, and activated using the END key. When it detects incorrect uses of punctuation, it calls **zl:parse-ferror** to signal an error caught by the input editor.

```
(defun read-sentence-2 (&optional (stream *standard-input*))
  (with-input-editing-options ((:prompt "Type a sentence:  "))
    (with-input-editing (stream :end-activation)
      (loop with sentence = nil
            do (multiple-value-bind (word nil delimiter)
                   (read-delimited-string
                     '(#\space #\return #/. #/? #/, #/: #/;) stream)
                 (if (not (equal word ""))
                     (push word sentence))
                 (cond ((memq delimiter '(#\space #\return)))
                       ((null sentence)
                        (if (eq delimiter #\end)
                            (return nil)
                            (sys:parse-ferror
                              "The punctuation mark /"~C/" occurred at the ~
                             beginning of the sentence."
                              delimiter)))
                       ((symbolp (car sentence))
                        (sys:parse-ferror
                          "The punctuation mark /"~C/" was typed after a ~@∧."
                          delimiter (car sentence)))
                       (t (selectq delimiter
                            (#/,
                             (push ':comma sentence))
                            (#/:
                             (push ':colon sentence))
                            (#/;
                             (push ':semicolon sentence))
                            (#/.
                             (push ':period sentence)
                             (return (nreverse sentence)))
                            (#/?
                             (push ':question-mark sentence)
                             (return (nreverse sentence))))))))))
```

Sometimes an error in parsing is detected not by the function that invokes the input editor, but by some function that it calls. In the next example, **read-time** invokes **time:parse-universal-time** to do its parsing. If we did not use the **condition-case** form in **read-time**, we would enter the Debugger when **time:parse-universal-time** encountered incorrect input. The **condition-case** form encapsulates the original error in one of flavor **zl:parse-ferror** so that the input editor catches it. Alternately, we could define **time:parse-error** to be a subflavor of **sys:parse-error**.

```
(defun read-time (&optional (stream *standard-input*))
  (with-input-editing (stream :line)
    (let ((string (readline-or-nil stream)))
      (when string
        (condition-case (error)
            (time:parse-universal-time string)
          (time:parse-error
            (sys:parse-ferror "~A" error)))))))
```

7.7 Input Editor Messages to Interactive Streams

:input-editor *read-function* &rest *read-args* of **si:interactive-stream** *Method*

Apply *read-function* to *read-args* after invoking the input editor. For more information: See the section "The Input Editor Program Interface", page 23.

Normally a program does not send this message itself; it uses the special form **with-input-editing**. See the special form **with-input-editing**, page 28.

:start-typeout *type* &optional *spacing* of **si:interactive-stream** *Method*

Informs the input editor that typeout to the window will follow. The word "typeout" is used in the name of this message because this is very similar to typeout in the editor, even though typeout windows are not actually used. *type* can be one of the following keywords:

Keyword	*Action*
:insert	Typeout is inserted before the current input, as is done with notifications or input editor documentation.
:overwrite	Like **:insert**, but the next time **:insert** or **:overwrite** typeout is performed, this typeout is overwritten.
:append	Typeout appears after the current input, which remains visible before the typeout. This is the style used by **zl:break**.
:temporary	Typeout appears after the current input and is erased after the user types a character.
:clear-window	The window is cleared, and typeout appears at the top.

spacing can be one of the following keywords:

Keyword	*Action*
:none	No spacing before typeout.
:fresh-line	Typeout begins at the beginning of a line.
:blank-line	A blank line precedes typeout.

If *spacing* is not specified, a default that depends on *type* is computed.

si:*typeout-default* *Variable*

Controls the style of typeout performed by the input editor. Permissible values are the keywords acceptable as the *type* argument to the **:start-typeout** method of **si:interactive-stream**. These are **:insert**, **:overwrite**, **:append**, **:temporary**, and **:clear-window**. The default value is **:overwrite**.

:finish-typeout &optional *spacing erase?* of **si:interactive-stream** *Method*

Completes typeout to the window and causes the input buffer to be refreshed. In the case of **:temporary** typeout, the *erase?* parameter is used to indicate whether or not the typeout overwrote part of the current input by wrapping around the screen. It is the responsibility of the program doing the typeout to keep track of how much is output.

spacing can be one of the following keywords:

Keyword	*Action*
:none	No spacing before typeout.
:fresh-line	Typeout begins at the beginning of a line.
:blank-line	A blank line precedes typeout.

If *spacing* is not specified, a default that depends on the *type* argument to the **:start-typeout** method is computed.

:rescanning-p of **si:interactive-stream** *Method*

This message can be sent by a read function that uses the input editor to determine whether the next character returned by **:tyi** will come from the input buffer or from the keyboard. If **t** is returned, the input is being rescanned and the next character will come from the input buffer. If **nil** is returned, the next character will come from the keyboard.

:force-rescan of **si:interactive-stream** *Method*

This message can be sent by a read function that uses the input editor to force a rescan of the current input. Before this message is sent, usually some global state has changed and the contents of the input buffer are interpreted differently.

:replace-input *n-chars string* &optional (*begin* **0**) *end* (*rescan-mode* **:ignore**) of **si:interactive-stream** *Method*

This message can be sent by a read function that uses the input editor to provide completion of the current input.

n-chars specifies the number of characters to be removed from the end of the input buffer and erased from the screen. It can be an integer, a string, or **nil**:

integer	Remove *n-chars* characters from immediately before the scan pointer
string	Remove as many characters as the string contains
nil	Remove characters from the beginning of the input buffer to the scan pointer

The substring of *string* determined by *begin* and *end* is then displayed on the screen. *end* defaults to **(string-length** *string***)**. The scan pointer is left after the string, and a rescan does not take place. If a rescan takes place at some later time, the characters in *string* are seen as input.

rescan-mode specifies what action to take if the **:replace-input** message is sent when the scan pointer is not at the end of the input buffer:

:ignore	Don't perform the **:replace-input** operation. This is the default.
:enable	Perform the operation.
:error	Signal an error.

:read-bp of **si:interactive-stream** *Method*

Returns the value of the scan pointer. This is for the benefit of read functions that might want to return a pointer into the input buffer when signalling an error of type **sys:parse-error**.

:noise-string-out *string* &optional (*rescan-mode* **:ignore**) of **si:interactive-stream** *Method*

This message can be sent by a read function to display a string that is not to be treated as input. For example, the string might prompt the user for a particular kind of input. *string* is displayed on the screen without changing the scan pointer, and a rescan does not take place. If a rescan takes place at some later time, the characters in *string* are ignored.

rescan-mode specifies what action to take if the **:noise-string-out** message is sent when the scan pointer is not at the end of the input buffer:

:ignore	Don't perform the **:noise-string-out** operation. This is the default.
:enable	Perform the operation.
:error	Signal an error.

8. The Command Processor Program Interface

8.1 The Command Processor Reader

cp:read-command-or-form &optional (*stream* ***standard-input***) &key (*command-table* **cp:*command-table***) (*dispatch-mode* **cp::*default-dispatch-mode***) (*blank-line-mode* **cp::*default-blank-line-mode***) (*prompt* **cp::*default-prompt***) (*exception-chars* **nil**) (*environment* **si:*read-form-environment*** *environment-p*) *Function*

Reads a form or a Command Processor command from *stream*. This is an appropriate function to use at top level in a command loop that uses the command processor.

If *stream* is not supplied or is **nil**, it defaults to ***standard-input***.

If **:dispatch-mode** is specified, it is a keyword that indicates the command processor dispatch mode. The default is the value of **cp::*default-dispatch-mode***. The initial default is **:command-preferred**.

The actions that **cp:read-command-or-form** takes depend on *dispatch-mode*:

:form-only Calls **zl:read-form** to read a form from *stream*.

:command-only Calls **cp:read-command** to read a command from *stream*.

:form-preferred Calls **zl:read-form** unless the first character typed is a command dispatch character (by default, a colon). In that case calls **cp:read-command**.

:command-preferred
If the first character typed is a command dispatch character or an alphabetic character, calls **cp:read-command**; otherwise, calls **zl:read-form**. The user can evaluate a form that begins with an alphabetic character by first typing a form dispatch character (by default, a comma).

For a general description of how the user enters a command: See the section "Entering a Command" in *User's Guide to Symbolics Computers*.

If **:command-table** is supplied, it is a command table of the acceptable commands. The default command table is the value of

cp:*command-table*. The initial default is the "User" command table. See the section "Command Processor Command Tables", page 58.

If **:blank-line-mode** is supplied, it is a keyword that determines what action the command processor takes when the user types a blank line:

:reprompt	Redisplay the prompt, if any.
:beep	Beep.
:ignore	Do nothing.

The default *blank-line-mode* is the value of **cp::*default-blank-line-mode***. The initial default is **:reprompt**.

If **:prompt** is supplied, it is a prompt option for the input editor to display at appropriate times. *prompt* can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor", page 36. The default prompt is the value of **cp::*default-prompt***. The initial default is "Command: ".

cp:read-command-or-form returns a form. If **cp:read-command-or-form** calls **zl:read-form** to read from *stream*, it returns the form that **zl:read-form** returns. If it calls **cp:read-command**, it returns a list whose first element is a symbol, the name of the command, which is defined as a function. The remaining elements of the list are the arguments to the command, coerced to the appropriate types. Usually you execute the command by evaluating the returned list.

For an overview of **cp:read-command-or-form** and related facilities: See the section "Overview of Advanced Command Facilities" in *Programming the User Interface, Volume A.*

cp:read-command &optional (*stream* ***standard-input***) &key (*command-table* **cp:*command-table***) (*blank-line-mode* **cp::*default-blank-line-mode***) (*prompt* **cp::*default-prompt***) *Function*

Reads a Command Processor command from *stream*, terminated by RETURN or END.

If *stream* is not supplied or is **nil**, it defaults to ***standard-input***.

From the user's point of view, a command consists of a command name, positional arguments, and keyword arguments: See the section "Parts of a Command" in *User's Guide to Symbolics Computers.* **cp:read-command** offers completion over command names, keyword argument names, and some argument values, and it completes any unspecified command components when the command is terminated: See the section "Completion in the Command Processor" in *User's Guide to Symbolics Computers.*

cp:read-command prompts for arguments and gives information about what sort of values are expected. Some arguments have default values. The user can press HELP to see documentation appropriate to the current stage of entering the command: See the section "Help in the Command Processor" in *User's Guide to Symbolics Computers.* For a general description of how the user enters a command: See the section "Entering a Command" in *User's Guide to Symbolics Computers.*

If **:command-table** is supplied, it is a command table of the acceptable commands. The default command table is the value of **cp:*command-table***. The initial default is the "User" command table. See the section "Command Processor Command Tables", page 58.

If **:blank-line-mode** is supplied, it is a keyword that determines what action the command processor takes when the user types a blank line:

:reprompt	Redisplay the prompt, if any.
:beep	Beep.
:ignore	Do nothing.

The default *blank-line-mode* is the value of **cp::*default-blank-line-mode***. The initial default is **:reprompt**.

If **:prompt** is supplied, it is a prompt option for the input editor to display at appropriate times. *prompt* can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor", page 36. The default prompt is the value of **cp::*default-prompt***. The initial default is "Command: ".

cp:read-command returns two values. The first is a symbol, the name of the command, which is defined as a function. The second is a list of the arguments, converted to the appropriate types. Usually you execute the command by applying the first value (the function) to the second (the arguments).

For an overview of **cp:read-command** and related facilities: See the section "Overview of Advanced Command Facilities" in *Programming the User Interface, Volume A.*

cp::*default-dispatch-mode* *Variable*

The default command processor dispatch mode for **cp:read-command-or-form**; a keyword. Possible values are **:form-only**, **:form-preferred**, **:command-only**, and **:command-preferred**. For the meanings of these values: See the section "Setting the Command Processor Mode" in *User's Guide to Symbolics Computers.* The default is **:command-preferred**.

The dispatch mode used in Lisp Listeners and **zl:break** loops is the value of **cp:*dispatch-mode***.

cp::*default-blank-line-mode* *Variable*

The default command processor blank line mode for **cp:read-command** and **cp:read-command-or-form**. This is a keyword that determines what action the command processor takes when you type a blank line:

:reprompt	Redisplay the prompt, if any. This is the default.
:beep	Beep.
:ignore	Do nothing.

The blank line mode used in Lisp Listeners and **zl:break** loops is the value of **cp:*blank-line-mode***.

cp::*default-prompt* *Variable*

The default command processor prompt option for **cp:read-command** and **cp:read-command-or-form**. The value of this variable is passed to the input editor as the value of the **:prompt** option. The value can be **nil**, a string, a function, or a symbol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor", page 36. The default is "Command: ".

The prompt used in Lisp Listeners and **zl:break** loops is the value of **cp:*prompt***.

8.2 Defining a Command Processor Command

define-cp-command, the focus of this section, is the pre-Genera 7.0 facility for defining Command Processor commands. It is supported in Genera 7.0, but the recommended command-definition facility is **cp:define-command**. For an overview of the latter and related facilities: See the section "Overview of Basic Command Facilities" in *Programming the User Interface, Volume A*.

define-cp-command *name args* &body *body* *Special Form*

Defines a command processor command. *name* is a specification for the command name. *args* is a specification for the command arguments. **define-cp-command** defines a function that executes the command, with *body* as the body of the function. The name of the function is derived from *name* and the arguments from *args*.

name is a symbol or a list. If *name* is a symbol, it is the name of the function that executes the command. By convention, the first four characters of the symbol's print name are usually "COM-".

If *name* is a list, the first element is a symbol, the name of the function that executes the command. The remaining elements are alternating keywords and values. Each keyword-value pair is optional. Following are the permissible keywords and values:

:name A string that represents the command name that the user types. If this option is not specified, the name is the result of calling **zl:string-capitalize-words** on the symbol's print name, except that if the symbol's print name begins with "COM-", those characters are omitted from the command name. This option is useful for special capitalization of words, as in "Start GC".

:comtab A command table or a string naming a command table in which to install the command. For example, to install a command in the "User" command table, you might specify "User" or the result of **(si:find-comtab "user")**. This option is evaluated. If it is not specified, the command is not installed in any command table and cannot be read. See the section "Command Processor Command Tables", page 58.

args is **nil** or a list of *argument specifications* for the arguments to the command and the function that executes the command. One element of *args* can be the symbol **&key** instead of an argument specification. All argument specifications preceding **&key** denote positional arguments to the command. All argument specifications following **&key** denote keyword arguments to both the command and the function that executes the command.

An *argument specification* is a list that describes one argument to the command.

The first element of an argument specification is a symbol. This symbol names a parameter in the arglist of the function that executes the command. This parameter is bound to the value of the argument when the function is called to execute the command. *body* can refer to the parameter. Unless a **:name** option is supplied later in the argument specification, the user-visible name of the argument is the result of calling **zl:string-capitalize-words** on the symbol's print name.

The second element of an argument specification is an *argument type specification*. This is a keyword or a list. If it is a keyword, it is the name of this argument's type. If it is a list, the first element is a keyword that is the name of this argument's type. The remaining elements supply information specific to the argument type. See the section "Command Processor Argument Types", page 53.

The remaining elements of an argument specification are alternating keywords and values. Each keyword-value pair is optional. None of the values is evaluated. Following are the permissible keywords and values:

:allow-multiple
: **t** if the argument can have multiple values; **nil** if the argument can have only one value. The user enters multiple values as a series separated by commas. These are passed to the command function as a list of values. The default is **nil**.

:confirm
: **t** if the argument requires confirmation by the user; **nil** if it does not. The default is **nil**.

:default
: A form to be evaluated when the argument is read to return the default value for the argument. If **:allow-multiple** is specified with a value of **t**, the form must return a list of values. The form can refer to parameters defined for any positional arguments (but not keyword arguments) specified in *args* before this argument specification. At the time the form is evaluated, these parameters are bound to the values of arguments already read.

  For a positional argument, if **:default** is not supplied the argument has no default value. When the command is read, the user is forced to supply a value.

  For a keyword argument, the default used depends on what combination of **:default** and **:mentioned-default** options is supplied:

  Both
  : Use the **:mentioned-default** default if the user types the name of the argument; otherwise use the **:default** default.

  :mentioned-default only
  : If the user types the name of argument, use the **:mentioned-default** default. Otherwise the default is **nil**.

  :default only
  : Use the **:default** default.

  Neither
  : If the user does not type the name of the argument, the default is **nil**. If the user types the name of the

argument, the argument has no default value, and the user is forced to supply a value.

:mentioned-default
For a keyword argument, a form to be evaluated when the argument is read to return the default value if the user types the name of the argument. If **:allow-multiple** is specified with a value of **t**, the form must return a list of values. The form can refer to parameters defined for any positional arguments (but not keyword arguments) specified in *args* before this argument specification. At the time the form is evaluated, these parameters are bound to the values of arguments already read.

The default used depends on what combination of **:default** and **:mentioned-default** options is supplied:

Both — Use the **:mentioned-default** default if the user types the name of the argument; otherwise use the **:default** default.

:mentioned-default only — If the user types the name of argument, use the **:mentioned-default** default. Otherwise the default is **nil**.

:default only — Use the **:default** default.

Neither — If the user does not type the name of the argument, the default is **nil**. If the user types the name of the argument, the argument has no default value, and the user is forced to supply a value.

Use this option when you want the default to depend on whether or not the user types the argument name. For example, the Delete File command has an Expunge keyword argument whose **:default** default is No and whose **:mentioned-default** default is Yes.

:use-type-default If non-**nil**, the default for this argument is determined by the current default for this type of argument, for example, a pathname for commands that deal with files. The default is **t**.

:set-type-default If non-**nil** the default for this argument becomes the current default for this type of argument (for example, a pathname for commands that deal with files). The default is **t**.

:documentation A string, usually short, that documents the meaning of the argument. The string is displayed after the argument name if the user presses HELP while entering the argument. For example, the string for the argument to the Show Hosts command is "Hosts about which to display status information". The default HELP display depends on the argument type.

:name A string that represents the user-visible name of the argument. The default name is the result of calling **zl:string-capitalize-words** on the print name of the symbol that is the first element of the argument specification. This option is useful when you want the user-visible name of the argument to differ from the parameter bound to the argument value. For example, you might want the user-visible name to be Base without binding the special variable **zl:base**.

:prompt A string that represents a prompt for the argument, or a form to be evaluated when the command is read to return a prompt string. The form is evaluated with the symbol =**default**= bound to the argument default. =**default**= is interned in the package that is the value of **zl:package** when the **define-cp-command** form is evaluated. The default prompt depends on the argument type. See the section "Command Processor Argument Types", page 53.

Example:

```
(define-cp-command (com-edit-file :comtab "Global")
  ((file :pathname
         :allow-multiple t
         :default '(,(fs:default-pathname))
         :prompt
         (format nil "file to edit [default ~A]" (first =default=))
         :documentation "Files to edit"))
   (ed file)
   (send standard-output :fresh-line)
   (send standard-output :tyo #\newline)
   (values))
```

8.3 Command Processor Argument Types

Following is a description of each command processor argument type. When you use **define-cp-command** to define a command, the argument type keyword is the second element of an *argument specification*, or the car of the second element. If the second element is a list, the elements of its cdr are the *type-specific information* described for each argument type. See the special form define-cp-command, page 48.

The default prompt and help message for each type provide information about the kind of values expected. In general, when the possible values are members of a restricted set, the default help message lists the possible values. The default prompt sometimes lists the possible values. For some types completion is provided over the set of possible values.

:boolean The value is **t** if the user types "Yes" and **nil** if the user types "No". Completion is provided over these choices.

Type-specific information: None.

:documentation-destination

The value is a keyword symbol or a local printer capable of serving as an output device for documentation display. If the keyword is **:screen**, the output is to be displayed on the screen. If the keyword is **:remote-printer**, the user should be prompted for the name of a nonlocal printer. Completion is provided over the possible values.

Type-specific information: None.

:enumeration The value is one of a restricted set of strings or objects that can be coerced to strings, specified by the type-specific information. The user must type a string associated with one of the elements of the set. Completion is provided over this set.

Type-specific information: The strings or objects that can be coerced to strings that make up the set of permissible values for the argument. Often these are keyword symbols. For example:

```
(:enumeration :yes :no :ask)
```

The default prompt lists strings formed by calling **zl:string-capitalize-words** (but keeping hyphens) on each element of the set of permissible values.

:number The value is a number.

Type-specific information: The symbol **:base** followed by an integer, the base in which the number is read. If **:base** is not supplied the number is read in decimal.

The default prompt displays the input base (if other than decimal) and the default.

:integer The value is an integer.

Type-specific information: Alternating keywords and values:

:base An integer, the base in which the integer is read. If **:base** is not supplied the integer is read in decimal.

:from A number. The integer must be greater than or equal to this. If **:from** is not supplied the integer has no lower limit.

:to A number. The integer must be less than or equal to this. If **:to** is not supplied the integer has no upper limit.

The default prompt displays the input base (if other than decimal) and the default.

:string The value is a string.

Type-specific-information: None.

:pathname If no type-specific information is supplied, the value is a pathname derived from merging the string the user types with the default and a default version of **:newest**. Completion is provided using the system pathname-completion facility.

Type-specific information: Alternating keywords and values:

:pathname-default
A form to be evaluated when the command is

read to return a default for pathname merging. The form can return anything suitable as the second argument to **fs:merge-pathnames**. This is used as the default only if the argument default is not a pathname; if the argument default is a pathname, that pathname is used as the default for merging. If the argument default is not a pathname and if **:pathname-default** is not supplied, the default is the result of **(fs:default-pathname)**.

:default-version	A number or symbol suitable as the third argument to **fs:merge-pathnames**, to be used as the default version for the merged pathname.
:or-none	If **t** and the user types "none", the value of the argument is **:none**.
:or-no	If **t** and the user types "no", the value of the argument is **:no**.
:or-query	If **t** and the user types "query", the value of the argument is **:query**.
:raw	The value of the argument is the result of calling **fs:parse-pathname** with arguments of the string the user types, **nil**, and the default.

The default prompt displays the default pathname.

:fep-pathname If no type-specific information is supplied, the value is a FEP pathname derived from merging the string the user types with the default and a default version of **:newest**. If the result is not a FEP pathname, an error is signalled.

Type-specific-information: The same as **:pathname**.

:host The value is the network host whose name the user types, unless the user types "Local", "All", or "None":

"Local"	The local host
"All"	**:all**
"None"	**nil**

Type-specific-information: None.

:printer The value is the printer object whose name the user types, unless the user types "None". In that case the value is **nil**. The value can be any printer accessible from the user's site. Completion is provided over the set of printers at the user's site.

Type-specific-information: None.

:date The value is a universal time integer. When the user's input is parsed, missing components are defaulted to the beginning of the smallest unsupplied unit of time. Thus, "5 pm" is the same as "5 pm today", whether typed before or after 5 pm.

Type-specific-information: None.

:package If no type-specific information is supplied, the value is the package whose name the user types. Completion is provided over the set of existing packages.

Type-specific information: The keyword **:all-allowed**. If this is supplied and the user types "All", the value of the argument is **:all**.

:font If no type-specific information is supplied, the value depends on what the user types:

Nothing — If no default exists, the value is **nil**.

Name of a loaded font — The value is the font.

Name of a known but not loaded font — The value is the print name of the symbol in the **fonts** package.

Name of an unknown font — The value is the string the user types.

Completion is provided over the set of known fonts.

Type-specific information: Alternating keywords and values:

:or-default — If **t** and the user types "Default-Font", the value of the argument is **:default-font**.

:known-only — If **t** and the user types the name of an unknown font, an error is signalled and caught by the input editor.

:loaded-only — If **t** and the user types the name of a font that is unknown or is not loaded, an error is signalled and caught by the input editor.

:system

If no type-specific information is supplied, the value is the system whose name the user types if the system is loaded, or the string the user types if it is not the name of a loaded system. Completion is provided over the set of loaded systems.

Type-specific information: Keywords:

:loaded-only If the user types "All", the value of the argument is **:all**. Otherwise, unless the user types the name of a loaded system, an error is signalled and caught by the input editor.

:patchable-only If the user types the name of a system that is loaded but not patchable, the value of the argument is the string the user types, unless **:loaded-only** is also specified. In that case an error is signalled and caught by the input editor.

:activity

The value is an element of the list that is the value of **tv:*select-keys***. This is a list of four elements, the third of which is the string that the user types naming the activity. For some activities, the user can also type a nickname for the name of the activity. In that case the string the user types is not the same as the third element of the returned list.

The elements of the returned list correspond to the first four arguments to **tv:add-select-key**. For information: See the function **tv:add-select-key**, page 152.

Completion is provided over the set of existing activities.

Type-specific information: None.

:documentation-topic

The value is an element of the completion aarray used by the Document Examiner. This is a list determined by the string the user types. The first element of the list is the string, and the remaining elements are associated function specs that have documentation available to the Document Examiner. Completion is provided over the set of defined documentation topics.

Type-specific information: None.

:make-system-version

The value is an integer, symbol, or string suitable as an argument to the **:version zl:make-system** option. If the user types a nonnegative integer, the value is that integer, unless **:no-number** is specified in the type-specific information. If the user types a string associated with one of the elements specified by the type-specific information, the value is that element. Otherwise, the value is the string the user types. Completion is provided over the set of values specified by the type-specific information.

Type-specific information: Strings or objects that can be coerced to strings that make up the set of permissible values for the argument. Usually this includes symbols like **:newest** or **:released**. If **:no-number** is one of these, integers (and **:no-number**) are not permissible values.

The default prompt lists strings formed by calling **zl:string-capitalize-words** (but keeping hyphens) on each element of the set of permissible values.

8.4 Command Processor Command Tables

A *command table* is an object that identifies a set of commands that are permissible in some context. Command tables can be arranged in a hierarchy, so that subordinate command tables inherit commands from their superiors. The set of permissible commands for a command table includes the commands in that command table and the commands in all superior command tables.

When a command is read, using **cp:read-command** or **cp:read-command-or-form**, the set of permissible commands is determined by the command table that is the value of the **:command-table** option to the reading function. Only commands in that command table or a superior can be read. You install a command in a command table at the time you define the command, using the **:command-table** option to **cp:define-command**. See the macro **cp:define-command** in *Programming the User Interface, Volume A.*

The Command Processor maintains a global registry of all command tables. You find a command table by using the function **cp:find-command-table**. This function is especially useful in supplying the **:command-table** argument to **cp:read-command** or **cp:read-command-or-form**. Use **cp:make-command-table** to create a command table, and **cp:delete-command-table** to delete one. Two useful existing command tables are the "Global" and the "User" command tables.

The variable **cp:*command-table*** is bound to the current command table in Lisp

Listeners and **break** loops. It is also the default command table for **cp:read-command** and **cp:read-command-or-form.**

cp:find-command-table *name* &key (*if-does-not-exist* **:error**) *Function*

Returns the Command Processor command-table object specified by the command-table name.

name The name (symbol or string) of the command table.

:if-does-not-exist

Specifies what happens if the named command table is not found. Three values are possible:

nil The function returns **nil.**

:error An error message is returned and the debugger is entered; this is the default.

:create A new command table named *name* is created and returned.

For an overview of **cp:find-command-table** and related facilities: See the section "Overview of Command Table Management Facilities" in *Programming the User Interface, Volume A.*

cp:make-command-table *name* &rest *init-options* &key (*if-exists* **:error**) &allow-other-keys *Function*

Creates and returns a Command Processor command table object.

name The name (symbol or string) of the command table.

init-options

Keyword-values pairs that are init options to the (internal) command-table flavor from which the command table object is created. Permissible options and values are as follows:

:inherit-from

Specifies a list of command tables from which to inherit commands.

:command-table-delims

Specifies a list of characters to use as delimiters of words in command names for commands in the table. The default list is `(#\Space #\Tab #\Return)`.

:command-table-size

An initial estimate of the number of commands the table will include (to preclude the table from having to grow substantially).

:kbd-accelerator-p
Boolean option specifying whether single-key accelerators may be used for commands; the default is **t**.

:accelerator-case-matters
Boolean option specifying whether single-key accelerators, if allowed, are case sensitive; the default is **nil**.

:if-exists Specifies what happens if the command table named *name* already exists. Four values are possible:

nil No new command table is made and the existing command table is returned.

:supersede
The new command table is made and replaces the old command table.

:update-options
The existing command table remains but its options are updated to those newly specified in the call to **cp:make-command-table.**

:error An error message is returned and the debugger is entered.

Example:

```
(cp:make-command-table "shell-cmds" :inherit-from '("user")
                                    :kbd-accelerator-p nil)
```

For an overview of **cp:make-command-table** and related facilities: See the section "Overview of Command Table Management Facilities" in *Programming the User Interface, Volume A.*

cp:delete-command-table *command-table-or-name* *Function*
Removes a Command Processor command table from the command table registry.

command-table-or-name
A command table object or the name (symbol or string) of a command table.

For an overview of **cp:delete-command-table** and related facilities: See the section "Overview of Command Table Management Facilities" in *Programming the User Interface, Volume A.*

cp:*command-table* *Variable*

Bound to the current command table, that is, the one used by the Command Processor when reading commands.

For an overview of **cp:*command-table*** and related facilities: See the section "Overview of Command Table Management Facilities" in *Programming the User Interface, Volume A.*

9. Querying the User

The following functions provide a convenient and consistent interface for asking questions of the user. Questions are printed and the answers are read on the stream ***query-io***, which normally is synonymous with ***terminal-io*** but can be rebound to another stream for special applications.

y-or-n-p &optional *format-string* &rest *args* *Function*

Provides a convenient and consistent interface for asking questions of the user. It types out a message (if supplied), reads a single character (Y or N), and returns **t** if the answer was the characters "y" or "Y", or **nil** if the answer was the characters "n" or "N".

y-or-n-p uses ***query-io*** to print the questions and read the answers. ***query-io*** is normally synonymous with ***terminal-io***, but can be rebound to another stream for special applications.

If *format-string* is supplied and non-**nil**, then a fresh-line operation is performed. After that a message is printed as if *format-string* and *args* were given to **format**. Otherwise it is assumed that any message has already been printed by other means.

Here are some examples of the use of **y-or-n-p**:

```
(y-or-n-p "Produce listing file?" *terminal-io*) =>
Produce listing file?(Y or N) y
T

(y-or-n-p "Cannot connect to network host ~S.  Retry?" host) =>
Cannot connect to network host TURKEY.  Retry?(Y or N) n
NIL
```

y-or-n-p should only be used for questions that the user knows are coming or in situations where the user is known to be waiting for a response of some kind. If the user is unlikely to anticipate the question, or if the consequences of the answer might be irreparable, then **y-or-n-p** should not be used because the user might type ahead and thereby accidentally answer the question. For such questions as "Shall I delete all of your files?", it is better to use **yes-or-no-p**.

zl:y-or-n-p &optional *message* (*query-io* **zl:query-io**) *Function*

This is used for asking the user a question whose answer is either "yes" or "no". It types out *message* (if any), reads a one-character answer, echoes it as "yes" or "**no**", and returns **t** if the answer is "yes" and **nil** if the answer is "no". The characters that mean "yes" are **#/y**, **#/t**, and

#\space. The characters that mean "no" are #/n and #\rubout. If any other character is typed, the function beeps and demands a "Y or N" answer.

If the *message* argument is supplied, it is printed on a fresh line (using the **:fresh-line** stream operation). Otherwise the caller is assumed to have printed the message already. If you want a question mark and/or a space at the end of the message, you must put it there yourself; **zl:y-or-n-p** does not add it. *query-io* defaults to the value of **zl:query-io**.

zl:y-or-n-p should only be used for questions that the user knows are coming. If the user is not going to be anticipating the question (for example, if the question is "Do you really want to delete all of your files?" out of the blue) then **zl:y-or-n-p** should not be used, because the user might type ahead a T, Y, N, space, or rubout, and therefore accidentally answer the question. In such cases, use **zl:yes-or-no-p**.

zl:y-or-n-p supplies a prompt that indicates which form of answer (single letter or full word plus RETURN) is required. This prompt is appended to any message that you supply with the function.

```
(y-or-n-p "More? ") =>
More? (Y or N) Yes.
```

yes-or-no-p &optional *format-string* &rest *args* *Function*

Provides a convenient and consistent interface for asking questions of the user. It types out a message (if supplied), reads a word (Yes or No), and returns **t** if the answer was the word "Yes", or **nil** if the answer was the word "No". **yes-or-no-p** allows completion, so you can type any subset of the word "Yes" or "No" followed by the END or RETURN keys.

yes-or-no-p uses ***query-io*** to print the questions and read the answers. ***query-io*** is normally synonymous with *terminal-io*, but can be rebound to another stream for special applications.

If *format-string* is supplied and non-**nil**, then a fresh-line operation is performed. After that a message is printed as if *format-string* and *args* were given to **format**. Otherwise it is assumed that any message has already been printed by other means.

Here are some examples of the use of **yes-or-no-p**:

```
(yes-or-no-p "Shall I delete all of your files?") =>
Shall I delete all of your files?(Yes or No) noRETURN
NIL

(yes-or-no-p "List the entire set of commands?") =>
List the entire set of commands?(Yes or No) yeEND
T
```

To allow the user to answer a yes or no question with a single character, use **y-or-n-p**. **yes-or-no-p** would be used for unanticipated or important questions, which is why it requires a multiple-action sequence to answer it.

zl:yes-or-no-p &optional *message* (*query-io* **zl:query-io**) *Function*

This is used for asking the user a question whose answer is either "Yes" or "No". It types out *message* (if any), beeps, and reads in a line from the keyboard. If the line is the string "Yes", it returns **t**. If the line is "No", it returns **nil**. (Case is ignored, as are leading and trailing spaces and tabs.) If the input line is anything else, **zl:yes-or-no-p** beeps and demands a "yes" or "no" answer.

If the *message* argument is supplied, it is printed on a fresh line (using the **:fresh-line** stream operation). Otherwise the caller is assumed to have printed the message already. If you want a question mark and/or a space at the end of the message, you must put it there yourself; **zl:yes-or-no-p** does not add it. *query-io* defaults to the value of **zl:query-io**.

To allow the user to answer a yes-or-no question with a single character, use **zl:y-or-n-p**. **zl:yes-or-no-p** should be used for unanticipated or momentous questions; this is why it beeps and why it requires several keystrokes to answer it.

zl:yes-or-no-p supplies a prompt that indicates which form of answer (single letter or full word plus RETURN) is required. This prompt is appended to any message that you supply with the function.

```
(yes-or-no-p "Detonate terminal? ") =>
Detonate terminal? (Yes or No) no
```

fquery *options* &optional *fquery-format-string* &rest *fquery-format-args* *Function*

Asks a question, printed by **(format query-io** *format-string format-args...***)**, and returns the answer. **fquery** takes care of checking for valid answers, reprinting the question when the user clears the screen, giving help, and so forth.

options is a list of alternating keywords and values, used to select among a variety of features. Most callers have a constant list that they pass as *options* (rather than consing up a list whose contents varies). The keywords allowed are:

:type What type of answer is expected. The currently defined types are **:tyi** (a single character), **:readline** (a line terminated by a carriage return), and **:mini-buffer-or-readline**. **:tyi** is the default.

:mini-buffer-or-readline is like the **:readline** value. The exception is that if **fquery** is called from inside Zwei or Zmail, the line of text is read from the minibuffer instead of from the **zl:query-io** stream. The idea of this feature is to let you write things that work equally well inside Zwei or on their own; if you use this value, you make it easier for your code to be integrated into a Zwei extension.

:choices Defines the allowed answers. The allowed forms of choices are complicated and explained below. The default is the same set of choices as the **zl:y-or-n-p** function. Note that the **:type** and **:choices** options should be consistent with each other.

:list-choices

If **t**, the allowed choices are listed (in parentheses) after the question. The default is **t**; supplying **nil** causes the choices not to be listed unless the user tries to give an answer that is not one of the allowed choices.

:help-function

Specifies a function to be called if the user presses the HELP key. The default help function simply lists the available choices. Specifying **nil** disables special treatment of HELP. If you specify a help function, it should take one argument, the stream on which to display the help message. The function can get the list of available choices from the value of the special variable **format:fquery-choices**.

:signal-condition

Basically a way to intervene and provide an answer to a query without asking the user.

The default for **:signal-condition** is **nil**. When its value is **t**, the **fquery** function signals an **fquery** condition with proceed type of **:choice** before prompting the user. Any handler can invoke the **:choice** proceed type in order to return a value from **fquery**. When no handler handles the condition, **fquery** proceeds normally and queries the user.

fquery *Flavor*

fquery is a simple condition built on **condition**. It is signalled by the **fquery** function when its **:signal-condition** option is **t**. The messages examine the arguments given to the **fquery** function.

Message	*Value returned*
:options	Returns the first argument to the **fquery** function.

:format-string Returns the second argument to the **fquery** function (its format control string or prompt).

:format-args Returns the rest of the arguments to the **fquery** function (the arguments to its format control string).

The **:choice** proceed type is provided. It has one argument, which is a value to be returned from the call to the **fquery** function.

The following example answers "yes" to every "Delete this entry?" query occurring inside do-it that has **:signal-condition** t:

```
(condition-bind
  ((fquery #'(lambda (condition)
               (and (send condition ':proceed-type-p ':choice)
                    (equal (send condition ':format-string)
                           "Delete this entry? ")
                    (values ':choice t)))))
  (do-it))
```

:fresh-line
If **t**, **zl:query-io** is advanced to a fresh line before asking the question. If **nil**, the question is printed wherever the cursor was left by previous typeout. The default is **t**.

:beep If **t**, **fquery** beeps to attract the user's attention to the question. The default is **nil**, which means not to beep unless the user tries to give an answer that is not one of the allowed choices.

:clear-input
If **t**, **fquery** throws away typeahead before reading the user's response to the question. Use this for unexpected questions. The default is **nil**, which means not to throw away typeahead unless the user tries to give an answer that is not one of the allowed choices. In that case, typeahead is discarded since the user probably wasn't expecting the question.

:select If **t** and **zl:query-io** is a visible window, that window is temporarily selected while the question is being asked. The default is **nil**.

:make-complete
If **t** and **zl:query-io** is a typeout-window, the window is "made complete" after the question has been answered. This tells the system that the contents of the window are no longer useful. The default is **t**.

:stream Has as its value the stream to use for both input and output. The default value is the value of the global variable **zl:query-io**.

:no-input-save
If **t**, tells the input editor not to put the response to the question into its history. The default is **nil**.

:status This option takes effect only if **zl:query-io** is a window and **:type** is **:tyi**. If the value is **:selected** and the window becomes deselected while **fquery** is waiting for input, **fquery** returns **:status**. If the value is **:exposed** and the window becomes deexposed or deselected while **fquery** is waiting for input, **fquery** returns **:status**. If the value is **nil**, **fquery** continues to wait for input when the window is deexposed or deselected. The default is **nil**.

This option is intended for queries that appear in temporary windows that might become deexposed or deselected before the user responds.

The argument to the **:choices** option is a list each of whose elements is a *choice* (with one exception, described in the next paragraph). A choice is a list whose cdr is a list of the user inputs that correspond to that choice. These should be characters for **:type :tyi** or strings for **:type :readline**. The car of a choice is either a symbol that **fquery** should return if the user answers with that choice, or a list whose first element is such a symbol and whose second element is the string to be echoed when the user selects the choice. In the former case nothing is echoed. In most cases **:type :readline** would use the first format, since the user's input has already been echoed, and **:type :tyi** would use the second format, since the input has not been echoed and furthermore is a single character, which would not be meaningful to see on the display.

The last element in the list of choices can be the symbol **:any** (instead of being a list, like all other choices). Then if the user gives some response that is not one of the other choices, **fquery** does not complain and reprompt the user, but instead returns what the user typed (a single character or a string, depending on the **:type** option).

For example, the **zl:yes-or-no-p** function uses this list of choices:

```
((t "Yes") (nil "No"))
```

and the **zl:y-or-n-p** function uses this list:

```
(((t "Yes.") #/Y #/T #\space)
 ((nil "No.") #/N #\rubout))
```

If you want to use the formatted output functions instead of **zl:format** to produce the prompting message, write:

```
(fquery options (format:outfmt exp-or-string exp-or-string ...))
```

format:outfmt puts the output into a list of a string, which makes **zl:format** print it exactly as is. There is no need to supply additional arguments to the **fquery** unless it signals a condition. In that case the arguments might be passed so that the condition handler can see them.

prompt-and-read *type* &optional *format-string* &rest *format-args* *Function*

prompt-and-read prompts the user, with *format-string* and its arguments as the prompt. It uses **zl:format** to **zl:query-io** to produce the prompt; it reads from the **zl:query-io** stream, calling the reading function associated with the *type* keyword. If *format-string* is not specified, it generates a prompt appropriate to *type*. The *type* argument can be a list in which the first element is the type keyword and the rest are keyword/value pairs to serve as arguments to the reading function. (For the **:object** and **:object-list** types, *type* must be a list with the **:class** keyword supplied.) **prompt-and-read** returns whatever the reading function returns.

This is an appropriate function to call for collecting input from the user. Its main advantages are that it does type checking on the input the user types and that it takes care of redisplaying the prompt at appropriate times (for example, after the screen has been refreshed or after a notification arrives).

```
(prompt-and-read :number "Please enter a number: ") =>
Please enter a number: 4
4
(prompt-and-read :string "Please enter a string: ") =>
Please enter a string: 4
"4"
```

It expects to collect input of type *type*, where *type* is a keyword. It handles the following types of input:

Option	*Action*
:eval-form	Reads a Lisp form. Evaluates it and returns the first value. Asks for confirmation of nonconstant values. The Debugger uses this to prompt for a form to evaluate.
:eval-form-or-end	Reads a Lisp form or just END. Evaluates it and returns the first value for a form. Returns two values, **nil** and **:end**, for END. Asks for confirmation of nonconstant values. The Debugger uses this to prompt for a form to evaluate.
:expression	Reads a Lisp expression and returns the expression without evaluating it.

:expression-or-end
Reads a Lisp expression or just END. It returns the expression without evaluating it. If the user just presses END, it returns two values, **nil** and **:end**.

:character Reads and returns a character. The returned value is a character code (an integer).

:symbol Reads and returns a symbol.

(:function-spec :defined-p *defined-p***)**
Reads and returns a function spec. If **:defined-p** is specified with a value other than **nil**, the function spec must be defined as a function. The default for *defined-p* is **nil**.

:string Reads a string terminated by RETURN, LINE, or END. It returns the empty string when the string is empty.

:string-trim Reads a string terminated by RETURN, LINE, or END. It trims any leading or trailing white space. It returns the empty string when the string is empty.

:string-or-nil Reads a string terminated by RETURN, LINE, or END. It trims any leading or trailing white space. It returns **nil** when the string is empty.

(:string-list :or-nil *or-nil***)**
Reads a series of strings separated by commas and terminated by RETURN, LINE, or END. It returns a list of the strings, unless *or-nil* is not **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **t**.

(:delimited-string :delimiter *delimiter* **:visible-delimiter** *visible-delimiter* **:buffer-size** *size* **:or-nil** *or-nil***)**
Reads characters until the user types a delimiter, then returns the input as a string without the delimiter.

:delimiter and **:visible-delimiter** are mutually exclusive. If one of them is specified, it must be **nil** or a list of characters that delimit the string. If neither is specified, or if one is specified with a value of **nil**, the only delimiter is **#\end**.

The difference between **:delimiter** and **:visible-delimiter** is that if a prompt is supplied as the second argument to **prompt-and-read**, the **:visible-delimiter** characters are displayed to the user after the prompt, but the **:delimiter** characters is not. If a prompt is supplied and neither **:delimiter** nor **:visible-delimiter** is specified, the delimiting character is not displayed. If no prompt is supplied, the delimiting characters are always displayed, whether they come from **:delimiter**, **:visible-delimiter**, or the default delimiter.

If **:buffer-size** is specified, an initial buffer of size *size* characters is allocated; otherwise, the initial size is 100. characters. It returns the empty string when the string is empty, unless **:or-nil** is specified with a value other than **nil**. In that case it returns **nil** when the string is empty. The default for *or-nil* is **nil**.

(:delimited-string-or-nil :delimiter *delimiter* **:visible-delimiter** *visible-delimiter* **:buffer-size** *size***)**

The same as **(zl-user:delimited-string :delimiter** *delimiter* **:visible-delimiter** *visible-delimiter* **:buffer-size** *size* **:or-nil t)**. This option is obsolete.

(:complete-string :alist *alist* **:delimiters** *delimiters* **:impossible-is-ok** *impossible-is-ok* **:or-nil** *or-nil* **:complete-on-space** *complete-on-space***)**

Reads and returns a (possibly completed) string, terminated by RETURN, LINE, or END.

If the user presses COMPLETE, the input so far is completed over the set of possibilities determined by *alist*. If *complete-on-space* is not **nil**, the input is also completed when the user presses SPACE at the end of the input buffer. The default for *complete-on-space* is **t**.

If the user presses c-?, the possible completions of the input are displayed. If the user presses HELP, the possible completions are displayed unless many exist; in that case a general help message is displayed.

The style of completion is the same as that offered by Zwei. *alist* can be **nil**, an alist, an **sys:art-q-list** array, or a keyword:

nil	No completion is offered. This is the default.
alist	The car of each alist element is a string representing one possible completion.

array	Each element is a list whose car is a string representing one possible completion. The array must be sorted alphabetically on the cars of the elements.
keyword	If the symbol is **:zmacs**, completion is offered over the definitions in Zmacs buffers. If the symbol is **:flavors**, completion is offered over all flavor names. If the symbol is **:documentation**, completion is offered over all documentation topics available to the Document Examiner.

Example:

```
(prompt-and-read
    '(:complete-string :alist :documentation))
Enter a string with completion, or <RETURN>
for none:  formatted output
=> "Formatted Output"
=> (("Formatted Output" DOC:|FORMATTED OUTPUT|))
```

delimiters is **nil** or a list of characters that delimit "chunks" for completion. As in Zwei, completion works by matching initial substrings of "chunks" of text. If *delimiters* is **nil**, the entire text of the input is a single "chunk". The default is **nil**.

If *or-nil* is **nil** and the user just presses RETURN, LINE, or END, **:complete-string** waits for more input. If *or-nil* is not **nil** and the user just presses RETURN, LINE, or END, it returns **nil**. The default for *or-nil* is **t**.

If the user presses RETURN, LINE, or END and the input buffer is not empty, the input is completed as far as possible. If the completed string is the car of an alist element, the completed string is returned. Otherwise, if the user pressed END or if *impossible-is-ok* is **nil**, **:complete-string** waits for more input. If the user pressed RETURN or LINE and if *impossible-is-ok* is not **nil**, the completed string is returned. The default for *impossible-is-ok* is **t**.

Unless **:complete-string** returns **nil**, it also returns a second value, a list of the alist elements that represent possible completions of the returned string.

(:flavor-name :impossible-is-ok *impossible-is-ok***)**

Reads and returns the name of a flavor, terminated by RETURN, LINE, or END. The user can type the flavor name with or without a package prefix.

If the user presses COMPLETE, the input so far is completed over the set of defined flavors. If the user presses c-?, the possible completions of the input are displayed. If the user presses HELP, the possible completions are displayed unless many exist; in that case a general help message is displayed.

If the user presses RETURN, LINE, or END and the input buffer is not empty, the input is completed as far as possible. If the completed input is the name of a flavor, the flavor name (a symbol in the appropriate package) is returned. Otherwise, if the user pressed END, **:flavor-name** waits for more input. If the user pressed RETURN or LINE and if *impossible-is-ok* is not **nil**, the completed input is returned as a symbol. If the user pressed RETURN or LINE and if *impossible-is-ok* is **nil**, an error is signalled and caught by the input editor. The default for *impossible-is-ok* is **t**.

(:number :base *input-base* **:or-nil** *or-nil***)**

Reads and returns a number, terminated by RETURN, LINE, or END. If **:base** is specified, the number is read in base *input-base*; otherwise, it is read as a decimal number. If **:or-nil** is specified with a value other than **nil**, it returns **nil** if the user just presses RETURN, LINE, or END. The default for *or-nil* is **nil**.

(:number-or-nil :base *input-base***)**

The same as **(:number :base** *input-base* **:or-nil t)**. This option is obsolete.

(:decimal-number :or-nil *or-nil***)**

The same as **(:number :base 10. :or-nil** *or-nil***)**. This option is obsolete.

:decimal-number-or-nil

The same as **(:number :base 10. :or-nil t)**. This option is obsolete.

(:integer :base *input-base* **:or-nil** *or-nil* **:from** *from* **:to** *to***)**

Reads and returns an integer, terminated by RETURN, LINE, or END. If **:base** is specified, the integer is read in base *input-base*; otherwise, it is read as a decimal number. If **:or-nil** is specified with a value other than **nil**, it returns **nil** if the user just presses RETURN, LINE, or END. The default for *or-nil* is **nil**. If **:from** is specified, the integer must be greater than or equal to *from*. If **:to** is specified, the integer must be less than or equal to *to*. The default for *from* and *to* is to place no limits on the integer.

(:date :past-p *past-p* **:never-p** *never-p* **:base-time** *base-time* **:or-nil** *or-nil***)**

Reads and returns a date, terminated by RETURN, LINE, or END. The returned date is a universal-time integer of the form returned by **time:parse-universal-time**. If **:past-p** is specified with a value other than **nil**, an ambiguous date is interpreted as being in the past, relative to the base time; otherwise, it is interpreted as being in the future. The default for *past-p* is **nil**. If **:never-p** is specified with a value other than **nil**, it returns **nil** if the user types "never". The default for *never-p* is **nil**. If **:base-time** is specified, it must be a universal-time integer that is used to fill in components that the user omits. If **:base-time** is not specified, the time when the user's input is read is used as the base time.

(:past-date :never-p *never-p* **:base-time** *base-time* **:or-nil** *or-nil***)**

The same as **(:date :past-p t :never-p** *never-p* **:base-time** *base-time* **:or-nil** *or-nil***)**. This option is obsolete.

(:date-or-never :past-p *past-p* **:base-time** *base-time* **:or-nil** *or-nil***)**

The same as **(:date :past-p** *past-p* **:never-p t :base-time** *base-time* **:or-nil** *or-nil***)**. This option is obsolete.

(:past-date-or-never :base-time *base-time* **:or-nil** *or-nil***)**

The same as **(:date :past-p t :never-p t :base-time** *base-time* **:or-nil** *or-nil***)**. This option is obsolete.

:time-interval-or-never

Reads a time interval, terminated by RETURN, LINE, or END. The interval must be either "never" or alternating numbers and units of time; the units can include seconds, minutes, hours, days, weeks, or years. It returns **nil** if the user types "never". Otherwise, it returns an integer representing the number of seconds in the time interval.

Example:

```
(prompt-and-read :time-interval-or-never)
Enter a time interval, or "never": 1 day 2 hrs 13 min =>
94380.
```

(:pathname :default *default* **:visible-default** *visible-default* **:default-version** *version* **:or-nil** *or-nil***)**

Reads a pathname, terminated by RETURN, LINE, or END, merging it with a default.

:default and **:visible-default** are mutually exclusive. If either is specified, its value can be **nil**, a pathname, a pathname string, or an alist of hosts and pathnames of the sort that is the value of **fs:*default-pathname-defaults***. If the value is **nil** or a defaults alist, the default used is the result of calling **fs:default-pathname** on the value. If the value is a pathname or a pathname string, the default used is the result of calling **fs:parse-pathname** on the value. If neither **:default** nor **:visible-default** is specified, the default used is the result of **(fs:default-pathname)**.

The difference between **:default** and **:visible-default** is that if a prompt is supplied as the second argument to **prompt-and-read**, the **:visible-default** pathname is displayed to the user after the prompt, but the **:default** pathname is not. If a prompt is supplied and neither **:default** nor **:visible-default** is specified, the default pathname is not displayed. If no prompt is supplied, the default pathname is always displayed, whether it comes from **:default**, **:visible-default**, or the default default.

If **:default-version** is not specified, the default version is **nil**. If **:default-version** is specified, its value should be an integer or keyword suitable as the third argument to **fs:merge-pathnames**.

If the user just presses RETURN or LINE this option returns the default pathname. If the user just presses END this option returns the default pathname, unless **:or-nil** is specified with a value other than **nil**. In that case it returns **nil**. Otherwise this option returns the pathname the user typed, merged against the default and the default version. The default for *or-nil* is **nil**.

If the user presses COMPLETE an attempt is made to complete the pathname string typed so far. If the user presses END after typing some text, an attempt is made to complete the pathname string, and if completion is successful the merged pathname is returned.

Example:

```
(prompt-and-read
  '(:pathname :visible-default ,my-defaults-alist)
  "Enter a name"))
```

(:pathname-or-nil :default *default* **:visible-default** *visible-default* **:default-version** *version***)**
The same as **(:pathname :default** *default* **:visible-default** *visible-default* **:default-version** *version* **:or-nil t)**. This option is obsolete.

(:pathname-list :default *default* **:visible-default** *visible-default* **:or-nil** *or-nil***)**
Reads a series of pathnames, separated by commas and terminated by RETURN, LINE, or END. The meaning of **:default** and **:visible-default** is the same as for the **:pathname** option. **:pathname-list** merges the pathnames with the default and with a default version of **:newest**. It returns a list of the merged pathnames, unless *or-nil* is not **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **t**.

(:host :host-type *type* **:default** *default* **:or-nil** *or-nil***)**
Reads the name of a host, terminated by RETURN, LINE, or END.

:host-type is a keyword that determines what kind of input is acceptable:

:physical	The name of a network host. This is the default.
:chaos-only	The name of a network host on the chaosnet.
:or-local	The name of a network host or "local", meaning the local host.
:pathname	The name of a pathname host or "local", meaning the local host.
:or-pathname	The name of a network host, a

pathname host, or "local", meaning the local host.

If **:default** is specified, it should be a network host or the name of a host as a symbol or string. If **:default** is specified and the user just presses RETURN, LINE, or END, it returns the host specified by **:default**.

If **:default** is not specified or is **nil**, **:or-nil** is specified with a value other than **nil**, and the user just presses RETURN, LINE, or END, it returns **nil**. Otherwise, it returns the host object whose name the user types. The default for *or-nil* is **nil**.

(:host-or-local :default *default* **:or-nil** *or-nil***)**
The same as **(:host :host-type :or-local :default** *default* **:or-nil** *or-nil***)**. This option is obsolete.

(:pathname-host :default *default* **:or-nil** *or-nil***)**
The same as **(:host :host-type :pathname :default** *default* **:or-nil** *or-nil***)**. This option is obsolete.

(:host-list :host-type *host-type* **:or-nil** *or-nil***)**
Reads a series of names of network hosts, separated by spaces or commas, and terminated by RETURN, LINE, or END. **:host-type** has the same meaning as for the **:host** option. **:host-list** returns a list of the host objects whose names the user types, unless *or-nil* is not **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **t**.

(:keyword :or-nil *or-nil***)**
Reads the name of a symbol to be interned in the **zl-user:keyword** package, terminated by RETURN, LINE, or END. The symbol name should not have a package prefix (that is, it should not be preceded by a colon). Lowercase letters in the name are converted to upper case. **:keyword** returns the keyword symbol whose name the user types, unless **:or-nil** is specified with a value other than **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **nil**.

(:keyword-list :or-nil *or-nil***)**
Reads a series of names of symbols to be interned in the **zl-user:keyword** package, separated by spaces or commas, and terminated by RETURN, LINE, or END. The

symbol names should not have package prefixes (that is, they should not be preceded by colons). Lower-case letters in the names are converted to upper case. **:keyword-list** returns a list of keyword symbols whose names the user types, unless *or-nil* is not **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **t**.

(:font :or-nil *or-nil***)**

Reads the name of a font, terminated by RETURN, LINE, or END. The font name should not have a package prefix (that is, it should not be preceded by **fonts:**), and it must be the name of a known font. **:font** returns the font (not the symbol) whose name the user types, unless **:or-nil** is specified with a value other than **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **nil**.

(:font-list :or-nil *or-nil***)**

Reads a series of names of fonts, separated by spaces or commas, and terminated by RETURN, LINE, or END. The font names should not have package prefixes (that is, they should not be preceded by **fonts:**), and they must be names of known fonts. **:font-list** returns a list of the fonts (not the symbols) whose names the user types, unless *or-nil* is not **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **t**.

(:object :class *class* **:or-nil** *or-nil***)**

Reads the name of an object in the network namespace, terminated by RETURN, LINE, or END. *class* is a keyword representing a namespace class or a string that is the print name of a class keyword. You must supply this argument. **:object** returns the namespace object whose name the user types, unless **:or-nil** is specified with a value other than **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **nil**.

(:object-list :class *class* **:or-nil** *or-nil***)**

Reads a series of names of objects in the network namespace, separated by spaces or commas, and terminated by RETURN, LINE, or END. *class* is a keyword representing a namespace class or a string that is the print name of a class keyword. You must supply this argument. **:object-list** returns a list of the namespace objects whose names the user types, unless *or-nil* is not **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **t**.

(:class :or-nil *or-nil***)**

Reads the name of a network namespace class, terminated by RETURN, LINE, or END. The name should not contain a package prefix (that is, it should not be preceded by a colon). It returns the keyword representing the class whose name the user types, unless **:or-nil** is specified with a value other than **nil** and the user just presses RETURN, LINE, or END. In that case it returns **nil**. The default for *or-nil* is **nil**.

Streams are permitted to have a handler for **:prompt-and-read** messages. The **prompt-and-read** function first determines whether the **zl:query-io** stream handles the **:prompt-and-read** message. If so, it sends a **:prompt-and-read** message with its own arguments on to the stream. The stream returns several values. The first value the stream returns says whether or not it wants to handle the interaction with the user itself. It returns **nil** to indicate that it declines to handle the message, in which case the **prompt-and-read** function continues its normal action of prompting the user. When the first value is not **nil**, the **prompt-and-read** function returns the rest of the values to its caller.

define-prompt-and-read-type *keyword parameter-list description* &body *body* *Special Form*

Defines a new type keyword for **prompt-and-read**. Defines a dispatch function to be called to get input from the user when **prompt-and-read** is called with a type keyword of *keyword*. The dispatch function is defined as the **prompt-and-read** property of *keyword*, which can be a symbol in any package. Its parameter list is derived from *parameter-list*, and its body is *body*. **prompt-and-read** returns whatever the dispatch function returns.

If the first argument to **prompt-and-read** is just *keyword*, the dispatch function is called with no arguments. If the first argument to **prompt-and-read** is (*keyword* . *type-args*), the arguments to the dispatch function are the elements of *type-args*. These are a series of alternating keywords and values.

parameter-list is **nil** if no *type-args* are allowed, or else a list of **&key** elements for the dispatch function's parameter list. **define-prompt-and-read-type** inserts **&key** in the parameter list itself; **&key** should not appear in *parameter-list*.

description can be **nil**, a **zl:format** control string, a list of a **zl:format** control string and **zl:format** control args, or a form to be evaluated. *description* is used to generate *input-type* in the default prompt, "Enter *input-type*: ":

description	*input-type*
nil	"a " followed by the print name of the type keyword.
zl:format control string	Generated by calling **zl:format** with arguments of **t** and the control string when it is time to display the prompt.
list of **zl:format** control string and args	Generated by calling **zl:format** with arguments of **t**, the control string, and the control args when it is time to display the prompt. The control args can examine any of the parameters in *parameter-list*.
form	Generated by evaluating the form when it is time to display the prompt. The form can examine any of the parameters in *parameter-list*. It should send output to **zl:standard-output**.

body is the body of the dispatch function. Often the body is a call to a more primitive reading function, such as **zl:read** or **zl:readline**. It is the responsibility of the body or a function it calls to provide input editing if needed.

Example:

```
(define-prompt-and-read-type :flavor-name
                             ((impossible-is-ok t))
                             "the name of a flavor"
  (sys:read-flavor-name query-io impossible-is-ok))
```

sys:read-flavor-name is a function that reads a flavor name with completion over the set of defined flavors.

PART II.

Using the Window System

10. Introduction to Using the Window System

"Using the Window System" 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 part 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 section does not explain how to extend the window system by defining your own flavors.

Most of the window system concepts and facilities covered in this part apply to Dynamic Windows as well as static windows. This is explicitly mentioned in a number of places. Where the two kinds of windows diverge, we also point that out. The reference documentation for **dw:dynamic-window** refers you to the particular sections in this part that describe facilities for use with Dynamic Windows: See the flavor **dw:dynamic-window** in *Programming the User Interface, Volume A*. For more general information on the relationship of static and Dynamic Windows: See the section "Overview of Window Substrate Facilities" in *Programming the User Interface, Volume A*.

To get the most out of this material, you should have a working familiarity with Symbolics Common Lisp. You should also have some experience with the user interface of the Symbolics 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. See the section "Flavors" in *Symbolics Common Lisp*.

11. Concepts

11.1 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.

The window system controls the keyboard, encoding the shifting keys, interpreting special commands such as the FUNCTION and SELECT 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.

11.2 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*** 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.

11.3 Hierarchy of Windows

Several Lisp Machine system programs and application programs present the user with a window that is split up into several sections, which are usually called *window panes* or *panes*. For example, the Inspector has six panes in its default configuration: the one you type forms into at the top, the menu, the history list, and the three inspection panes below the first three. The window Debugger and Zmail also use elaborate windows with panes. 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. For more on the screen manager: See the section "The Screen Manager", page 96.

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.

11.4 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.) 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, 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. See the section "Screen Arrays and Exposure", page 89.

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. See the section "The Screen Manager", page 96.

11.5 Screen Arrays and Exposure

This section discusses the concepts of screen arrays and of exposed windows. These have to do with how the system decides where to put a window's contents (its pixels), how the notion of visibility on the screen is extended into a hierarchy of windows, and how 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.

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).

The window system uses conformal indirect arrays for its screen arrays. This means that on the 3600 the bit-array in which a window saves its bits when it is not visible does not have to be the full width of the screen; it is just the width of the window, rounded up to the next multiple of 32 bits. Screen arrays do not use multilevel indirection; the screen array of a nonscreen sheet always indirects either to a bit-save array or to the screen array of its screen. The screen array of a screen is always a displaced array to the hardware screen buffer.

11.6 Window Exposure and 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.

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; that is, 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**. The default response to this is to notify the user that the window wants to type out and to make the window "interesting" so that `FUNCTION 0 S` can select it. Windows in the Terminal program 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 or 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: See the section "Temporary Windows", page 95.

tv:sheet-force-access (*sheet don't-prepare-sheet*) *body...* *Special Form*

Allows typeout on *sheet* if it has a screen array (that is, if it is exposed or has a bit-save array). If *don't-prepare-sheet* is **nil**, prepares the sheet before executing *body*. If *sheet* does not have a screen array, **tv:sheet-force-access** just returns without executing *body*. Use this to put output onto a deexposed window that has a bit-save array.

tv:prepare-sheet (*sheet*) *body...* *Special Form*

Prepares *sheet* for input or output. Ensures that *sheet* is not locked or in output-hold. Opens blinkers on inferiors and exposed superiors.

11.7 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) or 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.

11.8 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 visual impression of overlapping pieces of paper on a desktop; the deexposed 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 deexposed window that might be displayed in a given area of the screen, then the screen manager uses its priority ordering 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 not only manages 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. 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:

1. Exposed windows go in front of nonexposed windows.

2. If two windows are both exposed or both have the same value of priority, their order is not changed by the sorting.

3. If two nonexposed 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 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 (that is, **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 another 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.

The screen manager has a facility for *graying* areas of the screen that contain no windows or windows that are not fully exposed. See the section "Window Graying", page 101.

11.9 Window Graying

Screens and frames can *gray* areas that contain no windows or that contain windows that are not fully exposed. To gray an area of the screen is to cover it with a semitransparent texture pattern. There are two kinds of graying:

- *Background gray* is used to fill in areas of the screen that don't contain any windows. Normally this is just the borders around the screen, but if you reshape all the full-screen windows to be smaller, so that there is some area of the screen that doesn't have a window on it, the background gray appears there, also. The background gray in the two areas (the part of the screen where you can put windows and the part of the screen where you cannot put windows) joins smoothly.

- *Deexposed gray* is used to fill in the visible portion of a window that is not fully exposed. It tells you that you aren't seeing all of this window, because another window is covering part of it. Deexposed graying does not occur when a window is covered by a temporary window (like a momentary menu) because such a window isn't considered to be really deexposed and is often still a focus of the user's attention.

These concepts generalize to any window, dynamic or static, that has inferiors, not just the screen. You can make a flavor of frame that fills in any empty spots with gray or grays over any partially exposed panes.

Both kinds of graying are implemented by the screen manager, but are affected by messages to the screen and to the deexposed windows.

To disable both background and deexposed gray on the main screen:

```
(tv:set-screen-background-gray nil)
(tv:set-screen-deexposed-gray nil)
```

To get a light gray on both unused areas and deexposed windows:

```
(tv:set-screen-background-gray tv:6%-gray)
(tv:set-screen-deexposed-gray tv:6%-gray)
```

To get a light gray over deexposed windows and a darker gray in the background:

```
(tv:set-screen-background-gray tv:33%-gray)
(tv:set-screen-deexposed-gray tv:6%-gray)
```

11.9.1 Window Graying Specifications

A *graying specification* determines what pattern to use in graying areas of the screen that contain no windows or that contain windows that are not fully exposed. These specifications are used as arguments to functions and messages that deal with graying. See the section "Functions, Flavors, and Messages for Window Graying", page 103.

Following are the possible values of a specification and their meanings:

nil Disable graying. Background gray is white (in black-on-white mode); deexposed gray is completely transparent.

Two-dimensional bit array
A stipple pattern to be replicated by **bitblt**.

:white Opaque white.

:black Opaque black.

Instance An object that must handle the **:draw-blank-rectangle** message to draw a gray rectangle.

Function A function to be called with standard arguments to draw a gray rectangle.

List The first element is a function to be called, and the remaining elements are arguments to the function to be supplied after the standard arguments.

Following are the arguments to the **:draw-blank-rectangle** message and to a function to be called:

x-size Horizontal size of the rectangle in pixels.

y-size Vertical size of the rectangle in pixels.

x-pos X-position of the top left corner of the rectangle on *sheet.*

y-pos Y-position of the top left corner of the rectangle on *sheet.*

x-phase Starting x-coordinate of the source array.

y-phase Starting y-coordinate of the source array.

alu Alu function for drawing the rectangle.

sheet Sheet or array on which to draw the rectangle.

The variable **tv:*gray-arrays*** contains a list of variables that are bound to available predefined graying specifications.

tv:*gray-arrays* *Variable*

A list of variables bound to predefined graying specifications. You can use one of these as the source of a pattern for background or deexposed window graying. You can also make your own graying specifications and add them to this list. See the section "Window Graying Specifications", page 102.

11.9.2 Functions, Flavors, and Messages for Window Graying

tv:set-screen-background-gray *gray* &optional (*screen* **tv:main-screen**) *Function*

Specifies what pattern should be used to gray areas of a screen or frame that contain no windows. *gray* is a graying specification: See the section "Window Graying Specifications", page 102. Give an argument of **nil** to disable graying.

screen can be a screen or frame. It defaults to the main monochrome screen.

tv:set-screen-deexposed-gray *gray* &optional (*screen* **tv:main-screen**) *Function*

Specifies what pattern should be used to gray areas of a screen or frame that contain windows that are not fully exposed. *gray* is a graying specification: See the section "Window Graying Specifications", page 102. Give an argument of **nil** to disable graying.

screen can be a screen or frame. It defaults to the main monochrome screen.

:screen-manage-deexposed-gray-array *Message*

The screen manager sends this message to deexposed windows to give them an opportunity to override the kind of graying that their superior (or the screen) wants to provide. This message should return two values. Following are the possible pairs of values and their meaning:

graying specification and **nil**
Use *graying specification* to gray the window.

nil and **nil** Let the superior decide how to gray the window.

nil and **t** Disable graying of the window.

See the section "Window Graying Specifications", page 102.

tv:gray-unused-areas-mixin *Flavor*

This flavor, mixed into a screen or a frame, gives it the ability to gray areas within it that contain no windows.

:gray-array-for-unused-areas *gray* (for **tv:gray-unused-areas-mixin**) *Init Option*

Specifies *gray* as the graying specification to use in graying areas of this screen or frame that contain no windows. See the section "Window Graying Specifications", page 102.

:gray-array-for-unused-areas of **tv:gray-unused-areas-mixin** *Method*

Returns the graying specification that this frame or window uses in graying areas that contain no windows. See the section "Window Graying Specifications", page 102.

:set-gray-array-for-unused-areas *gray* of **tv:gray-unused-areas-mixin** *Method*

Sets *gray* as the graying specification to use in graying areas of this screen or frame that contain no windows. See the section "Window Graying Specifications", page 102.

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

This flavor, mixed into a screen or a frame, gives it the ability to gray areas within it that contain windows that are not fully exposed.

:gray-array-for-inferiors *gray* (for **tv:gray-deexposed-inferiors-mixin**) *Init Option*

Specifies *gray* as the graying specification to use in graying areas of this screen or frame that contain no windows. See the section "Window Graying Specifications", page 102.

:gray-array-for-inferiors of **tv:gray-deexposed-inferiors-mixin** *Method*

Returns the graying specification that this frame or window uses in graying areas that contain no windows. See the section "Window Graying Specifications", page 102.

:set-gray-array-for-inferiors *gray* of **tv:gray-deexposed-inferiors-mixin** *Method*

Sets *gray* as the graying specification to use in graying areas of this screen or frame that contain no windows. See the section "Window Graying Specifications", page 102.

11.10 Windows and Processes

tv:process-mixin *Flavor*

Creates a new process associated with each window of the dependent flavor. (The Dynamic Window flavor **dw:program-frame**, used by **dw:define-program-framework**, includes this mixin.)

:process (*initial-function* **.** *options*) (for **tv:process-mixin**) *Init Option*
options are options to **make-process.**

11.11 Activities and Window Selection

The concepts and facilities discussed in this section apply to both Dynamic Windows and static windows.

11.11.1 The Selected Window and the Selected Activity

When you type characters on the keyboard, they must be directed to some window. The window that receives keyboard input is the *selected window.* No more than one window can be selected at a time. Sometimes no window is selected, but usually this is a brief transitional state.

tv:selected-window *Variable*
The value of this variable is the currently selected window.

tv:cold-load-stream-old-selected-window *Variable*
At a cold-load-stream break, the value of this variable is the value of **tv:selected-window** at the time you entered the cold-load stream.

A window is selectable only if it has **tv:select-mixin** and **tv:stream-mixin** as components (**dw:dynamic-window** has both). **tv:select-mixin** allows the window to handle messages that select it. **tv:stream-mixin** provides the window an *I/O buffer*, which accumulates keyboard characters, and lets the window handle messages to get input. **tv:stream-mixin** also provides the window with *input editing.* When input editing is enabled and a reading function tries to get input from the window, the user can edit typein before the reading function sees it. See the section "Input From Windows", page 147.

An *activity* is a group of windows that the user regards as a single unit. Typically an activity consists of a top-level window – one that is a direct inferior of a screen – and all its direct and indirect inferior windows. An example of an activity is a top-level Lisp Listener. Sometimes an activity consists of a non-top-level window and all its direct and indirect inferior windows. One example is a Lisp Listener inside a Split Screen frame.

The concept of activity is only partially implemented in the window system. No separate object represents an activity. Instead, an activity is designated by a representative window from that activity. In the usual case, where the windows in an activity form a tree, the root of the tree serves as the representative.

The system contains several generic tools for selecting among activities: These include the SELECT key, FUNCTION S, and the [Select] menu in the System Menu.

The *selected activity* is the activity that contains the selected window. When you change the selected activity, you also change the selected window.

You usually select an activity by selecting the representative window of the activity. But this window might or might not be selectable itself; sometimes only its inferiors, or only some of its inferiors, can become the selected window. When you select an activity, the representative window of the activity usually decides which window within the activity should become the selected window.

We say that this window – the one that is to become the selected window when the activity is selected – is selected *relative* to its activity. When you select a window relative to its activity, you do not change the selected activity. If an activity happens to be the selected activity, then selecting a window relative to that activity also makes that window the new selected window. If an activity is not the selected activity, then selecting a window relative to that activity changes neither the selected activity nor the selected window.

Whenever you select a window that is part of an activity, that window is selected relative to its activity, and that activity becomes the selected activity.

11.11.2 Frames and Panes

A *frame* is a window that is designed to contain other windows inside it. A direct inferior window of a frame is called a *pane*. Many activities consist of a frame and its direct and indirect inferior windows. The frame is the representative window of this kind of activity.

A window that is a direct or indirect inferior of a frame can be the *selected-pane* of the frame. The *selected-pane* is the window that is selected relative to the frame. A frame usually cannot become the selected window. Instead, when you select a frame, its selected-pane becomes the selected window, unless the selected-pane is itself a frame. In that case the selected-pane of the selected-pane becomes the selected window.

You can change the selected-pane of a frame without selecting the activity that the frame represents. The next time that activity is selected, the new selected-pane becomes the selected window. If that activity happens to be the selected activity, then changing the selected-pane of the frame causes the new selected-pane to become the selected window.

If you select a window that is a pane of a frame, that window becomes the selected-pane of the frame, and the activity that the frame represents becomes the selected activity.

For more about panes and frames, including constraint frames: See the section "Frames", page 204.

11.11.3 Messages About Window Selection

:alias-for-selected-windows *Message*

When the **:alias-for-selected-windows** message is sent to a window, it returns the representative window of the receiver's activity. If two windows have the same alias-for-selected-windows, they belong to the same activity.

This message is sent by both the system and the user and may be received by either, although usually the system-supplied methods suffice. The default method (of **tv:sheet**) returns the window to which the message is sent, declaring the window to be in an activity by itself. **tv:select-relative-mixin** supplies a method that returns the superior's alias, unless the window to which the message is sent is a top-level window (that is, its superior is a screen); in that case it returns the window itself. **tv:pane-mixin** and **tv:basic-typeout-window** supply methods that return the superior's alias.

:name-for-selection *Message*

The **:name-for-selection** message to a window returns **nil** if the window is not supposed to be selected. Otherwise, it returns a string that serves as the name of the window in menus of selectable windows.

This message is sent by many parts of the user interface. Some use it just as a predicate; others put the returned string into a menu.

This message is usually received by the user. The default method (of **tv:sheet**) returns **nil**. **tv:select-mixin** provides a method that computes a name based on the window's label, if it has one, or else on the window's name.

Many application programs shadow this method and supply their own. This is especially so in the case of program frames. Typically, you do not want pane names to show up in select menus. The recommended procedure for addressing this issue is:

1. Make your frame's panes include **tv:pane-no-mouse-select-mixin** instead of **tv:pane-mixin** if you do not want them showing up in menus.

2. Give your frame a name that you do want to show up in menus.

3. If you want the name to be something separate, or if you have some panes that are menu-selectable for some reason, provide your own **:name-for-selection** method for the frame.

:selectable-windows *Message*

The **:selectable-windows** message to a window returns a menu item-list of activities containing or inferior to the window. The **:name-for-selection** and **:alias-for-selected-windows** messages are used to discover the available activities. When sent to a screen, this message returns a menu item-list of all the activities that screen contains.

This message is sent by [Select] in the System Menu and is received by the system. Users shouldn't need to send this message or to define methods for it.

:select-relative *Message*

The **:select-relative** message to a selectable window selects the window relative to its activity, but doesn't select a different activity.

If the window that receives this message belongs to the same activity as the currently selected window, the receiver becomes the new selected window. Otherwise, the window that receives this message sends the **:inferior-select** message to its superior to select the receiver relative to its activity.

User programs should send the **:select-relative** message rather than **:select** or **:mouse-select**, unless they are really responding to a user command to switch activities. Using **:select-relative** rather than **:select** to change windows within an activity ensures that the right thing happens when that activity is not the selected one and avoids suddenly changing the selected activity without the consent of the user.

This message returns no significant values. It is sent by the user and received by the system. Users should not need to define methods for it.

:inferior-select *sheet* *Message*

The **:inferior-select** message to a window returns non-**nil** if it is okay to select *sheet*, or **nil** if it is not okay. If the message returns **nil**, presumably some appropriate action such as selecting a different window has already been performed.

This message is sent and received by the system. It is normally sent under two circumstances:

- If a window is selected, and if the window includes a flavor that makes it participate in its superior's activity, the window sends its superior an **:inferior-select** message with itself as the argument. Flavors that make windows participate in their superiors' activities include **tv:select-relative-mixin**, **tv:pane-mixin**, and **tv:basic-typeout-window**.

- If a window receives a **:select-relative** message and the window's activity is not the currently selected activity, it sends its superior an **:inferior-select** message with itself as the argument.

The **:inferior-select** message is propagated upwards through all levels of the window hierarchy until it reaches a screen. This informs the direct and indirect superiors of window that it has been selected (or selected relative to its activity). When a frame receives an **:inferior-select** message, it saves *sheet* as its selected-pane and passes the message on, substituting itself for *sheet.*

All currently extant methods return a non-**nil** value. Only panes look at the returned value; they don't allow themselves to be selected if the returned value is **nil**. This permits a frame to refuse to allow its selected-pane to be changed.

:select-pane *pane* — *Message*

The **:select-pane** message to a frame makes *pane* the selected-pane of the frame. *pane* must be either an exposed inferior of the frame or **nil**, which means to set the selected-pane to **nil**. This message also deselects the current selected-pane if it is a window different from *pane*. Unless *pane* is **nil**, this message sends *pane* a **:select-relative** message.

:selected-pane — *Message*

The **:selected-pane** message to a frame returns the selected-pane of the frame. This message is sent by users and received by the system.

:selected-pane *pane* (for **tv:basic-constraint-frame**) — *Init Option*

Makes *pane* the selected-pane of this frame. *pane* can be the symbol used in the **:panes** init option to name the pane.

:mouse-select &optional (*save-selected* **t**) — *Message*

The **:mouse-select** message to a window selects the window as a result of a user command, usually clicking the mouse on it. This takes care of various window system issues, such as making sure that typeahead goes to the correct activity and getting rid of any temporary windows that are covering this window, preventing it from being exposed.

The operation fails and returns **nil** if this window is not contained inside its superior (it might be too large), which prevents it from being exposed. The operation can also fail and return **nil** if the message is sent to a frame whose selected-pane is **nil**. If the operation succeeds, the message returns **t**.

If *save-selected* is not **nil**, the previously selected activity is saved for restoring by the `FUNCTION S` command and the **:deselect** message.

The **:mouse-select** message to a pane (a window with **tv:pane-mixin**) selects the activity of which the pane is a part, without changing its selected-pane. Thus, the message does not necessarily select the window to which it is sent; it might select some other window in the same activity. **:mouse-select** is intended to be a command for switching activities.

User programs should send the **:select-relative** message rather than **:select** or **:mouse-select**, unless they are really responding to a user command to switch activities. Using **:select-relative** rather than **:mouse-select** or **:select** to change windows within an activity ensures that the right thing happens when that activity is not the selected one and avoids suddenly changing the selected activity without the consent of the user.

This message is sent by many parts of the user interface.

This message is usually received by the system, although users could define methods for it: either a method that returns **nil** to prevent a window from being selected, or a daemon. The default method is defined on **tv:essential-window**.

:select &optional (*save-selected* **t**) *Message*

The **:select** message is sent to a selectable window by a user program or by a part of the user interface to change the selected activity. It is also sent by the system to notify a window when it becomes the selected window, either because of a change of activities or because of selection of this window instead of a different window within the same activity.

This message is received by the system and is also received by user daemons that wish to be notified when a window becomes selected.

If *save-selected* is not **nil**, the previously selected activity is saved for restoring by the `FUNCTION S` command and the **:deselect** message.

The message returns **t** if it works, **nil** if it fails. It can fail when sent to a pane if the **:inferior-select** message that the pane sends to the frame returns **nil**. It can also fail when sent to a frame that has no selected-pane.

User programs should send the **:select-relative** message rather than **:select** or **:mouse-select**, unless they are really responding to a user command to switch activities. Using **:select-relative** rather than **:select** to change windows within an activity ensures that the right thing happens when that activity is not the selected one and avoids suddenly changing the selected activity without the consent of the user.

:deselect &optional (*restore-selected* **t**) *Message*

The **:deselect** message is sent to a selectable window by a user program or by a part of the user interface to change the selected activity. It is also sent by the system to notify a window when it ceases to be the selected window, either because of a change of activities or because of selection of a different window within the same activity. When sent by the system as a notification of deselection, *restore-selected* is always **nil**.

This message is received by the system and is also received by user daemons that wish to be notified when a window becomes deselected. Note that this message can be sent to a window that is not the selected window; in that case it is supposed to do nothing.

If **:deselect** is sent to the selected window and *restore-selected* is not **nil**, the previously selected activity is selected.

11.11.4 Flavors Related to Window Selection

tv:select-mixin *Flavor*

This flavor allows a window to be selectable. It provides methods for the **:select**, **:deselect**, **:select-relative**, and **:name-for-selection** messages.

tv:select-relative-mixin *Flavor*

This flavor makes a window participate in the same activity as its superior. It provides a method for the **:alias-for-selected-windows** message that returns the window if its superior is a screen, or the superior's alias otherwise. It also provides a daemon for the **:select** message that sends an **:inferior-select** message to the superior with an argument of the window.

This flavor does not provide a method for the **:select-relative** message; that is handled by **tv:select-mixin**.

tv:dont-select-with-mouse-mixin *Flavor*

This flavor provides a **:name-for-selection** message that returns **nil**, so that the user interface does not treat the window as a candidate for selection.

tv:basic-frame *Flavor*

This flavor provides methods that allow the frame to serve as the representative window of its activity. Usually a frame cannot become the selected window, but this flavor provides methods that handle messages about selection, typically by operating on the selected-pane instead of the frame. The **:select**, **:deselect**, and **:select-relative** methods just pass these messages on to the selected-pane when one exists; otherwise they return **nil**.

This flavor provides a handler for the **:select-pane** message that decides which pane should be selected when the activity is selected. The **:inferior-select** method saves the argument as the selected-pane and sends the message on to the frame's superior with the frame as argument. The **:name-for-selection** method returns the name-for-selection of the selected-pane if a selected-pane exists and has a name-for-selection; otherwise, the method returns the name of the frame.

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, the flavor **tv:window-pane**, used when you want a pane of a frame that understands everything that **tv:window** does, is defined as follows:

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

Among other things, **tv:pane-mixin** provides methods that let the pane participate in its superior's activity. The **:alias-for-selected-windows** method returns the superior's alias. When a window of this flavor receives a **:select** message, it first sends its superior an **:inferior-select** message. If the **:inferior-select** message returns **nil**, the **:select** message fails and just returns **nil**. When a window of this flavor receives a **:mouse-select** message, it passes the message on to its superior.

tv:pane-no-mouse-select-mixin *Flavor*

A mixin flavor to make a window a pane of a frame and ensure that it cannot be selected from a system menu. This flavor includes **tv:pane-mixin** and **tv:dont-select-with-mouse-mixin**.

11.11.5 Selecting a Window Temporarily

tv:window-call-relative (*window* &optional *final-action* &rest *final-action-args*) &body *body* *Special Form*

Temporarily selects a window relative to its activity, executes the body, then (in an **unwind-protect**) restores the previous selected-pane of that activity. This uses the **:select-relative** message.

window is a variable that is bound to the window to be selected. If *final-action* is specified, it is a message to be sent to *window* when done with it, and *final-action-args* are forms supplying arguments to that message. *final-action* is often **:deactivate**.

tv:window-call-relative is preferred over **tv:window-call** for use by application programs that are not responding to an explicit user command to switch activities.

tv:window-call (*window* &optional *final-action* &rest *final-action-args*) &body *body* *Special Form*

Temporarily selects a window – selecting a new activity if the window is not part of the currently selected activity – executes the body, then (in an **unwind-protect**) usually restores the previously selected activity. The previously selected activity is not restored if at that time the selected window is not *window* or a direct or indirect inferior of it. This heuristic deals with the case where the user has switched activities explicitly during the execution of *body*.

This uses the **:select** message but is different from using the *save-selected* and *restore-selected* arguments to **:select** and **:deselect**: **tv:window-call** restores the activity that was current when its execution began, not the second most recently selected activity, as sending a **:deselect** message with an argument of **t** would.

window is a variable that is bound to the window to be selected. If *final-action* is specified, it is a message to be sent to *window* when done with it, and *final-action-args* are forms supplying arguments to that message. *final-action* is often **:deactivate**.

tv:window-call-relative is preferred over **tv:window-call** for use by application programs that are not responding to an explicit user command to switch activities.

tv:window-mouse-call (*window* &optional *final-action* &rest *final-action-args*) &body *body* — *Special Form*

This is similar to **tv:window-call** but uses **:mouse-select** instead of **:select** to select *window*. It is used by parts of the user interface that want the temporary-window-clearing features of **:mouse-select**.

11.12 Window Status

The following methods respectively determine and set the status of a window. They may be used with static or Dynamic Windows.

:status of **tv:essential-activate** — *Method*

Returns one of **:deactivated**, **:deexposed**, **:exposed**, **:selected**, and **:exposed-in-superior**, indicating the current status of a window.

:set-status *new-status* of **tv:essential-activate** — *Method*

Sets the status of a window to **:deactivated**, **:deexposed**, **:exposed**, or **:selected**

12. Window Flavors and Messages

12.1 Overview of Window Flavors and Messages

In this section we 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 section 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). Prior to Genera 7.0, fonts for character output to a window were manipulated directly through various font messages. Currently, only a couple of these messages are supported. The preferred interface to character fonts is the *character style* system. Each window has a default character style, which you can specify as an init option: See the init option **(flavor:method :default-character-style tv:sheet)**, page 120. To override the default style, you can use one of several character style macros: See the section "Overview of Character Environment Facilities" in *Programming the User Interface, Volume A.* For more information on character styles generally: See the section "Character Styles" in *Symbolics Common Lisp.*

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, 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 **zl:read** or **zl:readline**) can be passed a window. Each window has an *I/O 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 I/O buffer; frequently two processes communicate by one process's forcing keyboard input into an I/O 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 status 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. 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.

12.2 Getting a Window to Use

12.2.1 Flavors of Basic Windows

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. **tv:minimum-window** itself is built on a number of other flavors that provide the "essential" attributes of windows. For reference, **tv:minimum-window** is defined as follows (ignoring **defflavor** options):

```
(defflavor tv:minimum-window ()
  (tv:essential-expose tv:essential-activate
   tv:essential-set-edges tv:essential-mouse
   tv:essential-window))
```

tv:essential-window, in turn, is built on the base flavor for all windows, **tv:sheet.**

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. **tv:window** has what 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: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.

For information on Dynamic Window flavors: See the section "Overview of Window Substrate Facilities" in *Programming the User Interface, Volume A.*

12.2.2 Creating a Window

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

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

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**). 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:

```
(defun make-window-example ()
  (let ((window (tv:make-window 'tv:window
                                :edges-from :mouse
                                :expose-p t
                                :blinker-p t
                                :default-character-style
                                  '(:fix :bold :large)
                                :save-bits t)))
    (format window "~2%Note the character style")))
```

The above function lets you specify the location of the upper-left and lower-right corners of the window with the mouse. Once the location is specified, the window is created and exposed. A blinker is visible; its size is that of the default character style for character output. Because the **:save-bits** init option is **t**, the formatted output to the window will still be visible after the window is de-exposed and then re-exposed.

:init *init-plist* of **tv:sheet** *Method*

Sets initial characteristics of the window, processing options in *init-plist*. This message is sent by the system; you might need to supply an **:after** daemon for it.

:blinker-p *t-or-nil* (for **tv:sheet**) *Init Option*

Boolean option specifying whether to provide a blinking cursor when the window is exposed; the default is **t**. For more information on blinkers: See the section "Blinkers", page 160.

:default-character-style *character-style* (for **tv:sheet**) *Init Option*

Specifies the character style for character output to the window. The default style is inherited from the screen (and is settable via the Set Screen Options command); the initial default character style is `(:fix :roman :normal)`. To change a window's default style, use the :set-default-style method: See the method **(flavor:method :set-default-character-style tv:sheet)**, page 126.

For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp*.

:save-bits *t-or-nil* (for **tv:sheet**) *Init Option*

Boolean option specifying whether output to the window is written to a bit-save array when the window is de-exposed; the default is **nil**. If **t**, the output is redisplayed following re-exposure of the window. For more information on bit-save arrays: See the section "Pixels and Bit-Save Arrays", page 88.

:superior *superior* (for **tv:sheet**) *Init Option*

Makes *superior* the superior window of the window being created.

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

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* (for **tv:essential-window**) *Init Option*

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).

tv:defwindow-resource *name parameters* &rest *options* *Macro*

Defines a resource of windows. *name* is the name of the resource. *parameters* is a lambda-list of parameters to **defresource**. *options* are alternating keywords and values:

Keyword	*Value*
:initial-copies	Number of windows to be created during evaluation of **defresource** form. Default: 1.
:superior	A form to be evaluated when the resource is allocated to return the superior window of the desired window. If this is not supplied, the superior is the value of **tv:mouse-sheet**.
:make-window	List of flavor name and options to **tv:make-window**, which will be called to make new windows. One of the options can be **:superior**.
:constructor	A form or the name of a function to make new windows. You must supply either **:make-window** or **:constructor**.
:reusable-when	Either **:deexposed** or **:deactivated**. Specifies when a window can be reused. Supply this when you use **allocate-resource** instead of **using-resource** to allocate resources. Default: reusable when not locked and not in use.

12.3 Character Output to Windows

The information included in this section applies to both Dynamic Windows and static windows.

12.3.1 How Windows Display Characters

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:sheet** implements the messages of the Lisp Machine output stream protocol. It implements a large number of optional messages of that protocol, such as **:insert-line**. The **tv:sheet** flavor is a component of all windows. 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; 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.)

When a character is drawn, it is combined with the existing contents of the pixels of the window according to an *alu function.* For a description of the different alu functions: See the section "Graphic Output to Windows", page 132. 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.

The *character style* of characters output to the window is gotten by merging the character style specified for the output against the window's *default character style.* The resulting style maps to a particular font. (For more information on character styles: See the section "Character Styles" in *Symbolics Common Lisp*. For more on specifying output character styles: See the section "Overview of Character Environment Facilities" in *Programming the User Interface, Volume A*. To specify a window's default character style: See the init option **(flavor:method :default-character-style tv:sheet)**, page 120.)

Details of fonts are gone into later: See the section "TV Fonts", page 155. For now, it is only important 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.

Character-width is the *char-width* attribute – the width of a space character – of the font currently being used for character output, that is, the *current font.* The line-height is the sum of the *vsp* of the window and the *char-height*s of the current font. 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 font is, plus vertical spacing added between lines by controlling the *vsp* of the window.

In some fonts, all characters have the same width; these are called *fixed-width* *fonts*. The default character style for the system, `(:fix :roman :normal)`, maps to a fixed-width font (**fonts:cptfont**) for character output to windows. 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; nonprinting 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 the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. 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 the section "Window Exposure and Output", page 93. 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:sheet** 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-rest-of-line** 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 (for example, 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:sheet** flavor does a **:clear-rest-of-line** 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-rest-of-line** 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, 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 (for example, 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:sheet** flavor advances the cursor to the next line just as typing a **#\return** character does normally. 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, 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:truncating-lines-mixin.**

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.

Normally **#\return** moves the cursor position to the inside left edge of the window and down by one line-height, and clears the line. 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.

12.3.2 Messages to Display Characters on Windows

:tyo *ch* of **tv:sheet** *Method*

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***)* of **tv:sheet** *Method*

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***)* of **tv:sheet** *Method*

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 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.

:fresh-line of **tv:sheet** *Method*

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.

:insert-char &optional (*char-count* **1**) (*unit* **':character**) of **tv:sheet** *Method*

Open up a space the width of *char-count* units 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. If *unit* is **:character**, *char-count* is interpreted as the number of character-widths to insert; if *unit* is **:pixel**, *char-count* is interpreted as the number of pixels to insert.

:insert-string *string* &optional (*start* **0**) (*end* **nil**) (*type-too* **t**) of **tv:sheet** *Method*

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 (*line-count* **1**) (*unit* **':character**) of **tv:sheet** *Method*

Take the line containing the cursor and all the lines below it, and move them down by *line-count* units. 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. Lines pushed off the bottom of the window are lost. If *unit* is **:character**, *line-count* is interpreted as the number of lines to insert; if *unit* is **:pixel**, *line-count* is interpreted as the number of pixels to insert.

:set-default-character-style *new-default-style* of **tv:sheet** *Method*

Changes the default character style of the window.

12.3.3 Messages to Read or Set Cursor Position

:read-cursorpos &optional *(units* **':pixel**) of **tv:sheet** *Method*

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**) of **tv:sheet** *Method*

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 point nearest to the specified coordinates that is within the window. Sending **nil** for x or y leaves the current value unmodified.

:home-cursor of **tv:sheet** *Method*
Move the cursor to the upper left corner of the window.

:home-down of **tv:sheet** *Method*
Move the cursor to the lower left corner of the window.

12.3.4 Messages to Remove Characters From Windows

:refresh &optional *type* of **tv:sheet** *Method*
Redisplays the window. Depending on *type* and the existence of a bit-save array, clears the window or restores the image from the bit-save array. This message is usually sent by the system. You might need to provide an **:after** daemon to reconstruct the contents of the window.

:clear-char &optional *char* of **tv:sheet** *Method*
Erase the character at the current cursor position. When using character styles mapping to 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-rest-of-line of **tv:sheet** *Method*
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-rest-of-window of **tv:sheet** *Method*
Erase from the current cursor position to the bottom of the window. In more detail, first do a **:clear-rest-of-line**, and then clear all of the window past the current line.

:clear-window of **tv:sheet** *Method*
Erase the whole window and move the cursor position to the upper left corner of the window.

:delete-char &optional (*char-count* **1**) (*unit* **':character**) of **tv:sheet** *Method*
Without an argument, delete the character at the current cursor position. Otherwise, delete *char-count* units, starting at the current cursor position. Move the display of the part of the current line that is to the right of the

deleted section leftwards to close the resultant gap. If *unit* is **:character**, *char-count* is interpreted as the number of characters to delete; if *unit* is **:pixel**, *char-count* is interpreted as the number of pixels to delete.

:delete-string *string* &optional (*start* **0**) *end* of **tv:sheet** *Method*

This is for deleting specific strings in the current font. It is one of the things to use when dealing with character styles that map to 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 (*line-count* **1**) (*unit* **':character**) of **tv:sheet** *Method*

Without an argument, delete the line that the cursor is on. Otherwise delete *line-count* units, starting with the one the cursor is on. Move up the display below the deleted section to close the resulting gap. If *unit* is **:character**, *line-count* is interpreted as the number of lines to delete; if *unit* is **:pixel**, *line-count* is interpreted as the number of pixels to delete.

12.3.5 Messages About Character Width and Cursor Motion

:character-width *ch* &optional *font* (*x* **tv:cursor-x**) *character-style* of **tv:sheet** *Method*

Return the width of the character *ch*, in pixels. The current font is used if *font* is not specified. If *ch* is a BACKSPACE, **:character-width** can return a negative number. For TAB, the number returned depends on the current cursor position. If *ch* is RETURN, the result is defined to be zero.

:compute-motion *string* &optional (*start* **0**) (*end* **nil**) *x* *y* *cr-at-end-p* (*stop-x* **0**) *stop-y* of **tv:sheet** *Method*

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* **nil**) *character-style* (*start-x* **0**) (*max-x* **0**) of **tv:sheet** *Method*

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 default character style (or that specified by *character-style*) 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.

: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).

12.3.6 Window Attributes for Character Output

The following messages and initialization options initialize, get, and set various window attributes which are relevant to the typing out of characters.

:more-p *t-or-nil* (for **tv:sheet**) *Init Option*

Initialize whether the window should have more processing. It defaults to **t**.

:more-p of **tv:sheet** *Method*

Return **t** if more processing is enabled; otherwise, return **nil**.

:set-more-p *more-p* of **tv:sheet** *Method*

If *more-p* is **nil**, turn off more processing; otherwise turn it on.

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**, so that a process can type out on a deexposed window and then have the window expose itself when a **MORE** break happens.

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

:vsp of **tv:sheet** *Method*
Return the value of *vsp* for this window.

:set-vsp *new-vsp* of **tv:sheet** *Method*
Set the value of *vsp* for this window to *new-vsp*.

:reverse-video-p of **tv:sheet** *Method*
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 `FUNCTION C`).

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

:deexposed-typeout-action *action* (for **tv:sheet**) *Init Option*
Initialize the deexposed typeout action of the window to *action*. It defaults to **:normal**.

:deexposed-typeout-action of **tv:sheet** *Method*
Return the deexposed typeout action of the window.

:set-deexposed-typeout-action *action* of **tv:sheet** *Method*
Set the deexposed typeout action of the window to *action*.

:deexposed-typein-action *action* (for **tv:sheet**) *Init Option*
Initialize the deexposed typein action of the window to *action*. It defaults to **:normal**.

:deexposed-typein-action of **tv:sheet** *Method*
Return the deexposed typein action of the window.

:set-deexposed-typein-action *action* of **tv:sheet** *Method*
Set the deexposed typein action of the window to *action*.

:right-margin-character-flag *x* (for **tv:sheet**) *Init Option*
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**.

:backspace-not-overprinting-flag *x* (for **tv:sheet**) *Init Option*

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**.

:cr-not-newline-flag *x* (for **tv:sheet**) *Init Option*

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* (for **tv:sheet**) *Init Option*

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.

12.3.7 Line-Truncating Windows

tv:truncatable-lines-mixin *Flavor*

If you mix in this flavor and the window's *truncate line out* flag is on, typeout does not wrap around when lines are too long. That is, when the cursor is near the right-hand edge of the window and an attempt is made to type out a character, the character is not typed out; text is truncated at the edge of the window. When the truncate line out flag is turned off, this flavor has no effect.

tv:line-truncating-mixin *Flavor*

An obsolete flavor that is the same as **tv:truncatable-lines-mixin**. The name is confusing; when this flavor is mixed in, truncation is enabled only if the window's truncate line out flag is on. Otherwise, it has no effect. **tv:truncatable-lines-mixin** is built on this flavor for the sake of two-argument **zl:typep**.

tv:truncating-lines-mixin *Flavor*

When this flavor is mixed in, lines of output that are too long to fit inside the window do not wrap around but are truncated at the edge of the window. This flavor is built on **tv:truncatable-lines-mixin**. It initializes the window's truncate line out flag to be on.

tv:truncating-window *Flavor*

This flavor is built on **tv:window** with **tv:truncating-lines-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.

:truncate-line-out of **tv:sheet** *Method*

Returns **t** if the window's truncate line out flag is set, or **nil** if it is not.

:set-truncate-line-out *new-value* of **tv:sheet** *Method*

Sets the value of the window's truncate line out flag. If *new-value* is **t** the flag is turned on; if **nil**, it is turned off.

12.4 Graphic Output to Windows

The facilities in this section can be used with both Dynamic Windows and static windows. For information on graphics functions introduced in Genera 7.0: See the section "Overview of Graphic Output Facilities" in *Programming the User Interface, Volume A.*

12.4.1 How Windows Display Graphic Output

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** and **dw:dynamic-window** flavors. Therefore, the messages documented below work on windows of these flavors or built on these flavors. (For information on a corresponding set of graphics functions: See the section "Overview of Graphic Output Facilities" in *Programming the User Interface, Volume A.*)

There are also some messages in this section that are in **tv:sheet** or **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 flavors.)

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 integers.

As with typing out text, before any graphics are typed the process must wait until it has the ability to output. 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.

There are a few simple microcoded primitives for drawing graphics. They can be used for drawing pictures into Lisp arrays. However, when drawing on windows you should send the documented messages 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.

12.4.2 Alu Functions

Most of the messages that produce graphic output on windows 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**, 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.

12.4.3 Drawing Points on Windows

:point *x y* of **tv:graphics-mixin** *Method*

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* of **tv:graphics-mixin** *Method*

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**, that is, a number with as many **1**'s as the number of bits in a pixel.

12.4.4 Copying Bit Rectangles to and From Windows

:bitblt *alu wid hei from-raster from-x from-y to-x to-y* of **tv:sheet** *Method*

Copy a rectangle of bits from *from-raster* 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-raster* is too small it is automatically replicated.

For complete details: See the function **bitblt** in *Symbolics Common Lisp.* Note that *to-raster* is constrained as described in the the description of the **bitblt** function. See the function **tv:make-sheet-bit-array**, page 135.

:bitblt-from-sheet *alu wid hei from-x from-y to-raster to-x to-y* of **tv:sheet** *Method*

Copy a rectangle of bits from the window to *to-raster.* All the other arguments have the same significance as in the **:bitblt** method of **tv:sheet**. Note that *to-raster* is constrained as described in the the description of the **bitblt** function. See the function **tv:make-sheet-bit-array**, page 135.

:bitblt-within-sheet *alu wid hei from-x from-y to-x to-y* of **tv:sheet** *Method*

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 the **:bitblt** method of **tv:sheet**.

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* *Function*

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 **zl:make-array** when the array is created, so you can control other parameters such as the area.

12.4.5 Drawing Characters and Strings on Windows

:draw-char *char x-bitpos y-bitpos* &optional (*alu* **tv:char-aluf**) of **tv:sheet** *Method*

Display *char* with its upper left corner at coordinates (*x-bitpos, y-bitpos*).

:draw-string *string from-x from-y* &optional (*toward-x* **(1+ tv:from-x)**) (*toward-y* **tv:from-y**) (*stretch-p* **nil**) *character-style* (*alu* **tv:char-aluf**) of **tv:graphics-mixin** *Method*

:draw-string draws a character string between two points.

The left baseline point of each character lies on the line between the two points defined by *from-x*, *from-y* and *toward-x*, *toward-y*.

The string is always written from left to right, starting at the leftmost point, regardless of whether that is the first point or the second point. When the string is longer than the line between the points, the full string appears anyhow.

toward-x, *toward-y* Controls the direction in which printing takes place. The default values specify ordinary horizontal output.

```
(send (tv:window-under-mouse) ':draw-string
      "hi there" 600 50)
```

stretch-p	Controls the spacing of the characters. When it is **nil** (the default), the characters appear literally, with no change to the spacing. Otherwise, the distance between the characters is adjusted so that the string starts and ends as close to the two points as possible.
character-style	Specifies the character style to use. The default is the default character style for the window, or that specified by a character style macro: See the section "Overview of Character Environment Facilities" in *Programming the User Interface, Volume A.*
alu	Controls how the pixels being drawn combine with pixels already in the window. The default is the **tv:char-aluf** for the window.

This message is useful for placing text at absolute screen positions (as opposed to treating the window as a stream), for labelling graphs, or for putting text into pictures.

12.4.6 Drawing Lines on Windows

:draw-line *x1 y1 x2 y2* &optional *alu (draw-end-point* **t***)* of **tv:graphics-mixin** *Method*

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* of **tv:graphics-mixin** *Method*

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-dashed-line *from-x from-y to-x to-y* &optional *(alu* **tv:char-aluf***) (dash-spacing* **20.***) space-literally-p (offset* **0***) dash-length* of **tv:graphics-mixin** *Method*

:draw-dashed-line draws a dashed line along the line lying between two points. All the dashes are the same length; all the spaces between the dashes are the same length. (The spaces, however, need not be the same length as the dashes). The spacing and lengths of the dashes are controlled by separate arguments.

alu — Controls how the pixels being drawn combine with pixels already in the window. The default is the **tv:char-aluf** for the window.

dash-spacing — Specifies the distance from the beginning of one dash to the beginning of the next dash. It is expressed in pixels. The default is **20**. (The spacing between dashes is *dash-spacing* minus *dash-length*.) This specifies the "frequency" of the line.

space-literally-p — Controls what happens when the distance between the points, given the specified spacings, would not produce a full-size dash connected to the endpoint.

The default value, **nil**, allows the size of *dash-spacing* to be adjusted slightly so that the dashes are all of equal size and both endpoints look the same, as far as dash length goes. In this case, the dash-length is always exactly half of the dash-spacing; any values for *offset* and *dash-length* are ignored.

The value **t** means to use *dash-spacing* exactly, with no adjustment. The endpoint might or might not have a dash connected to it, depending on the exact distances involved.

offset — Specifies a distance (in pixels) from the starting point (*from-x*, *from-y*) for the beginning of the first dash. This lets you control the "phase" of the dashed line.

dash-length — Specifies the length of the line segments, in pixels. It must be less than *dash-spacing*. This lets you control the "duty cycle" of the line. The default is half the value of *dash-spacing*.

You can make complex dashing by using **:draw-dashed-line** many times with *space-literally-p* as **t**. For example:

```
(progn
 (send terminal-io ':draw-dashed-line 0 0 200. 200. tv:alu-ior 25. t 0 10.)
 (send terminal-io ':draw-dashed-line 0 0 200. 200. tv:alu-ior 25. t 15. 5.))
```

This gives you alternating long and short dashes. Because the **nil** value for *space-literally-p* changes the spacing, this technique does not work well when *space-literally-p* is **nil**.

:draw-curve *x-array y-array* &optional *end alu* of **tv:graphics-mixin** *Method*

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-closed-curve *x-array y-array* &optional *end (alu* **tv:char-aluf***)* of **tv:graphics-mixin** *Method*

:draw-closed-curve draws a sequence of connected line segments, using the points in *x-array* and *y-array* as the *x* and *y* coordinates for the end-points of the lines. It ensures that each particular point is drawn only once, which is necessary for producing a connected line with **tv:alu-xor**. It plots the points in the arrays until *end* elements or until it encounters **nil** in either of the arrays. The default for *end* is the length of *x-array*. *alu* specifies how the pixels being drawn combine with those already there. It plots the points in the arrays until *end* elements or until it encounters **nil** in either of the arrays.

:draw-closed-curve is the same as **:draw-curve** except that it closes the figure by joining the first and last points.

:draw-wide-curve *x-array y-array width* &optional *end alu* of **tv:graphics-mixin** *Method*

Like **:draw-curve** but *width* is how wide to make the lines.

12.4.7 Drawing Polygons and Circles on Windows

:draw-rectangle *width height x y* &optional *alu* of **tv:sheet** *Method*

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* of **tv:graphics-mixin** *Method*

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* of **tv:graphics-mixin** *Method*

Draw the outline of a circle specified by its center and radius.

:draw-circular-arc *center-x center-y radius start-theta end-theta* &optional *(alu* **tv:char-aluf***)* of **tv:graphics-mixin** *Method*

Draws a circular arc for the circle centered at *center-x, center-y* with radius *radius*. It draws the part of the circle swept counterclockwise from the starting angle to the finishing angle. The angles are assumed to be in radians and are reduced mod 2pi before drawing. For example, drawing from pi/4 to -pi/4 draws a "C". The same "C" appears when you draw from pi/4 to 7pi/4.

For **tv:alu-xor**, the behavior with respect to points that would fall on the same pixel is not defined.

:draw-filled-in-circle *center-x center-y radius* &optional *alu* of **tv:graphics-mixin** *Method*

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* of **tv:graphics-mixin** *Method*

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 counterclockwise.

:draw-regular-polygon *x1 y1 x2 y2 n* &optional *alu* of **tv:graphics-mixin** *Method*

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").

12.4.8 Drawing Splines on Windows

:draw-cubic-spline *px py z* &optional *curve-width alu c1 c2 p1-prime-x p1-prime-y pn-prime-x pn-prime-y* of **tv:graphics-mixin** *Method*

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.

:anti-cyclic
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.

12.4.9 Primitives for Drawing Onto Arrays

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 y alu sheet-or-raster* *Function*
This is analogous to the **:draw-rectangle** message to **tv:stream-mixin.**

sys:%draw-line *x1 y1 x2 y2 alu draw-end-point sheet-or-raster* *Function*
This is analogous to the **:draw-line** message to **tv:graphics-mixin.**

sys:%draw-triangle *x1 y1 x2 y2 x3 y3 alu sheet-or-raster* *Function*
This is analogous to the **:draw-triangle** message to **tv:graphics-mixin.**

12.5 Notifications

This section applies to both static and Dynamic Windows.

12.5.1 Overview of Notifications

Notifications are messages that a process sends to the user asynchronously to inform the user of some change in the state of the process. Some examples:

- By default the garbage collector notifies the user as storage is used up and when the dynamic garbage collector flips and flushes oldspace.
- If a window's deexposed typeout action is **:notify**, the user is notified when an attempt is made to type out on that window.
- Converse messages can be received as notifications.

A process uses **tv:notify** to notify the user. This function constructs a notification and saves it on a queue. A central delivery process takes notifications from the queue and delivers them to the user. This process first gives the process associated with the selected window a chance to accept the notification itself. If the process associated with the selected window does not accept the notification within a short time, the delivery process usually tries to display the notification itself, in either the selected window or a pop-up window.

The notification delivery process tries to give the user process a chance to accept the notification by storing the notification in a locative obtained by sending the **:notification-cell** message to the selected window. If the user process wants to accept notifications, it usually checks the contents of this cell as part of the **:input-wait** wait function. The user process sends the **:receive-notification** message to accept the notification. When it wants to display a notification it usually calls **sys:display-notification**. By default, if the user process doesn't accept a notification, the notification delivery process displays the notification in a pop-up window. The user process can use the **tv:with-notification-mode** special form to control what happens to notifications it doesn't accept.

All notifications received since cold booting are displayed in a scroll window obtained by pressing SELECT N or by calling **zl:display-notifications**. You can display some or all notifications by using the Show Notifications command.

zl:display-notifications *Function*

Selects a scroll window that displays all notifications received since cold booting.

12.5.2 Notifying the User

tv:notify *window-of-interest format-control* &rest *format-args* *Function*

Issues an asynchronous notification to the user. Constructs a notification and pushes it onto a queue. A central notification delivery process delivers the notification to the user. The text of the notification is constructed from *format-control* and *format-args.* If *window-of-interest* is not **nil**, it is a window to be made available via FUNCTION 0 S.

12.5.3 Receiving and Displaying Notifications

When a process notifies the user, the central notification delivery process gives the process associated with the selected window a chance to accept the notification before the delivery process tries to display the notification itself. The notification delivery process stores the notification in a locative obtained by sending the **:notification-cell** message to the selected window, unless a notification is already there. In that case the notification delivery process usually tries to display the notification itself.

A user process that wants to accept notifications should send the selected window a **:notification-cell** message to find the locative that might contain a notification. The process should wait (usually in an **:input-wait** wait function) until the locative contains something other than **nil.**

When a notification cell contains a notification, a process can accept the notification by sending the selected window a **:receive-notification** message. If the process wants to display the notification, it usually passes it on to the function **zl:display-notifications.**

:notification-cell *Message*

This message to an interactive stream returns the locative in which the notification delivery process stores notifications. If some process notifies the user, the notification delivery process gives the process associated with the selected window a chance to accept the notification. It does this by trying to store the notification in the locative returned by the **:notification-cell** message to the selected window, unless the locative contains a notification already. In that case the notification delivery process usually tries to display the notification itself.

A user process that wants to accept notifications should find this locative by sending the **:notification-cell** message to the selected window. It should wait (usually in an **:input-wait** wait function) for the locative to contain something other than **nil.** The user process can receive the notification by sending the selected window a **:receive-notification** message.

:receive-notification *Message*

This message to an interactive stream returns a notification when one exists in the stream's notification cell. The message checks the contents of the locative returned by the **:notification-cell** message to the stream. When the locative contains a notification, **:receive-notification** returns the notification and stores **nil** in the locative. When the locative does not contain a notification, **:receive-notification** returns **nil**.

sys:display-notification *stream note* &optional *style window-width* *Function*

Displays a notification on *stream*. *note* is the notification, returned by the **:receive-notification** message to an interactive stream. The display includes the time and the text of the message as specified in the arguments to **tv:notify**.

style is **nil** or a keyword determining the style of the display:

nil	Displays the time and the text of the message at the current cursor position, with indentation. This is the default.
:stream	Sends a **:fresh-line** message, then displays the time and the text of the message, with indentation, in square brackets, then displays a Newline. This style is for merging the notification display with other output to the stream.
:window	Sends a **:fresh-line** message, then displays the time and the text of the message, with indentation, in square brackets. This style is for using the entire window to display the notification. It assumes the window has been cleared first.
:pop-up	Displays the time and the text of the message at the current cursor position, with indentation, then sends a **:fresh-line** message. This style is used by the notification delivery process to display notifications in a pop-up window.

window-width is **nil** or the number of characters available on a line to display the notification. If *window-width* is **nil** or not supplied, the default is the result of sending the stream a **:size-in-characters** message. This is used only to determine how much to indent lines other than the first in the notification. If *window-width* is about 110 or more, lines are indented to the beginning of the text of the message (following the time). If *window-width* is about 100 or less, lines are indented only one character. You can supply a large *window-width* to increase the indentation in a narrow window, or supply a small *window-width* to decrease the indentation in a wide window.

If *style* is **:stream**, **:window**, or **:pop-up** and if a "window of interest" was supplied as the first argument to **tv:notify**, a message is displayed that informs the user that FUNCTION 0 S selects the window of interest.

sys:display-notification does not return any interesting values, unless *style* is **:pop-up**. In that case it returns the X and Y coordinates, in pixels, of the beginning of the line following the text of the notification.

Following is a simple example of a command loop that waits for input, a notification, or a new selected-pane. When a notification arrives, it displays it in a pane reserved for notifications. When input arrives, it just displays a representation of the input in the selected pane.

```
(defun my-top-level (frame)
  (let ((notification-pane (send frame :get-pane 'notification-pane)))
    (error-restart-loop ((error sys:abort) "My top level")
      (let ((selected-pane (send frame :selected-pane))
            (note))
        (when selected-pane
          (send selected-pane :input-wait nil
                #'(lambda (note-cell)
                    (declare (sys:downward-function))
                    (or (neq selected-pane (send frame :selected-pane))
                        (not (null (location-contents note-cell)))))
                (send selected-pane :notification-cell))
          (cond ((neq selected-pane (send frame :selected-pane)))
                ((setq note (send selected-pane :receive-notification))
                 (sys:display-notification notification-pane note :stream))
                (t
                 (let ((char (send selected-pane :any-tyi-no-hang)))
                   (cond ((null char))
                         ((fixp char)
                          (format selected-pane "~&Character: ~C" char))
                         ((listp char)
                          (format selected-pane "~&Blip: ~S" char))
                         (t (format selected-pane "~&Unknown object: ~S" char))))))))))
```

After storing a notification in the selected window's notification cell, the notification delivery process gives the process associated with the selected window some time to accept the notification. The amount of time is determined by the variable **tv:*notification-deliver-timeout***.

tv:*notification-deliver-timeout* *Variable*

The length of time, in sixtieths of a second, that the notification delivery process waits for the process associated with the selected window to accept a notification. If the selected window's process does not accept the notification during this time, the delivery process takes the notification back and usually tries to display it itself. Default: **180.** (three seconds).

If the process associated with the selected window does not accept a notification within the specified time, or if the window's notification cell already contains a notification, the window's *notification mode* determines what the delivery process does with the notification. You can use the **:notification-mode** message to get the notification mode and the **:set-notification-mode** message to set it.

:notification-mode *Message*

This message to an interactive stream returns the stream's notification mode. The notification mode determines what the notification delivery process does with a notification when the process associated with the stream doesn't accept it:

:pop-up	The notification is displayed in a pop-up window. This is the default.
:blast	The notification is displayed on the stream.
:ignore	The notification is ignored but is added to the notification history for SELECT N and the Show Notifications command.
nil	The same as **:pop-up**.

:set-notification-mode *new-mode* *Message*

This message to an interactive stream sets the stream's notification mode. The notification mode determines what the notification delivery process does with a notification when the process associated with the stream doesn't accept it. *new-mode* can be a keyword or **nil**:

:pop-up	The notification is displayed in a pop-up window. This is the default.
:blast	The notification is displayed on the stream.
:ignore	The notification is ignored but is added to the notification history for SELECT N and the Show Notifications command.
nil	The same as **:pop-up**.

If you want to execute some code with a stream's notification mode bound to some value, use the special form **tv:with-notification-mode**.

tv:with-notification-mode (*new-mode* &optional *stream*) &body *body* *Macro*

Executes *body* with the notification mode of *stream* bound to *new-mode*. *stream* defaults to **zl:standard-output**. The notification mode determines what the notification delivery process does with a notification when the process associated with *stream* doesn't accept it. *new-mode* can be a keyword or **nil**:

:pop-up	The notification is displayed in a pop-up window. This is the default.
:blast	The notification is displayed on the stream.
:ignore	The notification is ignored but is added to the notification history for SELECT N and the Show Notifications command.
nil	The same as **:pop-up**.

12.5.3.1 Pop-up Notifications

When a notification is displayed in a pop-up window, the user is alerted with a beep and given some time to notice the beep and stop typing. Until that time elapses, all typein is directed to the previously selected window, except that the user can press ABORT to deexpose the pop-up window immediately. The amount of time is determined by the variable **tv:unexpected-select-delay**.

tv:unexpected-select-delay *Variable*

The amount of time, in sixtieths of a second, that a user is given to notice a pop-up notification and stop typing. Until that time has elapsed, all typein is directed to the previously selected window. During this time the user can press ABORT to deexpose the pop-up window. A value of **nil** means no delay time and no display of the message that typing any character deexposes the pop-up window. Default: **180.** (three seconds).

After the select delay, typing any character or selecting another window deexposes the pop-up window. If a "window of interest" was supplied as the first argument to **tv:notify**, a message is displayed that informs the user that FUNCTION 0 S or a mouse click on the pop-up window selects the window of interest. If another notification arrives while the pop-up window is exposed, the notification is displayed on the window. If after a time the user has typed nothing, the pop-up window is deexposed automatically. The amount of time the pop-up window remains exposed is determined by the variable **tv:*notification-pop-down-delay***.

tv:*notification-pop-down-delay* *Variable*

The amount of time, in sixtieths of a second, that a notification pop-up window remains exposed if the user types no characters to the window. A value of **nil** means that the window remains exposed indefinitely. Default: **54000.** (15 minutes).

12.6 Input From Windows

The material presented in this section applies to both static and Dynamic Windows.

12.6.1 Windows as Input Streams

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. The **tv:stream-mixin** flavor is a component of the **tv:window** and **dw:dynamic-window** flavors.

tv:stream-mixin *Flavor*

This flavor allows a window to function as an interactive stream. It should be mixed into any window that can be used for interacting with a user, and particularly into any window that can become the value of **zl:terminal-io**. It gives the window an I/O buffer, allows the window to handle input messages, and provides the window with input editing.

tv:stream-mixin includes **si:interactive-stream**, and windows support all the operations that interactive streams in general do: See the section "Interactive Streams", page 1. Windows have specialized versions of some input operations: See the section "Messages for Input From Windows", page 149.

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 go only 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.

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 with a message like "Process X wants typein", and the window is "made interesting" so that FUNCTION 0 S can select it.

Reading characters from a window normally returns an integer that represents a character in the Lisp Machine character set, possibly with extra bits that correspond to the CONTROL, META, SUPER, and HYPER keys. For information on the format of such integers and the symbolic names of the bit fields: See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*.

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 **zl:tyi**, **zl:read**, and **zl: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 *I/O 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 I/O buffer, letting you clear it and ask whether there is anything in it.

Normally, integers get into the I/O buffer because characters were typed on the keyboard. However, you can also get any Lisp object into a window's I/O 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 to tell a user process about activity on the mouse buttons. That is how characters with the **zl:%%kbd-mouse** bit can get read from the window. It is possible to put Lisp objects other than integers into an I/O 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 integers, in order to find out the mouse position at the time of the click. Using the mouse is explained later on.

You can explicitly manipulate I/O 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 I/O buffer; this is often used to make panes of a paned window all share the same I/O buffer. Another thing you can do is put properties on the I/O buffer's property list; this lets you request various special features.

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 I/O buffer; however, the format of the raw codes is complicated and dependent on the hardware implementation. It is not documented here.

The window system intercepts some characters specially. Some are intercepted when the user process is about to read the character from a window; others are intercepted as soon as they are typed. In the first category, the **io-buffer-output-function** of the I/O buffer defaults to **tv:kbd-default-output-function**, which intercepts certain characters when they are read. The value of the variable **sys:kbd-intercepted-characters** is a list of characters that are intercepted and not returned as input from the window. These characters default to #\abort, #\m-abort, #\suspend, and #\m-suspend. For more information: See the section "Intercepted Characters", page 17.

The second category of specially handled characters is those handled *asynchronously*. See the section "Asynchronous Characters", page 154.

12.6.2 Messages for Input From Windows

Windows support all the input operations that interactive streams in general do: See the section "Messages for Input From Interactive Streams", page 13. Windows have specialized versions of some of these operations, mainly involved in reading characters from I/O buffers.

:any-tyi &optional *eof-action* of **tv:stream-mixin** *Method*

Read and return the next character of input from the window, waiting if there is none. Where the character comes from depends on the value of the variable **sys:rubout-handler**. Following is a summary of actions for each possible value of **sys:rubout-handler**:

nil	If the input buffer contains unscanned input, take the next character from there. Otherwise, take the next character from the window's I/O buffer.
:read	If the input buffer contains unscanned input, take the next character from there. Otherwise, if an activation blip or character is present, return that. Otherwise, enter the input editor.
:tyi	Take the next character from the window's I/O buffer.

If *eof-action* is not **nil**, an error is signalled when an end-of-file is encountered. Otherwise, the method returns **nil** when an end-of-file is encountered. The default for *eof-action* is **nil**.

:any-tyi-no-hang &optional *eof-action* of **tv:stream-mixin** *Method*

Check the window's I/O buffer and return the next character if it is immediately available. If no characters are immediately available, return **nil**. It is an error to call this method from inside the input editor (that is, if the value of **sys:rubout-handler** is not **nil**). *eof-action* is ignored. 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* of **tv:stream-mixin** *Method*

Return *ch* to the proper buffer so that it will be the next character returned by **:any-tyi** or **:tyi**. *ch* must be the last character that was **:tyi**'ed, and it is illegal to do two **:untyi**'s in a row. Where *ch* is put depends on the value of the variable **sys:rubout-handler**. Following is a summary of actions for each possible value of **sys:rubout-handler**:

nil If the input buffer contains scanned input, decrement the scan pointer. Otherwise, put *ch* back into the window's I/O buffer.

:read Decrement the input editor scan pointer.

:tyi Put *ch* back into the window's I/O buffer.

This method is used by parsers that look ahead one character, such as **zl:read**.

:listen of **tv:stream-mixin** *Method*

Return **t** if there are any characters available to **:any-tyi** or **: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 of **tv:stream-mixin** *Method*

Clear this window's input and I/O buffers. This flushes all the characters that have been typed at this window, but have not yet been read.

12.6.3 SELECT And FUNCTION Keys

tv:add-function-key *char function documentation* &rest *options* *Function*

Adds *char* to the list of keys that can follow the FUNCTION key. Following is an explanation of the arguments:

char The character (an integer) that should be typed after FUNCTION to get the new command. Lower-case letters are converted to upper case.

function A specification for the action to be taken when the user presses FUNCTION *char*. *function* can be a symbol or a list:

- *Symbol:* The name of a function to be applied to one argument. The argument is the numeric argument to FUNCTION *char* (an integer) or **nil** if the user supplied none.
- *List:* A form to be evaluated.

function is applied or evaluated in a newly created process unless you supply the **:keyboard-process** option (see below).

documentation A form to be evaluated when the user presses `FUNCTION HELP` to produce documentation for the command. The form should return a string, a list of strings, or **nil** (of course, *documentation* can just be a string or **nil**):

- *String:* One line of text describing this command for `FUNCTION HELP`.
- *List of strings:* Each string is a line of text for `FUNCTION HELP` to print successively in describing this command. Usually *documentation* is a Lisp form that looks like `'("line 1" "line 2" ...)`.
- **nil**
  `FUNCTION HELP` prints nothing describing this command.

options A series of alternating keywords and values. Possible options are **:keyboard-process**, **:process-name**, **:process**, and **:typeahead**:

- **:keyboard-process**
  function is applied or evaluated in the keyboard process instead of a newly created process. This option exists because certain built-in commands must run in the keyboard process. You should not use this option for new commands. The cost of creating a new process is quite low.
- **:process-name** *string*
  string is the name of the newly created process in which *function* is applied or evaluated. If you don't supply this option or the **:process** option, the name of the process is "Function Key".
- **:process** *list*
  list is a list to be used as the first argument to **process-run-function**, called to create a new process in which *function* is applied or evaluated. This option takes precedence over **:process-name**.
- **:typeahead**
  Everything the user types before pressing the

FUNCTION key is treated as typeahead to the currently selected window. Use this option with commands that change windows to ensure that the user's typed input goes to the I/O buffer of the expected window.

Here is an example of a call to **tv:add-function-key**:

```
(tv:add-function-key #\refresh 'tv:kbd-screen-redisplay
                     "Clear and redisplay all windows.")
```

See the variable **tv:*function-keys***, page 152.

tv:*function-keys* *Variable*

The value of this variable is an alist, each entry of which describes a subcommand of the FUNCTION key. Entries are of the form:

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

For an explanation of the components of the entries: See the function **tv:add-function-key**, page 150. Use **tv:add-function-key** to add a new entry or redefine an existing one rather than changing the value of **tv:*function-keys*** yourself.

tv:add-select-key *char flavor name* &optional (*create-p* t) *clobber-p* *Function*

Adds *char* to the list of keys that can follow the SELECT key. Following is an explanation of the arguments:

char — The character (an integer) that should be typed after SELECT to get the new command. Lower-case characters are converted to upper case.

flavor — A specification for the window to be selected when the user presses SELECT *char*. *flavor* can be a symbol, an instance, or a list:

- *Symbol:* The name of a flavor. The SELECT command searches the list of previously selected windows and selects a window of flavor *flavor* if it finds one. (*flavor* can be the name of a component flavor of the window, not just the instantiated flavor.) Otherwise, if the currently selected window is of flavor *flavor*, it beeps. Otherwise, it takes the actions specified by *create-p*.

- *Instance:* A window. The SELECT command selects that window.

- *List:* A form to be evaluated (in the SELECT command's newly created process). The form should return a window to be selected or a symbol that is the name of a flavor of window to be selected.

name A string giving the colloquial name of the program to be selected. *name* is printed by SELECT HELP.

create-p A specification for actions that the SELECT command should take if it cannot find a previously selected window of flavor *flavor* and if the currently selected window is not of flavor *flavor*. *create-p* can be **nil**, **t**, another symbol, or a list:

- **nil**: Beeps.

- **t**: Calls **tv:make-window** with no options to create a window of flavor *flavor*. Selects that window.

- *Another symbol:* The name of a flavor. Calls **tv:make-window** with no options to create a window of flavor *create-p*. Selects that window.

  flavor and *create-p* can be names of different flavors. For example, *flavor* might be the name of a mixin that is a component of several flavors, all of which are suitable flavors of window to select.

- *List:* A form to be evaluated (in the SELECT command's newly created process). The form presumably selects a window.

clobber-p Boolean option specifying whether to reassign a key to select a new program without first requesting confirmation; a value of **t** suppresses the confirmation prompt.

If the user presses *char* with the c- modifier (after pressing SELECT), and if *flavor* is a symbol that names a flavor or is a form that returns the name of a flavor, the SELECT command does not search for previously selected windows of flavor *flavor*. Instead, it takes the actions specified by *create-p*. But if *flavor* is a window, the SELECT command selects that window even if the user presses *char* with the c- modifier.

Here is an example of a call to **tv:add-select-key**:

```
(tv:add-select-key #/E 'zwei:zmacs-frame "Editor" :clobber-p nil)
```

See the variable **tv:*select-keys***, page 154.

tv:*select-keys* *Variable*

The value of this variable is an alist, each entry of which describes a subcommand of the SELECT key. Entries are of the form:

(*char flavor name create-p*)

For an explanation of the components of the entries: See the function **tv:add-select-key**, page 152. Use **tv:add-select-key** to add a new entry or redefine an existing one rather than changing the value of **tv:*select-keys*** yourself.

12.6.4 Asynchronous Characters

The FUNCTION and SELECT 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 FUNCTION and SELECT commands, using the functions **tv:add-function-key** and **tv:add-select-key**. See the section "SELECT And FUNCTION Keys", page 150.

Other characters can also be intercepted as soon as they are typed. A special system process called the keyboard process calls a user-defined function as soon as the key is pressed. The main process of the program is left undisturbed. This function runs in parallel with the main program and could communicate with it.

Asynchronous character handling is available to any window that includes **tv:stream-mixin**. The window has a list that associates keyboard characters with functions. The default list contains c-ABORT, c-SUSPEND, c-m-ABORT, and c-m-SUSPEND. The default actions are the same as those of the corresponding keys without c- modifiers, except that the window's process is sent an **:interrupt** message so that the actions take place immediately.

The keyboard process checks each character coming in to see if it is defined as an asynchronous character for the selected window. When it is, the keyboard process calls the associated function in the context of the keyboard process.

The function that runs as a result of an asynchronous character is running in the keyboard process. It is called with two arguments, the character and **self**. It should be very short and must not do any I/O. An error in one of these functions would break the keyboard process and the keyboard along with it and you would have to warm boot. To avoid any possibility of errors, you can have the function create a new process with **process-run-function** and make the new process handle the real work.

You can set up your own handling of asynchronous characters by using the **:asynchronous-character-p, :handle-asynchronous-character,**

:add-asynchronous-character, and **:remove-asynchronous-character** messages and the **:asynchronous-characters** init option for **si:interactive-stream**. See the section "Interactive-Stream Operations for Asynchronous Characters", page 19.

12.7 TV Fonts

12.7.1 Using TV Fonts

In Genera, 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. The Font Editor (FED) is a program that allows you to create, modify, and extend fonts: See the section "Font Editor" in *Text Editing and Processing.*

A font is represented internally 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 interface to fonts is provided by *character styles* (for more information: See the section "Character Styles" in *Symbolics Common Lisp.*) You can (indirectly) control which font is used when output is done to a window by specifying the default character style for that window: See the init option **(flavor:method :default-character-style tv:sheet)**, page 120. Additional control over character styles is provided by several output macros: See the section "Overview of Character Environment Facilities" in *Programming the User Interface, Volume A.*

The character style resulting from merging the output character style against a window's default character style maps to a particular font. This is true of both static and Dynamic Windows. This font is the *current font* for the window; to access it you can use the **:current-font** message:

:current-font of **tv:sheet** *Method*

Returns the current font, as a font object.

Example:

```
(send *standard-output* :current-font) ==>
#<FONT CPTFONT 260000546>
```

To discover what font corresponds to a particular character style, use the following function:

si:backtranslate-font *font* *Function*

Returns the character style object corresponding to a specified screen *font*. Also returned are the character set, charset-offset, and device type. (The default device type for this function is **si:*b&w-screen***.)

Example:

```
(si:backtranslate-font fonts:eurex24i) ==>
#<CHARACTER-STYLE EUREX.ITALIC.HUGE 260273114>
#<STANDARD-CHARACTER-SET 260000540>
0
#<B&W-SCREEN-DISPLAY-DEVICE 260272253>
```

When you create a font of your own, there are basically two ways you can make use of it: 1) for defining a new character style; and 2) as a collection of glyphs for graphics output. To define a new character style and associate your font with it, use the function **si:define-character-style-families**: See the section "Mapping a Character Style to a Font" in *Symbolics Common Lisp*. To draw a glyph included in a font array, use **graphics:draw-glyph**: See the function **graphics:draw-glyph** in *Programming the User Interface, Volume A*.

One additional facility provided for interfacing with TV fonts is the **:baseline** method:

:baseline of **tv:sheet** *Method*

Returns the baseline of the current font. The bases of all output characters are so aligned as to be this many pixels below the top of the line on which the characters are printed.

The baseline is affected by the value of the **:bind-line-height** option to character style macros: See the section "**:bind-line-height** Option to Character Style Macros" in *Programming the User Interface, Volume A*.

12.7.2 Standard TV Fonts

You can use `Show Font HELP` in the Lisp Listener or the List Fonts (`m-X`) command in Zmacs to get a list of all the fonts that are currently loaded into the Lisp environment. The **fonts** package contains the names of all fonts. Here is a list of some of the useful fonts:

fonts:cptfont This is the default font, used for almost everything.

fonts:jess14 This is the default font in menus. It is a variable-width rounded font, slightly larger and more attractive than medfnt.

fonts:cptfonti This is a fixed-width italic font of the same width and shape as **fonts:cptfont**, the default screen font. It is most useful for italicizing running text along with **fonts:cptfont**.

fonts:cptfontcb This is a fixed-width bold font of the same width and shape as **fonts:cptfont**, the default screen font.

fonts:medfnt This is 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 nonselected items in Choose Variable Values windows.

fonts:hl10b This is a bold version of **hl10**, used for selected items in Choose Variable Values windows.

12.7.3 Attributes of TV Fonts

Fonts, and characters in fonts, have several interesting attributes.

Character Height Font Attribute

One attribute of each font is its *character height*. This is a nonnegative integer used to figure out how tall to make the lines in a window. 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 vertical spacing (in pixels) between the text lines (*vsp*) specified for the window, 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. This ensures that any line has enough room to display the largest character of the largest font 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 are still spaced far enough apart to accommodate characters from the large font. This is because the window system cannot predict when you might, in the middle of a line, suddenly switch to the large font.

Baseline Font Attribute

Another attribute of a font is its *baseline.* The baseline is a nonnegative integer 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 are aligned at their bases rather than at their tops or bottoms. So when you output a character at a certain cursor position, the window system first examines the baseline of the current font, then draws the character in a position adjusted vertically to make the bases of the characters all line up.

Character Width Font Attribute

The *character width* 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 merely defined to be the amount by which the cursor should be moved.

Left Kern Font Attribute

The *left kern* is an attribute of each character separately . Usually it is zero, but it can also be a positive or negative integer. When the window system draws a character at a given cursor position, and the left kern is nonzero, 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.

Fixed-width Font Attribute

A font that does not have separate character widths for each character and does not have any nonzero 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.

Blinker Width And *Blinker Height* Font Attributes

The *blinker width* and *blinker height* are two nonnegative integers that tell the window system an attractive width and height to make a rectangular blinker for characters in this font. These attributes are completely independent of all other attributes and are only used for making blinkers. Using a fixed width blinker for a variable-width font causes problems; the editor actually readjusts 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. The easiest thing to do is to use the blinker width as the width of the blinker. This works well with a fixed-width font.

Chars-exist-table Font Attribute

The *chars-exist-table* is **nil** if all characters exist in a font, or an **sys:art-boolean** array. This table is not used by the character-drawing software; it is 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.

12.7.4 Format of TV Fonts

The array leader of a font is a structure defined by **zl:defstruct**. Here are the names of the accessors for the elements of the array leader of a font.

zl:font-name *font* — *Function*

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.

zl:font-char-height *font* — *Function*

The character height of the font; a nonnegative integer.

zl:font-char-width *font* — *Function*

The character width of the characters of the font; a nonnegative integer. If the **zl: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".

zl:font-baseline *font* *Function*

The baseline of this font; a nonnegative integer.

zl:font-char-width-table *font* *Function*

If this is **nil** then all the characters of the font have the same width, and that width is given by the **zl:font-char-width** of the font. Otherwise, this is an array of nonnegative integers, one for each logical character of the font, giving the character width for that character.

zl:font-left-kern-table *font* *Function*

If this is **nil** then all characters of the font have zero left kern. Otherwise, this is an array of integers, one for each logical character of the font, giving the left kern for that character.

zl:font-blinker-width *font* *Function*

The blinker width of the font.

zl:font-blinker-height *font* *Function*

The blinker height of the font.

zl:font-chars-exist-table *font* *Function*

This is **nil** if all characters exist in the font, or an **sys:art-boolean** array with one element for each logical character of the file. The element is **t** if the character exists and **nil** if the character does not exist.

zl:font-raster-height *font* *Function*

The raster height of the font; a positive integer.

zl:font-raster-width *font* *Function*

The raster width of the font; a positive integer.

zl:font-indexing-table *font* *Function*

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.

12.8 Blinkers

Each static or Dynamic 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 an integer giving the number of 60ths 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** has its 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 the variable **tv:alu-xor**, page 133.

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).

12.8.1 General Blinker Operations

tv:make-blinker *window* &optional *(flavor* **'tv:rectangular-blinker***)* &rest *options* — *Function*

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* (for **tv:blinker**) — *Init Option*

Along with the **:y-pos** init option, set the initial position of the blinker within the window. This init option is irrelevant for blinkers that follow the cursor. The initial position for nonfollowing blinkers defaults to the current cursor position.

:y-pos *y* (for **tv:blinker**) — *Init Option*

Along with the **:x-pos** init option, set the initial position of the blinker within the window. This init option is irrelevant for blinkers that follow the cursor. The initial position for nonfollowing blinkers defaults to the current cursor position.

:read-cursorpos of **tv:blinker** — *Method*

Returns two values: the *x* and *y* components of the position of the blinker within the inside of the window.

:set-cursorpos *x* *y* of **tv:blinker** — *Method*

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* (for **tv:blinker**) — *Init Option*

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* of **tv:blinker** *Method*

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* (for **tv:blinker**) *Init Option*

Set the initial visibility of the blinker. This defaults to **:blink**.

:set-visibility *new-visibility* of **tv:blinker** *Method*

Set the visibility of the blinker. *new-visibility* should be one of **:on**, **nil**, **:off**, **t**, or **:blink**. For the meaning of these values: See the section "Blinkers", page 160.

:deselected-visibility *symbol* (for **tv:blinker**) *Init Option*

Set the initial deselected visibility. By default, it is **:on**.

:deselected-visibility of **tv:blinker** *Method*

Examine the deselected visibility of the blinker.

:set-deselected-visibility *new-visibility* of **tv:blinker** *Method*

Change the deselected visibility of the blinker.

:half-period *n-60ths* (for **tv:blinker**) *Init Option*

Set the initial value of the half-period of the blinker. This defaults to **15**.

:half-period of **tv:blinker** *Method*

Examine the half-period of the blinker.

:set-half-period *new-half-period* of **tv:blinker** *Method*

Change the half-period of the blinker.

:set-sheet *new-window* of **tv:blinker** *Method*

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* *Function*

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* *Function*

Set the visibility of all blinkers on *window* to **:off**.

12.8.2 Specialized Blinkers

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* (for **tv:rectangular-blinker**) *Init Option*

Set the initial width of the blinker, in pixels. By default, it is set to the width of a space character in the default character style of the window associated with the blinker.

:height *n-pixels* (for **tv:rectangular-blinker**) *Init Option*

Set the initial height of the blinker, in pixels. By default, it is set to the height of the default character style of the window associated with the blinker.

:set-size *new-width new-height* of **tv:rectangular-blinker** *Method*

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* (for **tv:ibeam-blinker**) *Init Option*

Set the initial height of the blinker. It defaults to the *line-height* 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* (for **tv:character-blinker**) *Init Option*

Set the font in which to find the character to display. This may be anything acceptable to the **:parse-font-descriptor** message of the window's screen. You must provide this.

:char *char* (for **tv:character-blinker**) *Init Option*

Set the character to display. You must provide this.

:set-character *nchar* of **tv:character-blinker** *Method*

Set the character to display to *nchar*.

12.9 Mouse Input

12.9.1 Introduction

The "Mouse Input" section describes the mouse process and mouse facilities in the pre-Genera 7.0 context of static windows. In this context, the mouse process has broader responsibilities than it does in Genera 7.0, and many applications have included considerable amounts of code running in the mouse process in addition to that running in the user process. Coordinating the two processes is sometimes tricky, and the facilities described below for "grabbing the mouse", "usurping the mouse", and so on are helpful in providing more control in the user process.

In Genera 7.0, the mouse process has fewer duties, being responsible primarily for communicating to the user process where the mouse cursor is and whether any actions involving the mouse have occurred. With Dynamic Windows and the presentation-type system, mouse sensitivity of displayed items is a built-in feature. Facilities in Genera 7.0 forming the interface to the mouse process are **dw:tracking-mouse** and the mouse handler facilities: See the section "Overview of Advanced User Input Facilities" in *Programming the User Interface, Volume A*.

12.9.2 Handling 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 elsewhere: See the section "Window System Choice Facilities", page 239.

The window system includes a process called **Mouse** that 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 boxes around things, usually to indicate what would happen if you were to click a button. (This has changed in Genera 7.0: See the section "Introduction to Mouse Input", page 165.) To do this, 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.

There are 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 (m-X) command in the Editor), menus, multiple-choice windows, and more.

Several of the following facilities are methods for **tv:essential-mouse**. This is a component flavor of both **tv:window** and **dw:dynamic-window**.

tv:mouse-sheet *Variable*

The superior window, usually the main screen, that contains the position of the mouse.

:handle-mouse of **tv:essential-mouse** *Method*

The mouse overseer sends this message when the mouse enters the window. The method calls the default mouse handler, which returns when the mouse moves outside the window.

:mouse-moves *x y* of **tv:essential-mouse** *Method*

The default mouse handler sends this message to the window when the mouse has moved or buttons have been pushed. *x* and *y* represent the current position of the mouse if it has moved or its position at the time of the click if buttons have been pushed. The arguments are in the window's outside coordinate system. The method tracks the mouse blinker.

:set-mouse-position *x y* of **tv:essential-mouse** *Method*

Positions the mouse blinker at window coordinates *x* and *y*.

To position the mouse blinker at absolute screen coordinates, use the function **tv:mouse-warp.**

:who-line-documentation-string of **tv:sheet** *Method*

The Scheduler periodically sends this message to the window owning the mouse. The returned value is displayed in the mouse documentation line. The value should be a string or, for no documentation, **nil**. This method returns **nil**; supply your own to provide mouse documentation.

tv:mouse-warp *x y* &optional (*mouse* **tv:main-mouse**) *Function*

Positions the mouse blinker at screen coordinates *x* and *y*. (The optional argument *mouse* is used in multiple-console systems.)

To position the mouse blinker at coordinates relative to a particular window, use **(flavor:method :set-mouse-position tv:essential-mouse)**.

tv:mouse-set-blinker-cursorpos *Function*

Positions the mouse blinker at point (**sys:mouse-x**, **sys:mouse-y**) on **tv:mouse-sheet.**

sys:mouse-wakeup *Function*

Causes **tv:mouse-input** to return as if the mouse had moved. This causes the default mouse handler to send the window owning the mouse a **:mouse-moves** message.

12.9.3 Mouse Blips

Mouse blips are lists inserted into the input buffer of a window when the mouse is clicked within that window. (Do not confuse these blips with presentation blips generated by translating mouse handlers when the mouse is clicked on a presentation in a Dynamic Window: See the section "Overview of Presentation Input Blip Facilities" in *Programming the User Interface, Volume A.*) The list contains five elements:

1. The keyword **:mouse-button**.
2. A mouse character corresponding to which button (left, middle, right) was clicked.
3. The window that received the blip.
4. The x-coordinate of the mouse cursor when the mouse was clicked.
5. The y-coordinate of the mouse cursor when the mouse was clicked.

Blips representing mouse clicks are sent by the **:mouse-click** method of **tv:essential-mouse**, a component of **tv:minimum-window**. You can receive mouse blips by sending the window a **:list-tyi** or **:any-tyi** message. (For an example: See the section "Mouse Characters", page 169.)

:mouse-click *buttons x y* of **tv:essential-mouse** *Method*

This method is called by the **:mouse-buttons** method of **tv:essential-mouse**, which is called by the default mouse handler when mouse buttons are pushed. *buttons* is a structure representing the buttons pushed; use reader macros like **#\mouse-r** to handle these structures in your program: See the section "Mouse Characters", page 169. *x* and *y* represent the position of the mouse at the time of the click, in the window's outside coordinates.

If the click is **#\sh-mouse-r**, the **:mouse-buttons** method pops up a system menu. Otherwise, if the window has an I/O buffer, **:mouse-click** sends it a blip of the form (**:mouse-button** *buttons window x y*). In addition, if the click is **#\mouse-l**, the window is selected.

:mouse-click methods are combined using **:or** combination, so the **:mouse-click** method of **tv:essential-mouse** runs only if no earlier method handles the message (and all earlier methods return **nil**). This allows a method to intercept only certain clicks and return non-**nil**, and to pass on other clicks and return **nil**.

12.9.4 Mouse Characters

Mouse characters are implemented as structures, not character objects, but the printed representation is similar. **#\mouse-l**, **#\mouse-m**, and **#\mouse-r** correspond to left, middle and right clicks. Mouse characters can be qualified by shift keys. For example, **#\c-mouse-m** indicates a click-middle with the CONTROL key depressed.

Mouse characters prefixed with **sh-**, for example, **#\sh-mouse-r**, can be generated in two ways. Either the user had the SHIFT key depressed when clicking the right mouse button, or the right mouse button was clicked twice in rapid succession. (The latter interpretation is only possible if the variable **tv:mouse-double-click-time** has not been set to **nil**.)

Because mouse characters are not implemented as other characters, they require their own set of manipulation functions. For example, to compare mouse characters, the function **char-mouse-equal** has been provided; and the predicate **mouse-char-p** is available for determining whether an object is a mouse character. Such functions are commonly used when handling mouse blips, as shown in the following example:

```
(defun get-mouse-char ()
  (let (blip mouse-char)
    (setq blip (send *graphics-window* :list-tyi))
    (setq mouse-char (second blip))
    (if (mouse-char-p mouse-char)
        (cond ((char-mouse-equal mouse-char #\mouse-l)
               (left-click-function))
              ((char-mouse-equal mouse-char #\mouse-r)
               (right-click-function))
              (t (send *graphics-window* :beep)))
        (send *graphics-window* :beep))))
```

char-mouse-equal, **mouse-char-p**, and three additional mouse character functions are documented in the following subsection.

12.9.4.1 Mouse Character Functions

char-mouse-equal *char1 char2* *Function*

Returns **t** if the mouse characters *char1* and *char2* are equal, **nil** otherwise.

mouse-char-p *char* *Function*

Returns **t** if *char* is a mouse character, **nil** otherwise.

char-mouse-button *char* *Function*

Returns the number corresponding to the mouse button that would have to be pushed to generate *char*. 0, 1, and 2 correspond to the left, middle, and right mouse buttons, respectively.

Example:

```
(char-mouse-button #\m-mouse-m) ==>
1
```

The complementary function is **make-mouse-char**.

char-mouse-bits *char* *Function*

Returns the value of the bits field of a mouse character. The bits field encodes the shift keys, if any, qualifying the root mouse character:

Bits	Shift Key
0	None
1	CONTROL
2	META
4	SUPER
8	HYPER
16	SHIFT

Every combination of shift keys corresponds to a unique *bits* value, for example:

```
(char-mouse-bits #\c-s-sh-Mouse-L) ==>
21
```

make-mouse-char *button* &optional (*bits* **0**) *Function*

Constructs a mouse character given a mouse button number. 0, 1, and 2 correspond to the left, middle, and right mouse buttons, respectively.

The optional *bits* argument is a number encoding the shift keys qualifying the root mouse character as follows:

Bits	Shift Key
0	None
1	CONTROL
2	META
4	SUPER

8	HYPER
16	SHIFT

The shift keys are additive with respect to the *bits* value, for example:

```
(make-mouse-char 0 31) ==>
#\h-s-m-c-sh-Mouse-L
```

12.9.5 Grabbing the Mouse

When the mouse is grabbed, the mouse process gets told that no window owns the mouse, and it changes the mouse blinker back to the default (a northeast arrow). The mouse process continues to track the mouse, and your process can now watch the position and the buttons by using the variables and functions described below. (In Genera 7.0, the corresponding facility for Dynamic Windows is **dw:tracking-mouse**: See the macro **dw:tracking-mouse** in *Programming the User Interface, Volume A.*)

tv:with-mouse-grabbed *Special Form* no effect ?

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. no effect ?

tv:with-mouse-grabbed-on-sheet (&optional (*sheet* **'self**)) &body *body* *Special Form*

Evaluates *body* with the mouse grabbed and confined to *sheet*. During execution the variables **sys:mouse-x** and **sys:mouse-y** are relative to the window's outside coordinates. The default value of *sheet* is **self**, so if *sheet* is not supplied, this form needs to appear inside a method or defun-method of a window flavor.

tv:with-mouse-and-buttons-grabbed &body *body* *Special Form*

The forms in *body* are evaluated with the mouse and buttons grabbed. When the buttons are grabbed, the mouse process does not maintain the value of **tv:mouse-last-buttons**. Instead, the user process can read directly from the mouse buttons, without losing clicks that the mouse process might fail to notice. Within the body of this form, you can call the functions **tv:mouse-wait**, **tv:wait-for-mouse-button-down**, **tv:wait-for-mouse-button-up**, and **sys:mouse-buttons**.

tv:with-mouse-and-buttons-grabbed-on-sheet (&optional (*sheet* **'self**)) &body *body* *Special Form*

Like **tv:with-mouse-and-buttons-grabbed**, except that the mouse is confined to *sheet*. During execution the variables **sys:mouse-x** and **sys:mouse-y** are relative to the window's outside coordinates. The default value of *sheet* is **self**, so if *sheet* is not supplied, this form needs to appear inside a method or defun-method of a window flavor.

sys:mouse-x *Variable*

The value is the x-coordinate of the position of the mouse, in pixels, measured from the upper-left corner of the screen the mouse is on (the value of **tv:mouse-sheet**). This variable is maintained by the process handling the mouse, normally the mouse process. It is in outside coordinates, since the mouse might be in the margins somewhere.

sys:mouse-y *Variable*

The value is the y-coordinate of the position of the mouse, in pixels, measured from the upper-left corner of the screen the mouse is on (the value of **tv:mouse-sheet**). This variable is maintained by the process handling the mouse, normally the mouse process. It is 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. (Except on the Symbolics 3600, chording is not supported; that is, if more than one button is depressed, the integer returned in not the sum of the individual button codes.)

tv:mouse-wait &optional (*old-x* **sys:mouse-x**) (*old-y* **sys:mouse-y**) (*old-buttons* **tv:mouse-last-buttons**) (*whostate* **"Mouse"**) (*timeout* **nil**) *Function*

This function waits until any of the variables **sys:mouse-x**, **sys:mouse-y**, or **tv:mouse-last-buttons** to become different from the values passed as arguments, or until *timeout* sixtieths of a second have elapsed. While waiting, *whostate* is displayed in the status line. 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-wait returns three values:

- An integer representing the state of the mouse buttons, in the format used by the variable **tv:mouse-last-buttons**

- The X-coordinate of the mouse
- The Y-coordinate of the mouse

tv:wait-for-mouse-button-down &optional (*prompt* **"Button"**) *Function*

If any buttons are down, waits until all the buttons are up, then waits for any mouse button to be pushed. If no buttons are down, waits for any button to be pushed. *prompt* is the whostate to display while waiting. Returns the same three values as **tv:mouse-wait**.

This must be called inside a **tv:with-mouse-and-buttons-grabbed** or a **tv:with-mouse-and-buttons-grabbed-on-sheet** form.

tv:wait-for-mouse-button-up &optional (*prompt* **"Release Button"**) (*timeout* **nil**) *Function*

Waits until all mouse buttons are up, or until *timeout* sixtieths of a second have elapsed. *prompt* is the whostate to display while waiting. Returns the same three values as **tv:mouse-wait**.

This must be called inside a **tv:with-mouse-and-buttons-grabbed** or a **tv:with-mouse-and-buttons-grabbed-on-sheet** form.

tv:mouse-button-encode *bd* *Function*

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 integer 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***)** computes this mask.

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. **tv:with-mouse-grabbed** and **tv: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 can **setq** this variable to a string to be displayed in the mouse documentation line. If your program has "modes" that affect how the click acts, each part of the program should **setq** this variable to its own documentation.

12.9.6 Usurping the Mouse

You can tell the mouse process not to 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**) *Function*

Wait until something happens with the mouse, and then return saying what happened. Six 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**) saying what buttons have changed since the last time **tv:mouse-input** was called. The last two values are the current x- and y-position of the mouse or, if any buttons changed, the position of the mouse at that time.

You can 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 will not work correctly.

The variables **sys:mouse-x** and **sys: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 does not wait; it can return with all zeroes, indicating that nothing has changed.

sys:mouse-buttons &optional *peek* *Function*

Return the current state of the mouse buttons. If *peek* is not **nil**, it looks at the state without pulling anything out of the buffer (of pending mouse-button transitions).

sys:mouse-buttons returns four values:

- An integer representing the state of the mouse buttons: 0 means no buttons were depressed; 1 means the left button was depressed; 2 the middle button; and 4 the right button. (Except on the Symbolics 3600, chording is not supported; that is, if more than one button is depressed, the integer returned in not the sum of the individual button codes.)

- An integer representing the time when that state was true

- The X-coordinate of the mouse at that time

- The Y-coordinate of the mouse at that time

To use some parts of the mouse software, such as **tv:mouse-button-encode**, you can store these four returned values into the variables **tv:mouse-last-buttons**, **tv:mouse-last-buttons-time**, **tv:mouse-last-buttons-x**, and **tv:mouse-last-buttons-y**, respectively. The mouse process does this itself when the mouse is not usurped.

12.9.7 Controlling the Mouse Outside a Window

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 from it before it vanishes.

(The **dw:dynamic-window** resource has a **:hysteresis** option, allowing you to get Dynamic Windows with this capability mixed in.)

:hysteresis *n-pixels* (for **tv:hysteretic-window-mixin**) *Init Option*

Set the initial value of the hysteresis, in pixels. It defaults to **25.** (decimal).

:hysteresis of **tv:hysteretic-window-mixin** *Method*

Examine the hysteresis of the window, in pixels.

:set-hysteresis *new-hysteresis* of **tv:hysteretic-window-mixin** *Method*

Set the hysteresis of the window, in pixels.

12.9.8 Scaling Mouse Motion

The following two variables apply to Dynamic Windows as well as static windows.

tv:mouse-x-scale-array *Variable*

The value of this variable is an array that, along with the array that is the value of **tv:mouse-y-scale-array**, can be used to control mouse scaling. These arrays determine the relation between the rates of motion of the mouse on the table and the mouse cursor on the screen. This relation can be nonlinear and can vary with the speed of the mouse. For example, fast mouse motion can move the cursor a distance that is proportionally greater than slow mouse motion.

Scaling is computed as follows. The even-numbered elements of **tv:mouse-x-scale-array** are compared with the value of **tv:mouse-x-speed**, and the even-numbered elements of **tv:mouse-y-scale-array** are compared with the value of **tv:mouse-y-speed**. **tv:mouse-x-speed** and **tv:mouse-y-speed** are the x- and y-components of the mouse speed on the table, typically in units of hundredths of an inch per second.

For each array, the first even array element that is greater than the mouse speed causes its corresponding odd-numbered array element to be multiplied by the mouse motion on the table and then divided by 1024 (decimal). The result is the mouse motion on the screen. Appropriate care is taken to save the fractions for the next computation.

The default array setup code is as follows:

```
;;; Use a scale of 2/3 in X, 3/5 in Y when moving at slow speed,
;;; double that at high speed
(aset 80. tv:mouse-x-scale-array 0)
(aset (// (lsh 2 10.) 3) tv:mouse-x-scale-array 1)
(aset 80. tv:mouse-y-scale-array 0)
(aset (// (lsh 3 10.) 5) tv:mouse-y-scale-array 1)
(aset #o17777777777 tv:mouse-x-scale-array 2)
(aset (// (lsh 4 10.) 3) tv:mouse-x-scale-array 3)
(aset #o17777777777 tv:mouse-y-scale-array 2)
(aset (// (lsh 6 10.) 5) tv:mouse-y-scale-array 3))
```

The following code provides for simple scaling of motion for the Hawley mouse. The microcode knows specially about each array. You can store into each array, but you cannot replace it with a new array or use **zl:adjust-array-size** on it.

```
;;; Aids to trying speed-dependent scaling
;;; Specs are scale-factor speed-break
;;; No attempt to treat X and Y differently
;;; Args of (1 80. 2) seem to be about right for the Hawley mouse
(defun mouse-speed-hack (&rest specs)
  (loop for (scale speed) on specs by 'cddr
        for i from 0 by 2
        do (aset (or speed #o37777777) tv:mouse-x-scale-array i)
        (aset (or speed #o37777777) tv:mouse-y-scale-array i)
        (aset (// (fix (* 2 scale 1024.)) 3)
              tv:mouse-x-scale-array (1+ i))
        (aset (// (fix (* 3 scale 1024.)) 5)
              tv:mouse-y-scale-array (1+ i))))

(defun hawley-mouse-hack ()
  (mouse-speed-hack 1 80. 2))
```

tv:mouse-y-scale-array *Variable*

The value of this variable is an array that, along with the array that is the value of **tv:mouse-x-scale-array**, can be used to control mouse scaling. See the variable **tv:mouse-x-scale-array**, page 176.

12.10 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.

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. You can use the variables **tv:mouse-double-click-time** and **tv:*mouse-incrementing-keystates*** to augment or replace double clicks with shifted clicks.

Mouse characters can be modified with the modifier keys CONTROL, META, SUPER, and HYPER, just as keyboard characters can. Which of these keys modify mouse characters depends on the value of the variable **tv:*mouse-modifying-keystates***.

The editor considers each modified mouse click to be a separate command. You can bind commands to particular modified mouse clicks. You can also use Install Mouse Macro (m-X) with modified mouse clicks to increase the number of mouse macros available.

You can use **login-forms** in an init file to set the variables

tv:mouse-double-click-time, **tv:*mouse-incrementing-keystates***, and **tv:*mouse-modifying-keystates*** and customize the behavior of the mouse.

tv:key-state *key-name* *Function*

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

key-name may be the symbolic name of a modifier key, from the table below, or the number of a nonmodifier key, which is the character you get when you type that key without any modifiers: a lowercase letter, a digit, or a special character. Modifier 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 modifier key names are:

:shift	:left-shift	:right-shift
:symbol	:left-symbol	:right-symbol
:control	:left-control	:right-control
:meta	:left-meta	:right-meta
:super	:left-super	:right-super
:hyper	:left-hyper	:right-hyper
:caps-lock	:repeat	:mode-lock

tv:mouse-double-click-time *Variable*

The maximum period of time (in microseconds) between mouse clicks for which the clicks are interpreted as a double click instead of two single clicks. Default: **200000** (decimal).

If you set this variable to **nil**, disabling double clicking entirely, mouse response time improves slightly in static windows and appreciably in Dynamic Windows. This is the recommended setting for Genera 7.0 and later systems.

tv:*mouse-incrementing-keystates* *Variable*

A list of names of keys, acceptable to **tv:key-state**. If one or more of these keys are pressed, single mouse clicks are interpreted as double clicks. Default: **(:shift)**.

tv:*mouse-modifying-keystates* *Variable*

A list of names of keys, acceptable to **tv:key-state**. If one or more of these keys are pressed, sets the corresponding modifier bits in the mouse character. Default: **(:control :meta :super :hyper)**. If a key appears as an element of both this list and the list that is the value of **tv:*mouse-incrementing-keystates***, the modifier bit is set and the click is interpreted as a double click.

tv:key-test *Function*

tv:key-test allows you to check that your keyboard and mouse hardware are functioning correctly. It displays a keyboard image and a mouse image. The mouse image tracks the mouse when mouse tracking is functioning correctly. Holding down a key or button causes the corresponding key or button on the screen to go into inverse video. The END key returns. This function is not loaded as part of the world load but is available:

```
Load System keytest
(tv:key-test)
```

12.11 Window Sizes and Positions

The messages and init options in this section are used to examine and set the sizes and positions of windows, both static and dynamic. 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. See the section "Character Output to Windows", page 121.

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.

The following 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.

12.11.1 Initializing Window Size and Position

:left *left-edge* (for **tv:sheet**) — *Init Option*

Specifies the x-coordinate of the left edge of the window.

:x *left-edge* (for **tv:sheet**) — *Init Option*

Specifies the x-coordinate of the left edge of the window.

:top *top-edge* (for **tv:sheet**) — *Init Option*

Specifies the y-coordinate of the top edge of the window.

:y *top-edge* (for **tv:sheet**) — *Init Option*

Specifies the y-coordinate of the top edge of the window.

:position *(left-edge top-edge)* (for **tv:sheet**) — *Init Option*

Specifies the x-coordinate of the left edge and the y-coordinate of the top edge of the window.

:right *right-edge* (for **tv:sheet**) — *Init Option*

Specifies the x-coordinate of the right edge of the window.

:bottom *bottom-edge* (for **tv:sheet**) — *Init Option*

Specifies the y-coordinate of the bottom edge of the window.

:width *outside-width* (for **tv:sheet**) — *Init Option*

Specifies the outside width of the window.

:height *outside-height* (for **tv:sheet**) — *Init Option*

Specifies the outside height of the window.

:size *(outside-width outside-height)* (for **tv:sheet**) — *Init Option*

Specifies the outside width and height of the window.

:inside-width *inside-width* (for **tv:sheet**) — *Init Option*

Specifies the inside width of the window.

:inside-height *inside-height* (for **tv:sheet**) *Init Option*
Specifies the inside height of the window.

:inside-size *(inside-width inside-height)* (for **tv:sheet**) *Init Option*
Specifies the inside width and height of the window.

:edges *(left-edge top-edge right-edge bottom-edge)* (for **tv:sheet**) *Init Option*
Specifies the x-coordinates of the left and right edges and the y-coordinates of the top and bottom edges of the window.

:character-width *spec* (for **tv:sheet**) *Init Option*
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 the default character style.

:character-height *spec* (for **tv:sheet**) *Init Option*
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* (for **tv:sheet**) *Init Option*
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* (for **tv:essential-window**) *Init Option*
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 the default character style.

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.

: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* (for **tv:essential-window**) *Init Option*

In combination with the **:edges-from :mouse** init option, this option and **:minimum-height** 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.

:minimum-height *n-pixels* (for **tv:essential-window**) *Init Option*

In combination with the **:edges-from :mouse** init option, this option and **:minimum-width** 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.

tv:set-default-window-size *flavor-name superior existing-windows* &rest *options* *Function*

tv:set-default-window-size allows you to modify the default size chosen by the system when you create a window without specifying either a size or a position for it. For example, when you create a Lisp Listener by pressing SELECT c-L, the default size is the full size of the screen, unless you modify it.

The arguments to **tv:set-default-window-size** are:

flavor-name	The flavor of window to be affected. Flavors built on top of this do not inherit this flavor's default window size. **nil** here means *all windows.*
superior	The window whose direct inferiors are to be affected; typically, the value of **tv:main-screen**.
existing-windows	An indicator as to whether existing windows must conform to these options. Any non-**nil** argument forces all existing windows of the specified *flavor-name* and *superior* to conform to the options.
options	Alternating keywords and values that are used as defaults in creating windows whose size or position is not specified. Valid keywords are **:width**, **:left**, **:right**, **:height**, **:top**, and **:bottom**. They have the same meaning as in **tv:make-window.**

For example:

```
(tv:set-default-window-size
  'zwei:zmacs-frame tv:main-screen t ':width 1400)
```

12.11.2 Messages for Window Size and Position

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.)

:change-of-size-or-margins &rest *options* of **tv:sheet** *Method*

Changes window size or margins, processing *options*. This message is sent by the system; you might need to provide an **:after** daemon for it.

:size of **tv:sheet** *Method*

Return two values: the outside width and outside height.

:set-size *new-width* *new-height* &optional *option* of **tv:essential-set-edges** *Method*

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 of **tv:sheet** *Method*

Return two values: the inside width and the inside height.

:set-inside-size *new-inside-width* *new-inside-height* &optional *option* of **tv:essential-set-edges** *Method*

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 of **tv:sheet** *Method*

Return two values: the inside size in characters, and the inside height in lines. The size of the default character style is used.

:set-size-in-characters *width-spec height-spec* &optional *option* of **tv:sheet** *Method*

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 of **tv:sheet** *Method*

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* of **tv:essential-set-edges** *Method*

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

:edges of **tv:sheet** *Method*

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* of **tv:essential-set-edges** *Method*

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 of **tv:sheet** *Method*

Return four values: the sizes of the left, top, right, and bottom margins, respectively.

:left-margin-size of **tv:sheet** *Method*

Returns the left margin size of the window in pixels.

:top-margin-size of **tv:sheet** *Method*

Returns the top margin size of the window in pixels.

:right-margin-size of **tv:sheet** *Method*

Returns the right margin size of the window in pixels.

:bottom-margin-size of **tv:sheet** *Method*

Returns the bottom margin size of the window in pixels.

:inside-edges of **tv:sheet** *Method*

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* of **tv:essential-set-edges** *Method*

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***)* of **tv:essential-set-edges** *Method*

If the window is not exposed, change its position according to *mode* and expose it (with the **:expose** message). 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 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 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 to the center of the window.

12.12 Window Margins, Borders, and Labels

There is a 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. With few exceptions, all of the facilities discussed are intended for static windows, not dynamic ones. For information on equivalent facilities for use with Dynamic Windows: See the section "Overview of Window Substrate Facilities" in *Programming the User Interface, Volume A*. More detailed information is available in the reference documentation for **dw:dynamic-window**: See the flavor **dw:dynamic-window** in *Programming the User Interface, Volume A.*

You can ask for the size of the margins with the **:margins** message.

tv:margin-space-mixin *Flavor*

This flavor provides a margin item that just leaves some blank space. It might be useful if you're using scroll bars, and you want to leave a little white space between the scroll bar and the inside of the window.

:margin-space (for **tv:margin-space-mixin**) *Init Option*

Initializes the amount of blank space in the margins of the window. Possible values:

nil	No space
t	One pixel blank in each of the four margins

n — *n* pixels of space in each of the four margins (*n* is an integer)

(*left top right bottom*) — *left* pixels blank in the left margin, *top* pixels blank in the top margin, and so on (values are integers)

:margin-space of **tv:margin-space-mixin** *Method*

Returns a list of four elements, (*left top right bottom*). These are integers representing the number of pixels of blank space in the four margins of the window.

:set-margin-space *new-space* of **tv:margin-space-mixin** *Method*

Specifies the amount of blank space to be left in the margins of the window. Possible values of *new-space*:

nil — No space

t — One pixel blank in each of the four margins

n — *n* pixels of space in each of the four margins (*n* is an integer)

(*left top right bottom*) — *left* pixels blank in the left margin, *top* pixels blank in the top margin, and so on (values are integers)

12.12.1 Window Borders

tv:borders-mixin *Flavor*

The **tv:borders-mixin** margin item creates the borders around windows that you often see when using the Lisp Machine. You can control the thickness of each of the four borders separately, or of all of them together. You can also specify your own function to draw the borders, if you want something more elaborate than simple lines.

The borders also include some white space 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* (for **tv:borders-mixin**) *Init Option*

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 which applies to each of the four borders.

a list (*left top right bottom*)

Specifications for each of the four borders of the window.

a list (*keyword1 spec1 keyword2 spec2...*)

Specifications 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" 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**. 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 is allocated, whereas in the latter case it is not.

:set-borders *new-borders* of **tv:borders-mixin** *Method*
Redefine the borders. *new-borders* can be any of the things that can be used for the **:borders** init option.

:border-margin-width *n-pixels* (for **tv:borders-mixin**) *Init Option*
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 of **tv:borders-mixin** *Method*
Return the value of the border margin width.

:set-border-margin-width *new-width* of **tv:borders-mixin** *Method*
Set the value of the border margin width.

12.12.2 Window Labels

Of the following facilities, only the **:name** and **:label** init options and the **:name** method apply to Dynamic Windows; the rest are intended for static windows. For information on equivalent facilities intended for use with Dynamic Windows: See the section "Overview of Window Substrate Facilities" in *Programming the User Interface, Volume A.*

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 character style in which it is displayed, and whether it appears at the top of the window or the bottom.

:name *name* (for **tv:sheet**) *Init Option*
The value is the name of the window, which should be a string. All windows have names; note that this is an init option of **tv:sheet**. It is mentioned here because the main use of the name is as the default string for the label, if there is a label.

:name of **tv:sheet** *Method*
Return the name of the window, which is a string.

:label *specification* (for **tv:label-mixin**) *Init Option*
Set the string displayed as the label, the character style in which the label is displayed, and whether the label is at the top or the bottom of the window. Anything you don't specify will default; by default, the string is the same as the name of the window, the character style is the default

character style 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 character style
The label is displayed in the specified character style.

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*.

:character-style *character-style*
The label is displayed in the specified character style, merged against the default character style.

:label-size of **tv:label-mixin** *Method*
Return the width and height of the area occupied by the label.

:set-label *specification* of **tv:label-mixin** *Method*
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. 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* of **tv:changeable-name-mixin** *Method*

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* of **tv:delayed-redisplay-label-mixin** *Method*

This is like the **:set-label** method, except that nothing actually happens until an **:update-label** message is sent.

:update-label of **tv:delayed-redisplay-label-mixin** *Method*

Actually do the **:set-label** operation on the *specification* given by the most recent **:delayed-set-label** message.

12.13 Text Scroll Windows

12.13.1 Concepts

A *text scroll window* maintains and displays an ordered list of Lisp objects, one on each line. The caller inserts objects into or deletes objects from the list by sending messages, and the window dynamically redisplays to show the changes. If there are more items in the list than lines in the window, the text scroll window displays some portion of the items. The portion that is shown is controlled by *scrolling* the window. The caller scrolls the window by sending messages, and the user scrolls it by using the mouse scroll bar.

12.13.2 Text Scroll Window Flavors

tv:text-scroll-window is the most basic text scroll window mixin. It simply displays the items and allows you to scroll the window using the mouse against the left edge.

tv:function-text-scroll-window lets you provide a function to print an item, replacing **prin1**, to give you finer control over how each item is displayed.

tv:mouse-sensitive-text-scroll-window makes the items displayed on the window sensitive to mouse clicks.

tv:margin-scrolling-with-flashy-scrolling-mixin provides the *More above/More below* facility.

12.13.2.1 Basic Use of Text Scroll Windows

You can use any of the usual options to **tv:make-window** to control such parameters as the size and shape of the window. When the window is first created, its item list is empty and it displays as an empty window.

tv:text-scroll-window *Flavor*

This is the base flavor of text scroll window, on which all the others are built. Each item displays using the **prin1** function, truncating at the end of the line.

tv:text-scroll-window must be treated as a mixin.

:insert-item *item-no new-item* of **tv:text-scroll-window** *Method*

Inserts *new-item* into the item list before *item-no*. *new-item* can be any Lisp object. *item-no* is an item number, and should be a non-negative fixnum.

If the item is inserted within the visible range, the window redisplays to show the new item.

:append-item *new-item* of **tv:text-scroll-window** *Method*

Inserts *new-item* after the last item in the list. *new-item* can be any Lisp object.

If the last item in the list is visible in the window and there is room to display the new item, the window redisplays to show the new item.

:delete-item *item-no* of **tv:text-scroll-window** *Method*
Deletes the item whose number is *item-no*.

If the item being deleted was visible, the window redisplays to show the new state of the item list.

:replace-item *item-no new-item* of **tv:text-scroll-window** *Method*
Replaces the item whose number is *item-no* with *new-item*.

If the item is currently visible, the window redisplays to show the new item.

:set-items *new-items* &optional (*new-top-item* **0**) of **tv:text-scroll-window** *Method*
new-items should be an array with a fill pointer. It becomes the new array used internally to hold the list of items. The window redisplays with the item whose number is *new-top-item* in the topmost line.

new-items can also be an integer, in which case this method allocates a new array of that length, and set its fill pointer to zero, making the list of items empty.

:items of **tv:text-scroll-window** *Method*
Returns the array that the window uses, internally, to hold the items. You should not modify the contents of this array or its fill pointer, because the window won't know that you did so, and redisplay will not work properly.

:number-of-items of **tv:text-scroll-window** *Method*
Returns the number of items in the item list.

:top-item of **tv:text-scroll-window** *Method*
Returns the number of the item being displayed in the topmost line of the window, or zero if the item list is empty.

:last-item of **tv:text-scroll-window** *Method*
Returns the last item in the item list.

:put-item-in-window *item* of **tv:text-scroll-window** *Method*
The first occurrence of *item* is located. If it occurs before the first item in the window, the window redisplays so that *item* appears in the top line. If it occurs after the last item in the window, the window redisplays so that *item* appears in the bottom line.

If *item* is already visible or is not in the list, nothing happens.

:put-last-item-in-window of **tv:text-scroll-window** *Method*

If the last item is not visible, the window redisplays so that the last item appears in the bottom line.

:item-value *item-no* of **tv:text-scroll-window** *Method*

Returns the item whose number is *item-no.*

:scroll-to *number type* of **tv:basic-scroll-bar** *Method*

Scrolls the window depending on *type.* If *type* is **:relative,** then scrolls the window *number* items in either the positive or negative direction. If *type* is **:absolute** then puts the item whose number is *number* in the topmost line.

Example of a Text Scroll Window

This example creates a small text scroll window in the upper left corner of the screen and uses most of the text scroll window methods. It then leaves the window on the screen so that you can also scroll the window using the mouse. Reselect the original window to deexpose it.

```
(defflavor test-window ()
          (tv:text-scroll-window tv:window)
  )

(defvar *test-window*
       (tv:make-window 'test-window
                       :edges '(0 0 400 100)
                       :expose-p nil))
```

```
(defun test-basic-scroll-window ()
  ;; Initialize window
  (send *test-window* :set-items 0)           ; Clear the items
  (send *test-window* :expose)
  (send *test-window* :scroll-to 0 :absolute)   ; Scroll to the top
  ;; Demonstrate appending of items to the end of the list
  (loop for i from 0 to 10
        do
    (send *test-window* :append-item (list 'appended i))
    (process-sleep 60 (format nil "appending ~d" i)))
  ;; Demonstrate absolute scrolling
  (loop for i from 1 to 10 by 2
        do
    (send *test-window* :scroll-to i :absolute)
    (process-sleep 60 (format nil "scrolled to item ~d" i)))
  ;; Scroll to a arbitrary point in the middle of the item list
  (send *test-window* :scroll-to 3 :absolute)
  ;; Demonstrate insertion of items
  (loop for i from 1 to 10
        for j from 10 by -1
        do
    (send *test-window* :insert-item j (list 'inserted i))
    (process-sleep 60 (format nil "inserting ~d at ~d" i j)))
  ;; Demonstrate replacement of items
  (loop for i from 1
        for j from 1 by 3
        until (> j (send *test-window* :number-of-items))
        do
    (send *test-window* :replace-item j (list 'replaced i))
    (process-sleep 60 (format nil "replacing ~d at ~d" i j)))
  ;; Scroll to bottom of item list
  (send *test-window* :put-last-item-in-window)
  (process-sleep 60 "put last item in window")
  ;; Demonstrate relative scrolling
  (loop until (zerop (send *test-window* :top-item))
        do
    ;; Scroll back two items
    (send *test-window* :scroll-to -2 :relative)
    (process-sleep 60 "scrolled back 2"))
  ;; Demonstrate deletion of items
  (loop until (< (send *test-window* :number-of-items) 10)
         do
```

```
    (send *test-window* :delete-item 0)
    (process-sleep 60 "deleting the first item")))
```

12.13.2.2 Formatting Text Scroll Window Items

The simple **tv:text-scroll-window** calls **prinl** on each item to display it on a line of the screen. **tv:function-text-scroll-window** lets you provide a function of your own to replace **prinl.**

When the window displays a line, the function is called with four arguments:

- The item to be printed.
- An object associated with the window. See the method **(flavor:method :set-print-function-arg tv:function-text-scroll-window)**, page 196.
- The window itself.
- The number of the item in the window's item list.

When the function is called, the window's cursor is positioned to the beginning of the appropriate line on the window, so you can just send stream output messages to the window (the third argument).

Do *not* output the new-line character to the window.

tv:function-text-scroll-window *Flavor*

Lets you provide a function to print the items in a text scroll window.

:set-print-function *function* of **tv:function-text-scroll-window** *Method*

Sets the printing function of the window to *function.*

:print-function of **tv:function-text-scroll-window** *Method*

Returns the window's printing function.

:set-print-function-arg *new-function-arg* of **tv:function-text-scroll-window** *Method*

Sets the object which the window passes as the second argument to the print function.

:print-function-arg of **tv:function-text-scroll-window** *Method*

Returns the object which the window passes as the second argument to the print function.

Example of Formatting Text Scroll Window Items

Change the previous example (See the section "Example of a Text Scroll Window", page 194.) as follows:

```
(defflavor test-window ()
           (tv:function-text-scroll-window tv:window)
  )

           .
           .
           .

(defun test-basic-scroll-window ()
  (send *test-window* :set-print-function
        #'(lambda (item object window number)
            (format window "~:r item which was ~a."
                    (second item)
                    (first item))))
  (send *test-window* :set-items 0)

           .
           .
           .
```

12.13.2.3 Mouse-Sensitive Items in Text Scroll Windows

The flavors **tv:mouse-sensitive-text-scroll-window** and **tv:mouse-sensitive-text-scroll-window-without-click** allow you to create *mouse-sensitive items*; that is, regions of each line can be made sensitive to mouse clicks.

Note that the word "item" is being used in two ways. One "item" of the item list is displayed on every line, but each line might have many "mouse-sensitive items" on it.

When the mouse is clicked, a *blip* is forced into the window's input buffer. The elements of the blip are:

- The type of the mouse-sensitive-item.
- The "item" which the "mouse-sensitive item" was in.
- The window itself.
- The mouse click character. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O.*

tv:mouse-sensitive-text-scroll-window *Flavor*

To use this flavor, you must create your own flavor based on this one, and redefine the **:print-item** message. Your new handler for **:print-item** can send the **:item** message to the window to create a new mouse-sensitive item.

tv:mouse-sensitive-text-scroll-window-without-click *Flavor*

This is just like **tv:mouse-sensitive-text-scroll-window**, but without the **:mouse-click** method, so that you can provide your own. (You can't just override it, because **:mouse-click** is combined with the **:or**) method combination.

:print-item *item line-no item-no* *Message*

A text scroll window sends itself this message to display *item* on a line of the screen. *line-no* is the number of the line on the screen, and *item-no* is the number of the item in the list of items. When this message is sent, the cursor is already positioned to the beginning of line *line-no*; your method should send stream output messages to the window (i.e. to *self*) to print *item*.

For "mouse-sensitive items" within the "item", send **:item** to *self*.

:item *item type* &optional (*function* **(function prinl)**) &rest *print-args* of **tv:mouse-sensitive-text-scroll-window-without-click** *Method*

Creates a new mouse-sensitive item. *item* may be any lisp object. *type* is a keyword which specifies the type of item. *function* is the function which is used to display the item in the window. *print-args* are further arguments to *function*.

This method prints *item* on the window at the current cursor position by calling *function*. The first argument to *function* is *item*; the second is the window itself; and the rest are the elements of *print-args*.

The portion of the window printed on by this method becomes mouse-sensitive, and a box appears around it when the mouse is moved into that area.

:mouse-sensitive-item *x y* of **tv:mouse-sensitive-text-scroll-window-without-click** *Method*

Returns the mouse-sensitive item at a given location.

The arguments are the *x* and *y* coordinates of the location. Two values are returned: the item and its type, or **nil** and **nil** if the mouse was not over any mouse-sensitive item.

This message is useful to send from your **:mouse-click** handler; the *x* and *y* parameters from **:mouse-click** can be passed along.

tv:sensitive-item-types *Variable*

This is a gettable, settable, and initable instance variable of **tv:mouse-sensitive-text-scroll-window-without-click** that controls which types of mouse-sensitive items are actually sensitive at any given time.

There are several possible values for **tv:sensitive-item-types**:

- **t**: All mouse-sensitive objects are sensitive, regardless of type. This is the default.
- A list: Only items whose type is an element of the list are sensitive.
- A function: The function must take as its only argument a mouse-sensitive item object and it should return **t** if it wants the item to be sensitive and **nil** otherwise.
- A symbol other than **t**: Taken to be a message to be sent to the window. The corresponding method should be a function of one argument returning **t** or **nil** as in the case of the function.

tv:displayed-item-item *mouse-sensitive-item* *Macro*

Given a mouse-sensitive item, returns the associated item.

tv:displayed-item-type *mouse-sensitive-item* *Macro*

Given a mouse-sensitive item, returns the type of the item.

Example of Mouse-Sensitive Items in Text Scroll Windows

This example creates a frame with a text scroll window and a plain window as panes. Clicks on the text scroll window display the blips on the plain window and toggle the mouse-sensitivity of the items.

```
(defflavor test-pane ()
           (tv:mouse-sensitive-text-scroll-window
            tv:pane-no-mouse-select-mixin
            tv:window)
  (:default-init-plist
   :sensitive-item-types :sensitive-type-p))
```

```
(defmethod (test-pane :print-item) (item ignore ignore)
  (send self :item item :whole-item
        #'(lambda (item window)
            (send window :item item :first-part
                  #'(lambda (item window)
                      (format window "~r" (car item))))
            (format window " and ")
            (send window :item item :second-part
                  #'(lambda (item window)
                      (format window "~r" (cdr item)))))))

(defmethod (test-pane :who-line-documentation-string) ()
  (let ((superior (send self :superior)))
    (format nil "L: Turn left ~:[on~;off~]; ~
M: Turn whole item ~:[on~;off~]; R: Turn right ~:[on~;off~]"
            (send superior :left)
            (send superior :both)
            (send superior :right))))

(defmethod (test-pane :sensitive-type-p) (mouse-sensitive-item)
  (let ((superior (send self :superior)))
    (selectq (tv:displayed-item-type mouse-sensitive-item)
      (:first-part (send superior :left))
      (:whole-item (send superior :both))
      (:second-part (send superior :right)))))

(defflavor test-frame ((left t) (both t) (right t))
           (tv:select-mixin
            tv:process-mixin
            tv:bordered-constraint-frame-with-shared-io-buffer)
  :settable-instance-variables
  (:default-init-plist
   :panes
   '((display-pane tv:window-pane)
     (scroll-pane test-pane))
    :constraints
    '((only . ((scroll-pane display-pane)
               ((scroll-pane .4))
               ((display-pane :even)))))
    :selected-pane 'display-pane
    :configuration 'only
    :process '(main-loop)))
```

```
(defun main-loop (frame)
  (send frame :main-loop))

(defmethod (test-frame :main-loop) ()
  (let* ((scroll-pane (send self :get-pane 'scroll-pane))
         (display-pane (send self :get-pane 'display-pane))
         (terminal-io display-pane))
    (loop for i from 1 to 5
          do
      (loop for j from 10 to 50 by 10
            do
        (send scroll-pane :append-item (cons i j))))
    (error-restart-loop ((sys:abort error)
                         "Silly program top level")
      (let ((blip (send display-pane :list-tyi)))
        (format t "~&Blip received was: ~%  ~s" blip)
        (selectq (if (eq (first blip) :mouse-button)
                     (second blip)
                     (fourth blip))
          (#/mouse-l (setq left (not left)))
          (#/mouse-m (setq both (not both)))
          (#/mouse-r (setq right (not right))))))))

(defvar *test-frame*
        (tv:make-window 'test-frame
                        :expose-p t))
```

12.13.2.4 Flashy Scrolling in Text Scroll Windows

To scroll a display with the familiar *More above* and *More below* style scrolling, use **tv:margin-scrolling-with-flashy-scrolling-mixin.**

When this flavor is used, **tv:borders-mixin** should be included in the flavor definition before **tv:margin-scrolling-with-flashy-scrolling-mixin.** If it isn't, the *More above* and *More below* messages appear outside the borders.

If a label is required, **tv:top-box-label-mixin** should be placed after **tv:borders-mixin** and before **tv:margin-scrolling-with-flashy-scrolling-mixin** to put the label in the right place.

tv:margin-scrolling-with-flashy-scrolling-mixin *Flavor*

Provides *More above* and *More below* style window scrolling for a text scroll window.

:margin-scroll-regions *regions* (for **tv:margin-scroll-mixin**) *Init Option*

Allows you to specify the messages at the top and bottom of the display.

regions is a list of lists. Each list contains 4 elements:

- **:top** or **:bottom.**
- A string that displays at the end of the item list in the given direction.
- A string that displays when there are more items to display in the given direction.
- The character style that the string prints in.

The keyword **:top** is identical to the list:

```
(:top "Top" "More above" (:dutch :italic :small))
```

The Keyword **:bottom** is identical to the list:

```
(:bottom "Bottom" "More below" (:dutch :italic :small))
```

:flashy-scrolling-region *scrolling-region* (for **tv:flashy-scrolling-mixin)** *Init Option*

Specifies the area in which the mouse maintains its "flashy-scrolling" shape.

scrolling-region is a list of two lists. The first list specifies the scrolling region for the top of the window, and the second for the bottom of the window.

Each list contains three numbers. The first number is the height, in pixels, of the scrolling region. The other two numbers are percentages of the window width specifying the width of the scrolling region. The defaults are 50, 0.40, and 0.60.

Example of Flashy Scrolling in Text Scroll Windows

Alter the previous example (See the section "Example of Mouse-Sensitive Items in Text Scroll Windows", page 199.) as follows:

```
(defflavor test-pane ()
          (tv:borders-mixin
           tv:top-box-label-mixin
           tv:margin-scrolling-with-flashy-scrolling-mixin)
           tv:mouse-sensitive-text-scroll-window
           tv:pane-no-mouse-select-mixin
           tv:window)
  (:default-init-plist
   :sensitive-item-types :sensitive-type-p
   :margin-scroll-regions '(:top :bottom)))
```

12.14 Typeout Windows

tv:window-with-typeout-mixin *Flavor*

Flavor to mix into a superior window to provide an inferior typeout window.

:typeout-window (*flavor-name* . *options*) (for **tv:essential-window-with-typeout-mixin**) *Init Option*

Provides a typeout window inferior to the window. *flavor-name* is the flavor of typeout window to create; *options* are options to **tv:make-window**.

tv:typeout-window *Flavor*

Standard flavor of typeout window.

tv:typeout-window-with-mouse-sensitive-items *Flavor*

A typeout window with **tv:basic-mouse-sensitive-items** mixed in.

tv:temporary-typeout-window *Flavor*

A flavor of typeout window that saves and restores the bits of its superior. When **tv:with-terminal-io-on-typeout-window** is used with a window that has this kind of typeout window over it, the program does not have to take any action to restore the display when the typeout window goes away.

tv:with-terminal-io-on-typeout-window (*window wait-for-space-p*) &body *body* *Special Form*

Binds **zl:terminal-io** to the typeout-window of *window* over the duration of the body, taking care of exposing and deexposing the typeout window, selection, etc. *wait-for-space-p*, if supplied and not **nil**, means that after executing the body the user should be prompted to type a space to get rid of the typeout window. Otherwise the typeout window goes away as soon as the body returns. All values of the body are returned.

12.15 Scrolling Windows

tv:basic-scroll-bar *Flavor*

Flavor that provides basic scroll-bar scrolling.

tv:margin-scroll-mixin *Flavor*

Flavor that provides scrolling by clicking on margin regions.

tv:flashy-scrolling-mixin *Flavor*

Flavor that provides slow scrolling by moving the mouse through margin regions.

12.16 Frames

The concepts and facilities discussed in this section apply generally to Dynamic Window-based frames created with the Frame-Up Layout Designer and **dw:define-program-framework**. (For an overview of these facilities and references to additional documentation: See the section "Overview of Top-level Facilities for User Interface Programming" in *Programming the User Interface, Volume A*.) In particular, the subsections on specifying panes and constraints, specification examples, and frame messages are relevant:

See the section "Specifying Panes and Constraints", page 208.

See the section "Examples of Specifications of Panes and Constraints", page 215.

See the section "Messages to Frames", page 223.

A *frame* is a window that is divided into subwindows, using the hierarchical structure of the window system. The subwindows are called *panes*. The panes are the inferiors of the frame, and the frame is the superior of each pane. Several heavily used systems programs use frames. For example, Inspector windows are frames. The default Inspector window has six panes: the interaction pane on top, the history pane and command menu pane below it, and three Inspect panes below that. The Window Debugger and Zmacs also use frames. In Zmacs, each new editor window is a pane of the Zmacs Frame. Zmail uses 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 Debugger 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. For full details: See the section "Specifying Panes and Constraints", page 208. In general, a frame can have many different *configurations*. Each configuration is described in the constraint language, and each specifies one way of splitting up the frame. While the program is running, it can switch a frame from one configuration to another. Some panes may appear in more than one configuration, but other panes may be left out of one configuration, and may only be visible when the frame is switched to another configuration. For example, in Zmail, when you click on [Mail], the frame changes to a new configuration showing the Headers and Mail panes.

12.16.1 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 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.) 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**.

(For information on Dynamic Window-based frames and related facilities: See the section "Overview of Top-level Facilities for User Interface Programming" in *Programming the User Interface, Volume A.*)

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, the flavor **tv:window-pane**, used when you want a pane of a frame that understands everything that **tv:window** does, is defined as follows:

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

Among other things, **tv:pane-mixin** provides methods that let the pane participate in its superior's activity. The **:alias-for-selected-windows** method returns the superior's alias. When a window of this flavor receives a **:select** message, it first sends its superior an **:inferior-select** message. If the **:inferior-select** message returns **nil**, the **:select** message fails and just returns **nil**. When a window of this flavor receives a **:mouse-select** message, it passes the message on to its superior.

tv:pane-no-mouse-select-mixin *Flavor*

A mixin flavor to make a window a pane of a frame and ensure that it cannot be selected from a system menu. This flavor includes **tv:pane-mixin** and **tv:dont-select-with-mouse-mixin**.

tv:window-pane *Flavor*

An instantiable flavor that includes **tv:pane-mixin** and **tv:window**.

The flavor of the frame itself might be any of several flavors. The simplest flavor of constraint frame is **tv:constraint-frame**.

tv:basic-frame *Flavor*

This flavor provides methods that allow the frame to serve as the representative window of its activity. Usually a frame cannot become the selected window, but this flavor provides methods that handle messages about selection, typically by operating on the selected-pane instead of the frame. The **:select**, **:deselect**, and **:select-relative** methods just pass these messages on to the selected-pane when one exists; otherwise they return **nil**.

This flavor provides a handler for the **:select-pane** message that decides which pane should be selected when the activity is selected. The **:inferior-select** method saves the argument as the selected-pane and sends the message on to the frame's superior with the frame as argument. The **:name-for-selection** method returns the name-for-selection of the selected-pane if a selected-pane exists and has a name-for-selection; otherwise, the method returns the name of the frame.

tv:constraint-frame *Flavor*

This flavor is the basic kind of constraint frame. 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 or a label. It also has **tv:pop-up-notification-mixin** as a component.

tv:bordered-constraint-frame *Flavor*

This flavor is just **tv:constraint-frame** with **tv:borders-mixin** 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, as 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 as if 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 I/O buffer. 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 I/O 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 I/O buffer.

What happens when many windows share an I/O 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 I/O buffer, in the chronological order in which they are generated. The process then does successive **:tyi** stream operations from any pane of the frame, and it receives anything that has been typed at any pane. When the I/O 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 I/O 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 I/O buffer used by the frame itself. However, the frame does not have **tv:stream-mixin** as a component, and it does not handle **:any-tyi** and **:tyi** messages.

(**tv:constraint-frame-with-shared-io-buffer** is a component flavor of the Dynamic Window flavor **dw:program-frame**.)

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* (for **tv:constraint-frame-with-shared-io-buffer**) *Init Option*

If this option is present, *io-buffer* is used as the I/O buffer for the frame and the panes. Otherwise, a default I/O buffer is created.

12.16.2 Specifying Panes and Constraints

This section gives the complete rules for specifying the panes of a constraint frame and for 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 **:configurations** 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).

:panes *pane-descriptions* (for **tv:basic-constraint-frame**) *Init Option*

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.

:configurations *configuration-specification-list* (for **tv:basic-constraint-frame**) *Init Option*

The **:configurations** init option to a constraint frame controls the sizes and arrangement of the panes in each possible configuration of the frame. It is required for all flavors of constraint frames.

In earlier releases, equivalent information was required to be specified under the **:constraints** init option; it is still accepted for compatibility. See the section "Specifying Panes and Constraints Before Release 6.0", page 225. To convert a **:constraints** option to a **:configurations** option: See the function **tv:back-convert-constraints**, page 232.

The value of the **:configurations** init option is an alist that associates configuration names with configuration specifications. Each configuration specification consists of a list of layout specifications and a list of size specifications. Thus the skeleton of a typical **:configurations** argument to **tv:make-window** looks like:

```
:configurations '((main-configuration
                    (:layout spec spec...)
                    (:sizes spec spec...))
                  (alternate-configuration
                    (:layout spec spec...)
                    (:sizes spec spec...)))
```

The **:layout** and **:sizes** clauses may appear in either order.

A configuration arranges *entities* within the frame. Each entity has a name (a symbol). There are four kinds of entity:

pane	A window inferior to the frame.
row	A linear arrangement of entities, side by side. All the entities in a row are the same height.
column	A linear arrangement of entities, one above the other. All the entities in a column are the same width.
fill	An area that does not contain any windows, but is simply filled with some pattern.

The entities in a row can be panes, fills, or columns. The entities in a column can be panes, fills, or rows. Rows and columns are collectively referred to as *stacks*. The subentities of a stack are referred to as the *members* of the stack. Different types of members can be mixed.

Configuration specifications have certain restrictions. All names used in a configuration specification must be defined as entities exactly once within that specification. Each entity must be used as a member of a stack exactly once, except for the entity with the same name as the configuration, which must not be a member of any stack. No stack can contain itself, directly or indirectly.

12.16.2.1 :layout Constraint Frame Specification

A configuration is itself a stack. Thus, the symbol that names a configuration must appear in that configuration's **:layout** list as the name of either a row or a column.

A configuration specification includes a list of layout specifications, introduced by the keyword **:layout**. Each layout specification defines one row, column, or fill. (The panes are defined by the **:panes** init option to the frame. See the init option **(flavor:method :panes tv:basic-constraint-frame)**, page 208.)

A layout specification for a *row* takes the following form:

(*name* `:row` *name1 name2*...)

name is a symbol, the name of the row. *name1*, *name2*, and so on are symbols, the names of the members of the row. The members are listed in left-to-right order.

A layout specification for a *column* takes the following form:

(*name* `:column` *name1 name2*...)

name is a symbol, the name of the column. *name1*, *name2*, and so on are symbols, the names of the members of the column. The members are listed in top-to-bottom order.

A layout specification for a *fill* takes one of the following forms. In each of these *name* is a symbol, the name of the fill.

(*name* :fill :white)
The area is filled with zero pixels (normally displayed as white).

(*name* :fill :black)
The area is filled with one pixels (normally displayed as black).

(*name* :fill *array*) The area is filled with the contents of the array, using **bitblt**. You probably want to use backquote (‘) to create the configuration description and insert the array at the appropriate point.

(*name* :fill *symbol*)
The symbol should be the name of a function of six arguments. The function is expected to fill 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. You should not need to do anything with this argument.

x-position
: X-coordinate of the top left corner of the rectangle to be filled.

y-position
: Y-coordinate of the top left corner of the rectangle to be filled.

width Width in pixels of the rectangle to be filled.

height Height in pixels of the rectangle to be filled.

screen-array
: This is a two-dimensional array into which the function should write the pattern it wants to put into the window.

(*name* **:fill** *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.

12.16.2.2 :sizes Constraint Frame Specification

A configuration specification includes a list of size specifications, introduced by the keyword **:sizes**. Each size specification defines how a stack is divided up among its members; it controls the width of each member of a row, or the height of each member of a column. No size specification exists for fills and panes.

A size specification is a list whose first element is the name of the relevant stack. The remaining elements consist of groups of *constraints* separated by the keyword **:then**. The groups are processed sequentially; all the constraints in a group are processed in parallel. Each constraint allocates some of the space available in a stack to a single member of that stack. (This space is width if the stack is a row, height if the stack is a column). After one group has been processed, the amount of space available is decreased by the sum of the space that was allocated, and then the next group is processed. This is the meaning of the parallel versus sequential distinction.

The division of constraints into groups matters when a constraint specifies the size of a member as some fraction of the space available. For example, suppose two constraints each specify that a member is to receive 50% of the available space. If these two constraints are in the same group (processed in parallel) they will allocate 100% of the space. If they are in separate groups (processed sequentially) they will allocate 75% of the space, and the first member will be twice as large as the second member. The first member gets 50% of the total space, then the second member gets 50% of what remains, which is 25% of the total space.

Note that the order of the constraints in a size specification is unrelated to the actual order of the members on the screen, which is controlled solely by the layout specification.

A constraint can take any of several forms. In each case the constraint is a list whose first element is the name of the member (a symbol).

(*name integer*)
: *integer* is the number of pixels to allocate.

(*name integer units*)
: *integer* is the number of characters of width or lines of height to allocate. *units* must be **:lines** or **:characters**. This form is illegal if *name* is not the name of a pane, since only panes have lines and characters. Use the following form if *name* is a stack or a fill.

(*name integer units pane*)
: *integer* is the number of characters of width or lines of height to allocate. *units* must be **:lines** or **:characters**. *pane* is the name of a pane that defines the units. Typically *name* is a stack and *pane* is a member, directly or indirectly, of the stack.

(*name fraction*)
: *fraction*, a floating-point number between 0.0 and 1.0, is the proportion of the available space to allocate.

(*name fraction units*)
: *fraction*, a floating-point number between 0.0 and 1.0, is the proportion of the available space to allocate. The allocation is rounded down to an integral number of lines or characters. *units* must be **:lines** or **:characters**.

(*name fraction units pane*)
: *fraction*, a floating-point number between 0.0 and 1.0, is the proportion of the available space to allocate. The allocation is rounded down to an integral number of lines or characters. *units* must be **:lines** or **:characters**. *pane* is the name of a pane that defines the units.

(*name* **:even**)
: The space available is divided evenly among all the constraints in the group. If any constraint in a group uses **:even**, every constraint in that group must use *:even*. Such a group must be the last group in a size specification. If the space available is not exactly divisible by the number

of constraints in the group, the division is slightly uneven so that exactly all of the space will be used, leaving no unsightly gaps or overlaps.

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.

(*name* **:ask** *message-name arg-1 arg-2 ...*)
This constraint lets you ask the window how much space it would like to take up.

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*, and so on, is sent to the pane; its answer says how much space the pane should take up. Note that *arg-1*, and so on, are not forms: They are the values of the arguments themselves (that is, 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** constraint except that the *constraint-node* is not passed. 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 not documented here.

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-pane-size** (also with no arguments), which makes the window take up enough room to be square.

(*name* **:ask-window** *pane-name message-name arg-1 arg-2 ...*)
This constraint is similar to **:ask**, except that the message is sent to the

pane named *pane-name* instead of the pane being described. Like **:ask**, the arguments that are actually sent along with the message are the same as the arguments passed when you use the **:funcall** constraint except that the *constraint-node* is not passed.

This constraint is primarily used for stacks, when the size of a stack should be controlled by the needs of a pane inside it.

(*name* **:funcall** *function arg-1 arg-2 ...*)
This constraint lets you supply a function to be called, which should compute the amount of space to use. The **:funcall** constraint is rarely used and you probably don't need to worry about it. 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*, and so on, values. The six arguments are:

constraint-node
This is an internal data structure. You should not need to do anything with this argument.

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.

(*name* **:eval** *form*)
This is like **:funcall**, but instead of providing a function and arguments, you provide a form. The **:eval** constraint is rarely used and you probably don't need to worry about it. 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****

(*name* **:limit** (*min max*) *rest-of-the-constraint...*)
Any constraint may, optionally, be preceded by a **:limit** clause. 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 integers giving the minimum and maximum values in pixels.

(*name* **:limit** (*min max units*) *rest-of-the-constraint...*)
If the list has a third element, it should be one of the symbols **:lines** or **:characters**, and it means that the integers are in units of lines or characters, computed by multiplying by the line-height or char-width of the pane.

(*name* **:limit** (*min max units pane*) *rest-of-the-constraint...*)
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 stack instead of a pane, and the third element of the list is present, then the fourth must be present as well, since stacks do not have their own line-height or char-width.)

For example, to make a pane called *interactor* the same height as a pane *menu* as long as that size is between 10 and 20 lines, you might use

```
(interactor :limit (10 20 :lines)
            :ask-window menu :pane-size)
```

12.16.3 Examples of Specifications of Panes and Constraints

The first set of examples below, 1-6, is meant to give you a feel for the basics of constraint-frame specification. These are followed by two, more complex examples. Additional examples can be found in the files sys:examples;constraint-frame-language-1.lisp, -2.lisp, and -3.lisp. Also, note that for Dynamic Window-based frames you can use the Frame-Up Layout Designer to help with the initial specification of a variety of layouts. (See the section "Frame-Up Layout Designer" in *Programming the User Interface, Volume A.*)

Example 1

```
;;; Two windows of equal size, one on top of the other.

(defflavor cframe1 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane
                    :blinker-p nil
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :label "Pane-2 label"))
   :configurations
         '((config1 (:layout (config1 :column pane-1 pane-2))
                    (:sizes  (config1 (pane-1 :even) (pane-2 :even)))))
   :configuration 'config1))
```

Example 2

```
;;; Two windows of equal size, side by side

(defflavor cframe2 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane
                    :blinker-p nil
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :label "Pane-2 label"))
   :configurations
         '((config1 (:layout (config1 :row pane-1 pane-2))
                    (:sizes  (config1 (pane-1 :even) (pane-2 :even)))))
   :configuration 'config1))
```

Example 3

```
;;; Here we have created a constraint frame with two
;;; possible configurations.  You can switch between these
;;; configurations at run time.
```

```
(defflavor cframe3 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-2 label"))
   :configurations
          '((config1 (:layout (config1 :column pane-1 pane-2))
                     (:sizes  (config1 (pane-1 :even) (pane-2 :even))))
            (config2 (:layout (config2 :row pane-1 pane-2))
                     (:sizes  (config2 (pane-1 :even) (pane-2 :even)))))
   :configuration 'config1))


;;; Before going on with more complex constraint frames,
;;; you have to know how to access the various panes of a
;;; constraint frame and how to tell the window to change
;;; to a different configuration.  Notice what happens to
;;; the circle drawn in Pane-2.

(defvar *win* (tv:make-window 'cframe3))

(defun one ()
  (let ((first-pane (send *win* :get-pane 'pane-1))
        (second-pane (send *win* :get-pane 'pane-2)))
    (send *win* :set-configuration 'config1)
    (send *win* :expose)
    (send *win* :send-all-panes :clear-window)
    (send first-pane :draw-circle 500 100 20)
    (send second-pane :draw-circle 600 100 20)
    (sleep 2)
    (send *win* :set-configuration 'config2)))
```

Example 4

```
;;; Now lets try organizing the panes in interesting patterns.
;;; For this we will need additional panes.  Also notice in
;;; config3 that you don't always have to use all the panes
;;; defined.

(defflavor cframe4 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-2 label")
            (pane-3 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-3 label")
            (pane-4 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-4 label"))
   :configurations
         '((config1 (:layout (config1 :column pane-1 row-panes)
                             (row-panes :row pane-2 pane-3 pane-4))
                    (:sizes  (row-panes (pane-2 :even)
                               (pane-3 :even) (pane-4 :even))
                             (config1 (pane-1 :even)
                                      (row-panes :even))))
            (config2 (:layout (config2 :row pane-1 column-panes)
                             (column-panes :column pane-2 pane-3 pane-4))
                    (:sizes  (column-panes (pane-2 :even)
                               (pane-3 :even) (pane-4 :even))
                             (config2 (pane-1 :even)
                                      (column-panes :even))))
            (config3 (:layout (config3 :row pane-1 pane-2 pane-4))
                    (:sizes  (config3 (pane-1 :even)
                               (pane-2 :even) (pane-4 :even)))))
   :configuration 'config1))
```

```
;
;;; Display all configurations of any constraint frame that is
;;; the value of *WIN*.  This function is useful for Examples 4,
;;; 5, and 6.
;

(defvar *win* (tv:make-window 'cframe4))

(defun display-all-configs ()
  (let ((configurations (loop for thing in (send *win* :constraints)
                              collect (car thing))))
    (loop for config in configurations
          do
      (print config)
      (sleep 2)
      (send *win* :set-configuration config)
      (send *win* :expose)
      (sleep 3)
      (send *win* :bury))))
```

Example 5

```
;;; Now we turn our attention to controlling the sizes of
;;; the panes.  In place of :EVEN, if you put a integer it
;;; will allocate that many pixels.  If you put a fraction
;;; 0.2 the pane will be allocated that percent of the room
;;; remaining to be allocated.  A size value of 10 :lines
;;; or 20 :characters will create a pane large enough to hold
;;; them.
```

```
(defflavor cframe5 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-2 label")
            (pane-3 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-3 label")
            (pane-4 tv:window-pane
                    :blinker-p nil
                    :save-bits t
                    :label "Pane-4 label"))
   :configurations
          '((config1 (:layout (config1 :row pane-1 pane-2 pane-3))
                     (:sizes  (config1 (pane-1 200)
                                       :then (pane-2 200)
                                       :then (pane-3 :even))))

            (config2 (:layout (config2 :row pane-1 pane-2 pane-3))
                     (:sizes  (config2 (pane-1 0.5)
                                       :then (pane-2 0.5)
                                       :then (pane-3 :even))))

            (config3 (:layout (config3 :row pane-1 pane-2 pane-3))
                     (:sizes  (config3 (pane-1 50 :characters)
                                       :then (pane-2 50 :characters)
                                       :then (pane-3 :even))))
            (config4 (:layout (config4 :column pane-1 pane-2 pane-3 pane-4))
                     (:sizes  (config4 (pane-1 200)
                                       :then (pane-2 0.5)
                                       :then (pane-3 5 :lines)
                                       :then (pane-4 :even)))))
   :configuration 'config1))
```

Example 6

```
;;; You might want to try reshaping the window by clicking on
;;; the system menu choice "Edit Screen" and then on "Reshape".
;;; You should notice two things:
;;;   1) The panes adjust themselves to fit into the space given.
;;;   2) If you make the window small enough the panes will
;;;      become uselessly small.
;;;
;;; To give your constraint frame size limits try the following:

(defflavor cframe6 ()
           (tv:bordered-constraint-frame)
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane
                    :blinker-p nil
                    :label "Pane-1 label")
            (pane-2 tv:window-pane
                    :blinker-p nil
                    :label "Pane-2 label"))
   :configurations
          '((config1 (:layout (config1 :column pane-1 pane-2))
                     (:sizes  (config1 (pane-1 :limit (5 10 :lines) :even) (pane-2 :eve
   :configuration 'config1))
```

Following are two examples of configuration definitions, slightly edited from the system source code.

Example 7

```
;;;Here is how the Font Editor (FED) specifies its
;;;standard configuration.  This code is extracted from a
;;;source file with package zl:fed and base 8.
```

```
(defmethod (fed :before :init) (init-plist)
  ...
  (setf (get init-plist :configurations)
        '((:standard
           (:layout
            (:standard :column character-pane prompt-pane top-section)
            (top-section :row fed-pane other-slab)
            (other-slab :column
                        draw-mode-menu
                        command-menu-1
                        command-menu-2
                        command-menu-3
                        status-pane
                        alphabet-menu
                        param-chvv
                        register-pane))
           (:sizes
            (other-slab (draw-mode-menu :ask :pane-size)
                        :then (command-menu-1 :ask :pane-size)
                        :then (command-menu-2 :ask :pane-size)
                        :then (command-menu-3 :ask :pane-size)
                        :then (status-pane 3 :lines)
                        :then (alphabet-menu :ask :pane-size)
                        :then (param-chvv 5 :lines)
                        :then (register-pane :even))
            (top-section (other-slab :limit (24 144 :characters prompt-pane)
                                     0.3)
                         :then (fed-pane :even))
            (:standard
             (character-pane :ask :wanted-size)
             :then (prompt-pane 4 :lines)
             :then (top-section :even))))
          (:wide ...))))
```

Example 8

```
;;;Here is how an early implementation of the
;;;Document Examiner specified its frame configuration.
;;;This code is extracted from a source file with package
;;;sage and base 10.
```

```
(defconst *dex-frame-constraints*
          '((main
             (:layout
              (main :column top-part bottom-part)
              (top-part :row title&viewer-pane candidates-and-bookmarks)
              (bottom-part :row command-pane menu-pane)
              (title&viewer-pane :column title-pane viewer-pane)
              (candidates-and-bookmarks :column candidate-pane bookmark-pane))
             (:sizes
              (main (bottom-part 4 :lines command-pane)
                    :then (top-part :even))
              (bottom-part (command-pane 660)
                           :then (menu-pane :even))
              (top-part (title&viewer-pane 660)
                        :then (candidates-and-bookmarks :even))
              (title&viewer-pane (title-pane 0 :lines)  ;label only
                                 :then (viewer-pane :even))
              (candidates-and-bookmarks (candidate-pane 0.5)
                                        :then (bookmark-pane :even))))))

(defmethod (dex-frame :before :init) (plist)
  (unless (variable-boundp tv:panes)
    (setq tv:panes *dex-frame-panes*))
  (unless (get plist :configurations)
    (setf (get plist :configurations) *dex-frame-constraints*))
  ...)
```

12.16.4 Messages to Frames

:select-pane *pane* *Message*

The **:select-pane** message to a frame makes *pane* the selected-pane of the frame. *pane* must be either an exposed inferior of the frame or **nil**, which means to set the selected-pane to **nil**. This message also deselects the current selected-pane if it is a window different from *pane*. Unless *pane* is **nil**, this message sends *pane* a **:select-relative** message.

:selected-pane *Message*

The **:selected-pane** message to a frame returns the selected-pane of the frame. This message is sent by users and received by the system.

:selected-pane *pane* (for **tv:basic-constraint-frame**) *Init Option*
Makes *pane* the selected-pane of this frame. *pane* can be the symbol used in the **:panes** init option to name the pane.

:get-pane *pane-name* of **tv:basic-constraint-frame** *Method*
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* of **tv:basic-constraint-frame** *Method*
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* of **tv:basic-constraint-frame** *Method*
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* of **tv:basic-constraint-frame** *Method*
Send the specified *message* with the specified *arguments* to all of the panes of this frame, including the nonexposed ones.

:send-all-exposed-panes *message* &rest *arguments* of **tv:basic-constraint-frame** *Method*
Send the specified *message* with the specified *arguments* to all of the exposed panes of this frame.

:configuration *configuration-name* (for **tv:basic-constraint-frame**) *Init Option*
Make the initial configuration of the frame be the one named by the symbol *configuration-name.*

:configuration of **tv:basic-constraint-frame** *Method*
Return the symbol naming the current configuration of the frame.

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

:constraints of **tv:basic-constraint-frame** *Method*
Returns the configuration description list of the frame.

For information on select menus and frames: See the message :name-for-selection, page 107.

12.16.5 Specifying Panes and Constraints Before Release 6.0

This section gives the complete rules for specifying the panes of a constraint frame, and for the constraint language, in releases before 6.0. The specification method described in this section is obsolete but supported in Genera 7.0 for compatibility.

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).

:panes *pane-descriptions* (for **tv:basic-constraint-frame**) *Init Option*

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* (for **tv:basic-constraint-frame**) *Init Option*

This initialization option was required for all flavors of constraint frames before Release 6.0. It has been replaced by the **:configurations** init option. See the init option **(flavor:method :configurations** tv:basic-constraint-frame), page 209. To convert a **:constraints** option to a **:configurations** option: See the function **tv:back-convert-constraints**, page 232.

The argument, *configuration-description-list*, is a list of configuration descriptions. For the format of configuration descriptions: See the section "Specifying Panes and Constraints Before Release 6.0", page 225.

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 an integer, a flonum, or one of various keyword symbols. Each type of constraint may take arguments, whose meaning depends on which kind of constraint this argument is passed to.

While descriptions of panes do not have the same format as descriptions of dummies, the same kind of constraints are used in both of them. So all the formats given below may be used inside the descriptions of either panes or dummies.

Any constraint may, optionally, be preceded by a **:limit** clause. If a constraint has a **:limit** clause, the constraint looks like:

(`:limit` *limit-specification key arg-1 arg-2* ...)

The **:limit** clause lets you set a minimum and a maximum value that will be applied to the size computed by the constraint. If the constraint returns a value smaller than the minimum, then the minimum value will be used; if it returns a value larger than the maximum, then the maximum value will be used. The *limit-specification* is normally a two-element list, whose elements are integers 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 integers are in units of lines or characters, computed by multiplying by the line-height or char-width of the pane. 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 or 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.

integer This lets you specify the absolute size. The value computed by the constraint is simply this integer. Optionally, an argument may be given: it may be the symbol **:lines** or the symbol **:characters**, meaning that the integer 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 or 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 an integer.

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*, and so on, is sent to the pane; its answer says how much space the pane should take up. Note that *arg-1*, and so on, are not forms: They are the values of the arguments themselves (that is, 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-pane-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 should 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*, and so on, 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.

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. You must use only the opposite kind of stacking from the kind currently happening; that is, successive levels of configuration-description must use alternating kinds of stacking. 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.

tv:back-convert-constraints *constraints* *Function*

Converts a list used as the **:constraints** init option for **tv:basic-constraint-frame** to a list suitable for the **:configurations** option. The **:configurations** option replaced the **:constraints** option in Release 6.0.

The function returns two values: a list suitable for use as the argument to the **:configurations** option, and a list of symbols naming the panes encountered in the list.

Example:

```
(tv:back-convert-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))))))
```

```
=> ((first-config (:layout
                   (first-config :column top-strip main-pane)
                   (top-strip :row huey dewey louie))
                  (:sizes
                   (top-strip (huey :even) (dewey :even) (louie :even))
                   (first-config (top-strip 0.3)
                                 :then (main-pane :even))))
    (second-config (:layout
                    (second-config :column main-pane bottom-strip)
                    (bottom-strip :row random-pane menu))
                   (:sizes
                    (bottom-strip (menu :ask :pane-size)
                                  :then (random-pane :even))
                    (second-config (bottom-strip 0.2)
                                   :then (main-pane :even)))))

   (random-pane menu main-pane louie dewey huey)
```

12.16.6 Examples of Specifications of Panes and Constraints Before Release 6.0

This section gives some examples of specifications of panes and constraints in the constraint language used before Release 6.0. The full description of how to use constraint frames, including the full constraint language, is rather complicated. For complete specifications of the pre-Release 6.0 language: See the section "Specifying Panes and Constraints Before Release 6.0", page 225.

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. 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))
```

For an explanation of how to use menus: See the section "Window System Choice Facilities", page 239.

Following is the last example, using the **:configurations** init option instead of the **:constraints** option used before Release 6.0:

```
(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")))
  ':configurations
    '((first-config (:layout
                      (first-config :column top-strip main-pane)
                      (top-strip :row huey dewey louie))
                     (:sizes
                      (top-strip (huey :even) (dewey :even) (louie :even))
                      (first-config (top-strip 0.3)
                                    :then (main-pane :even))))
      (second-config (:layout
                       (second-config :column main-pane bottom-strip)
                       (bottom-strip :row random-pane menu))
                      (:sizes
                       (bottom-strip (menu :ask :pane-size)
                                     :then (random-pane :even))
                       (second-config (bottom-strip 0.2)
                                      :then (main-pane :even))))))
```

For a description of the constraint language used in Release 6.0: See the section "Specifying Panes and Constraints", page 208.

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**.

PART III.

Window System Choice Facilities

13. The Choice Facilities

The window system for the Lisp Machine contains a variety of facilities to allow the user to make choices interactively. These all work by displaying some arrangement of items in a window. By pointing to an item with the mouse and pressing a mouse button, the user selects the item. The choice facilities are implemented in and accessed with the Flavors feature of Lisp.

13.1 Overview of the Choice Facilities

This section is a capsule description of the choice facilities. This should familiarize you with the possibilities, thereby helping you to decide which facility is appropriate to your application, without reading through each detailed description. (For an overview of choice facilities intended for use with Dynamic Windows: See the section "Overview of Facilities for Accepting Single Objects" in *Programming the User Interface.*)

13.1.1 List of Choice Facilities

Here is a brief explanation of each of the choice facilities.

Pop-up Menus

This facility puts a menu with items on the screen. The user is forced to make a choice among the items. (The menu does not disappear until a choice has been made.) See the section "Instantiable Pop-up and Momentary Menus", page 263.

Momentary Menus

Momentary menus appear on the screen with a list of choices. The user does not have to make a choice, however. By moving the mouse outside of the menu, the user can make the menu disappear. See the section "Basic and Mixin Pop-up and Momentary Menus", page 262.

Command Menus

Command menus are used when you want to pass a command to your own controlling process from a menu. The command is sent to the process via an input buffer that can be shared with other windows or processes. This way, the controlling process can be looking in the buffer for commands from several windows as well as for keyboard input. See the section "Command Menus", page 271.

Dynamic Item List Menus

A dynamic item list menu is provided for menus whose items change over time. The item list is updated whenever the menu is displayed. Both momentary and pop-up dynamic item list menus are available. See the section "Dynamic Item List Menus", page 277.

Multiple Menus

Multiple menus are provided for situations in which the user can select *several* items at a time. The selected items are displayed in inverse video. *Special choices* allow the user to specify operations on all the items selected. Both momentary and pop-up multiple menus are available. See the section "Multiple Menus", page 283.

Multiple Menu Choose Menus

This facility provides for menus with several columns. The user picks one item from each column. Special choices [Do It] and [Abort] are used to execute the choices and and deactivate the menu, respectively. See the section "The Multiple Menu Choose Facility", page 289.

Multiple Choice Menus

This facility displays a menu in which each item is displayed on a separate line. Each item is associated with several yes/no choices, in *choice boxes.* By pointing to a box and pressing the left mouse button, the user complements the yes/no state of the choice box for that item. Constraints can be imposed among the choices for an item, ensuring, for example, that if one box is selected, the others are automatically deselected. See the section "The Multiple Choice Facility", page 293.

Choose Variable Values

Each item is associated with a value printed next to it. Many different types of values can be specified, or the programmer can create new types. In operation, users select items and then alter the values associated with the item. See the section "The Choose Variable Values Facility", page 299.

User Options

The user option facility is based on the choose variable values facility. It is used to keep track of options to a program of the sort that users would want to specify once and then save. The option list can be associated with particular programs. See the section "The User Option Facility", page 309.

Mouse-sensitive Items and Areas

Mouse-sensitive behavior underlies all of the choice facilities. This mixin facility lets areas of the screen be sensitive to the mouse. Moving the mouse into such an area causes a box to be drawn around it. At this point, clicking the mouse invokes a user-defined operation. See the section "The Mouse-Sensitive Items Facility", page 323.

Margin Choices

Windows can be augmented with choice boxes in their margins. Choice boxes give the user a few mouse-sensitive points that are independent of anything else in the window. Margin choices can be added to any flavor of window in a modular fashion. See the section "The Margin Choice Facility", page 333.

13.2 Standard and Customizable Facilities

From the programmer's viewpoint, there are two ways of invoking the choice facilities.

- *Standard* facilities are provided with a reasonable set of defaults predefined in the system code. They are invoked with a simple function call.

- *Customizable* facilities require you to provide more specifications, but they allow more flexibility in the layout and behavior of the facilities. Customizable facilities are manipulated by the Flavor system, and include instantiable, basic, and mixin flavors.

Many of the documented choice facilities are provided in both standard and customizable forms.

13.3 Choice Facilities Use the Flavor System

The window system and the choice facilities are implemented using the Flavor system in Lisp. When a menu is instantiated, users communicate with it by pressing mouse buttons (sometimes called "mouse-clicking"), or by typing in values. Internally, programs communicate with a menu by sending it a message using the **send** function of Lisp.

Useful *initialization property-list options* (hereafter called *init-plist options*) and *messages* associated with each flavor are specified in this document.

13.3.1 Combining Choice Facilities

Since the choice facilities are implemented with the Flavor system, many of the behaviors listed previously can be integrated into one menu by means of flavor combination.

For example, one menu might include both of these features:

- Pop-up behavior, meaning that the window does not disappear until a choice has been made.
- Multiple menu behavior, allowing several menu items to be selected.

13.3.2 Instantiable, Basic, and Mixin Flavors

Each choice facility is based on either an *instantiable,* a *basic,* or a *mixin* flavor. Even the standard choice facilities (invoked by simple Lisp function calls) are based on these flavors.

Instantiable flavors are self-contained objects that are ready to be invoked. Instantiable facilities are built out of the basic and mixin facilities. An example of an instantiable facility is the **tv:momentary-menu** flavor.

Basic flavors (often denoted by the prefix **basic-** in the code) define a whole family of related flavors. Most of the basic flavors are noninstantiable and merely serve as a base on which to build other flavors. An example of a noninstantiable basic facility is the **tv:basic-mouse-sensitive-items** flavor.

Mixin flavors (often denoted by the suffix **-mixin** in the code) define a particular feature of an object. A mixin flavor cannot be instantiated, because it is not a complete object. An example of a mixin flavor is **tv:dynamic-multicolumn-mixin**.

In the descriptions of the different choice facilities that follow, the instantiable flavors will be discussed first, followed by the basic and mixin flavors.

13.3.3 Modifying the Choice Facilities

Although this document explains how to combine the features of the different choice facilities to suit different applications, it does not tell you how to modify the facilities provided with the system, except in the simplest of ways. In order to change the basic behavior of the choice facilities you will need to read some of the code that implements the facility in question. (For example, you should study window instance variables and internal messages that you might want to put daemons on or redefine.)

13.4 The User's Process and the Mouse Process

An asynchronous process called the *mouse process* handles interaction with the mouse. Some portions of these choice facilities execute in the process that calls them, while other portions execute in the mouse process. For example, when menu items are displayed on the screen and the mouse points to them, a box is drawn around the items. This drawing is performed by the mouse process.

This document does not attempt to explain the details of how the mouse and the window system interact. Indeed, the choice facilities are supposed to shield the user from such details, and they can be used effectively with no knowledge of how they are implemented internally.

However, the cases in which a portion of a facility runs in the mouse process are noted where they occur in this text. Excepting these cases, you can freely use side-effects (both special variables and **throw**), and not worry about errors in your program corrupting the system.

The choice facilities described in this document respond to messages sent by the mouse process. For example, **:mouse-buttons**, **:mouse-click**, and **:mouse-select** are all handled by any flavor built on **tv:menu**.

14. Introduction to the Menu Facilities

From the user's point of view, a *menu* is a group of choices, each identified by a word or short phrase. To see an example of a menu, click the right mouse button while in a Lisp Listener; this should cause the System menu to appear (Figure 1).

Windows	*This window*	*Programs*
Create	Attributes	Lisp
Select	Refresh	Edit
Split Screen	Bury	Inspect
Layouts	Kill	Mail
Edit Screen	Reset	Font Edit
Set Mouse Screen	Arrest	Trace
	Un-Arrest	Emergency Break
		Flavor Examiner
		Hardcopy
		File System

Figure 1. System menu.

You can select one of the choices by moving the mouse near it, which causes it to be highlighted (a box appears around it), and then clicking any mouse button. What happens when you select one of the choices depends on the particular type of menu. Typically the choices in a menu might be commands to some program or choices on which a command should operate.

The window system software automatically chooses the arrangement of the choices and the size and shape of the window. Naturally, there are ways for programmers to control these parameters if they desire. See the section "Init-plist Options For **tv:menu**", page 341.

The inverse-video *mouse documentation line* is provided near the bottom of the screen in order to convey the meaning of the mouse buttons at a given time. For example, in the System menu, with the mouse positioned over the "Create" item, the mouse documentation line normally displays the following text:

```
Create a new window.  Flavor of window selected from a menu.
```

The abbreviations L, M, and R stand for the left, middle, and right mouse buttons, respectively. The numeral 2 indicates a quick double click of the mouse button. (Note that the "double-click" effect can also be obtained by clicking once on the mouse while holding down the `SHIFT` key.)

14.1 Components of a Menu

It is important to understand the terminology for describing a menu. The components of a menu are shown in Figure 2.

14.2 Menu Items

From the viewpoint of the programmer, a menu has a list of *items;* each item represents one of the displayed choices. The user chooses an item, and then the program executes it.

An item, then, has three parts:

- A representation in the item list
- A displayed representation
- A specified action when it is executed; this can include a value (or values) to return as well as side-effects

14.3 The Form of a Menu Item

Generally speaking, a menu item can take any of several forms, noted in the list below. In practice, you find these forms in the specification of particular item types, described in the next section.

a string or a symbol
: The string or symbol is both what is displayed and what is returned. There are no side-effects when the item is executed. (Note: **nil** is not a valid menu item.)

a cons
: This is like an alist entry. The **car** is a string or symbol to display and the **cdr** is what to return. The **cdr** must be atomic to distinguish this case from the remaining ones. There are no side-effects.

a list (*name value*)
: Another form of alist entry. *name* is a string or a symbol to display, and *value* is any arbitrary object to return. There are no side-effects when the item is executed.

a list (*name type arg option1 arg1 option2 arg2...*)
: This is the "general list" form, described in more detail below. *name* is a string or a symbol to display. *type* is a keyword symbol specifying what to

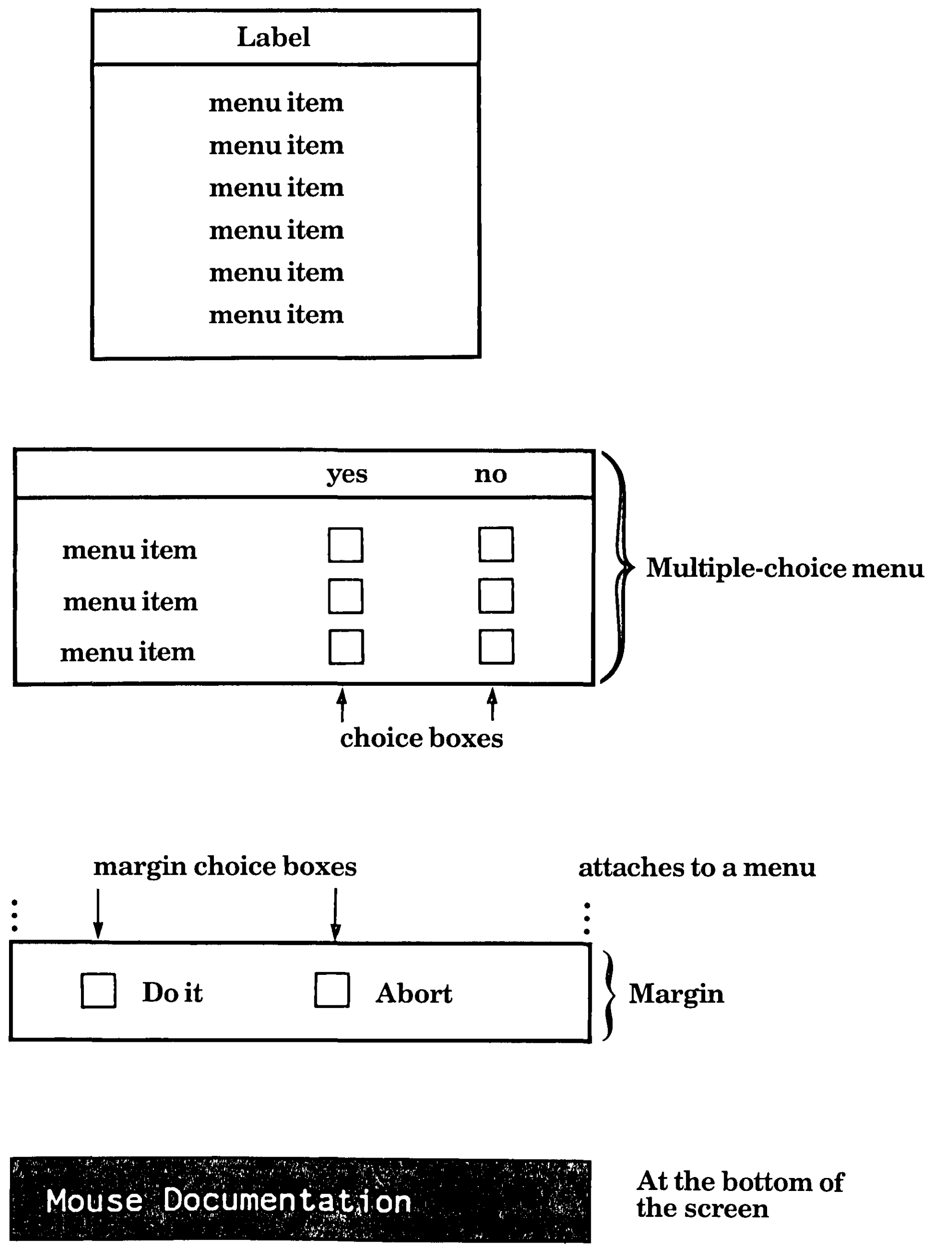


Figure 2. Components of a menu.

do when the item is executed, and *arg* is an argument to it. The *options* are keyword symbols specifying additional features desired, and the *args* following them are arguments to those options.

14.3.1 Types of Menu Items

Each menu item is an instance of a particular *type*. In most menus, you may not want to explicitly specify the type of the menu item. This is because in simple menus all the menu items are of the same type. Your code (which processes the selected items) presumably knows this type.

It is possible to specify the type of the menu items, however. This provides another dimension of flexibility in menu design. Since items of different types can be intermingled in a single menu, selecting different items can generate a variety of interesting effects. For example, some items can return a value, while others can generate new menus or perform other computations.

14.3.2 The "General List" Form of Item

To specify the type of an item, use the "general list" form of item.

(name type arg option1 arg1 option2 arg2 ...)

As described, an *arg* (argument) field follows each type specification. The predefined types of menu items and the meaning of their arguments are listed here.

:value *arg* is what to return when the item is executed. There are no side-effects.

:eval *arg* is a form to be evaluated in null environment. Its value is returned.

:funcall
arg is a function of no arguments to be called. Its value is returned.

:funcall-with-self
Like the **:funcall** item type, **:funcall-with-self** calls a function. However, the specified function is called with one argument: **self**, that is, the menu itself.

An example demonstrates its use:

```
;;; Specify the item list
...
;;; Specify the :funcall-with-self item
("Option 1" :funcall-with-self do-option-1)
...
(defun do-option-1 (menu)
  ;; send the :option-1 message
  (send menu ':option-1))
```

:no-select
This item cannot be selected. Moving the mouse near it does *not* cause it to be highlighted. This is useful for putting comments, headings, and blank spaces into menus. *arg* is ignored, but it must be present for syntactic consistency.

:kbd *arg* is sent to the selected window via the **:force-kbd-input** message. Typically it is either a character code that is to be treated as if it were typed in from the keyboard, or a list that is a command to the program. It is almost always preferable to use a command menu rather than **:kbd** menu items. See the section "Command Menus", page 271.

:menu *arg* is a new menu to choose from; it is sent a **:choose** message and the result is returned. Normally *arg* would be a momentary menu. If *arg* is a symbol it gets evaluated.

:buttons
arg is a list of three menu items. The item actually chosen (that is, the item to be executed) is one of these three, depending on which mouse button was clicked. The order in the list is (*left middle right*).

z.B. :buttons ("links" "mitte" "rechts")

:window-op
arg is a function of one argument. The argument is a list of three elements: the window the mouse was in before this menu was exposed and the X and Y coordinates of the mouse at that time. For a description of the **tv:window-hacking-menu-mixin**: See the section "Basic and Mixin Pop-up and Momentary Menus", page 262. This type is not useful unless the **tv:window-hacking-menu-mixin** is present in the window flavor.

14.3.3 Menu Item Options

Menu item options follow the arguments in the "general list" form of item. They have two purposes:

- Specifying the character style of a menu item
- Specifying the *mouse line documentation string* associated with an item

The menu item option keywords are as follows:

:character-style
This keyword is followed by a character style specification. The item is displayed in that character style, merged against the menu's default character style.

The **:character-style** option is for use with static-window-based menu facilities. For **dw:menu-choose** menu items and the **alist-member** presentation type, use the **:style** option instead.

:style This keyword is followed by a character style specification. The item is displayed in that character style, merged against the menu's default character style.

The **:style** option is for use with Dynamic Window-based menu facilities, **dw:menu-choose** and **alist-member** in particular. For static-window-based facilities, use the **:character-style** option instead.

:documentation
This keyword is followed by a string that briefly describes this menu item. When the mouse is pointing at this item, so that it is highlighted, the documentation string is displayed in the documentation line at the bottom of the screen. It is considered good practice to include documentation for all menu items.

An example of the use of menu item options is shown here:

```
("Item 2" :value 1.5 :documentation "Costs $1.50"
          :character-style (nil :bold nil))
```

The character style of the displayed item will be of the same family and size as the default character style for the menu, but its face will be bold.

14.4 Choosing and Executing

After an item has been chosen, it is *executed*. Executing a menu item does what the item type tells it to do. Depending on the type of item being executed, executing produces a value, performs a side-effect, or both.

Execution always takes place in the user process (rather than the mouse process). Thus, execution can depend on the special-variable environment and can perform actions that take a long time, interact with the user, or depend on being able to use the mouse.

The responsibility for executing the chosen menu item rests with either the system or the programmer, depending on how the menu is used. The **tv:menu-choose** function and the **:choose** message execute the chosen item and return its *value*, or they return **nil** if no item was chosen. When using command menus the chosen *item* is returned to the user program. See the section "Command Menus", page 271. The user program can execute it by sending the **:execute** message. See the section "Useful **tv:menu** Messages", page 265.

The importance of executing menu items depends on the function of the menu. Some menus contain items that act as "nouns". The user simply chooses one out of a group of similar items. In this case, executing the item serves only to translate from the item list. The item list contains the printed representation displayed in the menu and the documentation displayed in the mouse documentation line. For this kind of item, the **:value** item type is often used.

Other menus contain items that act more like "verbs". The program operating the menu might not be aware of the details of each item; it simply allows the user to choose one and then executes it. In this case, most of the complicated behavior is within the menu item. Typically, the **:eval** or **:funcall** item type is used.

15. The Geometry of a Menu

A menu has a *geometry* that controls its size, its shape, and the arrangement of displayed choices. The creator of a menu may specify some aspects of the geometry explicitly, leaving other aspects to be given by the system according to default specifications.

There are two ways the choices can be displayed. They can be shown in an array of rows and columns, or they can be in a "filled" format with as many to a line as will reasonably fit. Filled format is specified by giving zero as the number of columns.

The geometry of a menu is represented by a list of six elements:

columns
: The number of columns (0 for filled format).

rows
: The number of rows.

inside width
: The *inside width* of the window, in units of the screen (pixels). If you set the size or edges of the window the inside width is remembered here and acts as a constraint on the menu afterwards.

inside height
: The *inside height* of the window, in pixels. If you set the size or edges of the window the inside height is remembered here and acts as a constraint on the menu afterwards.

maximum width
: The maximum (inside) width of a window, in pixels. The window system prefers to choose a tall skinny shape rather than exceed this limit.

maximum height
: The maximum (inside) height of a window, in pixels. The system prefers to choose a short fat shape rather than exceed this limit. If both the maximum width and the maximum height are reached, the system displays only some of the menu items and enables scrolling to make the rest accessible.

Values of **nil** for parts of the geometry can be specified to leave that part unconstrained.

15.1 Geometry Init-plist Options

The init-plist options listed below initialize the geometry of any menu built on the **tv:menu** flavor.

:geometry *list* (for **tv:menu**) *Init Option*

Sets up the complete menu geometry, using a list to specify the columns, rows, inside-width, inside-height, max-width, and max-height. See the section "The Geometry of a Menu", page 255.

:rows *n-rows* (for **tv:menu**) *Init Option*

Sets the number of rows.

:columns *n-columns* (for **tv:menu**) *Init Option*

Sets the number of columns in a menu.

:fill-p *t-or-nil* (for **tv:menu**) *Init Option*

Specifies whether to use filled format or columnar format.

15.2 Geometry Messages

The following messages may be sent to any flavor of menu to access and manipulate its geometry:

:geometry of **tv:menu** *Method*

This message returns a list of six elements, which constitute the menu's geometry. These are the menu's default constraints, with **nil** in unspecified positions; contrast this with the **:current-geometry** message.

:current-geometry of **tv:menu** *Method*

Returns a list of six elements that constitute the geometry corresponding to the actual current state of the menu. This contrasts with the **:geometry** message, which returns the specified default geometry. Only the *maximum width* and *maximum height* can be **nil**.

:set-geometry &optional *columns rows inside-width inside-height max-width max-height* of **tv:menu** *Method*

Note that this message takes six arguments rather than a list of six things as you might expect. This is because you frequently want to omit most of the arguments. The geometry is set from the arguments, which can cause the menu to change its shape and redisplay. An argument of **nil** means to make that aspect of the geometry unconstrained. An omitted argument or an argument of **t** means to leave that aspect of the geometry the way it is.

:fill-p of **tv:menu** *Method*

:set-fill-p *t-or-nil* of **tv:menu** *Method*

Get (**:fill-p**) or set (**:set-fill-p**) the menu's fill mode. **t** is returned from **:fill-p** if the menu displays in filled form rather than columnar form. Thus, use **t** to set the fill characteristic. These messages are a special case of the **:geometry/:set-geometry** messages.

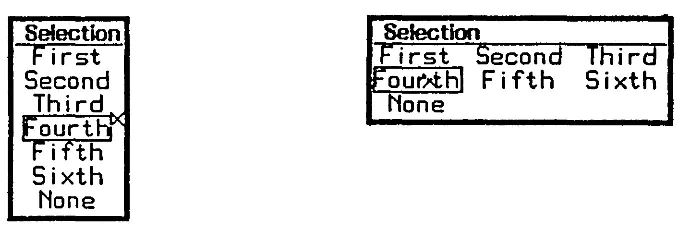
Note that the messages **:set-default-character-style** and **:set-item-list** (which do what they say) also cause the geometry of a menu to be recomputed.

15.3 Geometry Example 1: a Multicolumned Menu

It is not necessary to explicitly specify all six values for the geometry list. In the following example, only the *columns* value is supplied, and a one-column menu is specified. The rest of the geometry values are computed by using the column value to constrain the system-default settings.

```
(setq geometry-list (list 1))
```

Figures 3a and 3b show the result of setting the geometry of a menu first to a one-column form (3a), then a multicolumn format (3b, using the three-column code example below). In the example, the variable **result** holds the value of the item selected by the mouse.



(a) (b)

Figure 3. Adjusting a menu's column geometry. (a) One column (b) Three columns

The code used to generate Figure 3b is next.

```
;;; Geometry Example 1

;;; First element in the geometry list specifies three columns
(setq geometry-list (list 3))
```

```
;;; Make the menu
(setq my-menu (tv:make-window 'tv:momentary-menu
        ':label '(:string " Selection"
                 :character-style (:swiss :bold :normal))
        ':geometry geometry-list
        ':borders 3
        ':item-list '(("First"  :value 100)
                      ("Second" :value 200)
                      ("Third"  :value 300)
                      ("Fourth" :value 400)
                      ("Fifth"  :value 500)
                      ("Sixth"  :value 600)
                      ("None"   :value   0))))
;;; Expose the window, make a choice,
;;;   and leave the value in the variable "result"
(setq result (send my-menu ':choose))
```

15.4 Geometry Example 2: Retrieving Geometry Information

Figure 4 is an example of a simple menu from which we would like to retrieve geometry information.

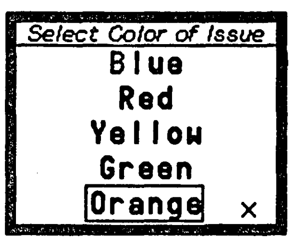


Figure 4. Simple menu from which geometry information is obtained.

The code that produced Figure 4 uses the **:current-geometry** message, which retrieves a description of a menu's geometry. Border and character-style specifications are used to customize the menu. (A list of the loaded screen fonts is accessible by using List Fonts (m-X) in the Zmacs editor.)

```
;;; Menu Geometry Example 2
;;; z is an instance of a momentary window created
;;;  by the make-window function
(setq z  (tv:make-window 'tv:momentary-menu
           ':borders 6
           ':label '(:string " Select Color of Issue "
                     :character-style (:swiss :italic :normal))))

;;; item-list is a list of menu items
(setq item-list '("Blue" "Red" "Yellow" "Green" "Orange"))

;;; This sends a message to set up an item list
(send z ':set-item-list item-list)
```

The next expression interrogates the menu and returns a list that describes its geometry. The list is returned in **geometry-facts.** (Nothing in particular is done with **geometry-facts** in this example).

```
(setq geometry-facts (send z ':current-geometry))
```

The final expression exposes the menu, allows a choice to be made, and returns the selected string in the variable **result.**

```
(setq result (send z ':choose))
```

16. Momentary and Pop-up Menus

A momentary menu appears on the screen with a list of items. The user does not have to make a choice, however. By moving the mouse outside the menu, the menu is made to disappear.

By contrast, a pop-up menu forces the user to make a choice. The menu does not disappear until an item has been selected.

16.1 The Standard Momentary Menu Interface

The standard form of a choice facility provides a simple function-call mechanism for invoking it without specifying its details. The standard momentary menu interface is based on the function **tv:menu-choose.**

tv:menu-choose *item-list* &optional *label near-mode default-item* *Function*

item-list is a list of menu items. See the section "Types of Menu Items", page 250. This function pops up a menu and allows the user to make a choice with the mouse. When the choice is made, the menu disappears and the chosen item is executed. The value of that item is returned. If the user moves the mouse out of the menu and far away, it pops down without making any choice and **nil** is returned.

label is a string to be displayed at the top of the menu, or **nil** (the default) to specify the absence of a label.

near-mode specifies where to put the menu on the screen. It defaults to the list **(:mouse)** and must be an acceptable argument to **tv:expose-window-near**.

default-item is the item over which the mouse should be positioned initially. This allows the user to select that item without moving the mouse. If *default-item* is **nil** or unspecified, the mouse is initially positioned in the center of the menu.

16.2 Standard Momentary Menu Example

The following code is an example of how to instantiate a simple momentary menu. Once the menu pops up, the user can make a choice with the mouse. (Any mouse button selects the chosen item.) The *item-list* is a list of menu items in the "general list" form. The **price** variable is set to the value of the selected item, specified in the item list.

```
(setq item-list
  '(("Meat and potatoes" :value 3.49 :documentation "Costs $3.49")
    ("Fish and chips" :value 3.79 :documentation "Costs $3.79")
    ("Hash" :value 1.49 :documentation "Costs $1.49")
    ("Chicken stew" :value 2.99 :documentation "Costs $2.99")))
(setq price (tv:menu-choose item-list "F & T Diner "))
```

16.3 The tv:mouse-y-or-n-p Facility

One of the simplest choice facilities in the system is based on the **tv:menu-choose** function. This is the **tv:mouse-y-or-n-p** function, which is useful for quick yes-or-no queries in a user interface.

tv:mouse-y-or-n-p *item* — *Function*

Takes an item as its argument and displays it in a one-item menu. *item* is usually a string. If the user clicks on this menu with the mouse button, the value of the item is returned. If the user moves the mouse out of the menu, **nil** is returned.

16.4 Basic and Mixin Pop-up and Momentary Menus

The *basic* and *mixin* flavors for ordinary kinds of menus are explained in this section. They cannot be instantiated themselves but they are the building blocks of the instantiable menus.

tv:basic-menu — *Flavor*

All the other menus in the standard menu facility are built on this flavor. The basic menu handles an item list, it remembers the last item selected, and it knows about its geometry. See the section "The Geometry of a Menu", page 255.

tv:basic-momentary-menu — *Flavor*

When this flavor is mixed with a window, it creates a kind of menu that is only momentarily on the screen. A **:choose** operation on a deexposed menu of this flavor causes it to position itself where the mouse is and expose itself. When the user selects an item in the menu, or alternatively moves the mouse far away from the menu, the menu disappears and deactivates.

tv:window-hacking-menu-mixin — *Flavor*

This menu flavor mixin provides for the **:window-op** item type. The window that the menu is exposed over is remembered. The remembered window is used if an item of type **:window-op** is selected. See the section "Types of Menu Items", page 250.

16.5 Instantiable Pop-up and Momentary Menus

The instantiable menu flavors are listed below, followed by an example of how to instantiate one of them. Two of the most important menu flavors are **tv:menu** and **tv:momentary-menu,** since many other menu flavors are built on them. For a diagram of the flavor network on which **tv:menu** and **tv:momentary-menu** are built: See the section "The Flavor Network Of **tv:menu**", page 339. For an enumeration of many of **tv:menu's** init-plist options and messages: See the section "Init-plist Options For **tv:menu**", page 341. See the section "Messages Accepted By **tv:menu**", page 345.

tv:menu *Flavor*

This is **tv:basic-menu** with borders and an optional label on top. By default, there is no label, but you can specify one with the **:label** init-plist option or the **:set-label** message. **tv:menu** is built on the **tv:basic-menu, tv:borders-mixin, tv:top-box-label-mixin, tv:basic-scroll-bar,** and **tv:minimum-window** flavors.

tv:momentary-menu *Flavor*

This is built on **tv:basic-momentary-menu** mixed with **tv:menu.** See the section "The Flavor Network Of **tv:menu**", page 339.

Momentary menus display a list of items. The user can avoid making a choice by moving the mouse outside the menu. In this case, the menu disappears.

tv:pop-up-menu *Flavor*

This menu is a combination of **tv:menu** and **tv:temporary-window-mixin,** but does not have the automatic expose and deexpose features of **tv:momentary-menu.** See the section "Temporary Windows", page 95. It is appropriate to use a pop-up menu rather than a momentary menu when you want to pop a menu up and make several choices from it before popping it back down. Another use is if you want to force the user to make a choice. Moving the mouse outside of the menu boundary does not deexpose the menu.

tv:momentary-window-hacking-menu *Flavor*

This is a momentary menu combined with **tv:window-hacking-menu-mixin.** The window that the menu is exposed over is remembered when the **:choose** message is sent. The remembered window is used if a **:window-op** type item is selected.

tv:momentary-menu &optional *(superior* **tv:mouse-sheet***)* *Resource*

This is a *resource* of momentary menus. **tv:menu-choose** allocates a window from this resource.

16.6 Useful tv:menu Init-plist Options

This is a list of some of the most frequently used init-plist options for the **tv:menu** flavor and menu flavors built on it, such as **tv:momentary-menu** and **tv:pop-up-menu.** For a list of more window-related init-plist options associated with any flavor built on **tv:menu**: See the section "Init-plist Options For **tv:menu**", page 341.

:borders *argument* (for **tv:menu**) *Init Option*

This option initializes the parameters of the borders. The *argument* can be **nil**, which specifies no borders, **t**, which specifies default borders, or it can be a *specification* of a border. The specification indicates which function is used to draw the border and how thick the border is, in pixels.

If the specification is a *number*, the border is drawn by the default function at the specified thickness. The default function is **tv:draw-rectangular-border**.

If the specification is a *symbol*, the border is drawn by the specified function at a default thickness. For more details on creating a function: See the section "Using the Window System", page 81.

If the specification is a *cons* in the form *(function . thickness)*, the borders are drawn by the specified function at a specified thickness.

The specification can also be a list of locations on the screen: *(left top right bottom)*.

:default-character-style *character-style* (for **tv:menu**) *Init Option*

Specify the default character style of the menu. Items whose character style is unspecified are displayed in the default style. If a character style is specified for an item, it is merged against the default style. (See the section "Menu Item Options", page 251.)

:item-list *list* (for **tv:menu**) *Init Option*

Initialize the item list for a menu. See the section "Types of Menu Items", page 250.

:label *specification* (for **tv:menu**) *Init Option*

Specifies the menu's label. The specification is usually a list in the following form:

`(:string "Foo" :character-style` *character-style-specification*`)`

:vsp *n-pixels* (for **tv:menu**) *Init Option*

Sets the vertical spacing between lines in the menu. The default is 2 pixels.

See the section "Geometry Init-plist Options", page 256.

16.7 Useful tv:menu Messages

This is a list of some useful window and menu-related messages associated with the **tv:menu** flavor and any flavor built on it. For a list of more window-related messages to **tv:menu**: See the section "Messages Accepted By **tv:menu**", page 345.

:choose of **tv:menu** *Method*

This message exposes the window and allows the user to make a choice with the mouse. It sends **:execute** to the window and performs the action specified by the item's type.

:execute *item* of **tv:menu** *Method*

This message extracts the value from a chosen item and returns it, or it performs a side-effect, or both. It decides what to return based on the item's type. See the section "Types of Menu Items", page 250.

In a program that uses command menus, the **:any-tyi** message can return a blip containing the menu and an item. The program sends the **:execute** message to the menu to execute the item. See the section "Command Menus", page 271.

:execute is sent by the system for other kinds of menus. For example, the **:choose** message, which returns a value and not an item, uses the **:execute** message to retrieve the value from the chosen menu item.

:deactivate of **tv:menu** *Method*

This message deactivates a window, deexposing it. In momentary menus, it is sent when the mouse is moved outside the borders of the menu.

16.8 tv:momentary-menu Example 1: Simple Momentary Menu

An example of a simple momentary window with three items in it from which to select is shown in Figure 5.

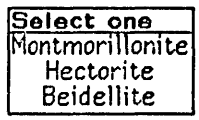


Figure 5. Momentary menu example.

The code to produce such a menu is given next. (In the example, there are no actions specified when an item is selected.)

```
;;; Momentary Menu Example 1

;;; z is an instance of a momentary menu created by the
;;;  make-window function
(setq z (tv:make-window 'tv:momentary-menu
           ':label '(:string "Select one"
                     :character-style (:swiss :bold :normal))))

;;; item-list is a list of menu items
(setq item-list '("Montmorillonite" "Hectorite" "Beidellite"))

;;; This passes a message to set up an item list
(send z  ':set-item-list item-list)

;;; The :choose message exposes the window and allows the mouse
;;;  to select an item.  choice holds the result.
(setq choice (send z ':choose))
```

16.9 tv:momentary-menu Example 2: Item List as Init-plist Option

Another way to set up the item list is to specify it as an init-plist option.

```
;;; Example 2
;;; Shows use of the init-plist to specify items

(setq z (tv:make-window 'tv:momentary-menu
              ':label " New Selection "
              ':item-list '("First" "Second" "Third")))

(setq choice (send z ':choose))
```

16.10 tv:momentary-menu Example 3: Centered Label and Use of General List Items

In Example 3, two new principles are shown. First, in order to have a centered label for the menu, the new flavor **momentary-menu-with-centered-label** is created.

Second, the "general list" form of item list is used. See the section "The "General List" Form of Item", page 250. This allows your program to invoke an operation or return a value when an item is selected. In the example, the variable *choice* is set to **nil** or one of the numbers 1.0, 2.0, or 3.0, depending upon the action taken by the user.

The **:documentation** option keyword has the following effect. When an item with the **:documentation** keyword is pointed at by the mouse, the specified documentation string appears in the inverse-video mouse documentation line at the bottom of the screen.

```
;;; Example 3
;;;  Shows use of flavor mixing and "general list" menu items

;;; Define a flavor with the centered-label-mixin
(defflavor momentary-menu-with-centered-label ()
           (tv:centered-label-mixin tv:momentary-menu))

;;; Create an instance of the window
(setq z (tv:make-window
           'momentary-menu-with-centered-label
           ':label "Selection"
           ':item-list '(("Orange" :value 1.0
                            :documentation "Select orange.")
                          ("Red" :value 2.0
                            :documentation "Select red.")
                          ("Yellow" :value 3.0
                            :documentation "Select yellow."))))

(setq choice (send z ':choose))
```

16.11 tv:momentary-menu Example 4: Using the Mouse Buttons

The general list form can include choices that depend on the three mouse buttons. **:buttons** is a menu itemtype that takes three arguments (*left middle right*), each of which specifies what to do if a particular button is pressed. If any argument to

:buttons is **nil,** a click on that button is ignored. See the section "Types of Menu Items", page 250. An example demonstrates its use.

```
;;; Example 4, shows the use of different mouse buttons

;;; Specify the item list
(setq button-list
      '(("One Item, Three Ways"
         :buttons ((l :eval (print "left"))
                   (m :eval (print "middle"))
                   (r :eval (print "right")))
         :documentation
         "L: Print left, M: Print middle, R: Print right")))


;;; Make the menu
(setq menu-1 (tv:make-window 'tv:momentary-menu
                             ':label "Test Buttons"
                             ':item-list button-list))

;;; Expose the window and choose
(setq choice (send menu-1 ':choose))
```

16.12 tv:pop-up-menu Example

Since a pop-up menu does not operate as automatically as a momentary menu, it requires a slightly different treatment. The normal mode of operation is to allow **:choose** to activate and expose it, and then send it a **:deactivate** message when done. This does not "destroy" the menu, it just makes sure it does not appear on the screen.

Figure 6 shows a simple example of a pop-up menu. We use the "general list" form of item to invoke a function that exposes a second menu and stores the results of the two selections in the variables **drink** and **price.**

The code that generated Figure 6 follows on the next pages.

```
;;; Pop-up menu example

(defvar drink nil)
(defvar grapefruit "Grapefruit Juice")
(defvar orange "Orange Juice")
(defvar apple "Apple Juice")
```

Figure 6. Pop-up menu example.

```
;;; This function dispatches according to the kind of
;;;  juice selected, and calls the second menu
(defun juice (fruit)
  (selectq fruit
    (gr (setq drink grapefruit))
    (oj (setq drink orange))
    (ap (setq drink apple)))
  (setq price (send two ':choose)))

;;; This function handles the no-juice item
(defun no-juice ()
  (setq drink nil))

;;; This the first menu, a pop-up menu that allows the user
;;;  to select a juice
(setq one (tv:make-window
             'tv:pop-up-menu
             ':label "Juice selection"
             ':borders 3
             ':item-list '(("Grapefruit" :eval (juice 'gr))
                           ("Orange" :eval (juice 'oj))
                           ("Apple" :eval (juice 'ap))
                           ("None" :funcall no-juice))))
```

```
;;; This is the second menu, a momentary menu that allows the user
;;;  to select a size of drink
(setq two (tv:make-window
             'tv:momentary-menu
             ':label "What size please?"
             ':borders 3
             ':item-list
                '(("Dinky" :value  .5
                  :documentation "Smallest size costs 50 cents.")
                  ("Large" :value 1.0
                  :documentation "Actually medium size, costs $1.")
                  ("Jumbo" :value 1.5
                  :documentation "Big, costs $1.50.")
                  ("None"  :value 0
                  :documentation "Cheapest selection by far."))))

;;; Operate the menu; explicit exposing and
;;;  deactivating are necessary for pop-up menus
(defun operate ()
  (send one ':expose-near '(:mouse))
  (send one ':choose)
  (send one ':deactivate))

;;; Invoke the juice selection menu
(operate)
```

Another way to implement this example would have been to use the **:menu** item type to invoke the second menu. See the section "Types of Menu Items", page 250.

17. Command Menus

Command menus are used when a menu does not stand alone but is part of a frame of several window panes, which can include other menus. The entire frame is controlled by a single process; each frame sends *commands* (or *blips*) to the controlling process from a menu. (For Dynamic Window-based frames, various high-level facilities are available for creating command menus: See the section "Overview of Top-level Facilities for User Interface Programming" in *Programming the User Interface, Volume A.*)

In order to understand the operation of a command menu, it is necessary to understand the difference between a menu item and a menu item's value.

17.1 Menu Items and Menu Values

A menu item consists of a list supplied by the programmer in the item list of a menu specification. In most menus, your program rarely receives menu items back from the window system; usually the *values* of the items are returned. There are two exceptions to this situation:

- Certain messages deal explicitly with items, such as the **:item-list** message, which returns the list of items associated with a menu.
- In command menus, your program receives a command (or blip) back from the window system. The blip contains an entire item as well as other information (explained in the next section). You send the **:execute** message to the menu to extract the item's value and perform side-effects.

17.2 Command Blips

Since the **:choose** message (which gets a value and not an item) does not operate on a command menu, the command is sent to the user process through an *I/O buffer* associated with the menu. (Many windows have an I/O buffer associated with them. See the section "Overview of Window Flavors and Messages", page 115.) Your controlling process can be looking in its I/O buffer for commands from several windows as well as for keyboard input.

The command chosen by the user is sent to the I/O buffer as a list in the following form:

(**:menu** *chosen-item button-mask window*)

Note: The button-mask is a bit mask with a bit for each button on the mouse. This provides the option of taking different actions depending on which mouse button was pressed. The bit assignments are as follows:

1	Left button
2	Middle button
4	Right button

17.3 Responsibilities of Your Program

Your program is responsible for performing each of the actions that the **:choose** message would normally do, including the following:

- Deciding where to put the menu. Usually this is specified in the definition of the frame, via **:panes** and **:constraints** specifications in a **tv:bordered-constraint-frame-with-shared-io-buffer** flavor.

- Exposing the menu. Usually the command menu is part of a frame and the entire frame is exposed.

- Receiving a choice from the mouse. This is received via an I/O operation like the **:any-tyi** message.

- Executing the choice. Example: (**send** *window* **':execute** *chosen-item*)

- Deciding whether to deactivate the frame. This is not normally performed on an individual command menu pane.

17.4 Command Menu Mixins

tv:command-menu-mixin *Flavor*

This is the basic mixin version of the command menu flavor. It is not instantiable on its own.

tv:command-menu-abort-on-deexpose-mixin *Flavor*

When a command menu built on this flavor receives the **:deexpose** message, it searches its item list for an item whose displayed representation is [Abort]. If such an item is found, a mouse blip is sent to the I/O buffer indicating that the [Abort] item was clicked on. See the flavor **tv:dynamic-pop-up-abort-on-deexpose-command-menu**, page 278.

17.5 Instantiable Command Menus

tv:command-menu *Flavor*

This is **tv:command-menu-mixin** mixed with **tv:menu** to make it instantiable.

tv:command-menu-pane *Flavor*

This version of the command menu flavor is meant to be used within a window frame. See the section "Frames", page 204.

17.6 tv:command-menu Init-plist Options

:io-buffer *buf* (for **tv:command-menu**) *Init Option*

The I/O buffer to be used by a command menu is usually specified when it is created. It can be shared with the I/O buffer of another window. I/O buffers are created with the **tv:make-io-buffer** function.

Note: By making a command-menu to be a pane in a a **tv:bordered-constraint-frame-with-shared-io-buffer**, you are supplied with an I/O buffer automatically. The frame puts an **:io-buffer** option into the init-plist of each pane. See the section "Frames", page 204.

17.7 tv:command-menu Messages

:io-buffer of **tv:command-menu** *Method*

This message *gets* the I/O buffer to which a command menu sends a command when an item is chosen.

:set-io-buffer *io-buffer* of **tv:command-menu** *Method*

This message *sets* the I/O buffer to which a command-menu sends a command when an item is chosen.

17.8 tv:command-menu Example

Figure 7 shows a simple command menu. The top pane contains a command menu that allows the user to draw an object on the screen. The middle pane is the drawing surface. The bottom pane is another command menu that allows the user to refresh the drawing surface or exit.

The Lisp code to produce the window in Figure 7 is shown next.

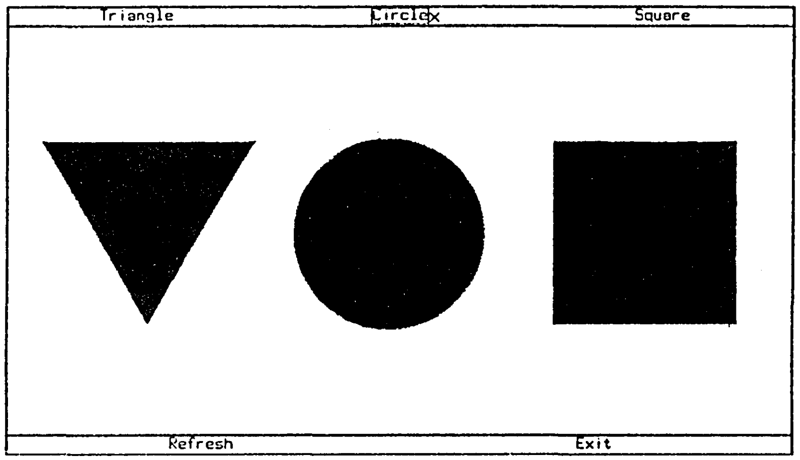


Figure 7. Command menu example.

```
;;; Define the frame and its panes
(setq *test-frame*
      (tv:make-window
        'tv:bordered-constraint-frame-with-shared-io-buffer
        ;; Select the graphics pane when it is exposed
        ':selected-pane 'graphics-pane
        ;; Specify the panes
        ':panes
        '((lower-menu-pane
            tv:command-menu-pane
            :item-list
            (("Refresh" :value :refresh
              :documentation "Refresh graphics pane")
             ("Exit" :value :exit
              :documentation "Exit this frame.")))
          (graphics-pane tv:window :label nil :blinker-p nil)
          (upper-menu-pane
            tv:command-menu-pane
            :item-list
            (("Triangle" :value :triangle
              :documentation "Draw a triangle.")
             ("Circle" :value :circle
              :documentation "Draw circle.")
             ("Square" :value :square
              :documentation "Draw square."))))

        ;; Specify the size constraints and ordering
        ':constraints
        '((main . ((upper-menu-pane graphics-pane lower-menu-pane)
                   ;; Big enough for the menu
                   ((upper-menu-pane :ask :pane-size))
                   ;; Big enough for graphics pane
                   ((graphics-pane :400.))
                   ;; Big enough for the menu
                   ((lower-menu-pane :ask :pane-size)))))))
```

```
;;; This function accesses the panes and looks for a blip
;;;  in the I/O buffer. It then draws, refreshes the
;;;  graphics pane, or exits
(defun work ()
  ;; Get access to the panes
  (let ((graphics-pane
          (send *test-frame* ':get-pane 'graphics-pane))
        (upper-menu-pane
          (send *test-frame* ':get-pane 'upper-menu-pane))
        (lower-menu-pane
          (send *test-frame* ':get-pane 'lower-menu-pane)))
    (send *test-frame* ':expose)
    ;; blip holds the list returned by :any-tyi
    (loop as blip = (send graphics-pane ':any-tyi)
          as result-value =
                 (cond ((and (listp blip) (eq (car blip) ':menu))
                        (send (fourth blip) ':execute (second blip)))
                       (t nil))  ;just ignore keyboard input
          do
          ;; Check the value and draw the appropriate object
          (selectq result-value
            (:square
             (send graphics-pane ':draw-rectangle 180. 180. 800. 110.))
            (:circle
             (send graphics-pane ':draw-filled-in-circle 530. 200. 94.))
            (:triangle
             (send graphics-pane ':draw-regular-polygon
                   82. 120. 282. 120. 3))
            (:refresh
             (send graphics-pane ':refresh))
            (:exit
             (send *test-frame* ':deactivate)
             (return))))))

(work)
```

18. Dynamic Item List Menus

A dynamic item list menu is a menu in which the items change in between exposures. You see an example of a dynamic item list menu when you click on the [Select] item on the System menu (Figure 8). At different times, a different item list appears, depending upon how many different processes were activated by the user.

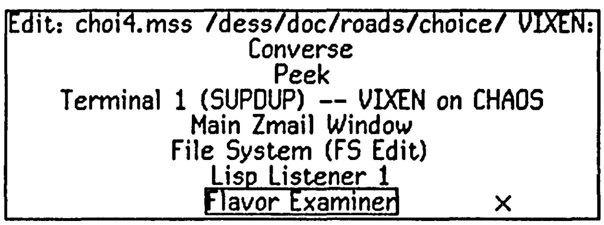


Figure 8. Select menu, an example of a dynamic item list menu.

You can add an item to the menu by changing the value of the variable supplied as the **:item-list-pointer** init-plist option. At appropriate times the menu checks to see if this variable has been changed. If it has, the menu automatically updates the item list. (Do not directly modify the item list yourself, as it is part of the menu.) For a description of the times when the menu checks the state of **:item-list-pointer** option, See the section "Messages to Dynamic Menus", page 279.

The dynamic item list feature is provided only for momentary and pop-up menus; it is not available for use in menus within fixed frames.

18.1 Dynamic Item List Mixins

tv:abstract-dynamic-item-list-mixin *Flavor*

This is a noninstantiable mixin flavor that implements the general notion of dynamically changing the item list. It causes the menu's item list to be updated at appropriate times. The actual item list is computed via the **:update-item-list** message.

tv:dynamic-item-list-mixin *Flavor*

This is a noninstantiable mixin flavor, built on **tv:abstract-dynamic-item-list-mixin** used as a building block to make instantiable versions listed later. This flavor provides a specific means of getting the latest item list, by evaluating a Lisp form, and provides the **:item-list-pointer** instance variable.

In the operation of this flavor, the old result of evaluating the value of **:item-list-pointer** is saved; if the new result of evaluating the value of **:item-list-pointer** is not the same (compared with the **zl:equal** function), then the item list is considered changed and the menu is updated. **:item-list-pointer** is evaluated when the **:choose** message is sent.

tv:dynamic-multicolumn-mixin *Flavor*

This is a noninstantiable mixin flavor. It makes a menu have multiple "dynamic" columns. Each column comes from a separate item list that is recomputed at appropriate times. The instance variable **tv:column-spec-list** is a list of columns. Each column list is in the form:

(heading item-list-form . options)

Heading is a string to go at the top of the column, and *options* are menu item options for it (typically a character style specification). *item-list-form* is a form to be evaluated (without side-effects) to get the item list for that column.

18.2 Instantiable Dynamic Item List Menus

tv:dynamic-momentary-menu *Flavor*

This is a momentary menu with the **tv:dynamic-item-list-mixin** and the **tv:abstract-dynamic-item-list-mixin.**

tv:dynamic-momentary-window-hacking-menu *Flavor*

This is a momentary menu with both the **tv:dynamic-item-list-mixin** and the **tv:window-hacking-mixin.**

tv:dynamic-pop-up-menu *Flavor*

This is a pop-up menu with the dynamic item-list mixin.

tv:dynamic-pop-up-command-menu *Flavor*

Specifies a command menu with the temporary-menu and dynamic item-list mixins. It is mixed in to form the hardcopy menu flavor **press:hardcopy-dynamic-pop-up-command-menu-with-highlighting.**

tv:dynamic-pop-up-abort-on-deexpose-command-menu *Flavor*

This is a command menu with the **tv:dynamic-pop-up-command-menu** and **tv:abort-on-deexpose** mixins.

18.3 Init-plist Option for Dynamic Menus

:column-spec-list *form* (for **tv:dynamic-multicolumn-mixin**) *Init Option*

Specified as a list of columns in the form:

(heading item-list-form . options)

Heading is a string to go at the top of the column, and *options* are menu item options for it (typically a character style specification). *item-list-form* is a form to be evaluated (without side-effects) to get the item list for that column.

:item-list-pointer *form* (for **tv:dynamic-...-menu**) *Init Option*

The ellipses in the name (...) indicate that this option works with several flavors of dynamic menus. The *form* is saved and evaluated periodically to get the item-list for the menu. *form* is usually a special variable but any Lisp form is valid. The evaluation may occur in any process, so only global variables should be accessed. If the result of evaluating *form* is not **zl:equal** to the item list, the message **:set-item-list** is sent to the menu to update the new list. Note that the Lisp function **equal** is used for comparison, not **eq.** (Do not directly and destructively modify a menu's item list yourself; the system will do this automatically.)

18.4 Messages to Dynamic Menus

:update-item-list of **tv:dynamic-...-menu** *Method*

Updates the item list if it needs to change;* this message is accepted by menus with the dynamic item-list mixin. The **:update-item-list** message sends a **:set-item-list** if one is necessary. The dynamic menu sends itself this message automatically at appropriate times. The appropriate times are before **:choose, :move-near-window, :center-around, :size,** and **:pane-size** messages.

*) nur, wenn Länge der Liste vergrößert oder verkleinert.
(ab 7.1. okay!)

18.5 Dynamic Menu Example

A graphic example of a dynamic-momentary-menu is given in Figure 9. The menu is shown in its state before updating (a) and after updating (b). This is followed by a listing of the code that produces it.

Figure 9. Dynamic menu example.

```
;;; Dynamic Menu Example
;;; Set up the initial item list and define the
;;;  dynamic-item-list pointer.
(defvar pointer
        '("Door Number 1"
          "Door Number 2"
          "Door Number 3"))

;;; Make the dynamic menu
(defvar doors (tv:make-window 'tv:dynamic-momentary-menu
                              ':borders 4
                              ':default-character-style
                                (:dutch :bold :normal)
                              ':label "CHOICES"
                              ':item-list-pointer 'pointer))

;;; Expose the menu, allowing a choice to be made
(send doors ':choose)
```

(In the example, nothing is being done with the result.)

Here is an example of dynamically updating the item list. The **:update-item-list** message is sent automatically and transparently by the menu to itself. The user does not have to explicitly send it.

```
;;; Add entries to the item list
(setq pointer
  (append pointer (list "Door Number 4" "Door Number 5")))

;;; Expose the menu with the new choices added
(send doors ':choose)
```

18.6 Adding an Item to the System Menu

Although they are not specifically a part of the dynamic item list facility, two functions exist for adding an item (such as the name of a program) to the System menu.

18.6.1 Adding an Item to the Programs Column

To add an item to the *Programs* column of the System menu, use the following function:

tv:add-to-system-menu-programs-column *name form documentation* &optional *after* — *Function*

Adds a program to the Programs column of the system menu. *name* is a string, the name to appear in the menu. *form* is a form to evaluate, in its own process, when the program is selected; often this is a call to **tv:select-or-create-window-of-flavor**. *documentation* is mouse documentation for the menu item. *after* determines the position of the new program name in the Programs column:

nil	Bottom of the column
t	Top of the column
string	After the program named *string* that is now in the menu

Example:

```
(tv:add-to-system-menu-programs-column
  "Concept Editor" 'crl:concept-editor
  "Edit the representation of a concept in the CRL system")
```

18.6.2 Adding an Item to the Create Column

To add an item to the *Create* menu used in the System Menu and the Screen Editor, use the following function:

tv:add-to-system-menu-create-menu *name flavor documentation* &optional *after* — *Function*

Adds an entry to the menu that appears when you click on [Create] in the System Menu or in the Edit Screen menu. *name* is a string, the name of the menu item. *flavor*, a flavor name, is the flavor of window that is created when the menu item is selected. *documentation* is mouse documentation for the menu item. *after* determines where in the [Create] menu the item should appear:

nil	Bottom of the menu
t	Top of the menu
string	After the item named *string* that is now in the menu

Example:

```
(tv:add-to-system-menu-create-menu
  "Concept Editor" 'crl:concept-editor
  "Edit the representation of a concept in the CRL system")
```

18.6.3 tv:select-or-create-window-of-flavor Function

tv:select-or-create-window-of-flavor *find-flavor* &optional (*create-flavor find-flavor*) *Function*

Selects the most recently selected window of flavor *find-flavor*. If no window of that flavor exists, makes a window of flavor *create-flavor* and selects it.

19. Multiple Menus

Multiple menus allow several items to be selected at a time. The selected items are highlighted in inverse video. Clicking the mouse on an item complements its selected state. Clicking the default special choice [Do It] associated with a multiple menu completes the selection, and returns the result of executing all the highlighted choices. The lower portion of Figure 10 is an example of a hardcopy multiple menu with several items selected.

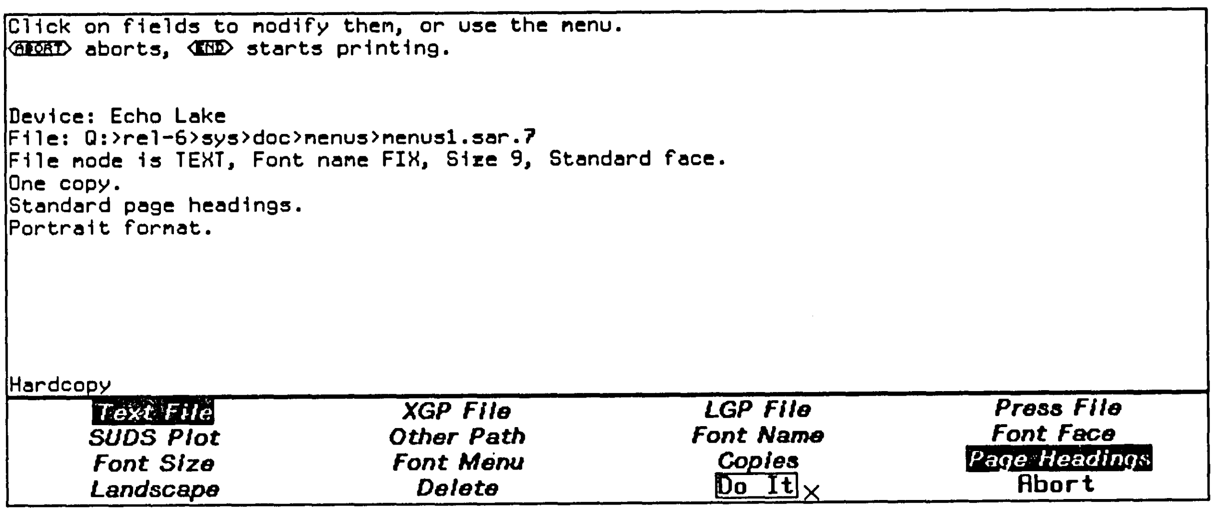


Figure 10. Hardcopy multiple menu.

19.1 Multiple Menu Mixins

These are the noninstantiable flavors that add multiple menu behavior to a window.

tv:menu-highlighting-mixin *Flavor*

This mixin flavor allows some of the menu items to be highlighted with inverse video. This is typically used with menus of options, where the options currently in effect are highlighted. The menu items corresponding to modes are typically set up so that when executed, they adjust the highlighting to reflect the enabling or disabling of a mode.

tv:multiple-menu-mixin *Flavor*

This mixin flavor gives a menu the ability to have multiple items "selected". Selected items are highlighted with inverse video, using the **tv:menu-highlighting-mixin.** Clicking on an item merely complements its selected state and does not execute it or return from the **:choose** message.

Normally (but not in the example above) at the top of the menu, in italics, are displayed some "special choices" (for example, [Do It] or [Abort]) that cannot be highlighted. Clicking on one of these behaves the same as clicking on an item of an ordinary menu.

By default, the only special choice is [Do It], which returns (from the **:choose** message) a list of the results of executing all the highlighted choices (that is, the result of the **:highlighted-values** message). You can define your own special choices with the **:special-choices** init-plist option, or get rid of them entirely by giving **nil** as the argument to this option.

19.2 Instantiable Multiple Menus

tv:multiple-menu *Flavor*

This instantiable menu flavor is a combination of **tv:multiple-menu-mixin** with **tv:menu.** It must be explicitly deactivated by the user program.

tv:momentary-multiple-menu *Flavor*

This instantiable flavor is built on **tv:multiple-menu-mixin** and **tv:menu-highlighting-mixin** with **tv:momentary-menu.**The menu is exposed near the mouse, and like any momentary menu, the menu disappears once the user has made a choice.

19.3 tv:multiple-menu-mixin Init-plist Options

:highlighted-items *items* (for **tv:menu-highlighting-mixin)** *Init Option*

When a menu with the menu-highlighting mixin is created, the list of items to be initially highlighted may be specified. The items in this list must be **eq** to items in the menu's **:item-list.** The default is **nil.**

:special-choices *choice-list* (for **tv:multiple-menu-mixin)** *Init Option*

Each element of *choice-list* specifies a menu item for a multiple menu. These are the items that behave like normal menu items; the items from the **:item-list** init option behave as on/off switches as described above. An element of *choice-list* may be any form of menu item.

19.4 tv:multiple-menu-mixin Messages

:highlighted-items of **tv:menu-highlighting-mixin** *Method*
Get the list of highlighted items.

:set-highlighted-items *list* of **tv:menu-highlighting-mixin** *Method*
Set the list of items to be highlighted.

:add-highlighted-item *item* of **tv:menu-highlighting-mixin** *Method*
Add an item to the list of items to be highlighted.

:remove-highlighted-item *item* of **tv:menu-highlighting-mixin** *Method*
Remove an item from the list of highlighted items.

:highlighted-values of **tv:menu-highlighting-mixin** *Method*

:set-highlighted-values *list* of **tv:menu-highlighting-mixin** *Method*

:add-highlighted-value *value* of **tv:menu-highlighting-mixin** *Method*

:remove-highlighted-value *value* of **tv:menu-highlighting-mixin** *Method*
These messages are similar to the preceding four, except that instead of referring to items directly you refer to their values, that is, the result of executing them. For instance, if your item-list is an association list, with elements (*string* **.** *symbol*), these messages use *symbol*. This only works for menu items that can be executed without side-effects, not, for example, the **:eval** and **:funcall** kinds.

When using the above methods, note that for those requiring an item from the menu's item list, the item must be **eq** to the **:item-list** item, that is, the item itself. Consider the following example:

You make a menu (probably in a constraint frame description):

```
      ...
   :panes '(...
            (tv:command-menu-pane :item-list
                  (("This" :funcall this
                   ("The other" :funcall the-other)))
            ...)
```

Later, in some function, you want to highlight the "This" menu item. So you use the **:set-highlighted-items** message:

```
    ...
    (send menu :set-highlighted-items
        '(("This" :funcall this)))
    ...
```

Doing this does not highlight anything. What you need to do instead is:

```
(defvar *item-list* '(("This" :funcall this)
                      ("That" :funcall that)
                      ("The other" :funcall the-other)))

;;; make the constraint frame, but use backquote:

  ...
:panes `(...
         (tv:command-menu-pane :item-list ,*item-list*)
         ...)
  ...


;;; And in the function, do this:

  ...
(send menu :set-highlighted-items (list (first *item-list*)))
  ...
```

19.5 tv:momentary-multiple-menu Example

A simple example of defining a momentary multiple menu is given in Figure 11. The example of a Thai restaurant is used to illustrate the situation where more than one choice is appropriate.

The Lisp code used to generate Figure 11 is given in this example of setting up and using a multiple menu. The variable **selections** is used to contain the selected items.

Figure 11. Momentary multiple menu.

```
;;; Multiple Menu Example
;;; Set up the item list. Each of the dishes has a name and
;;;  a number. When selected, the names are highlighted.
(setq items '(("Yum Hed Koong" 1)
              ("Nur Pud Nor-mai" 2)
              ("Nur Pud Pek" 3)
              ("Nam Sod" 4)
              ("Gai Pud Gra-prao" 4)
              ("Pla Preow Warn" 5)
              ("Pud Thai" 6)))


;;; This handles the "Do It" special item
(defun do-it ()
  ;; Get the names of the selected dishes
  (setq names
        (mapcar 'car (send Thai-menu ':highlighted-items)))
  ;; Get the numbers of the selected dishes
  (setq selections
        (send Thai-menu ':highlighted-values)))

;;; This handles the "None" special item
(defun none ()
  (send Thai-menu ':set-highlighted-items nil)
  (setq selections nil)
  (setq names nil))
```

```
;;; This handles the "All" special item
(defun all ()
  ;; Make all the items selected
  (send Thai-menu':set-highlighted-items items)
  ;; Get the names of the selected dishes
  (setq names (mapcar 'car (send Thai-menu ':highlighted-items)))
  ;; Get the numbers of the selected dishes
  (setq selections (send Thai-menu ':highlighted-values)))

;;; This sets up the special choice list.
;;; When one of these is selected, the menu exits.
(setq choices '(("Do it" :eval (do-it))
                ("None" :eval (none))
                ("All" :eval (all))))

;;; This instantiates the menu
(setq Thai-menu (tv:make-window
        'tv:momentary-multiple-menu
        ':item-list items
        ':special-choices choices))

;;; This exposes the menu, allowing choices to be made.
(send Thai-menu ':choose)
```

20. The Multiple Menu Choose Facility

The multiple menu choose facility provides menus with several columns. The user may choose one item from each column. The selected choice in each column is highlighted with inverse video. At the bottom of the leftmost two columns are two special choices, in italics. The [Do It] choice selects all the highlighted choices. [Abort] deactivates the menu with no further action.

An example of the multiple menu choose facility can be displayed by clicking right on the [Reply] item in the main Zmail window, as in Figure 12 below.

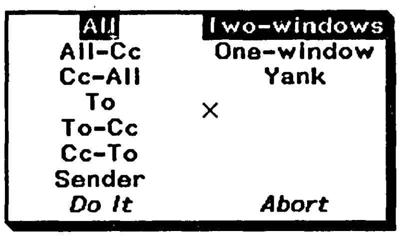


Figure 12. Multiple menu choose facility in Zmail.

Menus of this type are operated by the **:multiple-choose** message rather than the **:choose** message.

20.1 The Standard Multiple Menu Choose Function

This function provides all the default values necessary for a simple multiple-menu-choose menu.

tv:multiple-menu-choose *item-list defaults* &optional *near-mode* *Function*

item-list is a list of lists of menu items. Each sublist corresponds to a column. *defaults* is a list of menu items, one for each column, which are initially highlighted.

The function pops up a menu and allows the user to make choices with the mouse. The special choices [Do It] and [Abort] are supplied automatically. The function returns the list of selected menu items or **nil** if the user aborts. Note: The **tv:multiple-menu-choose** function executes items when they are chosen, not when the [Do It] choice is made. The menu items should not have any side-effects when executing.

tv:defaulted-multiple-menu-choose *item-list defaults* &optional *near-mode* — *Function*

item-list is a list of lists of menu items. Each sublist corresponds to a column.

defaults is a list of menu values, one for each column, which are initially highlighted.

This function is similar to **tv:multiple-menu-choose** but the defaults received by it and the values returned by it are values, not items.

20.2 tv:multiple-menu-choose Example

An example of a simple multiple-menu-choose menu is shown in Figure 13.

Figure 13. A standard multiple-menu-choose menu.

The code to produce the menu in Figure 13 follows.

```
;;;This sets up the three-row item list
(setq possibilities
        '((Item-AA Item-AB Item-AC)
          (Item-BA Item-BB Item-BC)
          (Item-CA Item-CB Item-CC)))

;;; This instantiates the menu
(setq new-menu (tv:multiple-menu-choose
                  possibilities '(Item-AA Item-BA Item-CA)))
```

20.3 Multiple Menu Choose Mixin and Resource

tv:multiple-menu-choose-menu-mixin — *Flavor*

This is the basic flavor that makes a window exhibit multiple-menu-choose behavior.

tv:pop-up-multiple-menu-choose-resource *Resource*

This is a *resource* of multiple-menu-choose menus.

20.4 Instantiable Multiple Menu Choose Flavors

tv:multiple-menu-choose-menu *Flavor*

This is the instantiable version of the multiple-menu-choose flavor, constructed by mixing **tv:multiple-menu-choose-menu-mixin** with **tv:menu.** It accepts the **:multiple-choose** message.

tv:pop-up-multiple-menu-choose-menu *Flavor*

This is a combination of **tv:multiple-menu-choose-menu-mixin** and **tv:pop-up-menu.** The arguments are the same as **tv:multiple-menu-choose-menu.** It accepts the **:multiple-choose** message.

20.5 tv:multiple-menu-choose-menu Example

Figure 14 shows an example of a momentary-multiple-item-list menu generated using the flavor **tv:multiple-menu-choose-menu.** The figure is followed by the code that generated the menu.

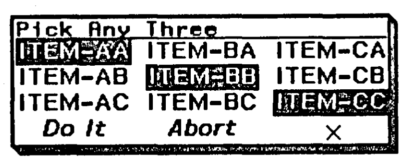


Figure 14. Momentary multiple-menu-choose menu.

```
;;; Multiple-menu-choose-menu Example

;;; Define the item list of lists
(setq items-3x3
        '((Item-AA Item-AB Item-AC)
          (Item-BA Item-BB Item-BC)
          (Item-CA Item-CB Item-CC)))

;;; Specify the default, highlighted items
(setq default-items '(Item-AA Item-BB Item-CC))
```

```
;;; Make the menu
(setq newer-menu
   (tv:make-window
            'tv:multiple-menu-choose-menu
            ':label
            '(:string "Pick Any Three"
              :character-style (:swiss :bold :normal))
            ':borders 2))

;;; Choose an item from each column; resultat holds result
(setq resultat
        (send newer-menu
                ':multiple-choose items-3x3 default-items))
```

21. The Multiple Choice Facility

The *Multiple Choice* facility produces a window containing several items, one per text line. For each item, there can be several yes/no choices for the user to make. For an example of a multiple-choice window, try selecting the [Kill or Save Buffers] operation in the Zmacs editor menu (see Fig. 15).

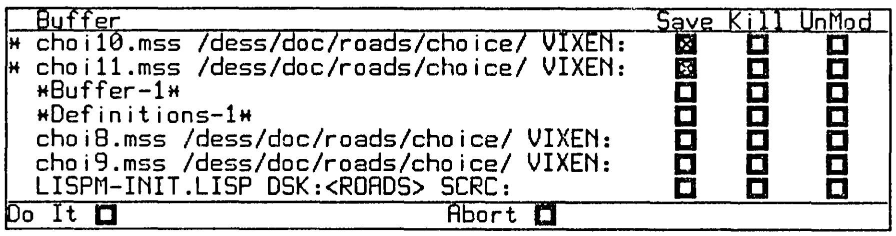


Figure 15. Multiple choice facility in the Zmacs menu.

Note that the window is arranged in columns, with headings at the top. The leftmost column contains the text naming each item. The remaining columns contain small boxes (called *choice boxes*). A "no" box has a blank center, while a "yes" box contains an "X".

Pointing the mouse at a choice box and clicking the left button complements its yes/no state. Each choice can be initialized by the program to yes or no as appropriate for a default set-up. Note that some items cannot allow some choices, so there can be blank places in the array of choice boxes.

There can be constraints among the choices for an item. For example, if they are mutually exclusive then clicking one choice box to "yes" automatically sets the other choice boxes on the same line to "no".

Several parameters are associated with a multiple-choice window:

- *Item-name* -- a string which is the column heading for the leftmost column.

- *Item-list* -- a list of representations of items. Each element is a list, (*item name choices*). *item* is any arbitrary object. *name* is a string which names that object; it is displayed on the left on the line of the display devoted to this item. *choices* is a list of keywords representing the choices the user can make for this item. Each element of *choices* is either a symbol, *keyword*, or a list, *(keyword default)*. If *default* is present and non-nil, the choice is initially "yes"; otherwise it is initially "no".

- *Keyword-alist* is a list defining all the choice keywords allowed. Each element takes the form (*keyword name*). *keyword* is a symbol, the same as in the *choices* field of an *item-list* element. *name* is a string used to name that keyword. It is used as the column heading for the associated column of choice boxes.

- An element of *keyword-alist* can have up to four additional list elements, called *implications*. These control what happens to other choices for the same item when this choice is selected by the user. Each implication can be **nil**, meaning no implication, a list of choice keywords, or **t** meaning all other choices.

  The first implication is *on-positive*; it specifies what other choices are also set to "yes" when the user sets this one to "yes."

  The second implication is *on-negative*; it specifies what other choices are set to "no" when the user sets this one to "yes."

  The third and fourth implications are *off-positive* and *off-negative*; they take effect when the user sets this choice to "no."

  The default implications are **nil t nil nil**, respectively. In other words the default is for the choices to be mutually exclusive. (If the implications are not specified, the defaults are **rplacd**'ed into the *keyword-alist* element by the system.)

- *Finishing-choices* -- the choices displayed in the bottom margin. When users click on one of these they are done. The variable **tv:default-finishing-choices** contains a reasonable pair of default finishing choices: [Do It] and [Abort].

21.1 The Standard Multiple Choice Function

This function interface to the multiple choice facility provides all the default values needed for a simple multiple choice menu.

tv:multiple-choose *item-name item-list keyword-alist* &optional *near-mode maxlines* — *Function*

This function pops up a multiple-choice window and allows the user to make choices with the mouse. The dimensions of the window are automatically chosen for the best presentation of the specified choices. If there are too many choices, scrolling of the window is enabled.

item-name, *item-list*, and *keyword-alist* are as described previously: See the section "The Multiple Choice Facility", page 293. The *finishing-choices*, [Do It] and [Abort], are prespecified by the system and cannot be changed by the user.

When the user clicks on one of the two finishing choices in the bottom margin ([Do It] and [Abort]), the window disappears and **tv:multiple-choose** returns. Two cases obtain:

- If the user finishes by choosing [Abort] the returned value is **nil**.
- If the user chooses [Do It], the returned value is a list with one element for each item. Each element is a list whose **car** is the *item* (that arbitrary object which the user passed in the *item-list* argument) and whose **cdr** is a list of the keywords for the "yes" choices selected for that item.

near-mode tells the window where to pop up. It is a suitable argument for **tv:expose-window-near**. The default is the list (**:mouse**). *maxlines*, which defaults to twenty, is the maximum number of choices allowed before scrolling is used.

21.2 tv:multiple-choose Menu Example

An example of a multiple-choice menu is shown in Fig. 16.

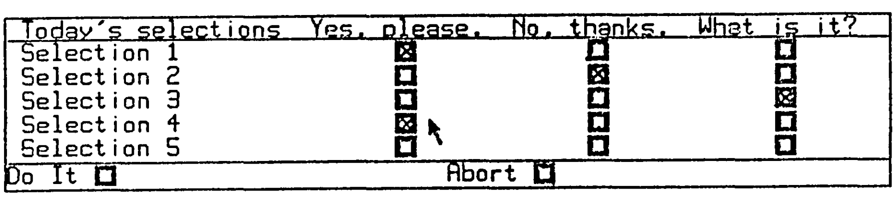


Figure 16. Multiple choice menu example.

The code to produce the multiple-choice menu in Fig. 16 follows.

```
;;; Multiple Choice Example

;;; These are the possible choices the user can make
(setq choices '(Yes No Explain))
```

```
(setq selection-item-list
  (list (list 1 " Selection 1" choices)
        (list 2 " Selection 2" choices)
        (list 3 " Selection 3" choices)
        (list 4 " Selection 4" choices)
        (list 5 " Selection 5" choices)))

;;; Set the choice boxes
(setq selection-keyword-alist
        (list '(Yes "Yes, please. ")
              '(No "No, thanks. ")
              '(Explain "What is it? ")))

;;;  Expose the menu,
(setq appetizer-order-list
        (tv:multiple-choose
             " Today's selections" selection-item-list
             selection-keyword-alist))
```

If a selection is made for each item, an example of the values assigned to the variable **appetizer-order-list** is the following:

```
((1 YES) (2 NO) (3 EXPLAIN) (4 NO) (5 NO))
```

If only one selection is made, the values assigned to the **appetizer-order-list** might look like this:

```
((1 YES) (2) (3) (4) (5))
```

21.3 The Basic Multiple Choice Flavor

The default multiple-choice facility described previously is useful for many applications, but sometimes more customization is desirable. The basic facilities provide many options, allowing you to tailor a multiple-choice menu to specific needs.

tv:basic-multiple-choice *Flavor*

The *basic* flavor that makes a window implement the multiple-choice facility. Like other basic flavors, it is not instantiable on its own but it does commit any window that incorporates it to being a multiple-choice window. **tv:basic-multiple-choice** is built out of **tv:text-scroll-window**.

21.4 Instantiable Multiple Choice Menu Flavors

tv:multiple-choice *Flavor*

An instantiable window flavor with the multiple-choice facility in it. It has borders and a label area on top which is used for the column headings.

tv:temporary-multiple-choice-window *Flavor*

This is a mixture of **tv:multiple-choice** and **tv:temporary-window-mixin**. Its behavior is that of a multiple-choice window that can be exposed and deexposed without deexposing the windows it covers up.

tv:temporary-multiple-choice-window &optional *(superior* **tv:mouse-sheet***)* *Resource*

This is a resource of temporary multiple-choice windows. It is used by the **tv:multiple-choose** function.

21.5 tv:multiple-choice Menu Messages

The following messages are useful to send to a multiple-choice window.

:setup *item-name keyword-alist finishing-choices item-list* &optional *maxlines* of **tv:multiple-choice** *Method*

This message sets up all the various parameters of the window. Usually one sends this message while the window is deexposed. The window decides what size it should be and whether all the items will fit or scrolling is required, then draws the display into its bit-array. Thus, when the window is exposed, the display appears instantaneously.

For an explanation of *item-name, keyword-alist,* and *finishing-choices,* See the section "The Multiple Choice Facility", page 293.

maxlines is the maximum number of lines the window can have; if there are more items than this only some of them are displayed and scrolling is enabled. *maxlines* defaults to 20.

:choose &optional *near-mode* of **tv:multiple-choice** *Method*

This message allows menu selection by the mouse. It first moves the window to the place specified by *near-mode*, which defaults to the list **(:mouse)**, (i.e., over the current mouse position) and exposes it. Then it waits for the user to make a finishing choice and returns the window to its original activate/expose status before the **:choose** operation. When it is sent to a multiple-choice menu, this message returns the same value as the function **tv:multiple-choose**. See the section "The Standard Multiple Choice Function", page 294.

21.6 tv:multiple-choice Example

This example shows how the **tv:multiple-choice** flavor can be used to define a multiple-choice menu.

```
;;; Specify the choice keywords
(setq choices '(Yes No))

;;; Set the choice boxes
(setq x-keyword-alist
        (list '(Yes "Yes")
              '(No "No")))

;;; Specify the item list
(setq x-item-list
  (list (list "Blue" "Blue" choices)
        (list "Red" "Red" choices)
        (list "Yellow" "Yellow" choices)
        (list "Green" "Green" choices)))

;;; Make the window
(setq x (tv:make-window 'tv:multiple-choice))

;;; Setup the window
(send p ':setup "Select Mode " x-keyword-alist
      tv:default-finishing-choices x-item-list)

;;; Expose the window and make a choice
(setq result (send p ':choose))
```

22. The Choose Variable Values Facility

The choose-variable-values facility is used throughout the Lisp Machine system software. The basic idea of choose-variable-values is to allow the user to interactively adjust the *value* of variables used in a program. (For an overview of related facilities intended for use with Dynamic Windows: See the section "Overview of Facilities for Accepting Multiple Objects" in *Programming the User Interface, Volume A*.)

More specifically, this facility displays a menu of names (standing for Lisp variables), followed by colons, and their values. After selecting a value with the left mouse button, users can interactively modify the value of the variable. Pressing the middle button preloads the input editor with the value of the variable, allowing the user to edit it. After the values are modified, the user can exit the menu.

For an example of a choose-variable-values window, try the [Edit Attributes] option of the System menu (see Fig. 17).

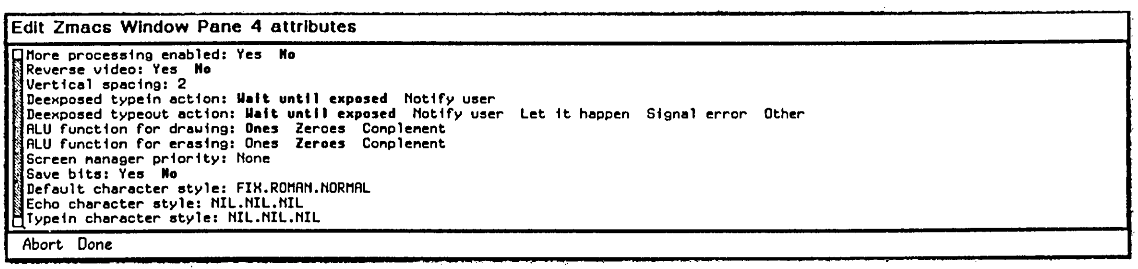


Figure 17. Choose-variable-values window accessed via the System menu.

22.1 Variables and Types

Each variable has a *type* that limits the values it can assume. The way the value is displayed and the way the user enters a new value depend on the type. The types fall into two categories:

Those with a small number of valid values.

Those with a large or infinite number of valid values.

The first category displays all the choices, with the current value of the variable in boldface. The second category displays the current value until it is selected, at which point the value disappears until the user types in a new value. If the user rubs out more characters than were typed in, the original value is restored.

Note that the type definition mechanism is extensible. You can define new types at any time. See the section "Defining Choose Variable Values Types", page 312.

All variables whose values are to be chosen must be declared **special**, so that they are represented by Lisp symbols and can be accessed non-locally to your program. (Note that the compiler automatically declares certain variables to be special. Good programming practice mandates that this should be done explicitly by the programmer.)

In most cases, the syntax for input and output is controlled by the binding of the Lisp system variables **zl:base**, **zl:ibase**, **zl:*nopoint**, **zl:prinlevel**, **zl:prinlength**, **zl:package**, and **zl:readtable**, as usual. However, the **:number**, **:number-or-nil**, **:integer**, and **:integer-or-nil** types take a **:base** parameter to specify the base for input and output. The default base is decimal.

Each line of the display is represented by an *item*, which can be one of the following:

String The string is displayed; strings are useful for putting headings and blank separating lines into the display.

Symbol The symbol is a variable whose type is **:sexp**; that is, its value can be any Lisp object. The name of the variable on the display is simply its print-name.

List in the form: (*variable name type args...*)

- *variable* is the object whose value is being chosen.
- *name* is optional; if it is omitted it defaults to the print-name of *variable*. If *name* is supplied it can be a string, which is displayed as the name of the variable, or it can be **nil**, meaning that this line should have no variable name, but only a value.
- *type* is an optional keyword giving the type of variable; if omitted it defaults to **:expression**.
- *args* are possible additional specifications dependent on *type*.

A list is the most general form of item. It is possible to omit *name* and supply *type* since *name* is always a string and *type* is always a symbol. For example, both of the following forms are valid item lists:

```
(base "Output Base" :integer)
```

and

```
(base :integer)
```

It is also possible to specify a locative in place of a variable. The value displayed and modified is the contents of the cell designated by the locative.

22.2 Predefined tv:choose-variable-values Variable Types

The following are the types of variables supported by default, along with any *args* that can be put in the item after the *type* keyword:

:boolean
The value of the variable is either **t** or **nil**. The choices are displayed as "Yes" for **t** and "No" for **nil**.

:inverted-boolean
The value of the variable is either **t** or **nil**. The choices are displayed as "Yes" for **nil** and "No" for **t**.

:expression
The value is any Lisp expression, read with **zl:read** and printed with **prin1**.

:sexp The same as **:expression**. This type is obsolete.

:princ The value is any Lisp expression, read with **zl:read** and printed with **princ**.

:eval-form
The value is the result of evaluating a Lisp form, read and evaluated with **zl:read-and-eval** and printed with **prin1**.

:choose *values-list print-function*
The value of the variable must be one of the elements of the list *values-list*. Comparison is by **zl:equal** rather than **eq**. All the choices are displayed, with the current value in boldface. A new value is entered by pointing to it with the mouse and clicking. *print-function* is the function to print a value; it is optional and defaults to **princ**.

:assoc *values-list print-function*
The displayed object is the **car** of one of the elements of *values-list*, while the **cdr** of the element is the value that goes in the variable.

print-function is the function to print a value; it is optional and defaults to **princ**.

:choose-multiple *values-list print-function*
This type takes arguments like the **:assoc** type, but permits the user to choose more than one element in the values list. The variable is set to a list of all the values chosen.

:menu-alist *item-list*
The items are specified in an *item-list*. See the section "Types of Menu Items", page 250. The usual menu mechanisms for specifying the string to display, the value to return, the function to call, and the mouse documentation work with this. **:menu-alist** is often used for its mouse documentation feature.

:character
The value is an integer that is a character code. It is printed as the character name (using the **~:@c zl:format** operator), and it is read as a single keystroke.

:character-or-nil
This is an integer like **:character**, but **nil** is also allowed as the value. **nil** displays as "none" and can be entered by pressing CLEAR-INPUT.

:string This value is a string, printed with **princ** and read with **zl:readline**.

:string-list
This value is a list of strings, whose printed representation for input and output consists of the strings separated by commas and optional spaces.

:string-or-nil
This value is a string or **nil** if the user just presses RETURN, LINE, or END.

:number :base *base* **:or-nil** *or-nil*
This value is a number. It is printed with **prin1** and read with **sys:read-number**. If **:base** is specified, the number is read and printed in base *base*. By default, the number is read and printed in decimal. If **:or-nil** is specified with a value other than **nil**, a value of **nil** is accepted when the user just presses RETURN, LINE, or END. **nil** displays as "none". The default for *or-nil* is **nil**.

:number-or-nil :base *base*
The same as **:number :base** *base* **:or-nil t**. This type is obsolete.

:decimal-number
The same as **:number :base 10.** This type is obsolete.

:decimal-number-or-nil
The same as **:number :base 10. :or-nil t**. This type is obsolete.

:integer :base *base* **:or-nil** *or-nil*
This value is an integer. It is printed with **prin1** and read with **sys:read-integer**. If **:base** is specified, the integer is read and printed in base *base*. By default, the integer is read and printed in decimal. If **:or-nil** is specified with a value other than **nil**, a value of **nil** is accepted when the user just presses RETURN, LINE, or END. **nil** displays as "none". The default for *or-nil* is **nil**.

:date This value is a universal date-time. An ambiguous date is interpreted as being in the future. (Compare this with **:past-date**.)

:date-or-never
This value is a universal date-time or **nil** if the user types "never". An ambiguous date is interpreted as being in the future.

:past-date
The value is a universal date-time. An ambiguous date is interpreted as being in the past.

:past-date-or-never
This value is a universal date-time or **nil** if the user types "never". An ambiguous date is interpreted as being in the past.

:time-interval-or-never
The value is an integer representing the number of seconds in a time interval, or **nil** if the user types "never". The interval is read and printed as either "never" or alternating numbers and units of time; the units can include seconds, minutes, hours, days, weeks, or years.

:time-interval-60ths
The value is an integer representing the number of sixtieths of a second in a time interval. The interval is read and printed as alternating numbers and units of time; the units can include seconds, minutes, hours, days, weeks, or years. The smallest unit read or displayed is second.

:pathname
The value is a pathname, represented as a string. The pathname read is merged with the result of **(fs:default-pathname)** and has a default version of **:newest**.

:pathname-or-nil
The value is a pathname, represented as a string, or **nil** if the user just presses RETURN, LINE, or END. The pathname read is merged with the result of **(fs:default-pathname)** and has a default version of **:newest**.

:pathname-list
The value is a list of pathnames, read as a series of pathnames separated by commas and optional spaces, and merged with the result of

(fs:default-pathname). The default version is **:newest**. The list is printed as a series of pathnames separated by commas and spaces.

:host The value is a network host, read and printed as the name of the host.

:host-or-local
The value is a network host. It is read as the name of a host or the string "local" to represent the local host. If the host is the local host, it is printed as "Local"; otherwise, it is printed as the name of the host.

:host-list
The value is a list of network hosts, read as a series of host names separated by commas or spaces, and printed as a series of host names separated by commas and spaces.

:pathname-host
The value is a pathname host, read and printed as the name of the host. The name can be "local", "sys", or the name of another logical host as well as the name of a physical host.

:keyword-list
The value is a list of symbols in the **keyword** package, read as a series of symbol names separated by commas or spaces, and printed as a series of symbol names separated by spaces. Symbol names are read and printed without package prefixes (that is, not preceded by colons).

:font-list
The value is a list of fonts, read as a series of font names separated by commas or spaces, and printed as a series of font names separated by commas and spaces. Font names are read and printed without package prefixes (that is, not preceded by **fonts:**).

A **:documentation** specification can be inserted where a variable type would normally be expected.

:documentation *doc type args...*
The actual type of the variable is *type*. *doc* is a string that is displayed in the mouse documentation line when the mouse is pointing at this item. The default, if no documentation is supplied using the **:documentation** specification, depends on the variable type. It is generally something like "Click left to input a new value from the keyboard".

22.2.1 The Optional Constraint Function

It sometimes is necessary to ensure that when one variable's value is changed, one or more of the others is changed as well. As an init-plist option, a choose-variable-values window can have an associated function, which is called whenever a variable's value is changed. This function can implement constraints among the variables.

The constraint function is specified by the **:function** init-plist option. See the section "**tv:choose-variable-values** Options", page 305. It is called with arguments *window*, *variable*, *old-value*, and *new-value*. The function should return **nil** if just the original variable needs to be redisplayed, or **t** if no redisplay is required; in this case it would usually **setq** several of the variables then send a **:refresh** message to the window to redisplay everything.

22.3 The Standard Choose Variable Values Function

The standard function interface to the choose-variable-values feature chooses the dimensions of the window and enables scrolling if there are too many variables to fit in the chosen height.

tv:choose-variable-values *variables* &rest *options* *Function*

This function exposes a window and displays the values of the specified variables, permitting the user to alter them. One or more choice boxes (as in the multiple-choice facility) appear in the bottom margin of the window. When the user clicks on the [Exit] choice box the window disappears and this function returns. The value returned is not meaningful; the result is expressed in the values of the variables.

variables is a list whose elements can be special variables or the more general items described above.

options is a list of alternating init-plist option keywords and values: See the section "**tv:choose-variable-values** Options", page 305.

22.4 tv:choose-variable-values Options

The following option keywords can be specified.

:label *string* (for **tv:choose-variable-values**) *Init Option*

The argument is a string that is the label displayed at the top of the window. The default is "Choose Variable Values".

:function *arg* (for **tv:choose-variable-values**) *Init Option*

Specifies the function to be called if the user changes the value of a variable. The default is **nil** (no function). See the section "The Optional Constraint Function", page 304.

:near-mode *arg* (for **tv:choose-variable-values**) *Init Option*
Specifies where to position the window. The default is the list **(:mouse)**. See the section "Input From Windows", page 147.

:width *arg* (for **tv:choose-variable-values**) *Init Option*
Specifies how wide to make the window. This can be a number of characters, or a string (it is made just wide enough to display that string). The default is to make it wide enough to display the current values of all the variables, provided that is not too wide to fit in the superior window.

:extra-width *arg* (for **tv:choose-variable-values**) *Init Option*
When **:width** is not specified, this specifies the amount of extra space to leave after the current value of each variable of the kind that displays its current value (rather than a menu of all possible values). This extra space allows for changing the value to something bigger. The extra space is specified as either a number of characters or a character string. The default is ten characters. If **:width** is specified, then **:extra-width** is ignored.

:margin-choices *list* (for **tv:choose-variable-values**) *Init Option*
The argument is a list of specifications for choice boxes to appear in the bottom margin. Each element can be a string, which is the label for the box that means "done," or a list containing a label string and a form to be evaluated if that choice box is clicked on. Since this form is evaluated in the user process it can do such things as alter the values of variables or **zl:*throw** out. With this facility, the default for **:margin-choices** is [Exit]. For an explanation of margin choices and their use: See the section "The Margin Choice Facility", page 333.

:superior *window* (for **tv:choose-variable-values**) *Init Option*
The argument is the window to which the pop-up choose-variable-values window should be inferior. The default is the value of **tv:mouse-sheet**, or the superior of *w* if the **:near-mode** option is already set to **(:window** *w***)**.

22.5 tv:choose-variable-values Examples

Here are some examples of how to call **tv:choose-variable-values**. The simplest kind of example is to display some variable names and values and let the user change them, as in Fig. 18. To see how it works, point at one of the variables, press the left mouse button, and then type in a new value and press `Return`. Recall that **zl:*nopoint** is a Lisp variable.

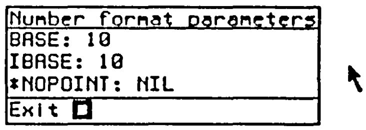


Figure 18. Choose-variable-values example 1.

The Lisp code used to produce Fig. 18 is shown here.

```
;;; Choose Variable Values Example 1

; Invoke the window
(tv:choose-variable-values '(base ibase *nopoint)
                            ':label "Number format parameters")
```

The same example can be done with better menu formatting in the next example (shown in Fig. 19).

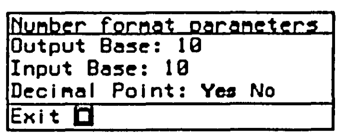


Figure 19. Choose-variable-values example 2: better formatting.

The Lisp code used to produce Fig. 19 is given here.

```
;;; Choose Variable Values Example 2

(tv:choose-variable-values
        '((base "Output Base" :number)
          (ibase "Input Base" :number)
          (*nopoint "Decimal Point"
                    :assoc (("Yes" . nil)
                            ("No" . t))))
        ':label "Number format parameters")
```

If we had not wanted to reverse the sense of **t** and **nil** the entry for **zl:*nopoint** would have been the following:

```
(*nopoint "No Decimal Point" :boolean)
```

If we wanted to use the name of the variable as the menu item, rather than spelling it out, we could have used the following expression:

```
(*nopoint :boolean)
```

As another example, we consider shopping for groceries via Lisp Machine. We have variables **fish, crustaceans, seafood-specialties, lettuce,** and **apples.** Many stores accept coupons for discounts on purchases, so the **coupon-value** variable (a floating-point number) allows users to enter a dollar value representing the value of the coupons they are redeeming.

As mentioned, clicking [Middle] on the mouse puts the variable in the input editor, allowing you to make changes in it. In Fig. 20 we display this situation and allow it to be modified, using several different kinds of items:

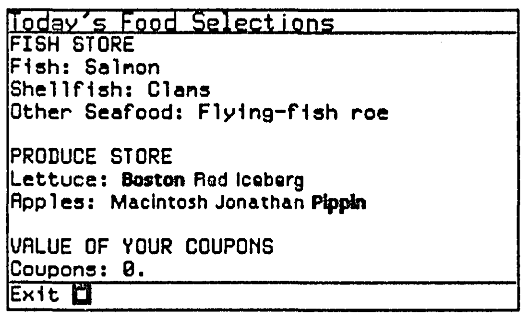


Figure 20. Choose-variable-values window: grocery store example.

The Lisp code used to produce Fig. 20 is provided next. Each "STORE" in the example is implemented with a different variation of the choose variable value facility. Note the use of strings to provide labels for the sections, and null strings to separate the sections with blank lines.

```
;;; Choose Variable Values Example 3

;;; Set up the variables
(setq fish '("Salmon"))
(setq crustaceans '("Clams"))
(setq seafood-specialties '("Flying-fish roe"))
(setq lettuce "Boston")
(setq apples "Pippin")
(setq Coupon-value 0)
```

```
(setq result (tv:choose-variable-values
   '("FISH STORE"
     (fish "Fish" :string-list)
     (crustaceans "Shellfish" :string-list)
     (seafood-specialties "Other Seafood" :string-list)
     ""
     "PRODUCE STORE"
     (lettuce "Lettuce" :choose ("Boston" "Red" "Iceberg"))
     (apples "Apples" :choose ("Macintosh" "Jonathan" "Pippin"))
     ""
     "VALUE OF YOUR COUPONS"
     (Coupon-value "Coupons"
        :documentation
          "Click left to enter the value of your coupons."
          :number))
     ':label "Today's Food Selections"))
```

22.6 The User Option Facility

The user option facility provides a simple window interface that allows you to set parameter options to your programs. The user option facility is based on the choose-variable-values facility.

A typical use would be in a program that requires several variables to be set before it is run. In a conventional system, a standard way to alter these values would be to alter the code, recompile the program, and then run it. By contrast, the user option facility generates a window with the names and default values of the variables. This gives you the option of resetting these variables before execution of the program. When the window is exited, the rest of the program runs.

For an example of a user option window, type the following function at a Lisp Listener window:

```
(choose-user-options zwei:*zmail-user-option-alist*)
```

The **choose-user-options** function is also used by the Zmail Profile mode, and elsewhere throughout the system.

Special forms are provided for defining options, and the **choose-user-options** function exists for putting all the options into a choose-variable-values window so that the user can alter them. In addition, the current state of the options can be written into an initialization file, or all the options can be set to their default initial values.

22.6.1 Functions for Defining User Option Variables

define-user-option-alist *name [constructor]* *Special Form*

(define-user-option-alist *name***)** defines *name* to be a global variable whose value is a "user option alist", something which may be used by the other functions below. This alist keeps track of all of the option variables for a particular program.

(define-user-option-alist *name constructor***)** also specifies the name of a constructor macro to be defined, which provides a slightly different way of defining an option variable from **define-user-option.** The form (*constructor option default type name*) defines an option in this user-option-alist. The arguments are the same as to **define-user-option.**

define-user-option *(option alist) default [type] [name]* *Special Form*

(define-user-option (*option alist***)** *default type name***)** defines the special variable *option* to be an option in the *alist*, which must have been previously defined with **define-user-option-alist.** The variable is declared and initialized via **(defvar** *option default***).** The value of the form *default* is remembered so that the variable can be reset back to it later.

type is the type of the variable for purposes of the choose-variable-values facility. It is optional and defaults to **:sexp.**

name is the name of the variable to be displayed in the choose-variable-values window. It is optional and defaults to a string that is the print-name of the variable except with hyphens changed to spaces and each word changed from all-upper-case to first-letter-capitalized. If the first and last characters of the print-name are asterisks, they are removed. For example, the default name for **so:*sunny-side-up*** would be **"Sunny Side Up"**.

22.6.2 Functions for Altering User Option Variables

choose-user-options *alist* &rest *options* *Function*

This function displays the values of the option variables in *alist* to the user and allows them to be altered. The *options* are passed along to **tv:choose-variable-values.**

reset-user-options *alist* *Function*

This function resets each of the option variables in *alist* to its default initial value.

write-user-options *alist stream* *Function*

This function specifies that for each option variable in *alist* whose current value is not **zl:equal** to its default initial value, a form is printed to *stream* which sets the variable to its current value. The form uses **zl:login-setq** so it is appropriate for putting into an initialization file.

22.7 User Options Example

Fig. 21 is an example of a user option window that sets three variables of a simple graphics program.

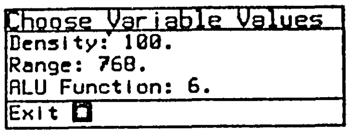


Figure 21. User options window example.

The Lisp code used to produce Fig. 21 is shown between the asterisk-marked (****) lines. The rest of the code generates the graphics.

```
;;; User Option Example

;;;****
;;; This names the user option alist
(define-user-option-alist options)

;;; These expressions set of the options
(define-user-option (alu-function options)
       tv:alu-ior :decimal-number "ALU Function")
(define-user-option (range options) 768. :decimal-number "Range")
(define-user-option (density options) 100. :decimal-number "Density")

;;; Expose the choose-option window
(choose-user-options options)
;;;****
```

```
;;; This is a random line-drawing function
(defun image (alu-function range density)
 (setq x (tv:make-window 'tv:window))
 ;; Temporarily select a window; the arguments
 ;;  are the window x and the final action on it
 (tv:window-call (x :deactivate)
  (setq n range)
  (loop for i below density do
        (send x ':draw-lines alu-function
              (random n) (random n) (random n) (random n)
              (random n) (random n) (random n) (random n))
        (send x ':draw-circle
              (random n) (random n) (random n)))
  (send x ':tyi)))

;;; Draw the image
(image alu-function range density)
```

22.8 Defining Choose Variable Values Types

The standard choose-variable-values facility supplies programmers with a range of predefined types. See the section "Predefined **tv:choose-variable-values** Variable Types", page 301. However, this list is extensible through two mechanisms:

1. Adding a type keyword property to a new type name

2. Adding a type decoding method

22.8.1 Adding a Type Keyword Property

The basic type definition mechanism is simple: put a **tv:choose-variable-values-keyword** property on the type name. In the following example, the new type is called **new-type**, the property value is *type-list*, and the property name is **tv:choose-variable-values-keyword.**

```
(defprop new-type type-list tv:choose-variable-values-keyword)
```

For a discussion of the contents of *type-list*: See the section "Elements of The **tv:choose-variable-values-keyword** Property", page 313. See the section "Type Decoding Message", page 313.

22.8.2 Adding a Type Decoding Method

The second way to extend the range of standard types is to define a new flavor of choose-variable-values window and give it a **:decode-variable-type** method -- circumventing the use of the standard variable types. This method must be careful to implement the **:documentation** keyword, which can appear in an item where a variable type would normally appear.

22.9 Type Decoding Message

:decode-variable-type *kwd-and-args* of **tv:basic-choose-variable-values** *Method*

The system sends this message to a choose-variable-values window when it needs to understand an item. *kwd-and-args* is a list whose **car** is the keyword for the item and whose remaining elements, if any, are the arguments to that keyword. Six values are returned. The default method for **:decode-variable-type** looks for two properties on the keyword's property list:

- **tv:choose-variable-values-keyword** -- The value of this property is a list of six values. See the section "Elements of The **tv:choose-variable-values-keyword** Property", page 313. Unnecessary values of **nil** may be omitted at the end.

- **tv:choose-variable-values-keyword-function** -- The value of this property is a function that is called with one argument, *kwd-and-args*. The function must return the six values.

22.9.1 Elements of The tv:choose-variable-values-keyword Property

The six elements of the **tv:choose-variable-values-keyword** property are listed below. Note that if the specified list is shorter than six elements, the others default to **nil**.

print-function
: A function of two arguments, *object* and *stream,* to be used to print the value. **prin1** is acceptable.

read-function
: A function of one argument, a *stream,* to be used to read a new value. **zl:read** is acceptable. If **nil** is specified, there is no read-function and instead new values are specified by pointing at one choice from a list. If the *read-function* is a symbol, it is called inside an input editor, and over-

rubout automatically leaves the variable with its original value. If *read-function* is a list, its **car** is the function, and it is called directly rather than inside an input editor.

choices A list of the choices to be printed, or **nil** if just the current value is to be printed.

print-translate
: If there are choices, and this function is supplied non-**nil**, it is given an element of the choice list and must return the value to be printed (for example, **car** for **:assoc** type items).

value-translate
: If there are choices, and this function is supplied non-**nil**, it is given an element of the choice list and must return the value to be stored in the variable (for example, **cdr** for **:assoc** type items).

documentation
: A string to display in the mouse documentation line when the mouse is pointing at this item. This string should tell the user that clicking the mouse changes the value of this variable, and any special information (for example, that the value must be a number).

Alternatively, the documentation element can be a symbol that is the name of a function. It is called with one argument, which is the current element of *choices* or the current value of the variable if *choices* is **nil**. It should return a documentation string or **nil** if the default documentation is desired. This can be useful when you want to document the meaning of a particular choice, rather than simply saying that clicking on this choice selects it.

Note that the function should return a constant string, rather than building one with **zl:format** or other string operations. This is because it will be called over and over as long as the mouse is pointing at an item of this type. (The function is called by the mouse documentation line updating in the scheduler, not in the user process.)

22.10 tv:choose-variable-values Type Definition Example

```
;;; Defining a Choose Variable Values Type Example
;;;  Adding the type keyword property

(defvar candidate-1 nil)
(defvar candidate-2 nil)
(defvar candidate-3 nil)
```

```
;;; Set up the type list
(setq type-list '(princ nil ("Yes" "No" "Abstain") nil nil nil))

;;; Put the type-list value on the
;;;  tv:choose-variable-values-keyword property
(putprop 'mytype type-list
   'tv:choose-variable-values-keyword)

;;; Use the newly created type
(tv:choose-variable-values
  '((candidate-1 " John Q. Public " mytype)
    (candidate-2 " Jane Doe " mytype)
    (candidate-3 " John Blevins " mytype))
  ':label "*** Select  One  Candidate ***")
```

22.11 Defining a Choose Variable Values Window

Up to this point, an easy-to-use but limited form of the choose-variable-values facility has been discussed, namely, the standard **tv:choose-variable-values** function.

In order to create a new flavor of window with choose-variable-values behavior, the *basic* and *instantiable* choose-variable-values window flavors are needed. These are described in this section. The basic flavor requires more parameter specifications from the programmer, but it is also the most flexible. The use of choose-variable-values windows as panes in a frame and as pop-up windows is also discussed.

22.12 The Basic Choose Variable Values Flavor

tv:basic-choose-variable-values *Flavor*

This is the *basic* flavor which makes a window implement the choose-variable-values facility. It is built out of **tv:text-scroll-window**. There are two ways to use this. In the first way, the programmer creates a window giving all of the parameters in the init-plist. In the second way one can create a window without specifying the parameters, then send the **:setup** message to start the display.

22.12.1 Instantiable Choose Variable Values Flavors

tv:choose-variable-values-window *Flavor*

This is a choose-variable-values window with a reasonable set of features, including borders, a label at the top, stream input/output, the ability to be scrolled if there are too many variables to fit in the window, and the ability to have choice boxes in the bottom margin.

tv:choose-variable-values-pane *Flavor*

This is a **tv:choose-variable-values-window** that can be a pane of a constraint-frame. For more on constraint frames: See the section "Specifying Panes and Constraints", page 208. It does not change its size automatically; the size is assumed to be controlled by the superior.

tv:temporary-choose-variable-values-window *Flavor*

This is a **tv:choose-variable-values-window** that is exposed temporarily. For an explanation of temporary windows: See the section "Temporary Windows", page 95.

22.12.2 I/O Buffers for Choose Variable Values Windows

I/O buffers can be associated with choose-variable-values windows. See the section "Menu Items and Menu Values", page 271. A choose-variable-values window has an I/O buffer, which the window uses to send commands (also known as *blips*) back to its controlling process. As usual these commands are lists, to distinguish them from keyboard characters that are numbers. If all panes send commands to the same I/O buffer, then when one of these commands arrives it can be processed in the appropriate pane. At the same time, the controlling process can be looking in the I/O buffer for other commands from other panes and for input from the keyboard. A choose-variable-values window uses the same I/O buffer to read a new value from the keyboard as it uses to send blips to the controlling process.

The following I/O buffer commands (blips) are sent by the choose-variable-values window to the user process.

(**:variable-choice** *window item value line-number mouse-gesture*)

This indicates that the user clicked on the value of a variable, expressing a desire to change it. *window* is the choose-variable-values window instance, *item* is the complete item specification, *value* is the value that was clicked on, and *line-number* is the line on which the item appears in the menu; the lines are numbered starting at 0. *mouse-gesture* is the mouse character (for example, #**mouse-m**) corresponding to the gesture used for clicking.

(**:choice-box** *window box*)

This indicates that the user clicked on one of the choice boxes in the bottom margin. *window* is the window instance, and *box* is the choice box specification.

The following sequence of events is a typical model for implementing a choose-variable-values window.

1. Set up and expose the window.

2. Loop within an **:any-tyi**, or **tv:io-buffer-get** loop, checking to see if a variable-choice or a choice-box selection has been made.

3. If a choice-box selection has been made, your "choice-box handler" routine is called. This routine returns the choice-box descriptor. If the choice-box was an [Abort] item, your process typically sends the window the **:deactivate** message.

tv:choose-variable-values-process-message *window command* *Function*

This function implements the proper response to the above commands. It should be called in the process and stack-group in which the variables being chosen are bound. The function returns **t** if the command indicates that the choice operation is "done", otherwise it performs the appropriate special action and returns **nil**. If *command* is a character, it is ignored unless it is the **#\refresh** key, in which case the choose-variable-values window is refreshed.

tv:temporary-choose-variable-values-window &optional *(superior* **tv:mouse-sheet***)* *Resource*

A resource of windows, from which **tv:choose-variable-values** gets a window to use.

22.13 tv:basic-choose-variable-values Init-plist Options

The following init-plist options are relevant to choose-variable-values windows. Note that if no dimensions are specified in the init-plist, the width and height are automatically chosen according to the other init-plist parameters. The height is dictated by the number of elements in the *item-list*. Specifying a height in the init-plist, using any of the standard dimension-specifying init-plist options, overrides the automatic choice of height. *Note:* the **:stack-group** option is required, unless the **:setup** message is used to initialize the window. See the section "**tv:choose-variable-values-window** Messages", page 319.

:function *function* (for **tv:basic-choose-variable-values**) *Init Option*

Specifies the function called when the value of a variable is changed. See the section "The Optional Constraint Function", page 304. The default is **nil** (no function).

:variables *item-list* (for **tv:basic-choose-variable-values**) *Init Option*
Specifies the list of variables whose values are to be chosen. These can be either symbols that are variables, or the more general *items* defined previously. See the section "Variables and Types", page 299.

:stack-group *sg* (for **tv:basic-choose-variable-values**) *Init Option*
This option specifies the stack group in which the variables whose values are to be chosen are bound. The window needs to know this so that it can get the values while running in another process, for instance the mouse process, in order to update the window display when it is refreshed or scrolled. This option is required, unless you use the **:setup** message.

:name-style *character-style* (for **tv:basic-choose-variable-values**) *Init Option*
This specifies the character style in which names of variables are displayed. The default is the system default character style.

:value-style *character-style* (for **tv:basic-choose-variable-values**) *Init Option*
This is the character style in which values of variables are displayed. The default is the system default character style.

:string-style *character-style* (for **tv:basic-choose-variable-values**) *Init Option*
This is the character style in which items that are just strings (typically heading lines) are displayed. The default is the system default character style.

:unselected-choice-style *character-style* (for **tv:basic-choose-variable-values**) *Init Option*
This option determines the character style in which choices for a value, other than the current value, are displayed. The default is a small distinctive character style.

:selected-choice-style *character-style* (for **tv:basic-choose-variable-values**) *Init Option*
This specifies the character style in which the current value of a variable is displayed, when there is a finite set of choices. This should be a bold-face version of the preceding character style. The default is the bold-face version of the default unselected-choice character style.

:margin-choices *choice-list* (for **tv:choose-variable-values-window**) *Init Option*
The default is a single choice box, labelled [Done]. For an explanation of the choice-box descriptors: See the section "The Margin Choice Facility", page 333. Note that specifying **nil** for this option suppresses the margin-choices entirely.

:io-buffer *buf* (for **tv:choose-variable-values-window**) *Init Option*

This specifies the I/O buffer to be used. The buffer can be associated with another window or it can be explicitly created for this window with the **tv:make-io-buffer** function. The I/O buffer is used both for reading keyboard input (new values) and for sending blips to the controlling process.

22.14 tv:choose-variable-values-window Messages

The following messages are useful to send to a choose-variable-values window.

:setup *items label function margin-choices* of **tv:choose-variable-values-window** *Method*

This changes the list of items (variables), the window label, the constraint function, and the choices in the bottom margin and sets up the display. This message remembers the current stack-group as the stack-group in which the variables are bound. If the window is not exposed this chooses a good size for it.

:set-variables *item-list* &optional *dont-set-height* of **tv:choose-variable-values-window** *Method*

This changes the list of items (variables) and redisplays. Unless *dont-set-height* is supplied non-**nil**, the height of the window is adjusted according to the number of lines required. If more than 25. lines would be required, 25. lines are used and scrolling is enabled. The **:setup** message uses **:set-variables** to do part of its work.

:appropriate-width &optional *extra-space* of **tv:choose-variable-values-window** *Method*

This returns the inside-width appropriate for this window to accommodate the current set of variables and their current values. Send this message after a **:setup** and before a **:expose**, and use the result to send an **:adjust-geometry-for-new-variables** message. The returned width is not larger than the maximum that fits inside the superior.

If *extra-space* is supplied, it specifies the amount of extra space to leave after the current value of each variable of the kind that displays its current value (rather than a menu of all possible values). This extra space allows for changing the value to something bigger. The extra space is specified as either a number of characters or a character string. The default is to leave no extra space.

:adjust-geometry-for-new-variables *width* of **tv:choose-variable-values-window** *Method*

The variable *width* is specified as **nil** if the size is not to be adjusted, otherwise the inside-width and height are also adjusted. The **:adjust-geometry-for-new-variables** message is normally sent after sending a **:setup** message. (It is not necessary to send it after a **:set-variables** message.)

:redisplay-variable *variable* of **tv:choose-variable-values-window** *Method*

This redisplays just the value of the specified variable.

22.15 tv:choose-variable-values-window Example

As we have discussed, in the simplest mode of operation, the **tv:choose-variable-values** function takes care of creating the window and establishes all necessary communication with it. When you make a choose-variable-values window (as in the example below), you need to handle the communication yourself, using the information given below. An example of a situation in which this is necessary is when you have a frame, some panes of which are choose-variable-values windows.

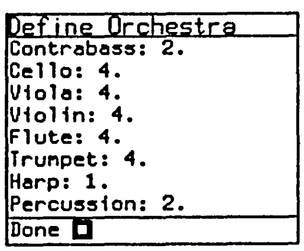


Figure 22. Example of making a choose-variable-values menu.

The Lisp code used to generate Fig. 22 is given next.

```
;;; Choose Variable Values Example 4

;;; In this example, the user specifies the number of
;;;  instrumentalists of each kind needed to define an orchestra.
```

```
(defvar contrabass 2)
(defvar cello 2)
(defvar viola 4)
(defvar violin 4)
(defvar flute 4)
(defvar trumpet 2)
(defvar harp 1)
(defvar percussion 2)

;;;; Define the variable list
(defvar instrument-list
          '((contrabass "Contrabass" :number)
            (cello "Cello" :number)
            (viola "Viola" :number)
            (violin "Violin" :number)
            (flute "Flute" :number)
            (trumpet "Trumpet" :number)
            (harp "Harp" :number)
            (percussion "Percussion" :number)))
;;; Define the margin choice list
(defvar margin-list '(("Done" nil
       tv:choose-variable-values-choice-box-handler nil nil)))

;;; Make the window
(defvar choix
        (tv:make-window 'tv:choose-variable-values-window))
```

```
;;; This function sets up the window, exposes it,
;;;  and calls appropriate routines
(defun display ()
        (let ((base 10.) (ibase 10.)) ; Set the base to 10
          (send choix ':setup
                        instrument-list
                        "Define Orchestra"
                        nil
                        margin-list)
          ;; The :setup message is normally followed by the
          ;;  :adjust-geometry-for-new-variables message in order
          ;;  to coordinate the size of the window with the number
          ;;  of variables. The numerical argument (180.) tells
          ;;  it to adjust the width of the window to the precise
          ;;  size I want it to be. I could also have sent
          ;;  the :appropriate-width message.
          (send choix ':adjust-geometry-for-new-variables 180.)
          (send choix ':set-position 200. 200.)
          (tv:window-call (choix :deactivate)
            ;; blip holds the list returned by :any-tyi
            ;;  Look for a :choice-box blip
            (loop as blip = (send choix ':any-tyi)
                  until (eq (car blip) ':choice-box)
                  do (tv:choose-variable-values-process-message
                          choix blip)))))
```

In order to invoke this menu, type the following form at the Lisp input editor:

```
(display)
```

The results are stored in **contrabass, cello, viola,** and the other instrument variables.

23. The Mouse-Sensitive Items Facility

The mouse-sensitive items facility is related to certain choice facilities such as the pop-up menus described previously. Like these facilities, the mouse is used to point at an object on the screen, and a box is drawn around an object when the mouse is over it. (Mouse sensitivity is a basic feature of Dynamic Windows and the presentation-type system. For an introduction to these facilities: See the section "Introduction to the User Interface Management System" in *Programming the User Interface, Volume A.*)

In contrast to a menu, in which mouse-sensitive behavior is limited to a relatively permanent item list, mouse-sensitive items are not a permanent part of a window. They disappear if the screen is cleared, for example. A main feature of a mouse-sensitive window is that graphical objects and text can be intermingled. The graphical objects themselves can be made mouse-sensitive. See the section "Mouse-Sensitive Areas Example", page 330.

For an example of mouse-sensitive items, try the [List Buffers] command in the Zmacs editor command menu (Figure 23). Move the mouse over the list of buffers and click the right-hand button. Another menu, keyed from a mouse-sensitive-item, is exposed.

```
Buffers in Zmacs:
  Buffer name:                                   File Version:     Major mode:

* choi9.mss /dess/doc/roads/choice/ VIXEN:                           (Text)
  choi10.mss /dess/doc/roads/choice/ VIXEN:                          (Text)
  choi7.mss /dess/doc/roads/choice/ VIXEN:                           (Text)
  choi8.mss /dess/doc/roads/choice/ VIXEN:                           (Text)
  choi5.mss /dess/doc/roads/choice/ VIXEN:                           (Text)
  choi4.mss /dess/doc/roads/choice/ VIXEN:                           (Text)
  chroot.mss /dess/doc/roads/choice/ VIXEN:                          (Text)
  choi3.mss /dess/doc/roads/choice/ VIXEN:                           (LISP)
  *Buffer-1*                                     [1 line]          (Fundamental)
```

Figure 23. Mouse-sensitive items.

Mixing **tv:basic-mouse-sensitive-items** into a window flavor equips the window with mouse-handling according to the paradigm described in this section. Mouse-sensitive items are something you add in when defining your own window, rather than a complete facility. Consequently, there is no instantiable version.

Note: The word "typeout" appears here and there in the mouse-sensitive items facility for historical reasons. Often mouse-sensitive items are typed out on top of some other display, such as an editor buffer. However, the mouse-sensitive-item facility has nothing to do with the *typeout-window* facility. See the section "Typeout Windows", page 203.

tv:basic-mouse-sensitive-items *Flavor*

Mixing this flavor into a window provides for areas of the screen that are sensitive to the mouse. Moving the mouse into such an area highlights the area by drawing a box around it. At this point clicking the mouse performs a user-defined operation. This flavor is called *basic* because it usurps the handling of the mouse by the window; do not mix it with another flavor that also expects to use the mouse. However it is less basic than many basic flavors in that it does not do anything special with the displayed image of the window.

23.1 Attributes of a Mouse-sensitive Item

A mouse-sensitive item has three main attributes:

- A *type* -- a keyword that controls what you can do to it
- An *item* -- an arbitrary Lisp object associated with it
- A *rectangular area* of the window -- typically something is displayed in that area at the same time as a mouse-sensitive item is created, using normal stream output to the window.

Unlike things such as menu items, mouse-sensitive items are not a permanent property of the window. They are just as ephemeral as the displayed text. This means they go away if you clear the window or if typeout wraps around and types over them.

23.2 Associating Actions with Mouse-sensitive Items

The **:item-type-alist** init-plist option specifies an alist that associates actions with types of items. Each element of the list contains the following elements:

- A *type keyword* -- for example, **:value**
- A *default operation* -- for example, a function name
- A *documentation string* -- displayed in the mouse documentation line when the mouse is pointing at an object of this type

- A *list of all the operations* -- (the default doesn't necessarily have to be a member of this list) This list is in the form of menu items, so typically each element is (*name* . *operation*) where the user sees the string *name* but the program identifies the operation by the symbol *operation*. In most cases *operation* is a function to be called, but it can be any atom.

Here is an example of an item-type-alist:

```
((zwei:file
   zwei:find-defaulted-file
   "Left: Find file this file.  Right: menu of Load, Find, Compare."
   ("Load" :value zwei:load-defaulted-file
    :documentation "Load this file.")
   ("Find" :value zwei:find-defaulted-file
    :documentation "Find file this file.")
   ("Compare" :value zwei:srccom-file
    :documentation "Compare file with newest version (srccom)."))
 (zwei:function-name
   zwei:edit-definition
   "Left: Edit function. Right: menu (Arglist, Edit, Disassemble, Document.)."
   ("Arglist" :value zwei:typeout-menu-arglist
    :documentation "Print arglist for this function.")
   ("Edit" :value zwei:edit-definition
    :documentation "Edit this function.")
   ("Disassemble" :value zwei:do-disassemble
    :documentation "Disassemble this function.")
   ("Documentation" :value zwei:typeout-long-documentation
    :documentation "Print long documentation for this function.")))
```

The **tv:item-type-alist** instance-variable can be initialized via the init-plist when the window is created. Normally, you do not create this alist directly. Instead, you use **tv:add-typeout-item-type** to build it up incrementally. See the section "**tv:basic-mouse-sensitive-items** Messages and Functions", page 327.

23.2.1 Mouse Behavior

The mouse works with a mouse-sensitive item in the following manner:

- Mouse-left -- Perform the default operation

- Mouse-right -- Pop up a menu of all the operations. Selecting one of these items performs it.

- Mouse-right-twice -- Call the System menu.

- Other mouse clicks and clicking on an item whose type is not in the type alist -- Cause a beep (the screen flashes) and generate an error.

Performing an operation means that a command (also known as a *blip*)is sent to the controlling process through the **:force-kbd-input** message to the window. This command is a list (**:typeout-execute** *operation item*), where *operation* is the operation and *item* is the arbitrary object remembered by the mouse-sensitive item. The ramifications of this, and how the *operation* is performed, are up to the application program.

tv:add-typeout-item-type *Special Form*

The following special form is used to declare information about a mouse-sensitive type by adding an entry to an alist kept in a special variable.

(tv:add-typeout-item-type
alist type name operation default-p documentation)

This alist can be put into the item-type alist of a mouse-sensitive window, using, for instance, the **:item-type-alist** init-plist option. Note that each possible operation on a particular mouse-sensitive item type is defined with a separate **tv:add-typeout-item-type** form; this allows each operation to be defined at the place in the program where it is implemented, rather than collecting all the operations into a separate table. It also allows new operations to be added in a modular fashion.

alist is the special variable that contains the alist. You should declare it **nil** with **defvar** before defining the first item type. Each program that uses mouse-sensitive items has its own alist of item types, so that there is no conflict in the names of the types.

type is the keyword symbol for the type being defined.

name is the string that names the operation.

operation is the action to be taken, for instance, the function to be called.

default-p is optional; if it is supplied and non-**nil**, it means that this operation is the default performed when you click the left button on an item of this type.

documentation is optional but highly recommended; it is a string that documents what *operation* does. When the user points the mouse at an item of this type, the documentation line at the bottom of the screen displays the documentation for the default operation (reachable by the left button) and a list of the operations in the menu (reachable by the right button). If the user clicks right, calling for a menu, then the screen displays documentation for the operation pointed at.

alist, *type*, and *operation* are not evaluated. *name*, *default-p*, and *documentation* are evaluated.

When *operation* is a function, the **tv:add-typeout-item-type** form is typically placed near the definition of the function in the program source file.

23.3 tv:basic-mouse-sensitive-items Init-plist Options

:item-type-alist *alist* (for **tv:basic-mouse-sensitive-items**) *Init Option*

Remembers *alist* as the set of item types allowed in this window. *alist* should be created by **tv:add-typeout-item-type**.

23.4 tv:basic-mouse-sensitive-items Messages and Functions

The following messages are useful to send to a window with mouse-sensitive items. To create and display a list of mouse-sensitive items, use the function **si:display-item-list**.

:item *type item* &rest *format-args* of **tv:basic-mouse-sensitive-items** *Method*

This creates and displays a mouse-sensitive item of type *type* with associated object *item*. If *format-args* are supplied, they are a **zl:format** control-string and arguments used to generate the display for this item. If *format-args* are not supplied, the display is generated with **princ**.

:primitive-item *type item left top right bottom* of **tv:basic-mouse-sensitive-items** *Method*

This is the primary means for creating a mouse-sensitive-area of the screen. It creates a mouse-sensitive item of type *type* with associated object *item*. When the mouse moves into the area, a box is overlaid around it. *left*, *top*, *right*, and *bottom* are the coordinates of a rectangular area of the window assumed to contain the display. The coordinates are "inside" coordinates. This is the same coordinate system that **:read-cursorpos** uses.

si:display-item-list *stream type list* &optional *item-string* (*order-columnwise* **t**) *Function*

Displays a list of items on *stream* in evenly spaced columns. *stream* must be interactive. If it supports mouse sensitivity, the items displayed are also made mouse sensitive.

list is a list of items to be displayed. Each item in the list is displayed by sending the stream an **:item** message with *type* as the first argument. If the item is not itself a list, the item is the second argument to the **:item** message.

If the item to be displayed is a list, the arguments to the **:item** message depend on *item-string*. If *item-string* is not **nil**, the second argument to the **:item** message is the first element of the item. If *item-string* is **nil**, the item should be an alist whose car is a string to be displayed and whose cdr is the item itself. In this case, the second argument to the **:item** message is the cdr of the item, the third argument is "~**a**", and the fourth argument is the car of the item. The default for *item-string* is **nil**.

If *order-columnwise* is not **nil**, the items are ordered down columns. If *order-columnwise* is **nil**, the items are ordered across rows. The default is **t**.

23.5 tv:basic-mouse-sensitive-items Example

An example of a mouse-sensitive items window is shown in Figure 24. It shows four mouse-sensitive items in a window. One of the items has been selected. Some graphic figures (not mouse-sensitive) have also been drawn in the window. For a description of the graphics operations: See the section "Graphic Output to Windows", page 132.

The point of this figure is to show how in mouse-sensitive windows (unlike in regular menus) graphics and text can be intermingled. Notice the technique of combining the mixin flavors **tv:borders-mixin** and **tv:top-box-label-mixin** before **tv:window** to generate the boxed-in label at the top of the window.

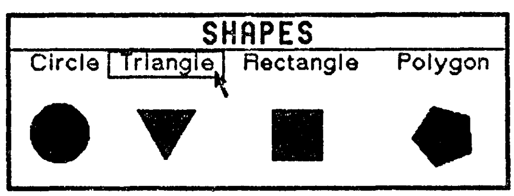


Figure 24. Mouse-sensitive items example.

In Figure 25 one of the items [Triangle] has been selected, causing a menu of alternative actions to the the default action (default function) to appear next to it.

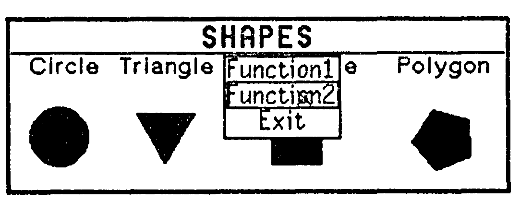


Figure 25. Result of selecting a mouse-sensitive item.

The Lisp code used to produce Figure 25 is listed next.

```
;;; Mouse-sensitive Example

;;; The functions called by the menus do nothing except increment
;;;  some values. Check their values after instantiating the
;;;  window to verify that the values were incremented. Also
;;;  look at the value of the variable "blip".


;;; Initialize variables
(defconst c1 0)
(defconst c2 0)
(defconst default 0)
(defvar alist-alpha nil)

;;; Define a new flavor of window, with a
;;;  centered top-label and a mouse-sensitive-item mixin
(defflavor new ()
           (tv:centered-label-mixin
            tv:borders-mixin tv:top-box-label-mixin
            tv:basic-mouse-sensitive-items
            tv:window))

;;; These define mouse-sensitive items
(tv:add-typeout-item-type alist-alpha
                          :new-type "Exit" (exit)
                          nil "Exit and kill window")

(tv:add-typeout-item-type alist-alpha
                          :new-type "Function2" (function2)
                          t "Add one to c2")

(defun function2 ()
  (setq c2 (+ 1 c2)))

(tv:add-typeout-item-type alist-alpha
                          :new-type "Function1" (function1)
                          nil "Add one to c1")

(defun function1 ()
  (setq c1 (+ 1 c1)))
```

```
;;; Make the mouse-sensitive window
(defvar sensitive-window
        (tv:make-window
          'new ; This is the flavor specification
          ':borders 2
          ':top 200.
          ':bottom 310.
          ':right 488.
          ':width 316.
          ':blinker-p nil
          ':label '(:string "SHAPES"
                    :character-style (:fix :roman :very-large))
          ':item-type-alist alist-alpha

;;; Expose the window and draw the objects
(defun set-up ()
  (tv:window-call (sensitive-window :deactivate)
    (send sensitive-window ':item ':new-type "  Circle ")
    (send sensitive-window ':item ':new-type " Triangle ")
    (send sensitive-window ':item ':new-type "  Rectangle ")
    (send sensitive-window ':item ':new-type "   Polygon")
    (send sensitive-window ':draw-filled-in-circle 30. 50. 18.)
    (send sensitive-window ':draw-triangle 79. 36. 116. 36. 97. 68.)
    (send sensitive-window ':draw-rectangle 32. 32. 164. 36.)
    (send sensitive-window
          ':draw-regular-polygon 265. 34. 288. 40. 5.)
    ;; blip holds the list returned by :any-tyi
    (loop as blip = (send sensitive-window ':any-tyi)
          ;; Invoke the operation returned by the blip
          ;;  unless the operation is (exit)
          until (equal (cadr blip) '(exit))
          do (eval (cadr blip)))))

; Do it
(set-up)
```

23.6 Mouse-Sensitive Areas Example

In Figure 26, we show how *areas* of the screen can be made mouse-sensitive, allowing the mouse to be used to select graphical entities, as well as text items.

To make the shapes mouse-sensitive, within the function **set-up**, add several lines of Lisp code after the following line:

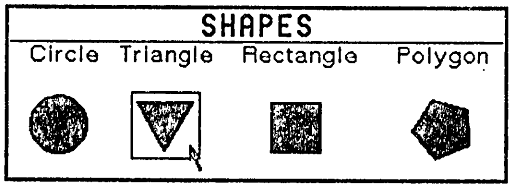


Figure 26. Mouse-sensitive areas example.

```
(send sensitive-window ':draw-regular-polygon 250. 34. 272. 40. 5.)
```

Next is the code to add to **set-up.**

```
(defun set-up ()
   .
   .
   .

;; The boxes are associated with the graphic area
(send sensitive-window
     ':primitive-item ':new-type 'box-1 10. 30. 52. 74.)
(send sensitive-window
     ':primitive-item ':new-type 'box-2 77. 31. 120. 72.)
(send sensitive-window
     ':primitive-item ':new-type 'box-3 160. 31. 201. 72.)
(send sensitive-window
     ':primitive-item ':new-type 'box-4 250. 31. 295. 75.)
   .
   .
   .
)
```

24. The Margin Choice Facility

A window can be augmented with choice boxes in its bottom margin using the flavor **tv:margin-choice-mixin**. See the section "The Multiple Choice Facility", page 293. Margin choice boxes give the user a few labelled mouse-sensitive points that are independent of anything else in the window. Thus margin-choices can be added to any flavor of window in a modular fashion. They are commonly used to implement "confirmation" choices (for example, [Do It] and [Abort]) following another selection.

Margin choices are not a complete choice facility and consequently do not come supplied in an instantiable version. The margin choice facility must be combined with another window flavor. For an example of a window with margin choices (as well as choice boxes in its interior), try the [Kill or Save Buffers] operation in the Zmacs editor menu (refer to Figure 15 shown previously, page 293.)

24.1 The tv:margin-choice-mixin Flavor

tv:margin-choice-mixin *Flavor*

This mixin flavor puts choice boxes in the bottom margin, according to a list of choice-box descriptors that can be specified with the **:margin-choices** init-plist option or the **:set-margin-choices** message. The choice boxes are spread evenly across the bottom margin.

A choice-box descriptor is a list, defined as follows:

(name state function x1 x2)

You can use a longer list as a choice-box descriptor and store your own data in the additional elements.

name is a string that labels the box. *state* is **t** if the box has an "X" in it, or **nil** if it is empty.

function is a function called by the system in a separate process if the user clicks on the choice box. It receives four arguments: the window containing the choice box, the choice-box descriptor for the choice box, the "margin region" that contains the choice boxes, and the *Y* position of the mouse relative to this window. (The last two arguments are usually ignored.)

The structure access functions **tv:choice-box-name** and **tv:choice-box-state** may be of use inside *function* (they are just more specific names for **car** and **cadr**). If *function* changes the state of the choice box, it should refresh the choice boxes in the following way:

```
(send (tv:margin-region-function region) :refresh window region)
```

where *region* is its third argument. This is why the *region* argument is passed. Note that automatic *implications* of a choice (things that happen to the other choice boxes when one choice box is selected), such as in the multiple choice facility are not implemented in the margin-choice facility. See the section "The Multiple Choice Facility", page 293. Programmers must write their own implication routines.

x1 and *x2* are used internally to remember the location of the choice boxes.

tv:margin-choice-mixin is built on the non-instantiable flavor **tv:margin-region-mixin**; the position of the latter in the list of component flavors controls where in the margins the choice boxes appear. The default puts **tv:margin-region-mixin** right after **tv:margin-choice-mixin**. To place the choice boxes inside the borders, use the following model:

```
(defflavor bordered-window-with-margin-choices ()
       (tv:borders-mixin tv:margin-choice-mixin tv:window)
```

24.2 tv:margin-choice-mixin Init-plist Option

:margin-choices *choices* (for **tv:margin-choice-mixin**) *Init Option*

This causes a line of choice-boxes to appear in the bottom margin of the window. *choices* is a list of choice-box descriptors, described previously. If *choices* is **nil**, there are no choice boxes and no space for them in the bottom margin; however, the window is still capable of accepting the **:set-margin-choices** message to create a line of choice boxes later.

24.3 tv:margin-choice-mixin Messages

:set-margin-choices *choices* of **tv:margin-choice-mixin** *Method*

This message changes the set of margin choices according to *choices*, which is **nil** to turn them off or a list of choice-box descriptors. If the choice boxes are turned on or off, the size of the window's bottom margin changes accordingly.

24.4 tv:margin-choice-mixin Example

A simple example of the margin choice facility is shown in Fig 27. In the example, the user can select one of three actions to be taken within a graphics window.

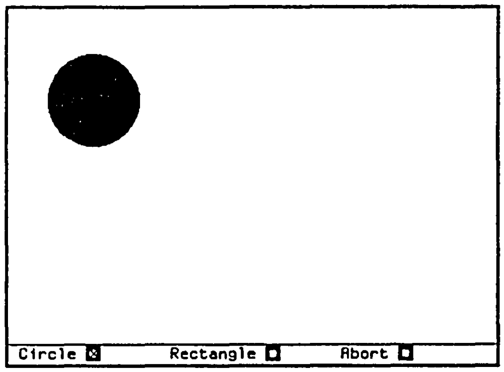


Figure 27. Example of a margin choice facility added to a window.

The Lisp code used to produce Figure 27 is listed below.

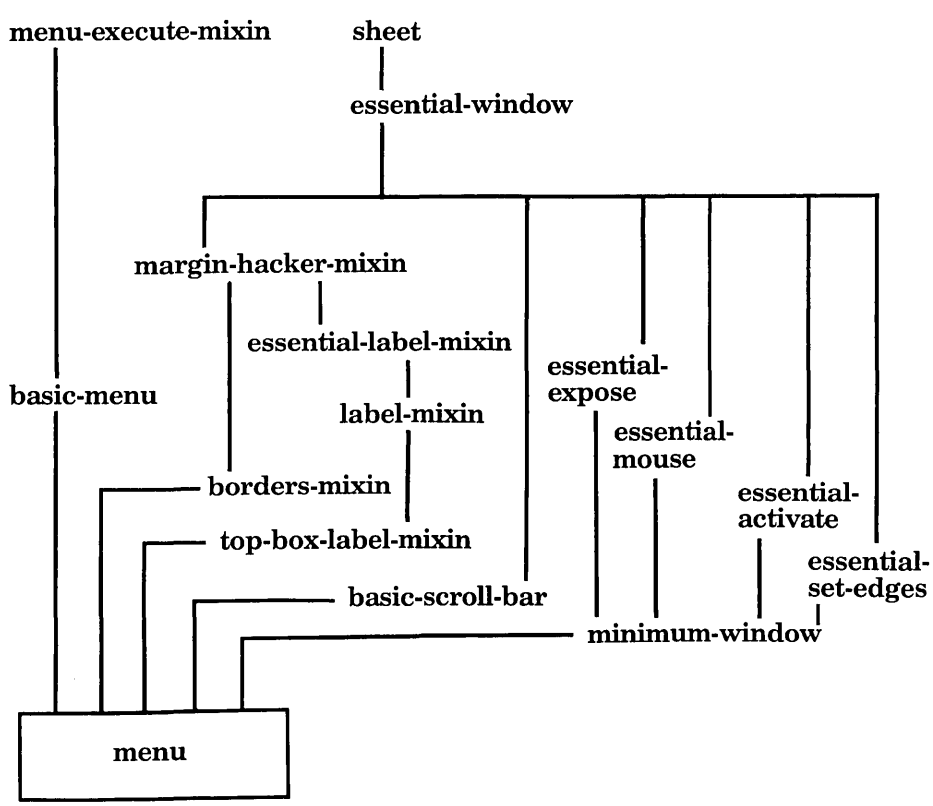
```
;;; Margin Choice Facility Example
;;;  Draws shapes or aborts based on the margin-choice selection.

;;; Specify the margin choice-box descriptors

(defvar choice-box-1 '(" Circle" nil shape-handler x y
                       (:draw-filled-in-circle 70. 75. 38.)))
(defvar choice-box-2 '("Rectangle" nil shape-handler x y
                       (:draw-rectangle 70. 70. 170. 50.)))
(defvar choice-box-3 '(" Abort" nil Abort-handler x y))
(defvar margin-list (list choice-box-1 choice-box-2 choice-box-3))

;;; Name of the window we create

(defvar test-window)

;;; Mixin the margin-choice facility with a window
(defflavor window-with-margin-choices ()
       (tv:borders-mixin tv:margin-choice-mixin tv:window))
```

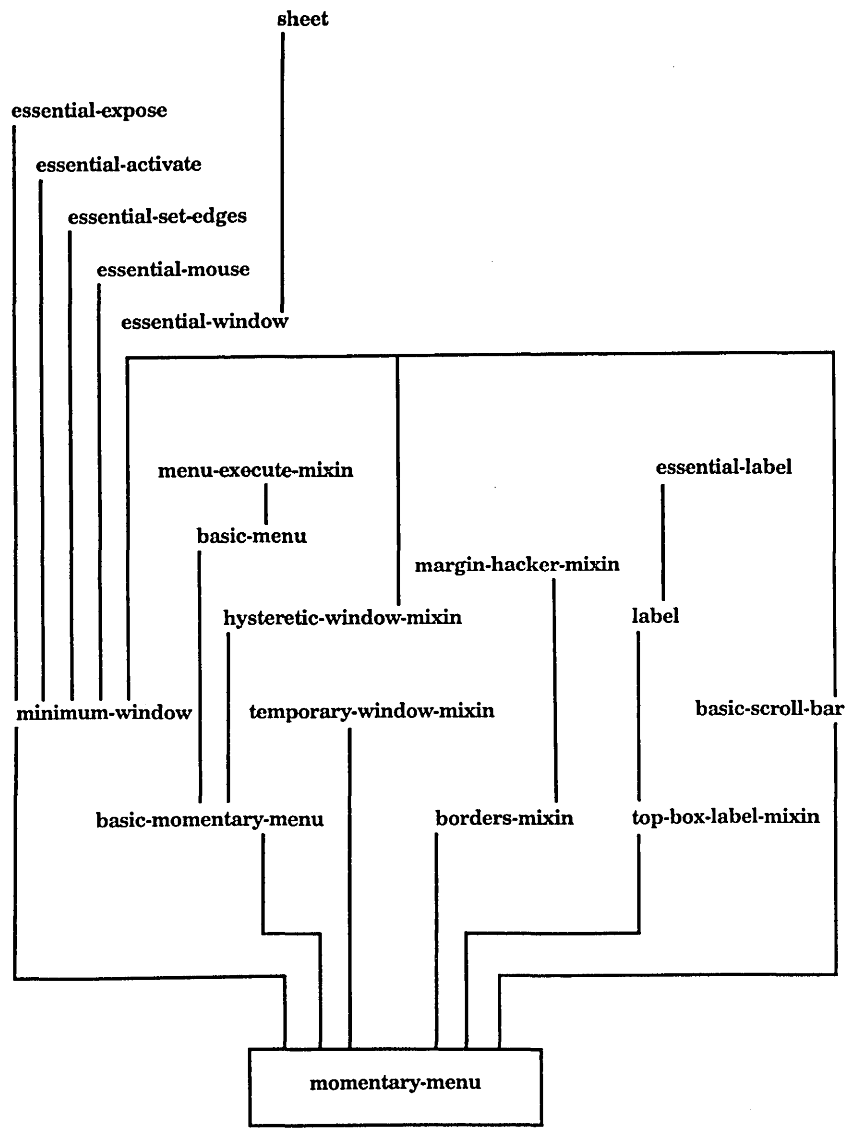
```
;;; Define a handler for the choice boxes that draw shapes
(defun shape-handler (window choice-box region y-pos)
  y-pos ;not used, suppress compiler warning
  ;; Make just this box be lit
  (clear-other-choice-boxes choice-box)
  ;; Erase the window
  (send window :clear-window)
  ;; Refresh the margin so new choice box X's are displayed
  (send (tv:margin-region-function region) :refresh window region)
  ;; Draw the shape the user requested
  (apply window (nth 5 choice-box)))

;;; Define a handler for the "Abort" box
(defun Abort-handler (window choice-box region y-pos)
  y-pos ;not used, suppress compiler warning
  ;; Make just this box be lit
  (clear-other-choice-boxes choice-box)
  ;; Refresh the margin so new choice box X's are displayed
  (send (tv:margin-region-function region) :refresh window region)
  ;; Remove the window from the screen
  (send window :deactivate))

;;; This function clears the non-selected choice boxes
;;; and sets the selected one
(defun clear-other-choice-boxes (selected-box)
  (dolist (box margin-list)
    (setf (tv:choice-box-state box) (eq box selected-box))))

;;; Test the window.
(defun Shapes (&optional (test-window (tv:make-window
        'window-with-margin-choices
        :borders 2
        :label nil
        :vsp 2  ; vertical spacing
        :top 200.
        :bottom 500.
        :right 650.
        :width 410.
        :margin-choices margin-list
        :blinker-p nil)))
  (send Test-Window :Expose))
```

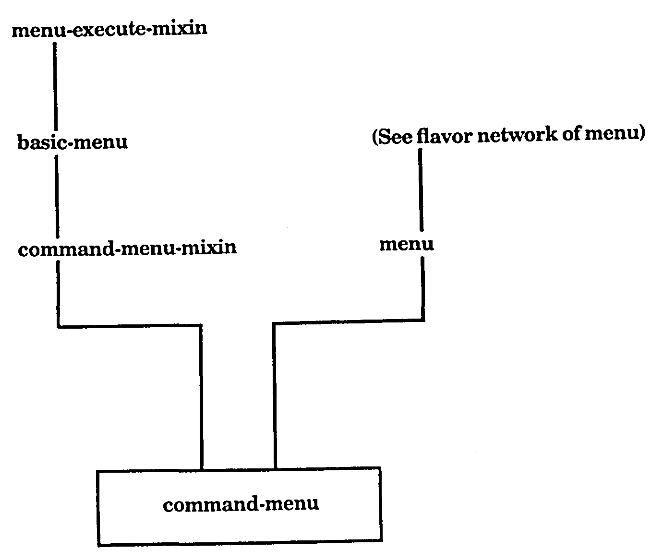
```
;;; Type (SHAPES) to try this out.
```

25. The Flavor Network Of tv:menu

tv:menu is the basis of many of the choice facilities described in this text. **tv:menu** is itself built on a network of flavors, shown in this diagram. **tv:momentary-menu** has a different network, which gives the flavor its own behavior. **tv:command-menu** is based on both **tv:menu** and the **tv:command-menu-mixin**. Knowing the derivation of these flavors can be useful in investigating all the available options and in modifying them for special applications.




sheet
essential-expose
essential-activate
essential-set-edges
essential-mouse
essential-window
menu-execute-mixin
essential-label
basic-menu
margin-hacker-mixin
hysteretic-window-mixin
label
minimum-window
temporary-window-mixin
basic-scroll-bar
basic-momentary-menu
borders-mixin
top-box-label-mixin
momentary-menu


menu-execute-mixin
basic-menu
(See flavor network of menu)
command-menu-mixin
menu
command-menu

26. Init-plist Options For tv:menu

This is a list of some useful window-oriented init-plist options accepted by the **tv:menu** flavor and flavors built on it. It is not meant to be a comprehensive list. Use the Flavor Examiner to find out all the init-plist options of a particular flavor. Most of these options are also documented elsewhere: See the section "Using the Window System", page 81.

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

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

:borders *argument* (for **tv:menu**) *Init Option*

This option initializes the parameters of the borders. The *argument* can be **nil**, which specifies no borders, **t**, which specifies default borders, or it can be a *specification* of a border. The specification indicates which function is used to draw the border and how thick the border is, in pixels.

If the specification is a *number*, the border is drawn by the default function at the specified thickness. The default function is **tv:draw-rectangular-border.**

If the specification is a *symbol*, the border is drawn by the specified function at a default thickness. For more details on creating a function: See the section "Using the Window System", page 81.

If the specification is a *cons* in the form *(function . thickness)*, the borders are drawn by the specified function at a specified thickness.

The specification can also be a list of locations on the screen: *(left top right bottom)*.

:bottom *bottom-edge* (for **tv:menu**) *Init Option*

This is specified in pixels and is relative to the outside of the superior window.

:character-height *spec* (for **tv:menu**) *Init Option*

This is a way of specifying the height of the window. The inside height of the window is made large enough to display *spec* number of lines in the default character style. If the *spec* is a string containing carriage returns, then it is made tall enough to accommodate the string.

:character-width *spec* (for **tv:menu**) *Init Option*

The *spec* is either an integer or a character string. This is one way to specify the width of the window. The inside width of the window is made large enough to display *spec* number of characters in the default character style. If the *spec* is a string, then it is made wide enough to display the string.

:columns *n-columns* (for **tv:menu**) *Init Option*
Sets the number of columns in a menu.

:default-character-style *character-style* (for **tv:menu**) *Init Option*
Specify the default character style of the menu. Items whose character style is unspecified are displayed in the default style. If a character style is specified for an item, it is merged against the default style. (See the section "Menu Item Options", page 251.)

:edges *(left-edge top-edge right-edge bottom-edge)* (for **tv:menu**) *Init Option*
Sets various position and size parameters. All the edge parameters are set relative to the outside of the superior window.

:edges-from *source* (for **tv:menu**) *Init Option*
Specifies that the window gets its edge information from the *source*. If the source is a *string*, the inside of the window is made large enough to display the string in the default character style. If the source is a list: (*left-edge top-edge right-edge bottom-edge*) it is the same as the **:edges** option. If the source is **:mouse,** the user is asked to point to where the left-top and right-bottom corners should go. If the source is a *window*, the window's edges are copied.

:expose-p *t-or-nil* (for **tv:menu**) *Init Option*
When this option is set to **t** the window is immediately exposed. Otherwise, it must be explicitly exposed with an **:expose** message.

:fill-p *t-or-nil* (for **tv:menu**) *Init Option*
Specifies whether to use filled format or columnar format.

:geometry *list* (for **tv:menu**) *Init Option*
Sets up the complete menu geometry, using a list to specify the columns, rows, inside-width, inside-height, max-width, and max-height. See the section "The Geometry of a Menu", page 255.

:height *arg* (for **tv:menu**) *Init Option*
Height in pixels. Includes margins, as opposed to **:inside-height**, which does not include margins.

:inside-height *arg* (for **tv:menu**) *Init Option*
Inside height specified in pixels. Excludes margins.

:inside-size *(inside-width inside-height)* (for **tv:menu**) *Init Option*

Inside size parameters specified in pixels.

:inside-width *arg* (for **tv:menu**) *Init Option*

Inside width of window specified in pixels.

:item-list *list* (for **tv:menu**) *Init Option*

Initialize the item list for a menu. See the section "Types of Menu Items", page 250.

:label *specification* (for **tv:menu**) *Init Option*

Specifies the menu's label. The specification is usually a list in the following form:

(:string "Foo" :character-style *character-style-specification*)

:left *arg* (for **tv:menu**) *Init Option*

Specifies the left edge of the menu, defined in pixels relative to the outside of the superior window.

:minimum-height *arg* (for **tv:menu**) *Init Option*

:minimum-width *arg* (for **tv:menu**) *Init Option*

In combination with the **:edges-from :mouse** init option, **:minimum-height** and **:minimum-width** specify the minimum size (in pixels) of the rectangle accepted from the user. If the user tries to specify a size smaller than one or both of these minimums, the screen beeps and the system prompts the user with a new left-corner.

:name *string* (for **tv:menu**) *Init Option*

This names the window. The name appears in such places as the list of windows generated by [Select] in the System Menu and in the window display option of Peek. The name is the default string for the label if another label string is not specified.

:position *(left-edge top-edge)* (for **tv:menu**) *Init Option*

Specifies the left and top edges of the window. All specifications are given with respect to the outside of the superior window.

:reverse-video-p *t-or-nil* (for **tv:menu**) *Init Option*

If this option is set to **t** the menu is displayed in reverse video, that is, white-on-black instead of black-on-white.

:right *right-edge* (for **tv:menu**) *Init Option*

Right edge of the window specified in pixels, relative to the outside of the superior window.

:rows *n-rows* (for **tv:menu**) *Init Option*
Sets the number of rows.

:screen *screen* (for **tv:menu**) *Init Option*
In a system with multiple screens, sets the screen on which the menu appears.

:top *top-edge* (for **tv:menu**) *Init Option*
Top edge of the window specified in pixels, relative to the outside of the superior window.

:vsp *n-pixels* (for **tv:menu**) *Init Option*
Sets the vertical spacing between lines in the menu. The default is 2 pixels.

:width *arg* (for **tv:menu**) *Init Option*
Specifies the width of the window in pixels.

:x *arg* (for **tv:menu**) *Init Option*
Specifies the left edge of the menu in pixels, relative to the outside of the superior window.

:y *arg* (for **tv:menu**) *Init Option*
Specifies the top edge of the menu in pixels, relative to the outside of the superior window.

27. Messages Accepted By tv:menu

These are some of the messages (arranged in alphabetical order) accepted by menu flavors built on **tv:menu.** The list is not meant to be comprehensive. Use the Flavor Examiner to find out all the messages accepted by a particular flavor. Most of these messages are also documented elsewhere: See the section "Using the Window System", page 81.

:deactivate of **tv:menu** *Method*

This message deactivates a window, deexposing it. In momentary menus, it is sent when the mouse is moved outside the borders of the menu.

:deexpose of **tv:menu** *Method*

Causes a menu to be deexposed. The window remains activated. This message is normally sent only by the system. It usually is meaningless if sent by a user program, because the window is exposed again immediately.

:expose of **tv:menu** *Method*

Causes a menu to be exposed, that is, displayed on the screen.

:refresh &optional *type* of **tv:menu** *Method*

Redraws the menu. The system sends this message with different *type* symbols depending on the event that caused redrawing. You can also send it; in this case the *type* argument is usually not supplied and is allowed to take on a default value. The menu refreshes itself from a bit-save array or redraws itself from scratch, as appropriate. If the bit-save array is invalid, or *type* is **:complete-redisplay** (this is the default), or the size of the menu has changed, it redraws from scratch.

:set-default-character-style *new-style* of **tv:menu** *Method*

Changes the default character style of the menu. All items displayed in the menu whose character style are not otherwise specified are displayed in the default character style.

:set-edges *new-left new-top new-right new-bottom* of **tv:menu** *Method*

This message sets the edges of the window to the four values supplied as arguments, in pixels relative to the superior window.

:set-item-list *list* of **tv:menu** *Method*

Sets the item list of a menu.

:set-label *label* of **tv:menu** *Method*

Sets the label of a menu.

PART IV.

Scroll Windows

28. Introduction to Scroll Windows

Scroll windows are a flavor of window provided by the Lisp Machine window system to facilitate building programs that display information that updates itself, changes its format, responds to the mouse, and shows other evidences of "live" behavior. To see many examples of this type of window, press SELECT P to invoke the **zl:peek** subsystem, and observe the behavior of its various displays as the objects they represent change state.

The basic service performed by scroll windows is that of *redisplay*. You provide a scroll window with a data structure defining what is to be displayed and how to display it. This is very different from other windows that you simply *instruct* to display text (and sometimes graphics) by telling them what to display. While a normal window simply draws what it has been asked to display, a scroll window remembers *how to display again* what it is now displaying, when instructed to do so. Also, a scroll window knows how to *update* its display, changing only those portions of the display that need changing. This is very much like what a real-time editor does when you change text. (Redisplay facilities for Dynamic Windows are introduced in another section: See the section "Overview of Redisplay Facilities" in *Programming the User Interface, Volume A*.)

A typical use of scroll windows is to display a structured representation of some data structure in your program. By clicking on mouse-sensitive items, you can ask to "display more detail" about some item on display. Your program and the scroll window would negotiate to display the more detailed items under the selected item, and move other items around. The file system editor and the **Window** hierarchy display in **Peek** do this. Another typical use is to display data about activity in the Lisp Machine going on simultaneously in other processes, while you watch the display. Such a display might have lines consisting of fixed text followed by numbers or strings that are the "values" of the quantities being "watched". For instance, some lines of such a display might read as follows:

```
Total polyhedra measured    603
Global eccentricity (av.)   .82%
```

while you watched; the numbers change (*update*) as the program measures new polyhedra.

Note that "scroll windows" have nothing, in particular, to do with the concepts of scrolling of windows in general and of mouse scrolling commands in particular. The name "scroll window" is something of a misnomer and a historical accident. Scrolling is not really what is important about scroll windows: the important thing that they provide is a convenient mechanism for getting information to redisplay.

Scroll window displays are exciting and enjoyable to watch and use, and add a touch of class to any program that uses them. ui, ui!

29. Basics of Scroll Windows

The flavor of scroll window most often used is **tv:scroll-window**. You can call **tv:make-window** to make a scroll window. There is also **tv:basic-scroll-window** that contains nothing more than the feature of being a scroll window, and can be used to build more highly specialized flavors. You might also be interested in **tv:scroll-window-with-typeout**. It provides an inferior typeout window should random program output occur directed at it.

The various fields to be displayed are described by *items*. Each item corresponds to some logical portion of the display, always an integral number of lines. Items often contain other items (in a hierarchical fashion), and items can be added and removed from items dynamically (which, as is the whole point of scroll windows, causes the objects on display to appear and reappear when the scroll window's display is *redisplayed*).

A scroll window displays exactly one *top-level item*. The top-level item is simply an item corresponding to *all* the data to be displayed in in the scroll window. When you have constructed the top-level item, you hand it to the scroll window via the **:set-display-item** message. You normally create and set the top-level item just once, when you create and initialize the scroll window.

:set-display-item *item* of **tv:basic-scroll-window** *Method*

Set the top-level item of the scroll window to *item*.

The display created by the items given to a scroll window may well be larger than the physical dimensions of the window. Scroll windows handle this elegantly by showing only a portion of the total display, and allowing the user to scroll the data of the display in the window by using the mouse scrolling commands.

You cause a redisplay by sending the window a **:redisplay** message.

:redisplay of **tv:basic-scroll-window** *Method*

When a scroll window is sent the **:redisplay** message, it examines all parts of the top-level item, including all items contained in it and all items contained in them and so on. It adds new lines to the display as they are found, removes ones no longer found, and updates ones still found, that are in need of updating.

There are two types of items: *line items* and *list items*. A line item describes information to be displayed on exactly one line of the display; that is, if the portion of the display controlled by a certain line item is visible in the window, then it uses up exactly one line of the window, and all of the information of the line item must fit in that line. Drawing a line item must not ever try to move to the next line (you shouldn't use RETURN characters).

A line item is built up of a sequence of *entries*. Each entry is responsible for controlling how one field of the line is drawn. The entries in a line item can be any mixture of constant strings or dynamically updated quantities. The descriptions of the dynamic quantities provide instructions for obtaining and displaying their values. The formats of these descriptions are given below. When the window is asked to redisplay, all of the dynamic entries of the line items on display are computed according to these instructions, and the fields of the line to which they correspond are dynamically and incrementally updated if they need to be.

List items describe multiple-line objects to be displayed. A list item is little more than a list of other items, themselves line items or list items. A list item is displayed by displaying all of the elements in it, in the order in which they appear in the list. The way you insert and remove lines of the display is by adding elements to and deleting elements from list items.

A list item is simply a Lisp list. Its first element is a *list item plist*, specifying some advanced options to be discussed below, and its remaining elements are the items logically comprising the list item. In most cases, the list item plist may be left empty (that is, **nil**).

30. Constructing Items

Line items are constructed by a specialized function, described below. List items are constructed by the standard Lisp list-building functions.

30.1 Constructing Line Items

Line items are constructed with the following function:

tv:scroll-parse-item &rest *line-item-spec* *Function*

This function receives its arguments as a single **&rest** argument that is a *line item spec*. It constructs and returns a *line item*. For the format of line item specs: See the section "Constructing Line Items", page 353.

The line item spec consists of two portions: *global line attributes* that are optional, and *entries*, specifying the fields to be displayed, in the order they are to be displayed on the line. The global line attributes are keyword/value pairs of elements. The first even-numbered element of the line item spec that is not a symbol is the first entry (all keywords are symbols). **nils** are ignored in any position of the line item spec; this sometimes makes the specs easier to construct. Every occurrence of **nil** is deleted from the spec before further processing.

Here is a simple call to **tv:scroll-parse-item.**

```
(tv:scroll-parse-item
        ':mouse '(DOUGHNUTS)
        "Number of doughnuts:  "
        '(:symeval food:doughnut-holes nil ("~D")))
```

Here the global line attributes are present, and consist of the following:

```
':mouse '(DOUGHNUTS)
```

There are two entries:

```
"Number of doughnuts:  "

(:symeval food:doughnut-holes nil ("~D"))
```

In the above example, the **:mouse** global line attribute makes the line displayed by this line item be mouse-sensitive, and the data item **(DOUGHNUTS)** will be encoded in the blip fed to the window's input buffer when this line is clicked upon. The meanings of the various global line attributes will be discussed in detail later.

There will be two fields displayed on this line: the fixed string **"Number of Doughnuts: "**, and the value of the global variable **food:doughnut-holes**. The latter value will be displayed as a decimal number (the "**~d**" is a **zl:format** control string), immediately after the **"Number of doughnuts: "** string, on the same line.

Whenever the window displaying this item is asked to redisplay, the displayed value of **food:doughnut-holes** will be updated if the value of that variable has changed.

30.1.1 Line Item Entries

An *entry* in a line item spec can either specify a constant string to be displayed, or it can specify how to find a value to be displayed. There are four types of entries: *string, symeval, function,* and *value.* An entry is ordinarily represented as a list, whose first element is one of the keywords **:string, :symeval, :function,** or **:value.**

There are two exceptions. First, when an entry is to be made mouse-sensitive, two extra elements are included at the front of the list. See the section "Mouse Sensitivity", page 357. Secondly, there are shorthand forms for some of the formats; they are listed in the table below.

Here are the four types of entries, and their respective formats:

:string

Format: **(:string** *string***)**
Shorthand format: *string*

where *string* is a string. This entry will display as the string, occupying as much of the line as it takes up.

:symeval

Format: **(:symeval** *symbol width* **(***format-ctl base *nopoint***))**
Shorthand format: *symbol*

where *symbol* is a symbol to be evaluated to produce the value to be displayed. The syntax *symbol* is equivalent to

```
(:symeval symbol nil ("~A" base *nopoint))
```

The third and fourth elements of the entry are optional. *width* specifies the field width in characters, on the line, to be allocated to the displayed data. If omitted, or given as **nil**, as much space as needed will be allocated.

If a value is given, it must be a positive number that must fit in the window's line length. The printed representation of the value should not use more than this many characters.

The value is printed using the **format** function. The fourth element of the entry is a list, whose first element specifies the **format** control string to be used. If there is no fourth element, "**~a**" is used. The second and third elements of this last element of the entry (which are also optional) give the values of the global variables **zl:base** and **zl:*nopoint** to be set up when **format** is called. If not given, the current values of these variables at redisplay time will be used.

Note that if you use the shorthand form of the **:symeval** entry type as the first entry in the line item spec, it will be mistaken for a keyword in the global line attributes. If you want the first entry to be a **:symeval** entry, you must use the longer syntax.

Here are some examples of **:symeval** entries:

```
(:symeval number-of-dogs)               ; Just display the value.
number-of-dogs                          ; (The same.)
(:symeval number-of-dogs 6 ("~S"))      ; Use six columns and
                                        ;   use slashification.
```

:function

Format: **(:function** *function arglist width* **(***format-ctl base *nopoint***))**
Shorthand format: **(lambda .....)**
Shorthand format: **(named-lambda ....)**
Shorthand format: *<an actual compiled code object>*

This is the most general type of entry. It specifies a function to be called at redisplay time, and the actual arguments to which it is to be applied. If obtaining the data to be displayed for an entry involves any action more complicated than the evaluation of a variable, you will need a **:function** entry. *function* specifies the function to be called. It may be a symbol, lambda expression, or named-lambda expression, or compiled code object. It will be applied to *arglist* at redisplay time to produce the value to be displayed. Keep in mind that *arglist* is a list of actual values, *not* a list of forms to be evaluated. If *arglist* is not given, it is assumed to be **nil**. It is often useful to use the backquote list-templating facility to create **:function** entries whose argument lists contain actual data objects obtained at the time **tv:scroll-parse-item** is called. See the section "Backquote" in *Symbolics Common Lisp.*

width, *format-ctl*, *base*, and **nopoint* are optional, and have the same meaning as they do with **:symeval** entries.

In the shorthand forms, in which only a function is supplied, *arglist* is assumed **nil** and default assumptions about the printing format are made as for **:symeval** entries.

Here are some examples of **:function** entries:

```
(:function #'compute-number-of-items '(dogs))
(:function #'compute-number-of-items '(dogs) 6 ("~S"))
(lambda () (compute-number-of-cats))
```

:value

Format: **(:value** *index width* (*format-ctl base *nopoint*)**)**

:value entries are a trick to obtain multiple results or decompose structured results from functions. Since **:function** entries can return only one value to be displayed, it is more difficult to display a complicated result, or multiple values returned by a function, than to display a single result. Scroll windows provide a one-hundred element array in which functions called by **:function** entries may store extra results. **:value** accesses elements of this array for display: *index* is a number that specifies what element of the array to access. By using this array as a temporary holding place, values computed by a **:function** entry early in the line item can be accessed by **:value** and **:function** entries later in the line item.

The array can also be accessed via the accessor **tv:value** from functions in **:function** entries. This accessor is applied to the array element index into the array **tv:value** in question. **zl:setf** may be used to store values into this array.

width, *format-ctl*, *base*, and **nopoint* are optional, and have the same meaning as they do with **:symeval** entries.

Here is an example of the use of a **:value** entry. We wish to display a line item that contains two constant strings and two variable fields. The line will represent the result of calling a function, **current-horse-lister**, that returns lists such as:

```
(Seabiscuit Silver Horace)
```

This function interrogates the state of some horse-processing system that is assumed to be running and continually processing horses. We wish to display on one line the number of horses currently being processed, and the actual list of their names.

A first attempt might look like

```
(tv:scroll-parse-item
        "Number of horses  :"
        '(:function (lambda ()
                      (length (current-horse-lister)))
                    5
                    ("~5D"))
        "Their names:   "
        '(:function #'current-horse-lister))
```

Although this will produce a display of the right format, it is inadequate because it calls **current-horse-lister** twice. It is possible that between the two calls to **current-horse-lister** the set of horses may have changed. Or we could be dealing with a function that has side effects, and must not be called twice if we really only want one answer. **:value** solves this problem. Here is the correct code.

```
(tv:scroll-parse-item
        "Number of horses  :"
        '(:function
             (lambda ()
               (setf (tv:value 0)
                     (current-horse-lister))
               (length (tv:value 0)))
             5 ("~5D"))
        "Their names:   "
        '(:value 0))
```

In this example, element **0** of the array is used to save the horse list between the display of the second and fourth entries in this item.

You should not use **tv:value** except for this purpose, and you should only expect its values to be saved during the display of one line item. It cannot be counted on to retain values between displays of different items, or repetitive displays of one item.

30.1.2 Mouse Sensitivity

Entire line items or individual entries in a line item may be made mouse-sensitive. This means that the display corresponding to the item or entry will be highlighted as the user moves the mouse over it, and if the user clicks on it, the program controlling the scroll window will be notified.

If you want to use any of the mouse sensitivity features, you must include the flavor **tv:scroll-mouse-mixin** in the flavor of window to be used. This mixin is not included in **tv:scroll-window**. (Note: this has nothing to do with mouse scrolling; the name means that it is the flavor of the scroll facility that deals with the mouse.)

To make a line item mouse-sensitive, put a specification of the form

```
:mouse      action
```

or

```
:mouse-self action
```

in the global line attributes of the line item spec when constructing the line item. *action* must be a list (actually, a cons). When a mouse-sensitive item is clicked on, the scroll window's handler, running in the mouse process, does one of the things described below, depending on the car of *action*.

If the car of *action* is **nil**, *action* is interpreted as a menu item. Clicking causes an **:execute** message is sent to the window, with *action* as its argument. Only those menu item types that produce side effects are meaningful here (that is, **:funcall**, **:eval**, **:kbd**, **:menu**, and **:buttons**). You can also use **:documentation** to provide a string to be displayed in the mouse documentation window in the who-line. Note that the car of *action* is not significant to **:execute**. For example:

```
(tv:scroll-parse-item
 ':mouse '(nil :eval (set-balance 0)
              :documentation "Set the balance to zero.")
 "Current balance:  " balance)
```

When you move the mouse over this line of the display, the entire line is highlighted, and the documentation string appears in the who line. If you click on the line, the function **set-balance** is applied to **0**.

If the car of *action* is a symbol other than **nil**, that symbol is looked up in the *type alist*, which is an association list. If the car is found, an **:execute** message is sent to the window. The argument to the message is the list

```
(nil op . action)
```

where *op* is the cadr of the entry found in the type alist for the car of *action*. The type alist can be set with the **:set-type-alist** message, or initialized with the **:type-alist** init option.

If the car of *action* is not found in the type alist (which will happen if you aren't using the alist feature) and is not **nil**, a blip of the form

(*type action window button*)

is forced into the window's input buffer. Here, *type* is the car of *action*, *window* is the window itself, and *button* is a mouse button encoding. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. This is the standard way to "read" the event of clicking on a sensitive item. The doughnut example above used this technique, putting blips of type **DOUGHNUT** in its input stream.

:mouse-self is just like **:mouse**, except that before returning the line item, **tv:scroll-parse-item** walks over *action*, and substitutes the actual line item that it constructed for all occurrences of the symbol **self** in *action*, so you can access its array leader. See the section "Line Item Array Leaders", page 359.

Individual entries in a line item can be made mouse-sensitive, as well. To make an entry mouse-sensitive, express it in the standard form, that is, (as opposed to the shorthand form), as follows:

```
(:string "Differential Amplifiers")
```

Then place either of the following at the head of the list:

:mouse *action*

or

:mouse-item *action*

The new entry will precede what was there before. For example:

```
(:mouse (nil :menu parts-menu
             :documentation "Pop up a menu of parts.")
  :string "Differential Amplifiers")
```

:mouse acts just like it does for entire line items, and *action* has precisely the same interpretation. Instead of **:mouse-self**, use **:mouse-item** to get the substitution feature: for mouse-sensitive entries, the *item* (that is, the item for the whole line) is substituted for all occurrences of the symbol **item** in *action* if **:mouse-item** is employed.

30.1.3 Line Item Array Leaders

You can use the array leader of a line item for arbitrary data storage. You can use **:mouse-self** or **:mouse-item** to get the items back at mousing time. Scroll windows use the first few entries in the array leader of a line item for its own purposes. The index of the first item available for your use is stored in the variable **tv:scroll-item-leader-offset**.

To specify that you want array leader space to be reserved at line item creation time, you must use the **:leader** global line attribute. Its formats are

`:leader` *size*

`:leader` *init-list*

size is the amount of array leader to be reserved for your purposes, and *init-list* is a list of elements to be placed at line item creation time in as many array leader elements as they require.

30.2 Constructing List Items

List items are normally constructed with the function **list**. The first element of a list item is the list item plist, and the rest of the elements are items that make up the list item.

Here is an example of constructing a list item for a three-line display:

```
(list ()                              ;list item plist
   (tv:scroll-parse-item ...)
   (tv:scroll-parse-item ...)
   (tv:scroll-parse-item ...))
```

The list item plist is a list of alternating keyword symbols and values. There are two defined keywords, as follows:

:pre-process-function

The **:pre-process-function** keyword takes any function object as an argument. This function is called at redisplay time, with the entire list item as as its one argument. Its returned value is ignored. The idea of this is to allow you to compute, at redisplay time, whether or not you still want all the items currently in the list item to remain in it, or want to add new ones and so on. Your "pre-process function" will have to walk over the cdr of the list item, and be aware that lists therein are list items and arrays are line items in whose array leader you may have stored identifying information meaningful to you.

:function

(Not to be confused with the **:function** entry type in line items.) The **:function** keyword takes any function object as an argument. When it is time to redisplay this list item, the function is called to process every item of this list item, and the returned value of the function is **rplaca**'ed back into the list item before the redisplay is done. This processing occurs *after* the pre-process function, if any, has been called.

The idea of the **:function** list item property is to allow scroll window redisplay to actually cause your subsystem to update its own data. Some subsystems might want or require this, although it is very uncommon.

The function is called on three arguments: *inferior-item*, *position*, and *plist*. *inferior-item* is the particular constituent item of the list item, *position* is an internal item index, and *plist* is a locative to the list item plist of the current list item. The result of *function* is **rplaca**'ed back into the list item when *function* returns.

31. Virtual List Maintenance

An elegant facility to construct and maintain list items is provided by **tv:scroll-maintain-list**. If you intend to construct displays in which lines and subdisplays dynamically appear and disappear, you probably want to use this facility to construct and update list items. It uses the list item plist facilities described above for its implementation.

The function **tv:scroll-maintain-list** constructs (and returns to you) a list item that updates itself to represent some object of yours and its inferior objects every time the scroll window is asked to redisplay. You provide **tv:scroll-maintain-list** with two functions, one (the *init function*) that will be called at redisplay time to produce some object of yours corresponding to a set of your objects that require associated displays, and a second (the *item function*) that, given an object of yours, produces the display item (line or list) representing it.

As just described, the set of objects is expected to be a list of your objects. **tv:scroll-maintain-list** will ask for it at each redisplay, and cdr down it, applying your item function to get display items. It is also possible to return a set of your objects in some other form than a list; in this case, you must provide a *stepper function* that knows how to extract the next object, the "rest" of the set, and tell whether the end has been reached.

tv:scroll-maintain-list *init-fun item-fun* &optional *per-element-fun stepper-fun compact-p pre-proc-fun* &rest *init-args* — *Function*

Constructs and returns a list item that updates itself when the scroll window is asked to redisplay. Takes the following arguments:

init-fun	The init function that will be called at redisplay time to provide a representation of the set of objects to be displayed.
init-args	Arguments to be passed to *init-fun* when called at redisplay time.
item-fun	The item function, to be applied to each object of yours to produce a display item.
per-element-fun	A function to be put in the list item plist of the list item as the **:function** function.
stepper-fun	The function that is called on the set of objects and all "rest"s of the set. It is expected to return three values: the next element, the "rest" of the set, and **t** if it has

returned the last element of the set. If not given, *stepper-fun* defaults to **tv:scroll-maintain-list-stepper**, a function that handles ordinary lists.

compact-p An optional flag that causes **tv:scroll-maintain-list** to copy the list it builds at each redisplay into a special area for such lists, in order to optimize paging performance. The list so constructed will be stored in compact (that is, cdr-coded) form.

pre-proc-fun A function to be put in the list item plist of the list item as the **:pre-process-function** function. If not given, *pre-proc-fun* defaults to **tv:scroll-maintain-list-update-function.**

Following is a simple example:

```
(tv:scroll-maintain-list #'(lambda (instance)   ;The init function
                                (send instance ':value-list))
                         #'(lambda (value)      ;The item function
                                (tv:scroll-parse-item
                                  '(:string ,(format nil "~S" value))))
                         nil nil nil nil
                         self)                  ;Argument to init function
```

Here is an example of code to construct a list item that displays the contents of a Lisp list on separate lines. The variable ***important-data*** contains the list.

```
(tv:scroll-maintain-list
   #'(lambda () *important-data*)      ; The init-fun.
   #'(lambda (list-element)            ; The item-fun.
       ;; Create an item from the list element.
       (tv:scroll-parse-item
          '(:string ,(format nil "~S" list-element)))))
```

PART V.

Digital Audio Facilities

32. Introduction to the Digital Audio Facilities

The 3600-family audio facilities consist of two 16-bit digital audio channels and supporting microcode. The facilities read arrays of samples from memory and feed them to the console at a rate of 50,000 pairs of samples per second. This rate is controlled in hardware by a crystal. When active, the audio microcode reads a pair of samples from main memory every 20 microseconds, supplying one 16-bit value to each channel.

In the standard console, the samples are sent to a 12-bit digital-to-analog converter (DAC). The signal emanating from the DAC is routed to a small speaker and an 8-ohm headphone jack, as well as a low-level analog output compatible with standard "auxiliary" inputs to consumer audio equipment. In the standard console, the monaural output sound is produced by combining the two DAC channels and routing the signal through a simple two-pole low-pass filter at 8 KHz.

The audio microcode also supports a *polyphony feature*. The polyphony feature allows the use of the audio facility for the performance of music, obviating the need to generate samples for an entire performance.

Use the online tools described elsewhere to find out more about a given object in the audio facility: See the section "Program Development Tools and Techniques" in *Program Development Utilities*.

The digital audio facilities are demonstrated through several code examples. See the section "Examples of Using the Audio Facilities", page 389. The code examples are distributed in the following file:

```
SYS:EXAMPLES;AUDIO-EXAMPLES.LISP.
```

Note: the digital audio facility works only on 3600-family Lisp Machines running System 5.2 (or later), with the Revision 6 (or later) I/O board (IO-REV.6) installed.

33. Setting the Console Volume

A function exists for checking and setting the volume (loudness) of the console audio.

sys:console-volume &optional (*console* **sys:*slb-main-console***) *Function*

Returns the current volume setting for the console, which is a number between 1 (loudest) and 63 (softest). The console volume can be changed with **zl:setf**, as in the example:

```
(setf (sys:console-volume) foo)
```

34. Microcode Support for the Digital Audio Facilities

34.1 The Audio Microtask

This section discusses the microcode interface, that is, the formats of commands and samples interpreted by the audio microcode. This is the lowest-level interface to this facility, and only the barest primitives are described here. The formats and commands given here might change in future versions of the hardware, microcode, and software.

The audio microcode runs in its own *microtask* and thus operates parallel with the execution of Lisp. The audio microtask is either *active* or *stopped* at any time. Since the microtask scheduler works according to a priority queue, when the audio task is active, it "wakes up" every 20 microseconds, and executes, preempting Lisp, until it either outputs an audio sample pair or stops. The generation of audio samples is not affected by the behavior of Lisp programs, including the masking of interrupts, and so forth.

When active, the audio microtask follows a *command list*, or program of its own, consisting of *audio commands*, stored by the programmer in main memory before the audio microcode is started. The command list is stored in sequential *physical* memory locations (although it can contain "jumps"). Each command occupies one or more 3600 words. The words are expected to be fixnums. The 32 data bits of each fixnum contain the data interpreted by the audio microtask. The commands include directives to control the flow of the command list as well as directives to output data to the console DAC. The audio microcode also maintains a *repeat counter* to facilitate generation of repetitive or continuous waveforms. See the section "Looping Through Audio Command Lists", page 384.

The audio microtask is started by the execution of the **%audio-start** instruction by Lisp; the evaluation of the form **(sys:%audio-start)** effects this. When this instruction is executed, the audio microtask fetches the physical address of the beginning of the command list from the variable **sys:%audio-command-pointer**. Therefore, this variable must be set to the physical address of the beginning of the command list *prior* to the execution of the form **(sys:%audio-start)**. The audio microcode stops when it encounters an explicit command to this effect in its command list.

The audio microtask is coded for real-time performance; it does no validity checking, and issues no diagnostics. If you program the audio microtask via the techniques described in this document, it is your responsibility, as always, to create valid programs. In the case of the digital audio facilities, however, the result of an invalid program could be a machine halt or destruction of the integrity of virtual memory, or both. If certain bit patterns are interpreted as audio commands, they can modify storage locations. Save your editor buffers often when debugging code for the audio microcode.

34.2 Sample Format

Each sample pair is expected to be a fixnum. The 32 data bits of each fixnum include two samples, one for each channel. The sample pair is read by the audio microtask in one operation, and the samples are sent to each channel in parallel. Each sample is a 16-bit unsigned integer, one in the lower (bits 0-15) half word (channel 0), and one in the upper (bits 16-31) half word (channel 1).

A sample value of 0 produces the lowest analog output voltage, and a sample value of all 1s (65535, octal 177777) produces the highest. A voltage of zero is represented by the midpoint value, 32768 (octal 100000).

Channel 0 is currently supplied with analog output hardware in the console; Channel 1 is not. The digital-to-analog converter in the console is only of 12-bit precision, and thus, it ignores the low 4 bits of Channel 0 samples.

34.3 Audio Command Format

Audio commands occupy one or more words of sequential physical memory. The command words are expected to be fixnums. The fixnum data (32 bits) for each command is described in this section.

The format of the first word of each command is as follows, described by byte specifiers in the **sys** package:

%%audio-command-op

A 4-bit *opcode* selecting the action to be performed by the audio microcode. Each of the currently assigned opcodes is described elsewhere. See the section "Audio Command Opcodes", page 371. See the section "Polyphony Command Opcodes", page 375.

%%audio-command-arg

A 28-bit quantity, whose meaning differs for each opcode. When the contents of this field, known as the *operand*, is described as an *address*, it must be a physical address. The usual way to obtain such a physical address is via the function **si:%vma-to-pma** (which does a virtual-to-physical translation). This function is given a fixnum virtual memory address. The usual way to derive such addresses, which are usually references to array element cells, is via the **%pointer** and **aloc** functions.

A physical address computed from a virtual address in this way cannot be validly used unless the relevant virtual address has been wired in advance. See the section "Notes on Wired Structures", page 376.

34.3.1 Audio Command Opcodes

These are the valid opcodes of audio commands, with the exception of those commands associated with the polyphony feature. See the section "The Polyphony Feature", page 373. The descriptions tell what action is performed by the audio microtask when a command having this opcode is encountered by the microtask. The opcodes are listed under the the name of the system constant (also in the sys package) that gives the opcode value.

%audio-command-stop
Causes the audio microtask to halt execution. No more commands are fetched, or samples sent to the console, until the next execution of the **sys:%audio-start** instruction. The operand is ignored.

%audio-command-jump
Causes the audio microtask to fetch its next instruction not from the next sequential location, but from the physical address that is the value of the operand. Sequential execution of commands continues at that physical address.

%audio-command-load-repeat
Loads the repeat register with the value of the operand. The operand is an unsigned 28-bit number to be loaded into the repeat register, not an address. See the description of the **%audio-command-loop** opcode for the use of this register.

%audio-command-loop
Decrements the repeat register by 1. If the result is greater than zero, the operand is interpreted as a jump address, and execution of commands continues at that address, as with **%audio-command-jump**. Otherwise, if the result is less than or equal to zero, command execution continues with the next sequential command.

%audio-command-samples
Designates a vector of sample pairs to be sent to the console. The operand is the physical address of the first sample pair; the remaining samples are fetched from successive words of physical memory. The word in the command stream after the **%audio-command-samples** command contains a fixnum that is

the count of the number of sample pairs to be fetched and sent to the console before the execution of **%audio-command-samples** terminates, and the microtask proceeds to the next sequential command. The **%audio-command-samples** command is thus a two-word command.

%audio-command-zero

A synchronization primitive. The operand is the *physical* address of a cell, usually an array element. The audio microcode stores a fixnum zero in that cell as the result of executing the command having the opcode **%audio-command-zero**. The software can use this facility to test if the audio microtask has passed a given point in its command list. This enables the software to ascertain when it is safe to unwire or reuse data structures containing audio commands and/or samples. It is important to remember that the audio task, when active, locks out Lisp execution until it either sends a sample or goes idle. For example, if **%audio-command-zero** is immediately followed by **%audio-command-stop**, the observation of the zeroed cell by Lisp software implies that the microtask has already read, interpreted, and executed the **%audio-command-stop**.

%audio-command-immediate

Designates a vector of sample pairs to be sent to the console. Unlike **%audio-command-samples**, the sample pairs appear in the command list, in consecutive physical memory locations immediately following the the **%audio-command-immediate** command word. The operand of **%audio-command-immediate** is a number, which is the count of sample pairs. That number of sample pairs is fetched from the command list and sent to the console, one every 20 microseconds (at a 50 KHz sampling rate). Execution of the command list proceeds with the next command after the vector of sample pairs, after all samples have been sent to the console.

It is critically important that the operand is equal to the number of samples provided, lest commands be interpreted as samples or vice versa.

34.4 The Polyphony Feature

Note: The polyphony feature is experimental in Release 7.0. It might be radically altered in function and/or interface in future releases, or might be removed entirely.

The polyphony feature of the Symbolics audio microcode provides a way to generate polyphonic music in real time. There is no need to precompute the samples and store them before playback from disk. The polyphony feature can produce six *voices*, where a voice is a rhythmically independent sequence of musical notes. Each voice can be assigned a predefined, programmer-specified waveform, which determines the spectrum and the amplitude of the notes that appear in that voice, regardless of their pitch (frequency). The waveform specification determines the shape and amplitude of *one cycle* only of the waveform. This waveform is repeated at different frequencies to produce musical tones.

The polyphony feature is not intended as a general-purpose music synthesis facility. For example, no control over the amplitude envelopes (attack, decay, and so forth) of the sounds produced is provided. The polyphony feature is intended for use in music system prototyping, that is, composition research, music editing programs, and so forth. Nevertheless, the square-envelope notes it produces are not very different from those produced by some electronic organs. When properly programmed and amplified, the digital audio facility is capable of reasonably authentic performance of much of the organ literature.

34.4.1 Operation of Polyphony

The basic function of the polyphony feature is to generate, in parallel, six separate wave signals, usually of different frequencies, and sum them, at the sampling times of the audio facility. The audio microcode accomplishes this by maintaining, for each voice, a *wavetable*, a *wavetable cursor*, and an *increment*.

The wavetable for each voice consists of 1024 fixnums stored in consecutive locations in physical memory, defining the *waveform* for notes in that voice. (Note: the size of the wavetables might change in a future release.) The fixnums constitute *wave values*, which digitally describe the waveform of the voice.

The detailed interpretation of the wave values is as follows: Each fixnum wavetable element is interpreted as the algebraic sum of the wave values for the channels 0 and 1, channel 1 having been shifted 16 bits left. In detail, the value for channel 0 is a 32-bit signed (31 bits and sign, 2's complement) value between -2**15 and 2**15-1, inclusive. The value for channel 1, also in the range -2**15 to 2**15-1, is shifted left 16 bits and added algebraically to the value for channel 0. The resulting number (which is always a fixnum) is the value of the wavetable entry. Note that this is not the same format as that of audio samples used by other parts of the audio facility.

When polyphony is running (that is, when the audio microtask is interpreting the command **%audio-command-polyphony**), one value from each of the six tables is extracted, and these values are added algebraically. The resulting value is then offset by 2**15 *in each halfword*, and the resulting two halfwords are sent as audio samples to the two audio channels.

You must ensure that the sum of the values from each table never exceeds the range -2**15 to 2**15-1 for either channel. The audio microcode clips or overflows into the other channel if this range is exceeded.

Associated with each voice is also a counter/pointer called the *wavetable cursor*. This quantity is a 32-bit unsigned number. The high-order ten bits of the wavetable cursor for each voice constitute an index, which selects the entry of its wavetable to be summed into the audio sample to be produced. The low bits are used to measure the passage of time, overflowing into the high bits 1024 times per cycle of that voice.

Also associated with each voice is a quantity called an *increment*. The increment is a 32-bit fixnum. It controls the frequency, or pitch, of the note in each voice, by controlling the rate of incrementing of the wavetable cursor for that voice. When the command **%audio-command-polyphony** is being interpreted by the audio microtask, the increment for each voice is added to the wavetable cursor for that voice, and the resulting quantity is made the new wavetable cursor. (This addition is performed *after* the wavetable sample is extracted). Thus, when this repeated addition produces enough change in the value of the wavetable cursor such that the top ten bits are affected, a different wavetable entry for that voice is fetched at the next sampling time. Note that continued incrementing in this manner "wraps around". In this way, the wavetable cursor is way reset to the beginning of the wavetable, after the last entry in the wavetable has been used.

The following function (available in the **audio** package) computes the increment for a voice from the frequency:

```
(defun frequency-polyphonic-increment (frequency)
  (round (* frequency (float 1_32.)) audio:*sample-rate*))
```

You simultaneously establish the increment and wavetable location for a voice by the audio command **%audio-command-load-voice**. You instruct the polyphony facility to output samples by the audio command **%audio-command-polyphony**. This command uses all of the wavetables and increments previously established by **%audio-command-load-voice**, and outputs as many samples as requested, one every 20 microseconds, generated by summing entries from the six wavetables, incrementing the six wavetable cursors by the six associated increments as each sample is generated.

Note: changing the wavetable and/or increment for a voice does not affect any other voice in any way. Since the audio microtask is awakened by an external timer, and runs until it either outputs a sample pair or stops, no discontinuity in notes played by other voices is observed when **%audio-command-load-voice** is interpreted to change the note in one voice.

34.4.1.1 Polyphony Command Opcodes

%audio-command-load-voice

Establishes a wavetable and increment for one voice of the polyphony feature The operand is the physical address of the base of the wavetable for the voice. The word in the command stream after **%audio-command-load-voice** is, in its 32 data bits, the increment for the voice. The low three (that is, the least significant) bits of this increment are the binary number of the voice whose wavetable and increment are to be established. **%audio-command-load-voice** is effectively a two-word command.

When polyphony is being performed, the audio microcode uses, for each voice, the wavetable and increment established for that voice. There is no way to assert that a voice does not exist, or has no wavetable, or no increment. A valid wavetable and increment must be established for each of the polyphonic voices before **%audio-command-polyphony** is executed by the audio microcode, regardless of whether that voice is needed for the performance of the particular composition.

%audio-command-load-voice does not affect the value of the wavetable cursor for the voice involved.

%audio-command-polyphony

The operand is an unsigned 28-bit number. The audio microcode sends out that many samples, one each 20 microseconds, generated from the currently established wavetables of the polyphony feature. The wavetable cursors of each voice used by the polyphony feature are incremented by the increment established for that voice as each sample is sent out. The values of the increments and the wavetable cursors are not reset in any way by either the start of **%audio-command-polyphony**, or its completion.

34.5 Simple Tone Generation With sys:%beep And sys:%slide

sys:%beep *half-period duration* *Function*

sys:%beep generates tones on 3600-family consoles that support the digital audio facilities. All new machines include this support. The arguments, *half-period,* (in microseconds) and *duration,* (in microseconds) are compatible with the version of **sys:%beep** that ran on the Symbolics LM-2 computer. In the following example, a 440 Hz tone is generated for 50,000 microseconds (i.e., 50 milliseconds).

```
;;; 440 Hz divided by 1 million is the period in microseconds.
;;; Divide by 2 to obtain the half-period.
(sys:%beep (// 1000000. 440. 2) 50000.)
```

nur 2 Argumente erlaubt! So nicht!

The standard system "beep" -- a short tone burst for signaling an error or notifying users of an important announcements, is defined by the function **beep**. See the function **beep** in *Reference Guide to Streams, Files, and I/O.*

sys:%slide *half-period duration* &optional (*console* **sys:*slb-main-console***)} *Function*

sys:%slide generates sliding tones (glissandi) on 3600-family consoles that support the digital audio facilities. All new machines include this support. The arguments are *half-period, delta-half-period, delta-time,* and *duration,* all specified in microseconds. The *console* argument checks to see if the console can support digital audio. By default, this is set to ***slb-main-console***.

In the following example, a sliding tone starting at 3000 Hz is generated for 500,000 microseconds (i.e., a half second). It changes 20 half-periods (10 Hz) every tenth of a second (i.e., every 100,000 microseconds).

```
;;; 3000 Hz divided by 1 million is the period.
;;; Divide by 2 to obtain the half-period.
(si:%slide (// 1000000. 3000. 2) 20 100000 5000000)
```

34.6 Notes on Wired Structures

The audio microtask fetches commands from sequential locations of physical memory. Branch addresses in the command list are physical addresses. Audio sample data pointed to by the command list are also described by physical address. Wavetables used by the polyphony feature are also described and accessed by physical address.

The audio microtask does not perform virtual address translation. Thus, the command list and sample data must be stored in data structures *wired*, or locked, in main memory. That is, they must be prevented from being paged out or moved by the Genera system. As a digital audio programmer, you must therefore be aware of page boundaries.

Audio command lists and sample vectors must be stored in wired pages consecutive in main memory, or scattered throughout main memory. If commands are stored in pages scattered throughout main memory, jumps must be programmed at the end of each page, to send the audio microcode on to the next page. If sample vectors are stored in pages scattered throughout main memory, you must use a separate **%audio-command-samples** command to describe the samples on each page. Wavetables for the polyphony feature must be in consecutive locations in main memory.

It is conventional to use Lisp arrays as the data structure containing audio commands, samples, and wavetables. Any type of array is usable for this purpose. **art-q** arrays allow one audio command or sample pair per element, and are also the only type of array whose elements can validly be addressed by the **aloc** function.

34.6.1 Lisp Primitives for Wiring Memory

The relevant Lisp primitives to wire data structures for the digital audio facility are **si:wire-structure**, **si:wire-words**, and **si:wire-consecutive-words**. **si:wire-words** wires any extent of virtual memory into physical memory, although the page frames into which successive pages are wired cannot be contiguous. **si:wire-consecutive-words** also wires any extent of virtual memory into physical memory, but successive pages are guaranteed to be stored in successive page frames in physical memory. **si:wire-structure** wires an entire structure (a convenience device to avoid having to calculate the location and extent of the virtual memory occupied by a structure) in the manner of **si:wire-words**.

Since commands must be stored in consecutive locations in physical memory, **si:wire-consecutive-words** suggests itself as the natural primitive for this application. However, success of this primitive depends on the availability of consecutive page frames of main memory not already containing wired pages, and it is thus less likely to succeed as more pages are wired. Use of **si:wire-structure** and **si:wire-words** for audio data does not encounter this problem, but requires explicit programmer handling of page boundaries, as outlined previously.

sys:%find-structure-header and **sys:%structure-total-size** are used to find the virtual memory location and extent of whole arrays or other structures to be wired. **si:page-array-calculate-bounds** can be used to calculate the virtual memory location and extent of portions of array that are to be be wired, when **si:wire-words** or **si:wire-consecutive-words** is used. **sys:%pointer-difference** can also be used to determine the length of the extent, in words, between two addresses obtained via these primitives or the **zl:aloc** function.

Structures, or portions thereof, wired by any of these primitives, should be unwired by **si:unwire-structure** or **si:unwire-words** (as appropriate) only after it has been ensured (via the techniques described) that the audio microtask is not fetching commands or samples from these structures.

35. Lisp Primitives for the Digital Audio Facilities

35.1 Functions, Variables, and Macros for Digital Audio

This section describes the functions, variables, and macros available to the Lisp programmer to aid in programming the 3600-family Digital Audio Facilities. All of these objects are tools for programming the audio microtask. Therefore, this section assumes that you already understand the microcode capabilities. See the section "Microcode Support for the Digital Audio Facilities", page 369.

All of the digital audio functions, variables, and macros appear in the **audio** package. Several comprehensive examples of their use are provided in the file sys:examples;audio-examples.lisp. See the section "Examples of Using the Audio Facilities", page 389.

These Lisp tools assume the existence of an audio *command array*, in which audio microtask commands are placed, and out of which they are executed by the audio microtask. A macro (**audio:with-audio**) manages the wiring and unwiring of command arrays within the scope of a program.

A default audio command array is provided as part of these audio support primitives. All of these primitives, however, allow the specification of any suitable user-provided array as a command array. Such an array must be a nonindirect, single-dimensional **sys:art-q** array, with a fill pointer, allocated in a static area (such as **audio:audio-area**).

Command arrays, as all arrays, are finite in extent. Carefully planned synchronization techniques must be utilized to allow uninterrupted sound to be produced from a single command array that is being serially reused for sequences of audio commands. See the section "Examples of Using the Audio Facilities", page 389.

35.2 Digital Audio Parameters

These are the critical constants of the audio facility. In programs these constants should be used instead of the numbers that are their current values in order to accommodate future modification of the audio facility.

audio:*sample-rate* *Variable*

The number of times per second that an audio sample is output when the audio microtask is active. This is a single-precision floating-point number. Its current value is **50e3**, as determined by the hardware.

audio:*number-of-polyphonic-voices* *Constant*

The number of polyphonic voices defined by the polyphony feature. See the section "The Polyphony Feature", page 373. This is a fixnum, and its current value is 6.

sys:%%audio-increment-integer *Constant*

A byte-spec to determine the integer portion of the per-channel increment. 2****sys:%%audio-increment-integer** is the wavetable size. Use this constant to parameterize your program with respect to wavetable sizes.

35.3 Testing for the Existence of Audio

audio:audio-exists *Variable*

This variable has a value of other than **nil** if and only if the machine on which it is evaluated has an operational audio facility.

35.4 The Audio Wrapping Form

audio:with-audio &optional *command-array* &body *body* *Macro*

Encases code that generates audio commands. It prepares a command array for use by wiring it in an appropriate fashion and unwires it when the body of the form is exited. When exited, it also unconditionally halts the audio microtask, silencing the audio output.

If *command-array* is given as **nil**, the default command array is used.

When the scope of **audio:with-audio** is entered, it also zeroes the fill pointer of the supplied command array. The various interface functions described later utilize the fill pointer of the command array to keep track of the current position in the audio command list being built. **audio:with-audio** also globally binds scheduler parameters to allow the process generating audio commands to gain control when necessary and more rapidly than usual.

35.5 Building Audio Command Lists

The functions listed in this section prepare arguments for, build, and store audio commands in a command array. They assume that the fill pointer of the array describes the next available location in the array, and they update the fill pointer as needed. The array must be wired, as some of these functions compute and store physical addresses of locations in the command array. Calling these functions does not produce sound. Sound is produced when the audio facility is directed (via **audio:audio-start**) to a command list produced by calling these functions.

The fill pointer of the array defines a logical pointer called the *audio index*. The function **audio:audio-index** (which defines a location accessible with **zl:setf**) is used to access this index (for example, for use as an argument to a later function call).

The current implementation uses command arrays that are wired into successive, contiguous page frames of physical memory. (Note: This might change in the future.) The exclusive use of these primitives hides this implementation detail. In order to accommodate future changes in this strategy, do not perform calculations on audio indices. Instead, request them whenever needed via **audio:audio-index**, and use them only as arguments to the primitives provided.

Use of the macro **audio:with-audio** is the recommended way to establish the proper context in which these functions can be validly used. Each of them takes an optional argument, which specifies the command array in question. This argument always defaults to the facility's default command array.

audio:audio-index &optional *command-array* *Function*

This function returns the audio index for the next command to be stored in the command array in question. The form **(audio:audio-index)** is suitable for use as the first operand of a **zl:setf** form.

audio:audio-room &optional *command-array* *Function*

This function returns the amount of available (unallocated) space, in single words, in the current command array.

audio:audio-limit &optional *command-array* *Function*

Returns a number one greater than the audio index of the last usable location in the command array.

audio:audio-push-audio-stop &optional *command-array* *Function*

Pushes a **%audio-command-stop** onto the command list in the command array. ("Push", as used in the names of these interfaces, means "add to the end of, at the current audio index, and increment the audio index appropriately.").

audio:push-audio-jump *target-index* &optional *command-array* *Function*

Pushes a **%audio-command-jump** onto the command list in the command array. The argument *target-index* is expected to be an audio index into the same command array, obtained previously from **audio:audio-index**.

audio:push-audio-zero-flag *flag-index* &optional *command-array* *Function*

Pushes a **%audio-command-zero** onto the command list in the command array. The argument *flag-index* is expected to be an audio index, into the same command array, of a "flag". Such flags are allocated, and their indices returned, by **audio:reserve-audio-flags**.

audio:push-audio-load-voice *voice-number wave-array wave-array-start-time wave-array-index-increment* &optional *command-array* *Function*

Pushes a **%audio-command-load-voice** onto the command list in the command array. *voice-number* is a number, zero or greater, below the value of **audio:*number-of-polyphonic-voices***, that specifies which polyphonic voice is to have its wavetable and increment loaded by the command to be built and stored. *wave-array-index-increment* is the value of that increment, which can be computed from the frequency of the tone desired by use of the function **audio:frequency-polyphonic-increment**. The wavetable for the voice is expected to be in the **sys:art-q** array *wave-array*. The argument *wave-array-start-index* is the index into that array where the 1024-word, wired, contiguous in physical memory, wavetable begins.

audio:push-audio-polyphony *number-of-samples* &optional *command-array* *Function*

Pushes a **%audio-command-polyphony** onto the command list in the command array. The argument *number-of-samples* specifies the sample count for the command to be built and pushed.

audio:modify-audio-command-arg *new-arg arg-type command-index* &optional *command-array* *Function*

Modifies an audio command that has already been pushed in the command array specified. This function must be used with extreme care: it can easily create invalid audio programs, which can destroy machine integrity. It modifies the 28-bit argument in the first word of the command whose index into the command array (*command-index*) is given. To be sure that this command can be validly used, read the description of the format of the individual audio command. See the section "Microcode Support for the Digital Audio Facilities", page 369. *new-arg* is the new value of the command whose index is given. The argument *arg-type* describes how it is converted to a 28-bit value for insertion in the existing command:

:immediate
No processing is done. *new-arg* is expected to be a non-negative fixnum, which must be a count.

:index
The argument is an audio index into the command array specified. The location of the corresponding array cell is computed, verified to be wired, and the physical address of that location stored in the command.

:location
The argument is a locative into a wired array of audio commands. The fact that this location is wired is verified, and the corresponding physical address stored in the command.

35.6 Storing Samples

The functions and macros described in this section place audio sample pairs into the command program. These commands can be either immediate (**%audio-command-immediate**) or stored elsewhere (**%audio-command-samples**).

audio:push-array-of-audio-samples *array* &optional *from to command-array immediate-p* — *Function*

Pushes appropriate commands onto the command list in the command array specified, to output all the sample pairs in the array *array* between indices *from* and (up to but not including) *to*. *from* defaults to 0, and *to* to the active length of *array*. *array* must be an **sys:art-q** array containing precomputed sample pairs.

If *immediate-p* is non-**nil**, the data are copied into the command array, and output by means of **%audio-command-immediate**.

If *immediate-p* is **nil**, *array* is assumed (and checked) to be wired, and as many **%audio-command-samples** commands as necessary to describe the data to be output are built and pushed. *array* need not be wired in contiguous page frames.

audio:computing-immediate-audio-samples (*count* &optional *command-array*) &body *body* — *Macro*

Facilitates the storing of immediate audio sample pairs. The code it wraps, *body*, is responsible for generating immediate audio sample pairs: it does so by calling the macro **audio:push-immediate-audio-sample**, within the scope of the use of **audio:computing-immediate-audio-samples**. Each use of **audio:push-immediate-audio-sample** stores one sample. The macro **audio:computing-immediate-audio-samples** arranges for an appropriate **%audio-command-immediate** to be constructed to describe all the samples stored. If the argument *count* is non-**nil** (at run time), it is expected to be a fixnum, which is the number of values to be stored. **audio:computing-immediate-audio-samples** checks, when it is exited, that that is the actual number of values stored, and signal an error if not. If *count* is **nil**, no checking is done, and

audio:computing-immediate-audio-samples assumes that the number of samples that have been pushed is the correct number, and modifies the commands it builds appropriately.

audio:push-immediate-audio-sample *sample* *Macro*

Stores one audio sample pair, which is the value of its argument. This macro can be used validly within the scope of **audio:computing-immediate-audio-samples.**

35.7 Looping Through Audio Command Lists

These two macros facilitate the use of **%audio-command-loop** to create loops in audio command lists. Keep in mind that the audio microcode does not support nested loops.

audio:audio-loop (*repeat-count-or-nil* &optional *command-array*) &body *body* *Macro*

This macro builds a loop (with **%audio-command-loop** and **%audio-command-load-repeat**) in the audio command list in the command array specified. The code, *body*, which is wrapped by this macro pushes commands for the body of the loop. The macro generates the audio command to loop back at the time its scope is exited. The argument *repeat-count-or-nil*, when non-**nil**, specifies how many times the loop is to be executed by the audio microtask. That is the number that is loaded into the repeat register. If *repeat-count-or-nil* is **nil** (at run time), the wrapped code must compute the number of loop repetitions, and invoke the macro **audio:set-audio-repeat-count**, whose argument is that number, some time before the scope of **audio:audio-loop** is exited. A diagnostic is issued (at run time) if the macro's scope is exited without the repeat count having been specified by one of these two means.

audio:set-audio-repeat-count *count* *Macro*

Sets the value *count* as the repeat count for an audio command list loop that is currently being built by **audio:audio-loop**. This macro can be validly used only within the scope of **audio:audio-loop**.

35.8 Synchronization Flags

These functions allocate, in the command array specified, locations to be used as synchronization flags (for **%audio-command-zero**), and allow the flags to be waited for and reset. The "reset", or "normal", state of these flags, is non-zero.

The audio microcode "sets" them, by setting them to zero, when a **%audio-command-zero** is executed. By means of these flags, the real-time progress of the audio microtask can be monitored.

audio:reserve-audio-flags *count* &optional *command-array* *Function*

Allocates, in the command list currently being built in the command array specified, *count* locations to be used as audio flags. The flags are reset. A **%audio-command-jump** is inserted in the command list being constructed, so that the audio microtask jumps around the locations being used as flags. The return value of this function is the index, in the command array given, of the first of the flags allocated. You can assume, if more than one flag was allocated by a call to **audio:reserve-audio-flags**, that the indices of flags other than the first are the sequential integers above the value returned.

audio:wait-for-audio-flag *flag-index* &optional *who-state audio reset-flag t command-array* *Function*

Waits for the audio flag specified by *flag-index*, in the command array specified, to be set. Normally, it is the audio microtask that sets these flags, by means of **%audio-command-zero**. *whostate* is the state to be displayed in the status line. If *reset-flag* is given as **nil** (this is *not* the default), the flag is not reset. The resetting, when requested, is performed *after* the flag has been observed to be set. The indices given to **audio:wait-for-audio-flag** should be those obtained from **audio:reserve-audio-flags**.

35.9 Starting and Stopping the Audio Microtask

These functions are used to start and stop the audio microtask.

audio:audio-start *index* &optional *command-array* *Function*

Starts the audio microtask, via the instruction **sys:%audio-start**, at the audio command specified by *index* in the command array specified. The array must be wired, and contain a valid, wired, audio command list.

audio:audio-stop &optional *command-array* *Function*

Stops the audio microtask immediately, causing immediate silence. **audio:audio-stop** accomplishes this by storing a **%audio-command-stop** instruction at location zero (0) of the command array given, and issuing **audio:audio-start** at that command. Thus, **audio:audio-stop** is destructive to the command array, and requires that it be wired.

35.10 Conversions Between Sample Formats

The following functions encode and decode sample pairs. They are provided to hide the internal representation of sample pairs. Some of these "functions" are actually implemented as macros to help make code that prepares audio samples as fast as possible.

These functions convert between three formats of samples, *float*, *fixnum*, and *sample*. Float and fixnum formats describe channel values. Sample format is the actual format of sample pairs stored in command arrays and sample arrays.

Fixnum format consists of integers in the range -1**15 ≤ x < 1**15. Float format consists of floating numbers and float channels are in the range -1.0 ≤ x < 1.0. You must ensure that a float format value is never +1.0.

audio:float-channel-fix *float* — *Function*

Converts a float format value to fixnum format.

audio:fix-channel-float *fix* — *Function*

Converts a fixnum format value to float format.

audio:fix-sample *right* &optional *left right* — *Function*

Takes one or two fixnum format values for the two channels and returns a sample pair in sample format containing those two values.

audio:float-sample *right* &optional *left right* — *Function*

Takes one or two float format values for the two channels and returns a sample pair in sample format containing those two values.

audio:sample-channels *sample* — *Function*

Takes a sample pair in sample format and returns two values, the right and left channel values of that sample, respectively, in fixnum format.

audio:sample-add-fix *sample right-increment* &optional *left-increment right-increment* — *Function*

Takes a sample pair and one or two increments, which are expected to be in fixnum format. The two channels of the sample pair are incremented by the two increments, and a new sample pair so constructed is returned. If the right channel goes out of range, it overflows into the left channel instead of clipping.

audio:sample-add-float *sample right-increment* &optional *left-increment right-increment* — *Function*

Takes a sample pair and one or two increments, which are expected to be in float format. The two channels of the sample pair are incremented by the two increments, and a new sample pair so constructed is returned. If the right channel goes out of range, it overflows into the left channel instead of clipping.

audio:sample-add-sample *sample1 sample2* *Function*
Takes two sample pairs, in sample format, and produces a new sample pair by adding them. The operation performed is the addition of the fixnum format values corresponding to the channel values in the sample pairs. In other words, it is as if **audio:sample-add-sample** extracted the sample values from the sample pairs using **audio:sample-channels**, then added the channel values and reconstructed a sample pair using **audio:fix-sample**. The actual operation of **audio:sample-add-sample** is considerably more efficient.

35.11 Conversions for the Polyphony Feature

These functions convert between fixnum and float format channel values and the values stored in wavetables used by the polyphony feature. See the section "The Polyphony Feature", page 373.

audio:fix-polyphonic-wave-table-entry *right* &optional *(left right)* *Function*
Takes one or two channel values in fixnum format and returns a fixnum representing those two values, in the format used in wavetables. This is not the same as sample format.

audio:float-polyphonic-wave-table-entry *right* &optional *(left right)* *Function*
Takes one or two channel values in float format and returns a fixnum representing those two values, in the format used in wavetables. This is not the same as sample format.

audio:polyphonic-wave-table-entry-channels *entry* *Function*
Takes as an argument an *entry* from a polyphonic wavetable, and returns two values in fixnum format, the right and left channel values encoded therein, respectively.

35.12 Computing Polyphonic Increments

This function computes the appropriate wavetable increment to specify the frequencies in polyphonic textures.

audio:frequency-polyphonic-increment *frequency* *Function*

Computes an increment value suitable for use with **%audio-command-load-voice**. The increment produced corresponds to a frequency of *frequency*. That is, the increment returned causes the wavetable for the voice with which it is used to be scanned *frequency* times per second.

36. Examples of Using the Audio Facilities

This chapter presents seven program examples that use the digital audio facilities, in both real-time and non-real-time synthesis applications.

36.1 Sine Wave Example

This example generates a sine wave at a specified frequency.

```
(defun sine-wave (frequency)
  (audio:with-audio ()                  ;Set up the audio environment
    (let* ((start (audio:audio-index))  ;Get the current (starting) index
           (samples-per-cycle (round audio:*sample-rate* frequency))
           ;; Spread out several cycles to get a more accurate
           ;; frequency.  Extra factor of 2 makes sure there is room.
           (number-of-cycles (max 1 (floor (audio:audio-limit) (* samples-per-cycle 2))))
           ;; Actual number of samples we are going to produce
           (number-of-samples (* samples-per-cycle number-of-cycles)))
      ;; Make sure we have room to play this frequency
      (when (> (+ number-of-samples 2) (audio:audio-limit))
        (ferror "Frequency too low"))
      ;; This form allows us to compute number-of-samples inline
      ;; (as opposed to computing them in a separate array). If we
      ;; didn't know how many samples we were going to produce we could
      ;; supply NIL for number-of-samples and the form will keep track
      ;; and adjust the command array when the form is exited.  Since we
      ;; do supply the number of samples, the form will check to make
      ;; sure we supply exactly that many.  This helps us to avoid writing
      ;; incorrect audio programs.
      (audio:computing-immediate-audio-samples (number-of-samples)
        (loop for sample-number below number-of-samples
              as phase =
                 ;; This is the phase (angle) that is passed to sin
                 ;; to get the sine wave.  (This will cons double-floats in
                 ;; systems where cl:pi is a double-float.)
                 (// (* 2 cl:pi sample-number number-of-cycles)
                     number-of-samples)
              as sample =
                 ;; Take the sin of the phase.  Also multiply it
                 ;; by something less than 1 so we never get a
                 ;; value of 1.0 (a restriction, see
                 ;; documentation).  Take the resulting floating
                 ;; point number in the range [-1.0, +1.0) and
                 ;; create a 'sample.'
                 (audio:float-sample (* (sin phase) 0.9))
              do ;; Now actually push the sample into the command array.
           (audio:push-immediate-audio-sample sample)))
      ;; All of the samples are computed and an appropriate command has
      ;; been generated to output them.  Now we cause a jump back to the
      ;; beginning to keep the sound going.
      (audio:push-audio-jump start)
      ;; The program is complete, we can now start the audio facility.
```

```
(audio:audio-start start)
;; When you've heard enough, just type anything.  with-audio
;; supplies code to turn off the audio facility when exited and do
;; other bookkeeping.
(tyi))))
```

36.2 Sawtooth Wave Example

This is roughly the same as sine wave, but instead produces a sawtooth and only generates one cycle for it.

```
(defun saw-wave (frequency)
  (audio:with-audio ()
    (let* ((start (audio:audio-index))
           (samples-per-cycle (round audio:*sample-rate* frequency)))
      (audio:computing-immediate-audio-samples (samples-per-cycle)
        (loop for sample-number below samples-per-cycle
              as value =
                 ;; create a sawtooth value in the range [-1.0,1.0).
                 ;; Note this can never be exactly 1.0 since
                 ;; sample-number never quite gets as large as
                 ;; samples-per-cycle.
                 (- (// (* 2.0 sample-number) samples-per-cycle) 1.0)
              do (audio:push-immediate-audio-sample (audio:float-sample value)))
        (audio:push-audio-jump start)
        (audio:audio-start start)
        (tyi)))))
```

36.3 Square Wave Example

This example demonstrates yet another type of waveform: a square wave. The **audio:audio-loop** form is also exemplified.

```
(defun square-wave (frequency)
  (audio:with-audio ()
    (let* ((start (audio:audio-index))
           (samples-per-cycle (round audio:*sample-rate* frequency))
           ;; Compute the number of samples for the high value and
           ;; low value.  Divide them as evenly as possible.
           (samples-first-half (// samples-per-cycle 2))
           (samples-second-half (- samples-per-cycle samples-first-half)))
      ;; Create a loop that will repeat samples-first-half times.  If we
      ;; weren't sure how many times we want to repeat, we could specify
      ;; NIL and then use set-audio-repeat-count to set the count.
      (audio:audio-loop  (samples-first-half)
        ;; Compute 1 value (the high value) for output.
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample 0.9))))
      ;; Do the same for the second half.
      (audio:audio-loop (samples-second-half)
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample -0.9))))
      ;; Jump back to the beginning so we get more than one cycle.
      (audio:push-audio-jump start)
      (audio:audio-start start)
      (tyi))))
```

36.4 Beep Example

This is basically a modified square-wave.

```
(defun %beep-ignoring-most-issues (frequency duration)
  (audio:with-audio ()
    (let* ((start (audio:audio-index))
           (samples-per-cycle (sys:round audio:*sample-rate* frequency))
           (samples-first-half (// samples-per-cycle 2))
           (samples-second-half (- samples-per-cycle samples-first-half)))
      ;; Can't nest loops, so we have to do the outer loop with a jump
      ;; and bash the location when time has elapsed.
      (audio:audio-loop (samples-first-half)
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample 0.9))))
      (audio:audio-loop (samples-second-half)
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample -0.9))))
```

```
;; This is the tricky part.  We need to put a jump to the
;; beginning, but we need to know where it is so we can cause it
;; to fall through.  We also need a flag so we know when the audio
;; has stopped so we can exit.  If we simply exited without
;; waiting, the with-audio form could turn off the sound prematurely.
(let* (;; get the index that we will eventually bash and put in a
       ;; jump back to the start.
       (jump-index (prog1 (audio:audio-index) (audio:push-audio-jump start)))
       ;; reserve (and reset) an audio flag.
       (flag-index (audio:reserve-audio-flags 1))
       ;; reserve-audio-flags puts in a jump command around the
       ;; flags it reserves, so we could have gotten the
       ;; fall-through index after pushing the jump command.
       ;; Anyway, get the index of the fall-through location.
       (fall-through-index (audio:audio-index)))
  ;; When we bash the jump command the microcode will jump to here
  ;; instead, which will cause the flag to get zeroed and the
  ;; audio facility to stop.  Both events happen atomically as far
  ;; as Lisp can tell because no samples are output in the
  ;; intervening time.
  (audio:push-audio-zero-flag flag-index)
  (audio:push-audio-stop)
  ;; Start the audio
  (audio:audio-start start)
  ;; Wait the appropriate number of microseconds.
  (loop with start-time = (sys:%microsecond-clock)
        until
        (≥ (%32-bit-difference (sys:%microsecond-clock) start-time) duration))
  ;; Here is where we bash the argument of the jump command to
  ;; instead jump to the fall-through code.
  (audio:modify-audio-command-arg fall-through-index :index jump-index)
  ;; Wait for the microcode to get to the flag and stop before we exit.
  (audio:wait-for-audio-flag flag-index "%BEEP")))))
```

36.5 Non-real-time Synthesis Example

Certain kinds of very high quality sound cannot be generated in real time (one sample computed every 20 microseconds). Small pieces (pieces that can fit in physical memory) can be computed and then played later.

```
(defun play-audio-sample-array
  (array &optional (from 0) (to (array-active-length array)))
  (audio:with-audio ()
    ;; with-wired-structure wires the structure on entry
    ;;  and unwires on exit. External sample arrays must be wired.
    (si:with-wired-structure array
      (let* ((flag-index (audio:reserve-audio-flags 1))
             (start (audio:audio-index)))
        ;; Cause the samples to be played.  If we supplied a non-NIL
        ;; immediate-p argument, we wouldn't have to wire the
        ;; structure, since the samples would be put in the command
        ;; array which is already wired.  However, most command arrays
        ;; are not very large and probably couldn't hold all the
        ;; samples.  It's a tradeoff.
        (audio:push-array-of-audio-samples array from to)
        ;; When the microcode finishes the samples, cause it to clear
        ;; the flag and stop.
        (audio:push-audio-zero-flag flag-index)
        (audio:push-audio-stop)
        ;; Start it up and wait for it to finish.
        (audio:audio-start start)
        (audio:wait-for-audio-flag flag-index "Play samples")))))
```

36.6 Playing Large Pieces Example

Larger pieces (those that are too big to fit in physical memory) can still be played. This program plays data that is stored on the FEP filesystem. Storage must be on the FEP filesystem for several reasons. The digital audio system must produce data at the rate of one sample every 20 microseconds (including all overhead). This is 1.6 megabits per second, which is a small factor away from raw disk speed. After overhead, this is getting close to the limits of the system. The LMFS file system incurs too much overhead. Also, we cannot copy (as LMFS would try to do if we used **:string-in** into an array) and we cannot spend time wiring buffers (as we would need to do with LMFS if we used **:read-input-buffer**).

The FEP filesystem allows us to do disk direct memory access (DMA) directly into a buffer that we can keep wired. We can also setup the audio facility to point to these buffers (using **push-array-of-audio-samples**) once so we do not have to do it often.

The macro **with-multi-disk-buffering** takes care of multibuffering bookkeeping. The user decides how many pages to devote to each buffer and the number of buffers. Disk arrays (the buffers) are allocated and wired on entry and unwired on exit.

```
(defmacro with-multi-disk-buffering
    ((npages nbuffers) (array-of-buffers size-of-each-buffer) &body body)
  "npages and nbuffers are inputs, array-of-buffers and size-of-each-buffer are outputs
  `(let ((,array-of-buffers (make-array ,nbuffers))
         (,size-of-each-buffer (* ,npages 288.)))
     (unwind-protect
       (progn (loop for .idx. below ,nbuffers
                    as .buffer. = (allocate-resource 'si:disk-array
                                                     (+ ,size-of-each-buffer 288.))
                    do (setf (aref ,array-of-buffers .idx.) .buffer.)
                    (si:wire-structure .buffer.))
              ,@body)
       (loop for .idx. below ,nbuffers
             as .buffer. = (aref ,array-of-buffers .idx.)
             do (when (si:structure-wired-p .buffer.)
                  (si:unwire-structure .buffer.))
             (deallocate-resource 'si:disk-array .buffer.)))))
```

The function **play-disk-file** is the workhorse. There are many "if we are fast enough" clauses in this example. As long as there is not much other activity (especially paging activity) we usually are fast enough.

```
(defun play-disk-file (pathname)
  (setq pathname (fs:merge-pathnames pathname "FEP:>↔.mus.newest"))
  ;; get the FEP file opened.
  (with-open-file (file pathname :direction :block
                        :if-exists :overwrite
                        :if-does-not-exist :error)
    ;; These numbers were picked after much experimentation and tuning.
    (let* ((npages 40.) (nbuffers 8))
      (audio:with-audio ()
        (with-multi-disk-buffering (npages nbuffers) (buffers buffer-size)
          ;; allocate a flag for each buffer for synchronization.
          (let* ((flags (audio:reserve-audio-flags nbuffers))
                 (start (audio:audio-index)))
            ;; build the audio program.  Push each buffer as an array of
            ;; samples and then cause the flag associated with the
            ;; buffer to be zeroed.
            (loop for buffer below nbuffers
                  do (audio:push-array-of-audio-samples (aref buffers buffer)
                        0 buffer-size) (audio:push-audio-zero-flag
                        (+ flags buffer)))
```

```
;; Loop back to the beginning.  To play new data (if we are
;; fast enough, there /will/ be new data in the buffers).
(audio:push-audio-jump start)
;; n-queued is the number of buffers filled with valid data
;; that the microcode can use.  (The microcode will use
;; all of them, but if we are fast enough we can keep them full.)
;; We fill up all the buffers and then start the audio facility.
;; This is done by an interaction with need-to-start and n-queued.
;; (There is also provision for small files.)  When all the buffers
;; are queued, we need to wait for the microcode to finish
;; the next one before we can do disk dma into it.
(loop with n-queued = 0
      with need-to-start = t
      with n-file-blocks = (sys:ceiling (send file :length) 1152.)
      with current-file-block = 0
      initially (format t "~&~F seconds~%"
                        (// (* n-file-blocks 288.) audio:*sample-rate*))
      as blocks-this-whack =
         ;; This is the number of blocks to do this time
         ;; around.  It is at most the number of pages of
         ;; buffering.  It is also at most the number of
         ;; blocks remaining in the file.
         (min npages (- n-file-blocks current-file-block))
      for buffer-number =
          ;; This is the current buffer number we are going
          ;; to try to fill.  It is gets incremented modulo
          ;; the number of buffers.
          0 then (\ (1+ buffer-number) nbuffers)
      as flag-index = (+ flags buffer-number)
      do ;; If all the buffers are queued, or if the end of
         ;; the file has been reached, wait for the
         ;; microcode to finish the buffer and then count it
         ;; as dequeued.
         (when (or (= n-queued nbuffers) (zerop blocks-this-whack))
           (audio:wait-for-audio-flag flag-index "Play disk file")
           (decf n-queued))
         ;; If we have some blocks to queue, make sure the
         ;; flag for this buffer is reset, read in the
         ;; blocks from the FEP file, increment the block
         ;; pointer into the file, and count another buffer
         ;; as queued.
         (when (not (zerop blocks-this-whack))
           (audio:reset-audio-flag flag-index)
```

```
          (send file :block-in current-file-block blocks-this-whack
                (aref buffers buffer-number))
          (incf current-file-block blocks-this-whack)
          (incf n-queued))
        ;; If the audio facility hasn't been started and
        ;; all buffers are filled, start the audio facility
        ;; (and remember we did start it).
        (when (and need-to-start
                   (or (= n-queued nbuffers)
                       (≥ current-file-block n-file-blocks)))
          (audio:audio-start start)
          (setq need-to-start nil))
    until
      ;; We are finished when nothing is queued and we are
      ;; at the end of the file.
      (and (zerop n-queued)
           (≥ current-file-block n-file-blocks))))))))
```

36.7 Polyphony Example

This is a simple muse. It uses roughly the same multibuffering strategy as the disk example, so that portion will not be commented as heavily. (See the section "Playing Large Pieces Example", page 394.) The muse muses some number of voices (user specified) between 1 and 6. All voices start at DO (C). Each step (approximately every 1/4 second) causes each voice to wander randomly between 2 diatonic tones below the previous value and 2 diatonic tones above the previous value.

```
;;; Figure out how large wave tables are in this release.

(defconst *samples-per-polyphonic-wave-table*
          (expt 2 (byte-size sys:%%audio-increment-integer)))


;; This is the wave-array for the muse.
;; It is big enough to ensure that there will be at least
;; *samples-per-polyphonic-wave-table* consecutive wired words.
```

```
(defvar *muse-wave-array*
        (make-array (+ *samples-per-polyphonic-wave-table* sys:page-size -1)
                                              :initial-value 0 :area audio:audio-area))

(defun polyphonic-muse (&optional (n-voices 4) &aux address wired)
  (check-arg n-voices (and (fixp n-voices)
        (≤ 1 n-voices audio:*number-of-polyphonic-voices*))
             "an integer between 1 and 6")
  (audio:with-audio ()
    (unwind-protect
      (let ((offset-to-page
              ;; This is how one gets to the number of Qs
              ;;  to the beginning of a page boundary
              (ldb sys:%%vma-word-offset
                   (- sys:page-size
                        (ldb sys:%%vma-word-offset
                        (%pointer (locf (aref *muse-wave-array* 0))))))))
        ;;; Wire words of the wave table, starting at
        ;;;  the location computed above.
        (setq address (locf (aref *muse-wave-array* offset-to-page)))
        (si:wire-consecutive-words
          address                                 ;where
          *samples-per-polyphonic-wave-table*)  ;how many, one per word.
        (setq wired t)                          ;Set a reminder to unwire it...
```

```
;; Set up the muse wave array for a 1/6 (minus a bit) amplitude
;;  sinewave (sawtooth doesn't seem to sound good here).  1/6
;;  allows all six voices to proceed without overflow.  The
;;  "minus a bit" avoids clipping at 1.0.
(loop for index below *samples-per-polyphonic-wave-table*
      do (setf (aref *muse-wave-array* (+ index offset-to-page))
               (audio:float-polyphonic-wave-table-entry
                 (// (sin (// (* 2.0 si:pi index)
                              *samples-per-polyphonic-wave-table*)) 6.2))))
;; Initialize each voice to a reasonable value.  It is essential
;; that each voice gets a proper wave-array pointer and
;; increment value.  An increment value of 0 will cause the
;; pointer never to be incremented.  (This isn't strictly true,
;; since the voice number is stored in the low 3 bits, but this
;; advances the pointer very slowly.)
(let ((start (audio:audio-index)))
  (loop for voice below audio:*number-of-polyphonic-voices*
     do
  (audio:push-audio-load-voice voice *muse-wave-array* offset-to-page 0))
  (audio:push-audio-stop)
  (audio:audio-start start)
  ;; put the audio index back to the start
  (setf (audio:audio-index) start))
(loop with nbuffers = 4
      with n-queued = 0
      with need-to-start = t
      with flags = (audio:reserve-audio-flags nbuffers)
      with start = (audio:audio-index)
      with chords-per-whack =
        ;; Take the room remaining, divide by the level of
        ;; buffering and then divide by the sum of [2 locations
        ;; per voice for the push-audio-load-voice command, one
        ;; for the push-audio-polyphony command, and one for a
        ;; possible flag or jump].
        (// (audio:audio-room) nbuffers (+ (* n-voices 2) 1 1))
      with half-tone-offsets =
        ;; 0 (and the multiples of 12) are DO.  The other
        ;; numbers are offsets (from 0) to consecutive notes in
        ;; the diatonic scale.
        '(-25. -24. -22. -20. -19. -17. -15. -13.
               -12. -10. -08. -07. -05. -03. -01.
               000. +02. +04. +05. +07. +09. +11.
               +12. +14. +16. +17. +19. +21. +23.
```

```
          +24. +26. +28. +29. +31. +33. +35.)
with half-tone-offsets-length = (length half-tone-offsets)
```

```
    with voice-indices =
      ;; A list, one element for each voice, starting at middle DO.
      (make-list n-voices
         :initial-value (find-position-in-list 000. half-tone-offsets))
    for buffer-number = 0 then (\ (1+ buffer-number) nbuffers)
    until (kbd-tyi-no-hang)   ; Stop when user hits a key
    do
(when (≥ n-queued nbuffers)
  ;; this also resets the flag
  (audio:wait-for-audio-flag (+ flags buffer-number) "Muse")
  (decf n-queued))
;; If this is buffer zero, make sure we are back to the start.
(when (zerop buffer-number)
  (setf (audio:audio-index) start))
;; setup the chords for this buffer
(loop repeat chords-per-whack
      do ;; update each voice
         (loop for voice-indices-scan on voice-indices
               as old-index = (car voice-indices-scan)
               as new-index =
                      (let ((index (+ old-index (random 5) -2)))
                           ;; clip at the boundaries of the list
                           (cond ((< index 0) 1)
                                 ((≥ index half-tone-offsets-length)
                                        (- half-tone-offsets-length 2))
                                  (T index)))
               do (setf (car voice-indices-scan) new-index))
         ;; And queue the new values to polyphony facility
         (loop for index in voice-indices
               for voice-number upfrom 0
               as half-tone-offset = (nth index half-tone-offsets)
               as octave-offset = (// half-tone-offset 12.0)
               as frequency-factor = (expt 2.0 octave-offset)
               as frequency = (* 256.0 frequency-factor)
               do (audio:push-audio-load-voice
                    voice-number *muse-wave-array* offset-to-page
                    (audio:frequency-polyphonic-increment frequency)))
         ;; Do polyphony for 1/4 second
         (audio:push-audio-polyphony (sys:round audio:*sample-rate* 4)))
;; synchronize this buffer
(audio:push-audio-zero-flag (+ flags buffer-number))
(incf n-queued)
(when (and (≥ n-queued nbuffers) need-to-start)
```

```
      (audio:push-audio-jump start)
      (audio:audio-start start)
      (setq need-to-start nil))))
(when wired
  (si:unwire-words address *samples-per-polyphonic-wave-table*)))))
```

PART VI.

Dates and Times

37. Representation of Dates and Times

The **zl:time** package contains a set of functions for manipulating dates and times: finding the current time, reading and printing dates and times, converting between formats, and other miscellany regarding peculiarities of the calendar system. It also includes functions for accessing the Lisp Machine's microsecond timer.

Times are represented in two different formats by the functions in the **time** package. One way is to represent a time by many numbers, indicating a year, a month, a date, an hour, a minute, and a second (plus, sometimes, a day of the week and time zone). The year is relative to 1900 (that is, if it is 1984, the *year* value would be **84**); however, the functions that take a year as an argument will accept either form. The month is 1 for January, 2 for February, and so on. The date is 1 for the first day of a month. The hour is a number from 0 to 23. The minute and second are numbers from 0 to 59. Days of the week are fixnums, where 0 means Monday, 1 means Tuesday, and so on. A time zone is specified as the number of hours west of GMT; thus in Massachusetts the time zone is 5. Any adjustment for daylight saving time is separate from this.

This "decoded" format is convenient for printing out times in a readable notation, but it is inconvenient for programs to make sense of these numbers, and pass them around as arguments (since there are so many of them). So there is a second representation, called Universal Time, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. This "encoded" format is easy to deal with inside programs, although it doesn't make much sense to look at (it looks like a huge integer). So both formats are provided; there are functions to convert between the two formats; and many functions exist in two forms, one for each format.

The Lisp Machine hardware includes a timer that counts once every microsecond. It is controlled by a crystal and so is fairly accurate. The absolute value of this timer doesn't mean anything useful, since it is initialized randomly; what you do with the timer is to read it at the beginning and end of an interval, and subtract the two values to get the length of the interval in microseconds. These relative times allow you to time intervals of up to an hour (32 bits) with microsecond accuracy.

The Lisp Machine keeps track of the time of day by maintaining a "timebase", using the microsecond clock to count off the seconds. When the machine first comes up, the timebase is initialized by querying hosts on the local network to find out the current time.

A similar timer counts in 60ths of a second rather than microseconds; it is useful for measuring intervals of a few seconds or minutes (or hours, which are longer than the microsecond timer's range) with less accuracy. Periodic housekeeping functions of the system are scheduled based on this timer.

38. Getting and Setting the Time

time:get-time *Function*

Get the current time, in decoded form. Return seconds, minutes, hours, date, month, year, day-of-the-week, and daylight-savings-time-p, with the same meanings as **time:decode-universal-time**.

get-universal-time *Function*

Returns the current time, in Universal Time form.

time:set-local-time &optional *new-time* *Function*

Set the local time to *new-time*. If *new-time* is supplied, it must be either a universal time or a suitable argument to **time:parse**. If it is not supplied, or if there is an error parsing the argument, you will be prompted for the new time. Note that you will not normally need to call this function; it is mainly useful when the timebase becomes unreliable for one reason or another.

38.1 The 3600-Family Calendar Clock

Machines in the 3600 family have a calendar clock that operates independently of the other Lisp Machine timers. When you cold boot and the machine fails to get the time from the network, it asks you to type in the time. If the calendar clock has been set, it uses the calendar clock reading as the default for the time you specify. If the calendar clock has not been set, it offers to set it to the time you type in. See the function **time:initialize-timebase**, page 423.

You can also set the calendar clock yourself using **time:set-calendar-clock** and read it using **time:read-calendar-clock**.

time:set-calendar-clock *new-time* *Function*

Sets the calendar clock to *new-time*, which must be either a universal time or a suitable argument to **time:parse**. Returns **t** if the calendar clock is set successfully, otherwise **nil**.

time:read-calendar-clock &optional *even-if-bad* *Function*

Attempts to read the calendar clock. If the attempt is unsuccessful, returns **nil**. If the attempt is successful and the time appears to be valid, returns the time in universal time form. If the attempt is successful but the time appears to be invalid, takes action depending on the value of *even-if-bad*:

nil or unspecified	Returns **nil**
Not **nil**	Attempts to convert the internal format to universal time. If the conversion is successful, returns the time in universal time form. Otherwise, signals an error.

38.2 Elapsed Time in 60ths of a Second

Rather than calendrical date/times, the following functions deal with elapsed time in 60ths of a second. These times are used for many internal purposes where the idea is to measure a small interval accurately, not to depend on the time of day or day of month.

zl:time *Function*

Returns a number that increases by 1 every 1/60 of a second, and "wraps around" less than once a day. Use the **time-lessp** and **time-difference** functions to avoid getting in trouble due to the wraparound. **zl:time** is completely incompatible with the Maclisp function of the same name.

time-lessp *time1 time2* *Function*

t if *time1* is earlier than *time2*, compensating for wraparound, otherwise **nil**. Also works for **time:fixnum-microsecond-time** values.

time-difference *time1 time2* *Function*

Assuming *time1* is later than *time2*, returns the number of 60ths of a second difference between them, compensating for wraparound. Also works for **time:fixnum-microsecond-time** values.

time-increment *time increment* *Function*

Adds *increment* to *time* and returns the resulting time value, compensating for wraparound. *time* should be a value of time, as returned by the **zl:time** function, and *increment* should be an amount of time expressed as a fixnum in units of 60ths of a second. Also works for **time:fixnum-microsecond-time** values.

time-elapsed-p *increment initial-time* &optional (*final-time* **(zl:time)**) *Function*

Returns **t** if at least *increment* 60ths of a second have elapsed between *initial-time* and *final-time*. Otherwise, returns **nil**.

initial-time and *final-time* should be time values as returned by the **zl:time** function. *final-time* defaults to the result of **(zl:time)**.

Example:

```
(defun process-sleep (interval &optional (whostate "Sleep"))
  (process-wait whostate #'time-elapsed-p interval (time)))
```

38.3 Elapsed Time in Microseconds

time:microsecond-time *Function*

Return the value of the microsecond timer, as a bignum. The values returned by this function "wrap around" about once per hour.

time:fixnum-microsecond-time *Function*

Return the value of the low 31 bits of the microsecond timer, as a fixnum. This is like **time:microsecond-time**, with the advantage that it returns a value in the same format as the **zl:time** function, except in microseconds rather than 60ths of a second. This means that you can compare fixnum-microsecond-times with **time-lessp** and **time-difference**. **time:fixnum-microsecond-time** is also a bit faster, but has the disadvantage that since you only see the low bits of the clock, the value can "wrap around" more quickly (about every half hour).

39. Printing Dates and Times

The functions in this section create printed representations of times and dates in various formats, and send the characters to a stream. To any of these functions, you may pass **nil** as the *stream* parameter, and the function will return a string containing the printed representation of the time, instead of printing the characters to any stream.

time:print-current-time &optional (*stream* **zl:standard-output**) *Function*

Print the current time, formatted as in **11/25/83 14:50:02**, to the specified stream.

time:print-time *seconds minutes hours day month year* &optional (*stream* **zl:standard-output**) *Function*

Print the specified time, formatted as in **11/25/83 14:50:02**, to the specified stream.

time:print-universal-time *ut* &optional (*stream* **zl:standard-output**) *timezone* *Function*

Print the specified time, formatted as in **11/25/83 14:50:02**, to the specified stream.

time:print-current-date &optional (*stream* **zl:standard-output**) *Function*

Print the current time, formatted as in **Friday the twenty-fifth of November, 1983; 3:50:41 pm**, to the specified stream.

time:print-date *seconds minutes hours day month year day-of-the-week* &optional (*stream* **zl:standard-output**) *Function*

Print the specified time, formatted as in **Friday the twenty-fifth of November, 1983; 3:50:41 pm**, to the specified stream.

time:print-universal-date *ut* &optional (*stream* **zl:standard-output**) *timezone* *Function*

Print the specified time, formatted as in **Friday the twenty-fifth of November, 1983; 3:50:41 pm**, to the specified stream.

time:print-brief-universal-time *ut* &optional (*stream* **zl:standard-output**) (*ref-ut* **(get-universal-time)**) *Function*

This is like **time:print-universal-time** except that it omits seconds and only prints those parts of *ut* that differ from *ref-ut*, a universal time that defaults to the current time. Thus the output will be in one of the following three forms:

```
02:59            ;the same day
3/4 14:01        ;a different day in the same year
8/17/74 15:30    ;a different year
```

zl:format accepts some directives for printing dates and times.

40. Reading Dates and Times

These functions accept most reasonable printed representations of date and time and convert them to the standard internal forms. The following are representative formats that are accepted by the parser:

```
"March 15, 1960" "15 March 1960" "3//15//60" "3//15//1960"
"3-15-60" "3-15" "15-March-60" "15-Mar-60" "March-15-60"
"1960-3-15" "1960-March-15" "1960-Mar-15"
"1130." "11:30" "11:30:17" "11:30 pm" "11:30 am" "1130" "113000"
"11.30" "11.30.00" "11.3" "11 pm" "12 noon"
"midnight" "m" "Friday, March 15, 1980" "6:00 gmt" "3:00 pdt"
"15 March 60" "15 March 60 seconds"
"fifteen March 60" "the fifteenth of March, 1960;"
"one minute after March 3, 1960"
"two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"
"now" "today" "yesterday" "two days after tomorrow"
"one day before yesterday" "the day after tomorrow"
"five days ago"
```

The parsing functions accept date strings in ISO standard format. These strings are of the form "yyyy-mm-dd", where:

yyyy	Four digits representing the year
mm	The name of the month, an abbreviation for the month, or one or two digits representing the month
dd	One or two digits representing the day

Following are some restrictions on strings to be parsed:

- You cannot represent any year before 1900.
- A four-digit number alone is interpreted as a time of day, not a year. For example, "1954" is the same as "19:54:00" or "7:54 pm", not the year 1954.
- The parser does not recognize dates in European format. For example, "3//4//85" or "3-4-85" is always the same as "March 4, 1985", never "April 3, 1985". A string like "15//3//85" is an error. In such strings, the first integer is always parsed as the month and the second integer as the day.

time:parse *string* &optional (*start* **0**) *end* (*futurep* **t**) *base-time* *must-have-time* *date-must-have-year* *time-must-have-second* (*day-must-be-valid* **t**) *Function*

Interpret *string* as a date and/or time, and return seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p, and relative-p. *start* and *end* delimit a substring of the string; if *end* is **nil**, the end of the string is used. *must-have-time* means that *string* must not be empty. *date-must-have-year* means that a year must be explicitly specified. *time-must-have-second* means that the second must be specified. *day-must-be-valid* means that if a day of the week is given, then it must actually be the day that corresponds to the date. *base-time* provides the defaults for unspecified components; if it is **nil**, the current time is used. *futurep* means that the time should be interpreted as being in the future; for example, if the base time is 5:00 and the string refers to the time 3:00, that means the next day if *futurep* is non-**nil**, but it means two hours ago if *futurep* is **nil**. The *relative-p* returned value is **t** if the string included a relative part, such as "one minute after" or "two days before" or "tomorrow" or "now"; otherwise, it is **nil**.

time:parse-universal-time *string* &optional (*start* **0**) *end* (*futurep* **t**) *base-time* *must-have-time* *date-must-have-year* *time-must-have-second* (*day-must-be-valid* **t**) *Function*

This is the same as **time:parse** except that it returns one integer, representing the time in Universal Time, and the *relative-p* value. It also returns a third value, which is **t** if hours, minutes, or seconds were specified by *string*, or **nil** if they were not.

time:parse-universal-time-relative *date-spec* *reference-date-spec* &optional (*future-p* **t**) *Function*

Like **time:parse-universal-time**, except that *date-spec* is parsed relative to *reference-date-spec*. The returned values are the same as those of **time:parse-universal-time**.

date-spec is a string suitable as the first argument to **time:parse-universal-time**. *reference-date-spec* is a universal-time integer or a string that can be parsed as an unambiguous time. If *future-p* is **nil**, an ambiguous *date-spec* is interpreted as being in the past relative to *reference-date-spec*; otherwise, it is interpreted as being in the future. The default for *future-p* is **t**.

For example:

```
(time:parse-universal-time-relative "5 pm" "today")
```

returns the same value when evaluated anytime today, whether or not the current time is before or after 5 pm.

time:parse-present-based-universal-time *time-being-parsed* *Function*

Like **time:parse-universal-time**, except that missing components in *time-being-parsed* are defaulted to the beginning of the smallest unsupplied unit of time. The returned values are the same as those of **time:parse-universal-time.** *time-being-parsed* is a string suitable as the first argument to **time:parse-universal-time.**

For example, "5 pm" is parsed as 5 pm on the current day, whether the current time is before or after 5 pm. "Thursday" is parsed as Thursday of the current week, whether today is Wednesday or Friday. "1 June" is parsed as June 1 of the current year, whether the date is before or after June 1.

41. Reading and Printing Time Intervals

Several functions read and print time intervals. They convert between strings of the form "3 minutes 23 seconds" and integers representing numbers of seconds.

time:print-interval-or-never *interval* &optional (*stream* **zl:standard-output**) *Function*

Prints the representation of *interval* as a time interval onto *stream*. If *interval* is **nil**, it prints "Never". *interval* should be a nonnegative integer, or **nil**.

time:parse-interval-or-never *string* &optional *start end* *Function*

string is the character-string representation of an interval of time. *start* and *end* specify a substring of *string* to be parsed; they default to the beginning and end of *string*, respectively. The function returns an integer if *string* represented an interval, or **nil** if *string* represented "never". If *string* is anything else, an error occurs. Examples of acceptable strings:

```
"4 seconds"          "4 secs"         "4 s"
"5 mins 23 secs"     "5 m 23 s"       "23 SECONDS 5 M"
"never"              "not ever"       "no"
""                   "3 yrs 1 week 1 hr 2 mins 1 sec"
```

Note that several abbreviations are understood, the components can be in any order, and case (upper versus lower) is ignored. Also, "months" is not acceptable, since months vary in length. This function accepts anything that **time:print-interval-or-never** produces, and it returns the same integer (or **nil**).

time:read-interval-or-never &optional (*stream* **zl:standard-input**) *Function*

Reads a line of input from *stream* (using **zl:readline**) and calls **time:parse-interval-or-never** on the resulting string.

42. Time Conversions

time:decode-universal-time *universal-time* &optional *timezone* *Function*

Convert *universal-time* into its decoded representation. The following values are returned: seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p. *daylight-savings-time-p* tells you whether or not daylight savings time is in effect; if so, the value of *hour* has been adjusted accordingly. You can specify *timezone* explicitly if you want to know the equivalent representation for this time in other parts of the world.

time:encode-universal-time *seconds minutes hours day month year* &optional *timezone* *Function*

Convert the decoded time into Universal Time format, and return the Universal Time as an integer. If you do not specify *timezone*, it defaults to the current time zone adjusted for daylight saving time; if you provide it explicitly, it is not adjusted for daylight saving time. *year* may be absolute, or relative to 1900 (that is, **84** and **1984** both work).

time:*timezone* *Variable*

The value of **time:*timezone*** is the time zone in which this Lisp Machine resides, expressed in terms of the number of hours west of GMT this time zone is. This value does not change to reflect daylight saving time; it tells you about standard time in your part of the world.

43. Internal Time Functions

These functions provide support for functions that deal with dates and time. Some user programs may need to call them directly, so they are documented here.

For more information on functions that deal with dates and times:

See the section "Getting and Setting the Time", page 409.
See the section "Elapsed Time in 60ths of a Second", page 410.
See the section "Elapsed Time in Microseconds", page 411.
See the section "Printing Dates and Times", page 413.
See the section "Reading Dates and Times", page 415.
See the section "Reading and Printing Time Intervals", page 419.
See the section "Time Conversions", page 421.

time:initialize-timebase &optional *ut* (*use-network* **t**) *Function*

Initializes the timebase. If *ut*, a universal-time integer, is supplied, uses *ut* as the current time. If *ut* is **nil** or unspecified and if *use-network* is not **nil**, queries local network hosts to find out the current time. (*use-network* is **t** by default.) If it cannot get the time from the network, or if *ut* and *use-network* are both **nil**, prompts the user for a string to parse as the current time. On machines in the 3600 family, if the calendar clock has been set, uses the calendar clock reading as the default time for the user to specify. If the calendar clock has not been set, offers to set it to the time that the user specifies.

This is called automatically during system initialization. You may want to call it yourself to correct the time if it appears to be inaccurate or downright wrong. See the function **time:set-local-time**, page 409.

time:daylight-savings-time-p *hours day month year* *Function*

Return **t** if daylight saving time is in effect for the specified hour; otherwise, return **nil**. *year* may be absolute, or relative to 1900 (that is, **84** and **1984** both work).

time:daylight-savings-p *Function*

Return **t** if daylight saving time is currently in effect; otherwise, return **nil**.

time:month-length *month year* *Function*

Return the number of days in the specified *month*; you must supply a *year* in case the month is February (which has a different length during leap years). *year* may be absolute, or relative to 1900 (that is, **84** and **1984** both work).

time:leap-year-p *year* *Function*

Return t if *year* is a leap year; otherwise return **nil**. *year* may be absolute, or relative to 1900 (that is, **84** and **1984** both work).

time:verify-date *day month year day-of-the-week* *Function*

If the day of the week of the date specified by *day*, *month*, and *year* is the same as *day-of-the-week*, return **nil**; otherwise, return a string that contains a suitable error message. *year* may be absolute, or relative to 1900 (that is, **84** and **1984** both work).

time:day-of-the-week-string *day-of-the-week* &optional (*mode* **':long**) *Function*

Return a string representing the day of the week. As usual, **0** means Monday, **1** means Tuesday, and so on. Possible values of *mode* are:

:short Return a three-letter abbreviation, such as "**mon**", "**tue**", and so on.

:long Return the full English name, such as "**monday**", "**tuesday**", and so on. This is the default.

:medium Same as **:short**, but use "**tues**" and "**thurs**".

:french Return the French name, such as "**lundi**", "**mardi**", and so on.

:german Return the German name, such as "**montag**", "**dienstag**", and so on.

:italian Return the Italian name, such as "**lunedi**", "**martedi**", and so on.

time:month-string *month* &optional (*mode* **':long**) *Function*

Return a string representing the month of the year. As usual, **1** means January, **2** means February, and so on. Possible values of *mode* are:

:short Return a three-letter abbreviation, such as "**jan**", "**feb**", and so on.

:long Return the full English name, such as "**january**", "**february**", and so on. This is the default.

:medium Same as **:short**, but use "**sept**", "**novem**", and "**decem**".

:french Return the French name, such as "**janvier**", "**fevrier**", and so on.

:roman Return the Roman numeral for *month* (this convention is used in Europe).

:german Return the German name, such as "**januar**", "**februar**", and so on.

:italian Return the Italian name, such as "**gennaio**", "**febbraio**", and so on.

time:timezone-string &optional (*timezone* **time:*timezone***) (*daylight-savings-p* **(time:daylight-savings-p time:timezone)**) *force-numeric-p punctuate* *Function*

Returns the printed representation of a timezone; the default timezone is the current one for the user's site. The value returned is either the commonly accepted abbreviation for the timezone, for example, `"EST"` (for Eastern Standard Time); or, if more than one or no abbreviation is available, a signed digit string, for example, `"-0500"`.

The sign of a returned digit string indicates the location of the timezone relative to Greenwich; positive means east, negative west. Note that the sign of the printed representation is opposite to that used internally; the printed digit string `"-0500"`, for example, corresponds to an internal representation of `5.0`.

timezone A number between -12 and 12 of the form *n*.0 or *n*.5. This number is the internal representation of the timezone whose printed representation is returned. Its sign is positive if you want to specify a timezone west of Greenwich, negative for one east of Greenwich. The value returned depends on the setting of the *daylight-savings-p* flag.

daylight-savings-p
Boolean option specifying whether the *timezone* argument refers to the daylight-savings timezone or non-daylight-savings timezone. For example, supplying `5` as the *timezone* argument returns `"EST"` when *daylight-savings-p* is **nil** and `"EDT"` (Eastern Daylight Time) when it is **t**.

For timezones for which straightforward rules exist governing the change from standard to daylight-savings time and back again, the timezone utility automatically switches over to the appropriate abbreviation. For other timezones, the switch must be made manually. For more information: See the section "Specifying a Time Zone for Your Site" in *Site Operations*.

force-numeric-p
Boolean option specifying whether to force the return of a signed digit string, even if a unique abbreviation is available.

punctuate

Boolean option specifying whether to insert a space at the beginning of the returned abbreviation string, for example, " EST" versus "EST".

PART VII.

Zwei Internals

44. Introduction to Zwei Internals

Zmacs, the Lisp machine editor, is built on a large and powerful system of text-manipulation functions and data structures, called *Zwei.*

Zwei is not an editor itself, but rather a system on which other text editors are implemented. For example, in addition to Zmacs, the Zmail mail reading system also uses Zwei functions to allow editing of a mail message as it is being composed or after it has been received. The subsystems that are established upon Zwei are:

- Zmacs, the editor that manipulates text in files
- Dired, the editor that manipulates directories represented as text in files
- Zmail, the editor that manipulates text in mailboxes
- Converse, the editor that manipulates text in messages

Since these subsystems share Zwei in the dynamically linked Lisp environment, many of the commands available as Zmacs commands are available in other editing contexts as well.

45. Stream Facility for Editor Buffers

zwei:open-editor-stream opens a stream to an editor buffer; it is analogous to **open** for files. **zwei:with-editor-stream** also opens a stream to an editor buffer; it is analogous to **with-open-file** for files.

45.1 The zwei:with-editor-stream Macro

zwei:with-editor-stream *(name options) body ...* *Macro*

zwei:with-editor-stream opens a bidirectional stream called *name* to a buffer, which is designated in one of the following ways:

an interval
a buffer name
a Zwei window
a pathname

It takes the same keyword options as **zwei:open-editor-stream**. See the section "Keyword Options", page 432. On exit, it sends a **:force-redisplay** message to the stream, which causes the editor to do any necessary redisplay.

45.2 The zwei:open-editor-stream Function

zwei:open-editor-stream *options* *Function*

zwei:open-editor-stream is used by **zwei:with-editor-stream**. You might sometimes need to call it directly for doing operations that need not be in the scope of a "with" form (for the same reasons that you would use **open** instead of **with-open-file** for file I/O). For example, you would use this in conjunction with **with-open-stream-case** for appropriate error signalling.

It takes the same keyword options as **zwei:with-editor-stream**. See the section "Keyword Options", page 432.

You can send a **:force-redisplay** message at any time while the stream is open.

45.3 Keyword Options

zwei:with-editor-stream and **zwei:open-editor-stream** both recognize the same set of keyword options. Some of the options are mutually exclusive and some are interdependent.

You specify where to find the text by using one of the following keywords, whichever is appropriate to the situation. The keywords appear here in priority order. When the options specify several of these, one from the top of the list overrides one from further down in the list, regardless of what order the keywords appear in the options list.

:interval
:buffer-name
:pathname
:window
:start

The options refer to an object called a *bp*. This is a Zwei data structure for representing a particular position in a buffer.

Option — *Values and meaning*

:buffer-name — The full name of a buffer to use for the stream.

```
(zwei:with-editor-stream
  (foo ':buffer-name (send zwei:*interval* ':name))
  ...)
```

The buffer does not need to exist (see **:create-p**). The following example creates a Zmacs buffer named temp and opens the stream **foo** to it.

```
(zwei:with-editor-stream (foo "temp")
  ...)
```

:create-p — Specifies what to do when the buffer does not exist. This applies only in conjunction with **:buffer-name** or **:pathname** with **:load-p**.

Value	*Meaning*
:ask	Queries the user before creating the buffer.
:error	Signals an error and provides proceed types for creating it or supplying an alternate.
t	Creates the buffer.
:warn	Notifies the user that a buffer is being created (the default).

:defaults Specifies the pathname defaults against which a **:pathname** option would be merged. These are necessary in case reprompting needs to occur. The default is **nil**, meaning to use the default defaults. This option applies only in conjunction with **:pathname**.

:end Specifies the conditions for terminating the stream (the "end of file" condition).

Value	*Meaning*
bp	Stops when this buffer bp is reached.
:end	Stops at the end of the buffer (the default). This applies only if **:start** was also a bp.
:mark	Stops when it reaches the mark. This option requires that you use the **:window** option as well.
:point	Stops when it reaches point. This option requires that you use the **:window** option as well.

:hack-fonts Specifies how to treat font shifts in the buffer.

Value	*Meaning*
nil	Ignores font shifts (the default).
t	Provides full font support. Encodes font shifts on both input and output using epsilons, as would go to a file.

:interval Specifies a Zwei interval to use for the stream.

:kill Specifies what to do with the buffer before using it as a stream.

Value	*Meaning*
nil	No action (the default)
t	Deletes all the text currently in the designated part of the buffer.

:load-p Specifies whether to read the file specified by **:pathname** into the editor before using the buffer as a stream. (This is analogous to Find File in Zmacs.) This works only from within Zmacs.

Value	*Meaning*
nil	No action (the default)
t	Loads the file into the editor.

:no-redisplay	Suppresses the redisplay of any windows associated with the interval being written into.
	`(zwei:with-editor-stream`
	`  (standard-output :buffer-name "Herald" :no-redisplay t)`
	`  (print-herald))`

:ordered-p	States whether **:start** and **:end** are guaranteed to be in forward order. The default is **nil**. This applies only when **:start** and **:end** are bps or **:point** and **:mark**.
:pathname	Specifies a pathname to use for the stream. This can be a pathname object or any file spec that can be coerced to a pathname by **fs:parse-pathname**.
:start	Specifies where to start the stream with respect to the buffer contents.

Value	*Meaning*
:append	Starts at the end of the buffer. (Same as **:end**.)
:beginning	Starts at the beginning of the buffer.
bp	Starts with this bp.
:end	Starts at the end of the buffer (the default). (Same as **:append**.)
:mark	Starts at the mark, which does not move as a result. This requires a Zmacs window.
:point	Starts at point, which does not move as a result. This requires that you use the **:window** option as well.
:region	Starts at point and ends at mark (or vice versa, depending on the ordering). This requires that you use the **:window** option as well. It ignores any **:end** in this case.

:window	Specifies a Zmacs window as the stream source.

zwei:with-editor-stream does not currently interlock to prevent simultaneous access to a single buffer by multiple processes. Neither does anything else. Trying to access the same buffer with several processes simultaneously is not guaranteed to work.

46. Making Standalone Editor Windows

You can create an editor window with the following properties:

- Should be standalone (have its own process).
- Need not have the buffer structure of Zmacs.
- Need not even have minibuffers. If I must have one, I want the pop-up style.
- Needs a special comtab. That comtab will have commands that make the window do something worthwhile.

To create such a window, follow this procedure:

Start with **zwei:standalone-editor-frame**. Send it an **:edit** message to make it edit. It does not have its own process by default; you can mix **tv:process-mixin** with it and make that process send the **:edit** message if you want it to have its own process.

Two other useful messages:

:set-interval-string
Inserts a string in the editor.

:interval-string Returns a string to the caller when **:edit** returns.

For providing a special comtab, you can initialize the instance variable **zwei:*comtab*** by using the **:*comtab*** keyword in the init plist.

You can exit from this kind of editor by using END.

Index

B B B

D D D

E

F

G G G

H H H

K

M M M

O

P

Q

R

S

T

U

V

W

Z