abbreviating-output (&optional *stream* &key *width height lozenge-returns newline-substitute show-abbreviation abbreviate-initial-whitespace*) &body *body* *Function*

Binds local environment such that character output is abbreviated. That is, output exceeding a specified width or height (in characters) is truncated.

stream The output stream; the default is ***standard-output***.

:width Specifies the width, in characters, beyond which abbreviation occurs, or **t** or **nil**. If **nil**, the default, individual lines are not truncated. If **t**, the width used is the value returned by the stream's **:size-in-characters**.

:height Specifies the height, in lines, beyond which abbreviation occurs, or **t** or **nil**. If **nil**, the default, no truncation occurs. If **t**, the height used is the value returned by the stream's **:size-in-characters**.

:lozenge-returns
Boolean option specifying whether **#\return** characters at line truncations are displayed within a lozenge, rather than causing a newline; the default is **nil**.

:newline-substitute
This is a generalization of **:lozenge-returns**. When given a *string* value, line truncations are displayed with that string, rather than causing a newline; the default is **nil**.

:show-abbreviation
Boolean option specifying whether an ellipsis (. . .) is displayed where output truncation occurs. The default is **nil**, meaning that there is no explicit indication that truncation has occurred.

You can also specify a string as an argument to :show-abbreviation. If you specify a string, the string appears after the abbreviation instead of ellipsis (...).

:abbreviate-initial-whitespace
Boolean option specifying that initial whitespace (spaces, tabs, newlines) be suppressed; the default is **nil**.

Example:

```
(defun abbrev-test (width height lozenge-p)
  (abbreviating-output (() :width width :height height
                           :lozenge-returns lozenge-p
                           :show-abbreviation t)
    (loop for row from 1 to 20 do
      (terpri)
      (loop for col from 1 to 100 do
        (format T " ~d:~d" row col)))))

(abbrev-test 42 10 nil)
```

The *body* code continues to run normally to completion, even though its output to *stream* may be truncated.

Within **abbreviating-output**, the **:set-cursorpos** operation is restricted. Only the *x* position may be specified, and then, only in characters.

For an overview of **abbreviating-output** and related facilities, see the section "Controlling Line Output".

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.

accept

presentation-type &key (*stream* ***query-io***) (*prompt* **:enter-type**) (*prompt-mode* **:normal**) *(original-type* **dw:presentation-type***) activation-chars additional-activation-chars blip-chars additional-blip-chars* (*inherit-context* **t**) (*default* **t**) (*provide-default* **'dw::unless-default-is-nil**) (*default-type* **dw::original-type**) (*display-default* **dw::prompt**) *present-default history* (*prompts-in-line* **dw::*accept-active***) (*initially-display-possibilities* **nil**) *input-sensitizer* (*handler-type* **#'dw::presentation-type-find-parser**) *query-identifier* (*separate-inferior-queries* **nil**) *(confirm* **nil***)* *Function*

Reads printed representation of a Lisp object from a stream. If the representation is entered via a mouse gesture, it returns the object; if the representation is entered as a series of keyboard characters, it parses the series and returns the object.

presentation-type
: Presentation type of the object to be accepted.

:stream
: Specifies stream from which object is read; the default is ***query-io***.

:prompt Specifies characteristics of the input prompt. (See the section "Displaying Prompts in the Input Editor".) Allowable values for this option are:

nil No prompt is printed.

string String to be used as prompt.

function Function to display a prompt string. It must take two positional arguments. The first is the stream on which the prompt is to be displayed. The second is a keyword indicating the origin of the function call; for available keywords and related information, see the section "Displaying Prompts in the Input Editor".

You typically provide a prompt function when you want the prompt to change dynamically. In such cases, you can **ignore** the second argument.

Note: If there is no distinguishing default prompt for them, you should specify either a prompt or a query-identifier for each **accept** form within a **dw:accepting-values** form; otherwise, there will be no way that the accept-variable-values stream can identify which **accept** form is being run.

list A list of a format string and format arguments, for example, where somewhere previously *i* and *j* have been set to 5 and 3:

```
(accept 'string
          :prompt '("Enter a string with ~D
letters, ~D of them vowels" ,i ,j))
=> Enter a string with 5 letters,
   3 of them vowels:
```

:enter-type
Causes the prompt "Enter a <*presentation type*>" to be used. The presentation type is that specified by the *presentation-type* argument to **accept**.

If **:prompt** is not **nil**, the default, if any, is displayed automatically after the prompt string. For example, a prompt string of "to file" for a presentation type of **pathname** is displayed as "to file (default Q:>foo.bar):". See the section "**:display-default** Option to **accept**". If you provide a prompt string, whether **accept** provides trailing punctuation is determined by the "**:prompt-mode** Option to **accept**".

:prompt-mode
Specifies whether a colon and space is appended to a user-supplied prompt. A value of **:normal** causes a trailing colon and space to be appended; a value of **:raw** does not. This option also controls the parenthesizing in recursive calls to **accept**.

:original-type
The original type supplied, to be passed through in successive recursive calls to **present** (or **present-to-string** or **accept**).

:activation-chars
Takes a list of characters that are used as activation characters for the duration of the call to **accept**. The default activators are **#\return** and **#\end**.

Activation characters signal the end of user input to the **accept** function. If input to the function is via the keyboard, the user must necessarily press an activation character to activate the **accept**.

If input is via a translating mouse handler, defined by **define-presentation-to-command-translator** or **define-presentation-translator**, then whether an activation character is necessary depends on whether the translator returns an `:activate t` keyword-value pair. See the function **define-presentation-translator**.

:additional-activation-chars
Similar to **:activation-chars**; the list of characters supplied is added to the list of activators. Additional activation characters may be useful for activating **accept** when called recursively.

:blip-chars
Takes a list of characters that serve as delimiters of input fields for the duration of the call to **accept**.

:additional-blip-chars
Similar to **:blip-chars**; the list of characters supplied is added to the list of delimiters. Additional blip characters may be useful for terminating input fields when **accept** is called recursively.

:inherit-context
Boolean option specifying whether the current invocation of **accept** inherits the existing input context or establishes a new root node; the default value is **t**. This option is useful for controlling the input contexts at different levels in a recursive call to **accept**.

:default Specifies the object to be used as the default value for this **accept**. If no object is specified by this option — and a default is to be given (see the **:provide-default** option) — then the object offered is the one at the top of the presentation history for the presentation type specified in the *presentation-type* argument.

:provide-default
Specifies whether to provide a default value for this **accept**. The default is **'dw::unless-default-is-nil**. If no value is supplied for this option, a default value is given unless it is **nil**. If **nil** is a valid default that you want to be provided, then you must specify **:provide-**

default t.

:default-type
Specifies the presentation type of the object offered as the default for completing the call to **accept**. The default for this option is **dw::original-type**, which in turns defaults to the type given by the *presentation-type* argument.

This option is used for specifying explicitly the presentation type of the default when accepting compound presentation types created with the **or** presentation type. The value supplied should be one of the cases listed in the **or** or an even more specific (sub)type of one of these. This allows the default to be presented properly. See the presentation type **or**.

:display-default
Controls the display of the default object. A value of **t** causes the default to be displayed whether or not a prompt is displayed; **nil** suppresses the display of the default whether or not a prompt is displayed.

The default value for this option, **dw::prompt**, causes the default to be displayed when a prompt is displayed, and the default display to be suppressed when a prompt is not displayed.

:present-default
Boolean option specifying whether the default object is presented and accepted. This option is for the internal use of **dw:accepting-values** and related facilities.

:history Specifies which presentation-type history to use for yanking purposes. A value of **nil**, the default, causes the history of the type specified by the *presentation-type* argument to be used.

Aside from providing another presentation type, you may also supply as the value to this option a history object. This would be appropriate if you constructed the presentation-type history yourself, rather than letting the presentation substrate do it for you.

:prompts-in-line
Boolean option specifying whether prompt is displayed in-line with parentheses or with a trailing colon. The default is **t** if the **accept** was called recursively, **nil** otherwise.

:initially-display-possibilities
Boolean option specifying whether to display the objects that could be used as input in the current context; the default is **nil**. If **t**, the possibilities are presented before the prompt appears. This is the same list of possibilities that is displayed when the user presses HELP after the initial prompt appears. Note that this option only works reliably if there is a specific enumeration set of possibilities.

:input-sensitizer

This option is used internally by **dw:accepting-values** and related facilities.

:handler-type

This option is used internally by **dw:accepting-values** and related facilities.

:query-identifier

Specifies a unique identifier for this call to **accept**; the default is derived from the prompt.

This option is used when the **accept** is used within an **dw:accepting-values** form and the prompt is not unique.

If the queries do not have unique prompts, use the **:query-identifier** option to **accept** to distinguish them. Identity will be based on **equal**. Examples:

Wrong:

```
(dw:accepting-values ()
  (loop for i from 1 to 10
        collect (accept 'integer)))
```

Right:

```
(dw:accepting-values ()
  (loop for i from 1 to 10
        collect (accept 'integer
                  :prompt
                  (format nil "Number #~D" i))))
```

or

```
(dw:accepting-values ()
  (loop for i from 1 to 10
        collect
          (accept 'integer
                  :query-identifier i)))
```

also (valid case for same prompts)

```
(dw:accepting-values ()
  (loop for i from 1 to 5
     collect (let ((num (accept 'integer
                          :prompt
                          (format nil "Number #~D" i))))
                (list num
                   (when (oddp num)
                      (accept 'boolean
                        :query-identifier
                        (list :subfield i)
                         :prompt
                         "  Subfield for that"))))))
```

See the function **dw:accepting-values**.

:separate-inferior-queries
Boolean option specifying whether recursive calls to **accept** go on separate lines when executing an **dw:accepting-values** function; the default is **nil**.

:confirm
Specifies, when set to **t**, that the user must confirm input before it will be accepted. The default is **nil**. This option is ignored unless the call to **accept** is made from the Command Processor (CP).

For an overview of **accept** and related facilities: See the section "Using Presentation Types for Input".

accept-from-string *presentation-type string* &rest *args* &key *index* (*start* **0**) *end* &allow-other-keys *Function*

Reads the printed representation of a Lisp object from a string and returns the object with a specified presentation type. This function is the presentation-system equivalent of the Common Lisp function **read-from-string**.

presentation-type
: Presentation type of the object to be accepted.

string
: String from which to accept the object.

args
: Keyword options to **accept**.

:start
: Specifies the position of the first character to be parsed. The default is `0`, the position of the first character.

:end
: Specifies the position of the first character not to include in the parsing of the string.

Examples:

```
(accept-from-string 'string "Test 1") ==>
"Test 1"
STRING

(accept-from-string 'integer "Test 2" :start 5) ==>
2
INTEGER
```

For an overview of **accept-from-string** and related facilities: See the section "Using Presentation Types for Input".

dw:accept-values *descriptions* &key (*prompt* **nil**) (*near-mode* **'(:mouse)**) (*stream* ***query-io***) (*own-window* **nil**) (*temporary-p* **dw::own-window**) (*initially-select-query-identifier* **nil**) *Function*

Reads a series of printed representations of Lisp objects from a stream and returns one value for each object read. The objects may be entered via mouse gestures or as keyboard input.

descriptions
List of descriptions. Each description is a list of a presentation type and a set of the keyword options; available keywords are those allowed by **accept**. being returned for all occurrences of that type.

Example:

```
(dw:accept-values '((integer :prompt "Half-life"
                             :default 24000)
                    (pathname :prompt "Log file")
                    (integer :prompt "Session number"))
          :prompt "Atomic experiment")
```

:prompt Specifies a string, or a function returning a string, serving as the prompt or heading for the whole series of input prompts that follow.

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)**.

This option is applicable only when the value of the **:own-window** option is **t**.

:stream Specifies the stream to be used for input and output; the default is ***query-io***.

:own-window
Specifies whether the input/output interaction occurs in a separate, momentary window or runs "in place" in the current window like ordinary input/output; the default is **nil**.

:temporary-p
Boolean specifying whether the menu window is temporary, that is, whether the menu locks the underlying window without graying it out. If the value of the **:own-window** option is **t**, then the default for this option is **t**, a temporary window; if the value of **:own-window** is **nil**, this option is inapplicable.

:initially-select-query-identifier
Specifies that a particular field is preselected when the user interaction begins. The field to be selected is tagged by the **:query-identifier** option to **accept**, passed through to **accept** by **dw:accept-values**. Use this tag as the value for the **:initially-select-query-identifier** keyword, as shown in the following example:

```
(dw:accept-values '((integer :prompt "Number of times"
                             :query-identifier fred)
                    (boolean :prompt "Backwards"))
                  :initially-select-query-identifier 'fred)
```

When the initial display is output, the mouse cursor appears after the prompt of the tagged field, just as if the user had selected that field by clicking on it. The default value,if any, for the selected field is not displayed.

For an overview of **dw:accept-values** and related facilities: See the section "Using Presentation Types for Input".

dw:accept-values-choose-from-sequence *stream sequence value query-identifier* &key *(type* **'t***) highlighted-type printer (key* **#'identity***) (highlighting-test* **#'eq***) highlighting-function (select-action* **#'(lambda (dw::new ignore) dw::new)***) fill-p multiple-choices (single-box* **t***)* *Function*

The function to use when writing a **:choose-displayer** in defining a presentation type. Here is an example from an internal presentation-type definition for **dw:alist-subset**:

```
:choose-displayer
  ((stream object query-identifier &key original-type)
  (dw:accept-values-choose-from-sequence
   stream alist object query-identifier
   :type original-type
   :multiple-choices t
   :highlighting-test #'(lambda (e list)
                          (member e list :test highlighting-test))
   :printer #'(lambda (element stream)
                (princ (token-element-string element) stream))
   :key #'tv:menu-execute-no-side-effects
   :select-action #'(lambda (new list)
                      (if (member new list :test highlighting-test)
                          (remove new list :test highlighting-test)
                          (adjoin new list :test highlighting-test)))))
```

stream The stream on which to display the accept-values menu.

sequence The sequence from which to choose a value.

value The current value of the query. Normally, this is the object argument to the **:choose-displayer** function. *value* is used as an argument to the **:highlighting-test** function.

query-identifier
Specifies a unique identifier for this accept-values call.

:type The presentation type of the object to be accepted.

:highlighted-type
Presentation type of choice(s) to highlight.

:printer Specifies a function of two arguments, the object and a stream, to use to print the menu choices.

:key Specifies a function to obtain the value of a selected item from an element of *sequence*. The default is **#'identity**.

:highlighting-test
Specifies a function that determines which value derived from *sequence* by **:key** is a currently selected choice, and therefore should be highlighted. The first argument is the value derived from *sequence*, the second argument is the current value of the query, initially *value*. The default is **(function eq)**. The function in the example above allows more than one choice to be selected, by interpreting *value* as a list of selected choices.

:highlighting-function
Specifies a function to be used to highlight the choices. **:highlighting-function** gets called with *continuation* &rest *continuation-args* where the *continuation-args* are choice, stream, and type. The default if **:highlighting-function** is unspecified or **nil** is to present the choice in boldface.

:select-action
The function to execute when an item is selected. It takes two arguments, the value of the selected item and the current value of the query, initially *value*. The value it returns is the new value of the query. The default is a function that simply returns its first argument.

:fill-p Boolean specifying when **t** that the line on which the choices are displayed should be filled, that is, displayed with newline characters to prevent wrapping of output for long lines.

:multiple-choices
Boolean specifying when **t** that more than one choice can be accepted.

:single-box
Boolean specifying when **t**, the default, that the mouse-sensitivity of objects output by this form be highlighted by a single box.

dw:accept-values-command-button *(*&optional *(stream* **'*standard-output***)) *prompt* &body *conditional-forms* *Function*

Used within **dw:accepting-values** form, displays *prompt* on *stream* and creates an area, the "button," in which when a mouse button is clicked the *conditional-forms* are evaluated.

Example:

```
(defun pathname-test ()
   (let ((number 10)
         (pathname #P"foo.bar"))
     (dw:accepting-values ()
        (dw:accept-values-fixed-line "Here are some questions:")
        (dw:accept-values-command-button ()
              (write-string "Click here to synchronize version.")
           (setq pathname (send pathname :new-version number)))
        (terpri)
        (list (setq number (accept 'integer :prompt "Version"
                                            :default number))
              (setq pathname (accept 'pathname :prompt "File"
                                               :default pathname))))))
```

dw:accept-values-display-exit-boxes &key *(stream* ***query-io****) (level* **:top-level***)* *Function*

Used within **dw:accepting-values** form, displays the mouse-sensitive exit boxes "ABORT aborts, END uses these values" on *stream*.

dw:accept-values-fixed-line *string* &optional *(stream* ***query-io****)* *Function*

Used within a **dw:accepting-values** form, displays *string* on *stream*.

dw:accept-values-for-defaults *continuation* *Function*

Runs *continuation* with a stream argument, causing calls to **accept** to return their defaults. This is like the user encountering **dw:accepting-values** and pressing END right away.

dw:accept-values-into-list *descriptions* &key *:prompt (:near-mode* **'(:mouse)***) (:stream* ***query-io****) :own-window (:temporary-p* **dw::own-window***) :initially-select-query-identifier* *Function*

Performs the same operation as **dw:accept-values**, but returns a list rather than multiple values. See the function **dw:accept-values**.

dw:accept-variable-values *variables* &key (*prompt* **"Choose Variable Values"**) (*near-mode* **'(:mouse)**) (*delayed* **t**) (*stream* ***query-io****) (*own-window* **nil**) (*temporary-p* **dw::own-window**) (*initially-select-query-identifier* **nil**) *Function*

Provides a menu-like facility for setting the values of special variables to values provided by the user. The value for each variable is read via a call to **accept** using a specified presentation type.

(Usage note: **dw:accept-variable-values** is intended for use with special variables, not local ones. As such, it is useful for conversion from **tv:choose-variable-values** but is, in general, less appropriate for new applications of accept-values technology. For the latter, we recommend using **dw:accept-values** and **dw:accepting-values**.)

variables A list of variable descriptions. Each description is a list of the form

```
(place &optional (prompt :enter-type) (type 'sys:expression))
```

where *place* is usually a variable name; *prompt* is either a prompt string, **:enter-type** (the default), or a function for returning a prompt string; and *type* is the presentation type of the variable (**sys:expression** by default).

Example:

```
(dw:accept-variable-values
        '((*a* "Number" integer)
          (*b* "File" pathname)
          (*c* "Printer" sys:printer))) ==>

Choose Variable Values
Number: an integer
File: the pathname of a file
Printer: a printer
 aborts,  uses these values
NIL
```

:prompt Specifies a string, or a function returning a string, serving as the prompt or heading for the whole series of input prompts that follow.

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)**.

This option is applicable only when the value of the **:own-window** option is **t**.

:delayed Boolean option specifying whether variables are updated with user-supplied values after the entire accept-variable-values interaction is complete, or individually after input to each variable field is terminated; the default is **t**.

:stream Specifies the stream to be used for input and output; the default is ***query-io***.

:own-window
Specifies whether the input/output interaction occurs in a separate, momentary window or runs "in place" in the current window like ordinary input/output; the default is **nil**.

:temporary-p

Boolean specifying whether the menu window is temporary, that is, whether the menu locks the underlying window without graying it out. If the value of the **:own-window** option is **t**, then the default for this option is **t**, a temporary window; if the value of **:own-window** is **nil**, this option is inapplicable.

:initially-select-query-identifier

Specifies that a particular field is pre-selected when the user interaction begins. The field to be selected is tagged by the **:query-identifier** option to **accept**; use this tag as the value for the **:initially-select-query-identifier** keyword, as shown in the following example:

```
(dw:accept-variable-values
  '(( a "The file" 'pathname)
    ( b  "The number" 'integer)
    ( c "The printer" 'sys:printer))
       :initially-select-query-identifier 'the-tag)
```

When the initial display is output, the mouse cursor appears after the prompt of the tagged field, just as if the user had selected that field by clicking on it. The default value, if any, for the selected field is not displayed.

For an overview of **dw:accept-variable-values** and related facilities: See the section "Using Presentation Types for Input".

dw:accepting-values *(*&optional *(stream* **'*query-io***) &key *:own-window (:display-exit-boxes* **(not dw::own-window)**) *(:temporary-p* **dw::own-window**) *(:label* **"Multiple accept"**) *(:near-mode* **'(:mouse)**) *:initially-select-query-identifier :resynchronize-every-pass :queries-are-independent (:changed-value-overrides-default* **t**) *(:query-entry-mode* **:inline**)) &body *body* *Function*

Causes all calls to **accept** within *body* to appear in a single, **dw:accept-variable-values**-like menu that can be modified dynamically. If this macro is called from a remote terminal or some other device that does not support menus, it just performs successive calls to **accept**.

stream Stream for input and output; the default is ***query-io***.

If the body of an accepting-values form assigns the returned value to a variable, as, for example, with

```
(setq a (accept ...))
```

and the user never submits any new input for this call to **accept**, that variable gets set to the default value of the accept.

:own-window
Specifies whether the input/output interaction occurs in a separate, momentary window or runs "in place" in the current window like ordinary input/output; the default is **nil**.

:display-exit-boxes
Boolean option specifying whether the Abort-End exit message is displayed. The default is to display it unless the interaction is in its own window (see the **:own-window** option).

:temporary-p
Boolean specifying whether the menu window is temporary, that is, whether the menu locks the underlying window without graying it out. If the value of the **:own-window** option is **t**, then the default for this option is **t**, a temporary window; if the value of **:own-window** is **nil**, this option is inapplicable.

:label Specifies a string to serve as the title of the interaction menu. This option is applicable only if the value of the **:own-window** option is **t**.

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)**.

This option is applicable only when the value of the **:own-window** option is **t**.

:initially-select-query-identifier
Specifies that a particular field is pre-selected when the user interaction begins. The field to be selected is tagged by the **:query-identifier** option to **accept**; use this tag as the value for the **:initially-select-query-identifier** keyword, as shown in the following example:

```
(let (a b c)
  (dw:accepting-values (*query-io*
        :initially-select-query-identifier 'the-tag)
    (setq a (accept 'pathname :prompt "The file"))
    (setq b (accept 'integer :prompt "The number"
        :query-identifier 'the-tag))
    (setq c (accept 'sys:printer
              :prompt "The printer")))
  (format t "Printing ~D copies of
              file ~A on ~A" b a c))
```

When the initial display is output, the mouse cursor appears after the prompt of the tagged field, just as if the user had selected that

field by clicking on it. The default value, if any, for the selected field is not displayed. **Note**: you must specify either a unique prompt or a query-identifier for each **accept** form within an **dw:accepting-values** form; otherwise, there will be no way that the accept-variable-values stream can identify which **accept** form is being run.

:resynchronize-every-pass

Boolean option specifying whether earlier queries depend on later values; the default is **nil**.

You can use this option to alter dynamically the multiple-accept display. The following is a simple example. It initially displays an integer field that disappears if a value other than 1 is entered; in its place a two-field display appears.

```
(defun alter-multiple-accept ()
  (fresh-line)
  (let ((flag 1))
    (dw:accepting-values
        (t :resynchronize-every-pass t)
      (if (= flag 1)
          (setq flag (accept 'integer :default flag))
          (accept 'string)
          (accept 'pathname)))))
```

As the example shows, to use this option effectively, the controlling variable(s) must be initialized outside the lexical scope of the **dw:accepting-values** macro.

body The body is run in order to generate the initial prompt/value display. The body (or some part of it) is re-run each time a change is made; so the dependencies that later calls to **accept** may have on earlier ones will be correctly resolved. Because the body is run repeatedly, you must be careful of side-effects in the body code. Also, because the stream carries the state information, all input/output calls within the body must use the stream specified in the **dw:accepting-values** options list.

If you had a file `V:>RJ>tst.dwg` and a printer named Audubon, you could do something like the following. (Supply a pathname at your site and a local printer to try these examples.)

Good examples:

```
(let ((a #P"V:>RJ>TST.DWG")
        (b 2)
        (c (net:find-object-named :printer "audubon")))
 (dw:accepting-values (*query-io*
                              :prompt "Good Example")
  (setq a (accept 'pathname
                        :prompt "The file"
                        :default a))
  (setq b (accept 'integer
                        :prompt "The number":default b))
  (setq c (accept 'sys:printer
                   :prompt "The printer"
                        :default c)))
 (format t "Printing ~D copies of
                file ~A on ~A" b a c))

(multiple-value-bind (a b c)
   (dw:accepting-values ()
    (values
     (accept 'pathname :prompt "The file")
     (accept 'integer :prompt "The number")
     (accept 'sys:printer
                 :prompt "The printer")))
 (format t "Printing ~D copies of
                file ~A on ~A" b a c))
```

Poor example:

```
(let ((the-list nil))
  (dw:accepting-values ()
    (push
      (list
        (accept 'pathname :prompt "The file")
        (accept 'sys:printer :prompt "The printer"))
      the-list))
  (format t "The list = ~S" the-list))
```

The above example is a poor one because the output list will have an unpredictable number of elements; this detracts from its usefulness.

A useful presentation type to use with **accept** functions in the body of a **dw:accepting-values** macro is **alist-member**. Its usefulness derives from the keyword options available for inclusion in the item lists contributing to the alists. Three options exist: **:documentation**, **:style**, and **:selected-style**.

The value of the **:documentation** keyword is a string that appears in the mouse documentation line when the mouse cursor is over the item (that is, the item is highlighted).

:style specifies the character style for the item when it is displayed. **:selected-style** specifies the character style of the item when it is selected, that is, after it has been clicked on. The **:selected-style** defaults to the boldface version of the unselected style.

Use of the **alist-member** presentation type with **dw:accepting-values** is illustrated by the following example:

```
(defun filter-a-v ()
  (let ((low-pass-list
          '(("Mean" :value :mean
             :documentation "1  1  1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil))
            ("Gaussian" :value :gauss
             :documentation "1  2  1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil))))
        (edge-list
          '(("Laplacian, HP" :value :lpl-hp
             :documentation "-1  3  -1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil))
            ("Laplacian, ED" :value :lpl-ed
             :documentation "-1  2  -1 mask"
             :style (:swiss :roman :normal)
             :selected-style (:dutch :bold nil)))))
    (dw:accepting-values (*query-io* :own-window t)
      (fresh-line)
      (setq lo-pass-f (accept `((alist-member
                                  :alist ,low-pass-list)
                                  :description "a low-pass filter")))
      (setq edge-f (accept `((alist-member
                               :alist ,edge-list)
                               :description "a hi-pass/edge filter"))))))
```

For an overview of **dw:accepting-values** and related facilities: See the section "Using Presentation Types for Input".

For additional examples, see the file `sys:examples;accepting-values.lisp`

(flavor:method :activate-p tv:essential-window) *t-or-nil* *Init Option*

If this option is specified non-**nil**, the window is activated after it is created. The default is to leave it deactivated. Note that **:activate-p** and **:expose-p** are arguments in init-options which cannot be specified in the flavor's **:default-init-plist**.

(flavor:method :activate-p tv:menu) *t-or-nil* *Init Option*

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

: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.

(flavor:method :add-asynchronous-character si:interactive-stream) *character handler* *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.

The standard handler that the system uses to intercept c-m-SUSPEND, c-ABORT, and so on, is the function **tv:kbd-asynchronous-intercept-character** . Therefore, if you have, for example, removed one of these system asynchronous characters, you can restore it through:

```
(send stream :add-asynchronous-character character
        'tv:kbd-asynchronous-intercept-character)
```

cp:add-command-accelerator *command-table function-name characters* *Function*

Adds a keyboard accelerator for an existing command named *function-name* in the table *command-table*. The list *characters* includes every character that will invoke the command. See the function **cp:define-command-accelerator**.

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 that should be typed after FUNCTION to get the new command. Lowercase letters are converted to uppercase.

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. (Note: you can accomplish the same effect by using a single string containing NEWLINE characters.) 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 keywords sometimes followed by 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***.

(flavor:method :add-highlighted-item tv:menu-highlighting-mixin) *item* *Method*

Add an item to the list of items to be highlighted.

(flavor:method :add-highlighted-value tv:menu-highlighting-mixin) *value* *Method*

Adds an item to the list of items to be highlighted. Refers to the item by value. For instance, if your item-list is an association list, with elements (*string* . *symbol*), this message uses *symbol*. This only works for menu items that can be executed without side-effects, not, for example, the **:eval** and **:funcall** kinds.See the section "**tv:multiple-menu-mixin** Messages".

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 (character object) that should be typed after SELECT to get the new command. Lower-case characters are converted to upper case. Number keys are not permitted.

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)
```

As of Genera 7.3 Ivory, the variable **tv:*select-keys***, previously used by the SELECT key, is obsolete. it is retained for compatibility. (The SELECT key now uses an internal database.) Use **tv:add-select-key** where possible.

dw:add-standard-menu-accelerator *command-table command-symbol* &optional *accelerator-name (menu-level* **'(:top-level)***)* *Function*

Defines normal command menu actions. This is what the **define-program-command :menu-accelerator** option expands into. Use this to add items to the command menu if you do not use **:menu-accelerator t** or to add synonyms.

For an overview of this and related topics: See the section "How Command Menus Work".

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")
```

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")
```

tv:add-typeout-item-type *Function*

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.

(flavor:method :adjust-geometry-for-new-variables tv:choose-variable-values-window) *width* *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.)

: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.

tv:*allow-pop-up-notifications* *Variable*

If the value is **t**, asynchronous notifications not handled by the selected process will be displayed in a pop-up window. If the value is **nil** and the window does not handle asynchronous notifications, any notification will just be held and an alert will appear in the progress note area at the very bottom of the screen: `"~D pending notifications"`. A user who sees this can switch to a Lisp listener or press `SELECT N` to see the notification.

Note that whether an asynchronous notification is "handled" by a process depends on the process state and the activity in progress. Lisp Listeners, for example, are almost always prepared to handle notifications, by printing them at the current output point on the window, unless other unusual output is occurring. A Zwei-based application (Zmacs, Zmail, Converse) prints notifications in the mode-line window if it is in the `User Input` state and the message will fit in that window. Most other applications are not prepared to handle notifications.

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.

This is the same as the Common Lisp **boole-and** function.

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.

This is the same as the Common Lisp **boole-andc1** function.

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.

This is the same as the Common Lisp **boole-ior** function.

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.

This is the same as the Common Lisp **boole-1** function.

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.

This is the same as the Common Lisp **boole-xor** function.

(flavor:method :any-tyi si:interactive-stream) &optional *eof-action* *Method*

Reads and returns the next character or blip 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 signaled 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**.

(flavor:method :any-tyi tv:stream-mixin) &optional *eof-action* *Method*

Reads and returns the next character or blip 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, takes the next character from there. Otherwise, takes the next character from the window's I/O buffer.
:read	If the input buffer contains unscanned input, takes the next character from there. Otherwise, if an activation blip or character is present, returns that. Otherwise, enters the input editor.
:tyi	Takes 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**.

(flavor:method :any-tyi-no-hang si:interactive-stream) &optional *eof-action* *Method*

Returns the next character from the stream if it is immediately available. If no characters are immediately available, returns **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.

(flavor:method :any-tyi-no-hang tv:stream-mixin) &optional *eof-action* *Method*

Checks the window's I/O buffer and returns the next character if it is immediately available. If no characters are immediately available, it returns **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.

(flavor:method :append-item tv:text-scroll-window) *new-item* *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.

(flavor:method :appropriate-width tv:choose-variable-values-window) &optional *extra-space* *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.

(flavor:method :asynchronous-character-p si:interactive-stream) *character*
Method

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

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

Specifies the asynchronous characters for the stream. *spec-list* 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)))
```

Thus, **tv:kbd-asynchronous-intercept-character** is the standard handler for all of the system's asynchronous characters. The **:handle-asynchronous-character** method for **si:interactive-stream** calls this function with two arguments, the character and the stream.

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.

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 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)
```

(flavor:method :backspace-not-overprinting-flag tv:sheet) *x* *Init Option*

If *x* is **0**, typing **#\back-space** will move the cursor position backward; if it is **1**, typing **#\back-space** will display "overstrike" in a lozenge (that is, **#\back-space** will be just like other special characters). It defaults to **0**.

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>
```

(flavor:method :baseline 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 "Table of Program Output Facilities".

tv:basic-choose-variable-values *Flavor*

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.

tv:basic-frame *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: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".

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: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.

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**.

tv:basic-scroll-bar *Flavor*

Provides basic scroll-bar scrolling.

(flavor:method :bitblt tv:sheet) *alu wid hei from-raster from-x from-y to-x to-y*
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**. Note that *to-raster* is constrained as described in the the description of the **bitblt** function. Use **:draw-1-bit-raster** rather than **:bitblt** in programs that run without modification on color screens. See the function **tv:make-sheet-bit-array**.

(flavor:method :bitblt-from-sheet tv:sheet) *alu wid hei from-x from-y to-raster to-x to-y* *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**.

(flavor:method :bitblt-within-sheet tv:sheet) *alu wid hei from-x from-y to-x to-y* *Method*

Copies a rectangle of bits from the window to some other place in the window. All the other arguments have the same significance as in the **:bitblt** method of **tv:sheet**.

(flavor:method :blinker-p tv:sheet) *t-or-nil* *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".

: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-1-1
            (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.

(flavor:method :border-margin-width tv:borders-mixin) *n-pixels* *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**), 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**.

(flavor:method :border-margin-width tv:borders-mixin) *Method*

Returns the value of the border margin width.

tv:bordered-constraint-frame *Flavor*

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.

tv:bordered-constraint-frame-with-shared-io-buffer *Flavor*

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.

(flavor:method :borders tv:borders-mixin) *argument* *Init 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.

(flavor:method :borders tv:menu) *argument* *Init 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".

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)*.

tv:borders-mixin *Flavor*

Creates the borders around windows that you often see when using Genera. 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.

(flavor:method :bottom tv:menu) *bottom-edge* *Init Option*

Specified in pixels and is relative to the outside of the superior window.

(flavor:method :bottom tv:sheet) *bottom-edge* *Init Option*

Specifies the y-coordinate of the bottom edge of the window.

(flavor:method :bottom-margin-size tv:sheet) *Method*

Returns the bottom margin size of the window in pixels.

tv:box-blinker *Flavor*

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**, so all of **tv:rectangular-blinker**'s init options and messages work on this too.

dw:box-bottom *box* *Function*

Returns the location of the bottom of *box*.

dw:box-contained-in-region-p *box other-left other-top other-right other-bottom* *Function*

Returns **t** if *box* is contained in the region defined by *other-left*, *other-top*, *other-right*, and *other-bottom*; otherwise, returns **nil**. Any of the other coordinates can be **nil**, meaning positive or negative infinity in the appropriate direction.

dw:box-edges *box* *Function*

Returns four values in the following order: the location of the left-hand edge of *box*, of the top, of the right-hand edge, and of the bottom.

dw:box-left *box* *Function*

Returns the location of the left-hand side of *box*.

dw:box-right *box* *Function*

Returns the location of the right-hand side of *box*.

dw:box-top *box* *Function*

Returns the location of the top of *box*.

dw:boxes-overlap-p *box-1 box-2* *Function*

Returns **t** if *box-1* and *box-2* overlap; otherwise, **nil**.

cp:build-command *command-name* &rest *command-arguments* *Function*

Constructs the internal representation of a Command Processor command.

command-name
: Symbol or string naming the command to invoke; if a string, it must be in the command table to which **cp:*command-table*** is currently bound.

command-arguments
: Positional and keyword arguments to the named command, either strings or appropriate objects (or single objects when a sequence is required.

Examples:

```
(cp:build-command "show file" "test-data.text") =>
(SI:COM-SHOW-FILE (#P"V:>elm>test-data.text.newest"))

(cp:build-command 'si:com-load-system "doc"
              :condition :always :redefinitions-ok t) =>
(SI:COM-LOAD-SYSTEM #<SCT:SYSTEM DOC 274003470>
                          :CONDITION :ALWAYS :REDEFINITIONS-OK T)
```

Note how, in the first example, **cp:build-command** "knows" the fact that `com-show-file` requires a sequence as its argument. It also parses strings into objects. It does all this by running the command parser in a mode that takes its input from the command arguments instead of from the keyboard. This is what makes **cp:build-command** useful. Suppose, for example, you want to output a command as a presentation:

```
(present (cp:build-command 'si:com-show-file #p"Y:>elm>foo.bar")
                                                        'cp:command) =>
Show File (file) Y:>elm>foo.bar
#<DW::DISPLAYED-PRESENTATION (SI:COM-SHOW... CP:COMMAND 447504140>
```

You should not use **cp:build-command** within a **define-presentation-to-command-translator** macro. It is too slow and you can encounter context problems in this use if the command is a program command whose parsing depends on the program's dynamic state. Instead of:

```
(define-presentation-to-command-translator com-add-comment
   (node
    :gesture nil)
   (node)
   (cp:build-command 'com-add-comment node))
```

Do this instead:

```
(define-presentation-to-command-translator com-add-comment
   (node
    :gesture nil)
   (node)
  `(com-add-comment ,node))
```

dw:call-presentation-menu *menu-type* &key *:presentation :original-presentation :window :label :x :y* &allow-other-keys *Function*

Used within a **define-presentation-action** that uses the **:define-menu** option, calls up the specified presentation menu. For example, the following form is used to set up the marking and yanking menu in the Lisp Listener:

```
(define-presentation-action marking-and-yanking-menu
   (raw-text si:input-editor
             :documentation "Marking and yanking menu"
             :gesture :marking-and-yanking-menu
             :menu (t :style (nil :italic nil))
             :defines-menu :marking-and-yanking)
   (ignore &key presentation
           (original-presentation presentation) window x y)
  (let ((use-presentation
          (or original-presentation presentation)))
    (return-from marking-and-yanking-menu
      (call-presentation-menu :marking-and-yanking
                              :label
                              (when
                                (eq use-presentation
                                    *null-presentation*)
                                "Marking and yanking operations")
                              :presentation use-presentation
                              :window window :x x :y y))))
```

Note that the *menu-type* argument to **dw:call-presentation-menu** is the one specified by **:defines-menu**. The **:label** keyword argument is used to specify the resulting menu's label. The values of the other keyword arguments are obtained from values available within the **define-presentation-action** form. Also note the use of **return-from** to cause the presentation action to return whatever values the handler invoked from the menu returns. This goes outside the scope of the **nil** from the expansion of **define-presentation-action**, since actions normally only have side effects.

dw:call-presentation-mouse-handler *presentation* &rest *arguments* &key *:mouse-char :window* &allow-other-keys *Function*

Invokes the translator for the mouse-click handler of *presentation*. This is for use within an input-blip handler. For example:

```
(defun dynamic-window-presentation-input-blip-handler (blip)
  (destructuring-bind (nil mouse-char window x y) blip
    (call-presentation-mouse-handler
      (send window :displayed-presentation-at-position x y t)
      :mouse-char mouse-char
      :x x
      :y y
      :window window)))
```

The arguments are the values of the blip. See the section "Mouse Blips".

(flavor:method :center-around tv:essential-set-edges) *x y* *Method*

Without changing the size of the window, positions 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.

(flavor:method :change-of-size-or-margins tv:sheet) &rest *options* *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.

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.

(flavor:method :char tv:character-blinker) *char* *Init Option*

Sets the character to display. You must provide this.

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
```

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-equal *char1 char2* *Function*

Returns **t** if the mouse characters *char1* and *char2* are equal, **nil** otherwise. **char-mouse-equal** checks that its arguments are really mouse characters and signals an error otherwise. You can also use **eql**, which is slightly faster, to compare mouse characters, when you do not require the argument checking.

tv:character-blinker *Flavor*

Draws itself as a character from a font. You can control which font and which character within the font it uses.

(flavor:method :character-height tv:menu) *spec* *Init Option*

Specifies 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.

(flavor:method :character-height tv:sheet) *spec* *Init Option*

Specifies 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.

(flavor:method :character-width tv:menu) *spec* *Init Option*

Specifies 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.

(flavor:method :character-width tv:sheet) *spec* *Init Option*

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.

(flavor:method :character-width tv:sheet) *ch* &optional *font* (*x* **tv:cursor-x**) *character-style* *Method*

Returns the width of the character *ch*, in pixels. The font used is *font* or the font resulting from merging *character-style* with the current character style. See the section "Merging Character Styles". 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.

dw:check-presentation-type-argument *type-arg* &key (*evaluated* **t**) (*function* **compiler:default-warning-function**) (*definition-type* **compiler:default-warning-definition-type**) *Function*

Checks an argument that is expected to be a presentation type for validity.

type-arg A form evaluating to a presentation type.

:evaluated
Boolean option specifying whether *type-arg* is expected to be quoted; the default is **t**.

:function Specifies a symbol naming the function for which the compiler warning is issued. This name is displayed in the warning instead of the name of the function in which the error occurred; the latter behavior is the default.

:definition-type

Specifies the definition type (`'defun`, `'defvar`, etc.) of the Lisp object that caused the compiler warning. The name for objects of this type ("Function", "Variable", etc.) is displayed in the warning instead of the name for the type of object in which the error occurred; the latter behavior is the default.

This function should be used in macros that take presentation types as arguments and in style-checkers for functions that take presentation types.

Here is an example of the use of **dw:check-presentation-type-argument** in a macro:

```
(defmacro with-value ((variable-name presentation-type) &body body)
  (dw:check-presentation-type-argument presentation-type :evaluated nil)
  `(let ((,variable-name (accept ',presentation-type)))
     ,@body))
```

If you try to compile the following function, which contains an invalid specification of the **integer** presentation type inside an invocation of `with-value`, you get a compiler error diagnosing the problem:

```
(defun check-type-test ()
  (with-value (x ((integer 3 5 extra-argument)))
    (format t "~&Value is ~S" x)))
```

The **:evaluated** keyword is used to control whether **dw:check-presentation-type-argument** expects the presentation type to be quoted or not. In the macro example above, the presentation type is inserted unquoted into the invocation of the `with-value` macro. If you wanted `with-value` to evaluate its presentation-type argument (for instance, so that a variable that was bound to a presentation type could be used), then you would supply `:evaluated t` (the default). The rewritten example follows:

```
(defmacro with-value ((variable-name presentation-type) &body body)
  (dw:check-presentation-type-argument presentation-type :evaluated t)
  `(let ((,variable-name (accept ,presentation-type)))
     ,@body))

(defun check-type-test ()
  (with-value (x '((integer 3 5 extra-argument)))
    (format t "~&Value is ~S" x)))
```

See the section "Defining Your Own Presentation Types".

(In both of the above examples, multiple error messages result because **accept** itself uses **dw:check-presentation-type-argument** to validate its arguments.)

For an overview of **dw:check-presentation-type-argument** and related facilities:

(flavor:method :choose tv:menu) *Method*

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.

(flavor:method :choose tv:multiple-choice) &optional *near-mode* *Method*

Allows menu selection by the mouse. It first moves the window to the place specified by *near-mode*, which defaults to the list **(:mouse)**, (that is, 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".

cp:choose-command-arguments *command-name* &rest *args* &key *(initial-arguments* **nil***) (start* **(length cp::initial-arguments)***) (end* **nil***) (command-table* **cp:*command-table***) (stream* ***standard-input****) (typeout-stream* **nil***) (help-stream* **cp::typeout-stream***) (prompt-mode* **:normal***) (near-mode* **'(:mouse)***) (mode* **:accept-values***) (typeout-stream* **nil***) (help-stream* **cp::typeout-stream***) (prompt* **nil***) (own-window* **nil***) (full-rubout* **nil***) (erase-input-editor* **nil***) (initially-select-query-identifier* **nil***)* *Function*

Returns arguments for the command *command-name* in one of three possible modes: **:accept-values**, the default, displays an accept-values-type of menu with which the user selects arguments; **:keyboard** allows the user to edit the command arguments from the keyboard; and **:none** returns default values for the positional arguments.

:initial-arguments
The arguments initially supplied. These are used as defaults or initial values in non **:accept-values** mode.

:start The number of arguments initially supplied that should be retained unmodified. This option is only used when the **:mode** is **:keyboard**.

:end The number of arguments initially supplied that should be retained unmodified. This option is only used when the **:mode** is **:accept-values**.

:command-table
Specifies the command table containing the command; the default is the current command table (bound to **cp:*command-table***).

:stream Specifies the input stream; the default is ***standard-input***.

:typeout-stream
The typeout window to use in **:accept-values** mode.

:help-stream

The stream to which responses to requests for help should be directed. The default is the value specified for **:typeout-stream**.

:prompt Specifies a string, or a function returning a string, serving as the prompt or as the window label if **:own-window** is true.

:prompt-mode
Specifies how the prompt is to be displayed. The default is **:normal**, which means display the prompt even if the accept-values display is not in ":own-window." The value **:own-window** means only show the prompt as a label.

:own-window
A Boolean option specifying when **t** that the accept-values-like menu be displayed in its own window. The default is **nil**.

:full-rubout
Specifies when non-**nil** that if the user rubs out all the characters that were typed, control is returned immediately. Two values are returned: **nil** and the value specified for **:full-rubout**. If this option is **nil**, the input editor waits for more characters to be typed. This option applies only when the **:mode** is **:keyboard**.

:erase-input-editor
A Boolean specifying when **t** that the menu used should be erased when done if called from inside the input editor. This is useful if you are going to **:replace-input** the command line next anyway. The default is **nil**.

:near-mode
Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. See the method **(flavor:method :expose-near tv:essential-set-edges)**.This option is applicable only when the value of the **:own-window** option is **t**.

:mode Specifies mode for choosing. One of

:accept-values	The default. Provides a menu-like facility for setting the values of the command arguments to values supplied by the user.
:menu	The same as **:accept-values**.
:keyboard	Applies **cp:read-command-arguments** for the specified command and returns them.
:none	Defaults the positional arguments of the specified command.

:initially-select-query-identifier

Specifies a field to be pre-selected when the user interaction begins. This is the same as the **dw:accept-values :initially-select-query-identifier** option, see the function **dw:accept-values**.

choose-user-options *alist* &rest *options* *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**.

tv:choose-variable-values *variables* &rest *options* *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:

The following option keywords can be specified.

(flavor:method :label tv:choose-variable-values)
(flavor:method :function tv:choose-variable-values)
(flavor:method :near-mode tv:choose-variable-values)
(flavor:method :width tv:choose-variable-values)
(flavor:method :extra-width tv:choose-variable-values)
(flavor:method :margin-choices tv:choose-variable-values)
(flavor:method :superior tv:choose-variable-values)

See the section "**tv:choose-variable-values** Examples".

tv:choose-variable-values-pane *Flavor*

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". It does not change its size automatically; the size is assumed to be controlled by the superior.

tv:choose-variable-values-process-message *window command* *Function*

Implements the proper response. 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 ap-

propriate special action and returns **nil**. If *command* is a character, it is ignored unless it is the **#/si:refresh** key, in which case the choose-variable-values window is refreshed.

tv:choose-variable-values-window *Flavor*

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.

(flavor:method :clear-char tv:sheet) &optional *char* *Method*

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

(flavor:method :clear-history dw:dynamic-window) *Method*

Eliminates all items in the output history of the window, and resets the viewport to the top of the history.

For an overview of **(flavor:method :clear-history dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :clear-input si:interactive-stream) *Method*

Clears 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.

(flavor:method :clear-input tv:stream-mixin) *Method*

Clears this window's input and I/O buffers. It flushes all the characters that have been typed at this window but have not yet been read.

dw:clear-presentation-input-context *Function*

Clears the current input context. This is useful for eliminating the input context established by a function's callers in order to establish a new input context that doesn't inherit from the callers.

For an overview of **dw:clear-presentation-input-context** and related facilities: See the section "Presentation Input Context Facilities".

(flavor:method :clear-region dw:dynamic-window) *left top right bottom* *Method*

Clears the output display in a rectangular area of the window. Specify the region in terms of absolute window coordinates. Any coordinate can be given a value of **nil** to indicate infinite extent in that direction.

left	The x-coordinate for the left edge of the cleared area.
top	The y-coordinate for the top edge of the cleared area.
right	The x-coordinate for the right edge of the cleared area.
bottom	The y-coordinate for the bottom edge of the cleared area.

For an overview of **(flavor:method :clear-region dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :clear-rest-of-line tv:sheet) *Method*

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

(flavor:method :clear-rest-of-window tv:sheet) *Method*

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

(flavor:method :clear-window dw:dynamic-window) *Method*

Scrolls the window forward in the vertical dimension far enough to eliminate previous output from the current display. Note that only the display is affected, not the window's output history.

For an overview of **(flavor:method :clear-window dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :clear-window tv:sheet) *Method*

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

tv:cold-load-stream-old-selected-window *Variable*

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

(flavor:method :column-spec-list tv:dynamic-multicolumn-mixin) *form*
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.

(flavor:method :columns tv:menu) *n-columns* *Init Option*

Sets the number of columns in a menu.

: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.

cp:command-in-command-table-p *command-symbol command-table* &optional *(need-name* **t***)* *Function*

Determines the presence of a command in a Command Processor command table. The function returns three values: **t** if the command is either in the specified command table or in a table from which the specified table inherits; the command's name (a string) or, if *need-name* is **nil**, **nil**; and the command table in which the command was found.

command-symbol
: The command symbol.

command-table
: The command table.

need-name
: A Boolean specifying whether the name (a string) of the command should be returned. By default, it is. The function runs faster when this is **nil**.

For an overview of **cp:command-in-command-table-p** and related facilities: See the section "Managing Your Program Frame".

tv:command-menu *Flavor*

This is **tv:command-menu-mixin** mixed with **tv:menu** to make it instantiable.

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**.

dw:command-menu-choose-arguments *command-name* &rest *args* &key *:initial-arguments (:command-table* **cp:*command-table***) (:gesture* **:left***) :mode :own-window (:full-rubout* **t***)* &allow-other-keys *Function*

Performs the normal actions taken by command menus. You can call this in addition to special actions of your own for a command menu.

command-name
A string, which identifies the command-menu item.

args The command arguments.

:initial-arguments
Arguments already specified to the command (that do not need to be chosen).

:command-table
The command table that contains the command *command-name*.

:gesture A single keyword, or list of keywords, identifying which gestures this handler applies to. A single handler can apply to more than one gesture, and multiple handlers can be defined on different gestures. The possible actions will then naturally be their union.

:mode The mode in which to accept arguments. Possible values are **:keyboard**, **:menu**, and **:accept-values**.

:own-window
A Boolean option specifying when **t** that the accept-values-like menu be displayed in its own window. The default is **nil**.

:full-rubout
Specifies when non-**nil** that if the user rubs out all the characters that were typed, control is returned immediately. Two values are returned: **nil** and the value specified for **:full-rubout**. If this option is **nil**, the input editor waits for more characters to be typed. This option applies only when the **:mode** is **:keyboard**.

For an overview of related topics: See the section "How Command Menus Work".

tv:command-menu-mixin *Flavor*

The basic mixin version of the command menu flavor. It is not instantiable on its own.

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".

dw:*command-menu-test-phase* *Variable*

Bound to **t** during the **:tester** phase when a command form body is running, and to **:documentation** during the documentation phase. This for use within a **dw:define-command-menu-handler** form. The command body can throw to the tag **dw:command-menu-test-phase** with a command (list of command name and arguments) or with a string (in the documentation case). Note that if the command body pops up a menu or reads from the keyboard to get arguments, it must look at this flag to prevent doing so when except when the user really clicked.

For an example of the use of the throw tag:See the section "How Command Menus Work".

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 "System Command Tables".

dw:compare-char-for-accept *char-from-accept comparandum* *Function*

Compares an input character with a specified character. Use this function instead of **char-equal** when manipulating characters read with **dw:read-char-for-accept**.

char-from-accept
: The input character (returned by **dw:read-char-for-accept**).

comparandum
: The comparison character. This may be any standard character.

For an overview of **dw:compare-char-for-accept** and related facilities, see the section "Defining Your Own Presentation Types".

dw:complete-from-sequence *sequence stream* &key *type* (*name-key* **#'string**) (*value-key* **#'identity**) (*delimiters* **dw::*standard-completion-delimiters***) (*allow-any-input* **nil**) (*enable-forced-return* **nil**) (*initially-display-possibilities* **nil**) (*partial-completers* **nil**) (*complete-activates* **nil**) (*compress-choices* **20**) (*compression-delimiter*) *Function*

Provides input completion from a sequence of possible completions for input to **accept**. Returned values are the object associated with the completion string; **t** or **nil** depending on whether or not the completion was the only one possible; and the completion string.

sequence The sequence of possible completions.

stream The input stream.

:type Specifies the presentation type to use when displaying help information for possible completions. This makes the displayed possibilities mouse-sensitive.

If the completion utility is being called from the parser of a presentation type, that type should be supplied as the value of this option.

:name-key
Specifies the function called on each element in the sequence for extracting the completion string. The default function is **string**. Another useful function is **string-capitalize-words**.

:value-key
Specifies the function called on each element in the sequence for extracting the value to be associated with the element's completion string. The default function is **identity**, which extracts the element itself.

:delimiters
Specifies a list of characters used by the standard completion mechanism to tokenize completion strings. The default value is the binding of **dw::*standard-completion-delimiters***; this variable is preset to "- " (hyphen and space).

:allow-any-input
Boolean option specifying whether the completer accepts keyboard input from the user that does not match any of the possible completion strings; the default is **nil**.

Most parsers should specify `:allow-any-input nil`. In a call to **accept** for which you want to allow input that does not match any of the completions, use the **type-or-string** presentation type.

Note that the completion facilities always signal the error **dw:input-not-of-required-type** when a user types `RETURN` at blank input. This is intended to allow **accept** to fill in the default in the blank case. It means, however, that a caller of a completion facility that passes `:allow-any-input t` must also **condition-bind** for **dw:input-not-of-required-type**, if you want a null line to be treated the same as any other input.

:enable-forced-return

Boolean option specifying whether the user can force a response that is not a member of the completion set; the default is **nil**.

If this option is **t**, the user can terminate input with c-RETURN, causing the completion utility to return to the caller whatever input the user supplied. This is useful in situations where you expect the user to specify a member of a set of possibilities, but want to provide a way for supplying a new name to be added to the set. (The Zmacs Select Buffer (c-X B) command uses this feature to allow the user to create new buffers.)

:initially-display-possibilities

Boolean option specifying whether to display the entire set of completion possibilities before prompting for input; the default is **nil**. If **t**, the behavior is as if the user typed Help before any other input.

Most parsers should supply to this option the same value that was supplied to them by **accept**. **accept**, in turn, has an **:initially-display-possibilities** option controlled by its caller. See the function **accept**.

:partial-completers

Specifies a list of characters that trigger partial completion when entered by the user.

Partial completion restricts completion to only one token of the completion set possibilities, even if enough characters have been supplied to uniquely identify one of the members of the completion set. For example, the Command Processor uses **#\space** as a partial completer.

The syntax of a token is defined by the **:delimiters** option.

:complete-activates

Boolean option specifying whether the COMPLETE key causes activation, that is, whether the completion utility returns if a unique completion was found. The default is **nil**.

This option is used to control completion behavior in a multi-field input context, such as in the command processor. Normally, the END key performs completion and then returns if the resulting completion is unique.

:compress-choices

Specifies whether to compress the display of completion possibilities that have a common left token as defined by the **:compression-delimiter** option. Three values are possible:

An integer

When the possibilities exceed this number, the display is

compressed. The default value is 20.

:always Whenever more than one possibility exists, the display is compressed.

:never The display is never compressed, regardless of the number of possibilities.

Compressed displays have the form "*token* ... (*n*)", where *token* is the shared left token and *n* is the number of possible completions.

To see an example of choice compression, press HELP to the command processor prompt in a Dynamic Lisp Listener. You get the following display (abbreviated for this example):

```
You are being asked to enter a command or form.
Use the Help :Format Detailed command to see a full
list of command names.

These are the possible command names:
  Add Paging File
  Append
  Clean File
  Clear ... (3)
  Close File
  Compare Directories
  Compile ... (2)
  Copy ... (5)
  Create ... (4)
  Debug Process
```

"Add Paging File", "Append", and "Clean File" are full command names. "Clear" is a left token shared by three commands, Clear All Breakpoints, Clear Breakpoint, and Clear Output History. These three completion choices have been compressed to "Clear ... (3)". The user can expand this and other compressed choices by clicking on them with the mouse.

:compression-delimiter
Specifies a character used for delimiting the shared left tokens in a display of completion possibilities. The default value is **#\space**.

For an overview of **dw:complete-from-sequence** and related facilities, see the section "Introduction: More Presentation-Type Concepts". For a table of available functions relating to Presentation Types, see the section "Table of Facilities for Defining Presentation Types".

: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".

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

dw:complete-input *stream function* &key (*allow-any-input* **nil**) *enable-forced-return partial-completers* (*type* **nil**) *parser* (*compress-choices* **20**) (*compression-delimiter*) (*help-offers-possibilities* **t**) (*initially-display-possibilities* **nil**) (*complete-activates* **nil**) (*documenter* **nil**) (*document* **(not (null dw::documenter))**) *Function*

Provides input completion for input to **accept**. Returns three values: the object associated with the completion string; **t** or **nil** depending on whether or not the completion was the only one possible; and the completion string. **dw:complete-input** is a low level function that is called by **dw:completing-from-suggestions**. In most cases, the latter is easier to use and is recommended. Use **dw:complete-input** only when you require finer control over the completion operations than that allowed by **dw:completing-from-suggestions**.

stream The input stream.

function The completion function. The function receives two arguments, the input supplied by the user and a keyword specifying an operation.

Operations are divided into two categories, completion operations and possibility operations. The former attempt to complete and return the completion; the latter return either a list of possible completions or the number of possible completions. Available keywords for each type are described below:

Completion Operations

:complete Complete and return as much as possible based on the input so far.

:complete-limited
Complete and return the current input "chunk" only, even if the input uniquely identifies a full completion possibility. The meaning of "chunk" depends on the type of input. For example, in the case of command processor commands, a chunk is a word in the command name.

:complete-maximal
Complete and return as much as possible based on the input so far, even if that means adding empty tokens between delimiters.

Regardless of the completion operation, the completion function must return the following five values:

1. The string resulting from completing the input string.

2. A boolean indicating that the completion is successful, that is, that the string given to the user-supplied function is itself a valid completion.

3. The object associated with the completion if it is successful. (If the completion is not successful, the value of this is undefined, although it is often one of the objects in the ambiguous case where the completion is unsuccessful and the number of possibilities is greater than one.)

4. The index in the completion string of the first point of ambiguity if the string is not unique, that is, the leftmost place in the string where a difference arises between two or more completion possibilities. The completer generally tries to position the input cursor at that point so that the user can resolve the ambiguity.

5. The number of possible input completions; this may be `0`.

Possibility Operations

:possibilities
Return a list of completion possibilities that begin with the input string. (This list is used, for example, when the user presses the HELP key.)

:apropos-possibilities
Return a list of the completion possibilities that contain the input string anywhere in the completion string. (This list is used, for example, when the user types CONTROL-/.) The function may split the input into tokens and search for possibilities that contain all the tokens somewhere in the completion string. In this case, it should return as a second value the list of tokens extracted from the original input string.

:apropos-initial-possibilities
Return a list of the completion possibilities that contain the input string in the completion string.

:possibilities-quick-length
Returns the number of completion possibilities that begin with the input string.

:apropos-possibilities-quick-length
Return the number of completion possibilities that contain the input string anywhere in the completion string.

If the possibility operation is **:possibilites** or **:apropos-possibilities**, the function must return a list of three-element lists, each of which is `(string object presentation-type)`.

The completion function can return **nil** to indicate that it does not support the "quick-length" operations. In this case, the completer utility asks for a full **:possibilities** or **:apropos-possibilities** list and counts the number of elements to return.

:allow-any-input
Boolean option specifying whether the completer accepts keyboard input from the user that does not match any of the possible completion strings; the default is **nil**.

Most parsers should specify `:allow-any-input nil`. In a call to **accept** for which you want to allow input that does not match any of the completions, use the **type-or-string** presentation type.

Note that the completion facilities always signal the error **dw:input-not-of-required-type** when a user types `RETURN` at blank input. This is intended to allow **accept** to fill in the default in the blank case. It means, however, that a caller of a completion facility that passes `:allow-any-input t` must also **condition-bind** for **dw:input-not-of-required-type**, if you want a null line to be treated the same as any other input.

:enable-forced-return
Boolean option specifying whether the user can force a response that is not a member of the completion set; the default is **nil**.

If this option is **t**, the user can terminate input with `c-RETURN`, causing the completion utility to return to the caller whatever input the user supplied. This is useful in situations where you expect the user to specify a member of a set of possibilities, but want to provide a way for supplying a new name to be added to the set. (The Zmacs Select Buffer (`c-X B`) command uses this feature to allow the user to create new buffers.)

:partial-completers
Specifies a list of characters that trigger partial completion when entered by the user.

Partial completion restricts completion to only one token of the completion set possibilities, even if enough characters have been supplied to uniquely identify one of the members of the completion set. For example, the Command Processor uses **#\space** as a partial completer.

The syntax of a token is defined by the **:delimiters** option.

:type Specifies the presentation type to use when displaying help information for possible completions. This makes the displayed possibilities mouse-sensitive.

If the completion utility is being called from the parser of a presentation type, that type should be supplied as the value of this option.

:parser Specifies the function called to translate input strings into objects of the desired type. The function is called with one argument, the string entered by the user.

This option is typically used when the set of possible completions is not known in advance, and can therefore not be enumerated. If they can be enumerated, use **dw:complete-from-sequence** or **dw:completing-from-suggestions** instead.

The parser function is called on each possible completion string when a list of possibilities is generated, and on the user-supplied input when the completion utility is about to return a value.

:compress-choices

Specifies whether to compress the display of completion possibilities that have a common left token as defined by the **:compression-delimiter** option. Three values are possible:

An integer
When the possibilities exceed this number, the display is compressed. The default value is `20`.

:always Whenever more than one possibility exists, the display is compressed.

:never The display is never compressed, regardless of the number of possibilities.

Compressed displays have the form "*token* ... (*n*)", where *token* is the shared left token and *n* is the number of possible completions.

To see an example of choice compression, press `HELP` to the command processor prompt in a Dynamic Lisp Listener. You get the following display (abbreviated for this example):

```
You are being asked to enter a command or form.
Use the Help :Format Detailed command to see a full
list of command names.
```

```
These are the possible command names:
  Add Paging File
  Append
  Clean File
  Clear ... (3)
  Close File
  Compare Directories
  Compile ... (2)
  Copy ... (5)
  Create ... (4)
  Debug Process
```

"Add Paging File", "Append", and "Clean File" are full command names. "Clear" is a left token shared by three commands, Clear All Breakpoints, Clear Breakpoint, and Clear Output History. These three completion choices have been compressed to "Clear ... (3)". The user can expand this and other compressed choices by clicking on them with the mouse.

:compression-delimiter
Specifies a character used for delimiting the shared left tokens in a display of completion possibilities. The default value is **#\space**.

:help-offers-possibilities
Boolean option specifying whether the full list of completion possibilities is displayed when the user presses the HELP key; the default is **t**.

:initially-display-possibilities
Boolean option specifying whether to display the entire set of completion possibilities before prompting for input; the default is **nil**. If **t**, the behavior is as if the user typed Help before any other input.

Most parsers should supply to this option the same value that was supplied to them by **accept**. **accept**, in turn, has an **:initially-display-possibilities** option controlled by its caller. See the function **accept**.

:complete-activates
Boolean option specifying whether the COMPLETE key causes activation, that is, whether the completion utility returns if a unique completion was found. The default is **nil**.

This option is used to control completion behavior in a multi-field input context, such as in the command processor. Normally, the END key performs completion and then returns if the resulting completion is unique.

:documenter
Specifies a function called to generate documentation for the elements of a possibilities display. The function receives two arguments, a completion possibility and the output stream for displaying the documentation.

:document
Specifies how each possibility displayed as a result of a HELP request is documented. Three values are possible:

t Display the documentation. If a documentation function is specified by the **:documenter** ,option, it is called on each possibility; otherwise, the Common Lisp function **documentation** is called.

nil Do not display any documentation.

:if-unique Display documentation only if there is a unique completion of the input supplied by the user.

The default for this option is **t** if a **:documenter** function is supplied, **nil** otherwise.

For an overview of **dw:complete-input** and related facilities, see the section "Defining Your Own Presentation Types".

dw:completing-from-suggestions (*stream* &key (*allow-any-input* **t**) (*delimiters* **dw::*standard-completion-delimiters***) (*enable-forced-return* **nil**) (*partial-completers* **nil**) (*type* **nil**) (*parser* **nil**) (*complete-activates* **nil**) (*compress-choices* **20**) (*compression-delimiter* **nil**) (*initially-display-possibilities* **nil**)) &body *body* *Function*

Binds local environment to build a completion table for input to **accept**. Three values are returned: the object associated with the completion string; **t** or **nil** depending on whether or not the completion was the only one possible; and the completion string, that is, the fully completed string generated from what the user typed. Within this environment, you must use **dw:suggest** within *body* to add each possible choice to the completion table.

stream The input stream.

Here is an example, excerpted from a system definition:

```
(define-presentation-type keyword-argument ((&rest keywords))
   :parser ((stream &key initially-display-possibilities type)
                              .
                              .
                              .
            (multiple-value-bind (word nil nil)
               (dw:completing-from-suggestions
                  (stream :force-complete t
                          :partial-completers
                            *keyword-key-partial-completers*
                          :initially-display-possibilities
                            initially-display-possibilities
                          :type type
                          :compress-choices :never)
                  (loop for keyword-name being
                             the array-elements of keyword-names
                        for key being the array-elements of keys
                           doing
                        (dw:suggest keyword-name key)))
             word)))))
```

:allow-any-input

Boolean option specifying whether the completer accepts keyboard input from the user that does not match any of the possible completion strings; the default is **nil**.

Most parsers should specify `:allow-any-input nil`. In a call to **accept** for which you want to allow input that does not match any of the completions, use the **type-or-string** presentation type.

Note that the completion facilities always signal the error **dw:input-not-of-required-type** when a user types `RETURN` at blank input. This is intended to allow **accept** to fill in the default in the blank case. It means, however, that a caller of a completion facility that passes `:allow-any-input t` must also **condition-bind** for **dw:input-not-of-required-type**, if you want a null line to be treated the same as any other input.

:delimiters

Specifies a list of characters used by the standard completion mechanism to tokenize completion strings. The default value is the binding of **dw::*standard-completion-delimiters***; this variable is preset to "- " (hyphen and space).

:enable-forced-return

Boolean option specifying whether the user can force a response that is not a member of the completion set; the default is **nil**.

If this option is **t**, the user can terminate input with `c-RETURN`, causing the completion utility to return to the caller whatever input

the user supplied. This is useful in situations where you expect the user to specify a member of a set of possibilities, but want to provide a way for supplying a new name to be added to the set. (The Zmacs Select Buffer (c-X B) command uses this feature to allow the user to create new buffers.)

:partial-completers
Specifies a list of characters that trigger partial completion when entered by the user.

Partial completion restricts completion to only one token of the completion set possibilities, even if enough characters have been supplied to uniquely identify one of the members of the completion set. For example, the Command Processor uses **#\space** as a partial completer.

The syntax of a token is defined by the **:delimiters** option.

:type Specifies the presentation type to use when displaying help information for possible completions. This makes the displayed possibilities mouse-sensitive.

If the completion utility is being called from the parser of a presentation type, that type should be supplied as the value of this option.

:parser Specifies the function called to translate input strings into objects of the desired type. The function is called with one argument, the string entered by the user.

This option is typically used when the set of possible completions is not known in advance, and can therefore not be enumerated. If they can be enumerated, use **dw:complete-from-sequence** or **dw:completing-from-suggestions** instead.

The parser function is called on each possible completion string when a list of possibilities is generated, and on the user-supplied input when the completion utility is about to return a value.

:complete-activates
Boolean option specifying whether the COMPLETE key causes activation, that is, whether the completion utility returns if a unique completion was found. The default is **nil**.

This option is used to control completion behavior in a multi-field input context, such as in the command processor. Normally, the END key performs completion and then returns if the resulting completion is unique.

:compress-choices
Specifies whether to compress the display of completion possibilities that have a common left token as defined by the **:compression-delimiter** option. Three values are possible:

An integer
When the possibilities exceed this number, the display is compressed. The default value is `20`.

:always Whenever more than one possibility exists, the display is compressed.

:never The display is never compressed, regardless of the number of possibilities.

Compressed displays have the form "*token* ... (*n*)", where *token* is the shared left token and *n* is the number of possible completions.

To see an example of choice compression, press `HELP` to the command processor prompt in a Dynamic Lisp Listener. You get the following display (abbreviated for this example):

```
You are being asked to enter a command or form.
Use the Help :Format Detailed command to see a full
list of command names.

These are the possible command names:
  Add Paging File
  Append
  Clean File
  Clear ... (3)
  Close File
  Compare Directories
  Compile ... (2)
  Copy ... (5)
  Create ... (4)
  Debug Process
```

"Add Paging File", "Append", and "Clean File" are full command names. "Clear" is a left token shared by three commands, Clear All Breakpoints, Clear Breakpoint, and Clear Output History. These three completion choices have been compressed to "Clear ... (3)". The user can expand this and other compressed choices by clicking on them with the mouse.

:compression-delimiter
Specifies a character used for delimiting the shared left tokens in a display of completion possibilities. The default value is **#\space**.

:initially-display-possibilities
Boolean option specifying whether to display the entire set of completion possibilities before prompting for input; the default is **nil**. If **t**, the behavior is as if the user typed `Help` before any other input.

Most parsers should supply to this option the same value that was supplied to them by **accept**. **accept**, in turn, has an **:initially-display-possibilities** option controlled by its caller. See the function **accept**.

For an overview of **dw:completing-from-suggestions** and related facilities, see the section "Defining Your Own Presentation Types".

(flavor:method :compute-motion tv:sheet) *string* &optional (*start* **0**) (*end* **nil**) *x y cr-at-end-p* (*stop-x* **0**) *stop-y* *Method*

Determines 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.

dw:computing-outline-from-path *(*&optional *(stream* ***standard-output****)* &key *:transform)* &body *body* *Macro*

Returns a sequence of lines suitable for **:highlighting-boxes** corresponding to unfilled graphics drawn to *stream* by *body*, possibly with a transform specified by the keyword **:transform**. For example:

```
(dw:computing-outline-from-path (stream)
 (graphics:draw-circle center-x center-y (+ radius 2)
       :stream stream :filled nil)
 (graphics:draw-circle center-x center-y (- radius 2)
       :stream stream :filled nil))
```

(flavor:method :configuration tv:basic-constraint-frame) *Method*

Returns the symbol naming the current configuration of the frame.

(flavor:method :configuration tv:basic-constraint-frame) *configuration-name*
Init Option

Makes the initial configuration of the frame be the one named by the symbol *configuration-name*.

(flavor:method :configurations tv:basic-constraint-frame) *configuration-specification-list*
Init Option

Controls the sizes and arrangement of the panes in each possible configuration of a constraint 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 in Non-Dynamic Windows". To convert a **:constraints** option to a **:configurations** option, see the function **tv:back-convert-constraints**.

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.

tv:constraint-frame *Flavor*

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:constraint-frame-with-shared-io-buffer *Flavor*

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**.)

(flavor:method :constraints tv:basic-constraint-frame) *configuration-description-list* *Init Option*

Required for all flavors of constraint frames before Dynamic windows. It has been replaced by the **:configurations** init option. See the init option **(flavor:method :configurations tv:basic-constraint-frame)**. To convert a **:constraints** option to a **:configurations** option, see the function **tv:back-convert-constraints**.

The argument, *configuration-description-list*, is a list of configuration descriptions. For the format of configuration descriptions, see the section "Specifying Panes and Constraints in Non-Dynamic Windows".

(flavor:method :constraints tv:basic-constraint-frame) *Method*

Returns the configuration description list of the frame.

dw:continuation-output-size *continuation stream* &optional (*unit* **:pixel**) *Function*

Determines the amount of space a specified continuation would require for output on a specified stream. Six values are returned: *width*, *height*, *cursor-motion-x*, and *cursor-motion-y*, *left, top*.

The *continuation* is **funcall**ed with a single argument, an internal stream, which tracks the cursor motion caused by the output code of *continuation*.

continuation
: The continuation to run.

stream
: The output stream. This must be supplied, even though no output is actually sent to it, because information about the stream is necessary. For example, if a **:string-length** message is involved, the default character style for the stream is needed information.

unit The unit of measure. The default is **:pixel**; the other possible value is **:character**.

Example:

```
(defmacro centering-about-point ((stream x y) &body body)
  `(centering-about-point-internal
     (zl:named-lambda centering-about-point (,stream) ,@body)
     ,stream ,x ,y))

(defun centering-about-point-internal (continuation stream x y)
  (multiple-value-bind (width height)
      (dw:continuation-output-size continuation stream)
    (let ((start-x (- x (round width 2)))
          (start-y (- y (round height 2))))
      (dw:in-sub-window (stream start-x start-y width height)
        (funcall continuation stream))
      ;; Drawing the lines is just to verify the centering
      (graphics:draw-line start-x start-y (+ start-x width)
                                  (+ start-y height))
      (graphics:draw-line (+ start-x width) start-y start-x
                                  (+ start-y height))
      )))

;;; Some code to test it
(defun test-centering ()
  (send *standard-output* :clear-window)
  (multiple-value-bind (left top right bottom)
      (send *standard-output* :visible-cursorpos-limits)
    (let ((center-x (round (+ left right) 2))
          (center-y (round (+ top bottom) 2)))
      (centering-about-point (*standard-output* center-x center-y)
        ;; Surround with border just to show
        ;; the bounding box of the output
        (dw:surrounding-output-with-border (*standard-output*)
          ;; Generate some output
          (cp:execute-command "Show Flavor Handler"
                              ':tyo 'dw:dynamic-window
                              :code :detailed))))
    ;; Drawing the lines is just to verify the centering
    (graphics:draw-line left top right bottom)
    (graphics:draw-line right top left bottom)
    ;; Pause to read a character before the command prompt
    ;;clobbers our carefully crafted output.
    (read-char)))
```

```
;; In a full-size Lisp Listener, try
;; (with-character-size (:large) (test-centering))
```

For an overview of **dw:continuation-output-size** and related facilities: See the section "Writing Formatted Output Macros".

(flavor:method :cr-not-newline-flag tv:sheet) *x* *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.

(flavor:method :current-font tv:sheet) *Method*

Returns the current font, that is, the font used for NIL.NIL.NIL, as a font object.

Example:

```
(send *standard-output* :current-font) ==>
#<FONT CPTFONT 260000546>
```

(flavor:method :current-geometry 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**.

dw:current-program &key *window (type* **'dw::program***) (error-p* **t***)* *Function*

Returns the current program of the type specified by **:type** given the starting window specified by **:window**. This useful for command translators that need to get to the program associated with the window in which you clicked.

time:daylight-savings-p *Function*

Returns **t** if daylight savings time is currently in effect; otherwise, return **nil**.

time:day-of-the-week-string *day-of-the-week* &optional (*mode* **':long**) *Function*

Returns 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.

(flavor:method :deactivate tv:menu) *Method*

Deactivates a window, deexposing it. In momentary menus, it is sent when the mouse is moved outside the borders of the menu.

dw:dead-blip *Flavor*

The error signalled when a mouse click goes unhandled. This is also the blip format returned when an attempt to convert a mouse blip to a presentation blip fails. This flavor has instance variable **mouse-char** and component flavor **condition** Methods on this flavor include **:report**.

decode-universal-time *universal-time* &optional *timezone* *Function*

Given a universal time, returns nine values for the corresponding decoded time: second (0-59); minute (0-59); hour (0-23); date (1-31); month (1-12); year (A.D.); day-of-week (0[Monday]-6[Sunday]); a flag (**t** or **nil**) indicating whether daylight savings time is in effect; and the timezone (hours west of GMT).

universal-time Seconds, plus or minus, since midnight, January 1, 1900 GMT.

timezone Hours west (postive) or east (negative) of GMT; it defaults to the timezone set for the site (for more information: See the section "Specifying a Time Zone for Your Site".) Any real number is allowed, but only numbers of the form *n* or *n.5* can correspond to actual timezones.

Examples:

```
(decode-universal-time 0) =>
0
0
19
31
12
1899
6
NIL
5

(decode-universal-time 18000) =>
0
0
0
1
1
1900
0
NIL
5

(decode-universal-time 0 0) =>
0
0
0
1
1
1900
0
NIL
0
```

time:decode-universal-time *universal-time* &optional *timezone* *Function*

Converts *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. Note that **decode-universal-time** is preferred for new code.

(flavor:method :decode-variable-type tv:basic-choose-variable-values) *kwd-and-args* *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.

- 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.

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.)

(flavor:method :deexpose 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.

(flavor:method :deexposed-typein-action tv:sheet) *Method*

Returns the deexposed typein action of the window.

(flavor:method :deexposed-typein-action tv:sheet) *action* *Init Option*

Initializes the deexposed typein action of the window to *action*. This is the action taken when typein is required and the window is not exposed. The possibilities are **:normal** and **:notify**. The default is **:normal**.

(flavor:method :deexposed-typeout-action tv:sheet) *Method*

Returns the deexposed typeout action of the window.

(flavor:method :deexposed-typeout-action tv:sheet) *action* *Init Option*

Initializes the deexposed typeout action of the window to *action*. This is the action taken when typeout is attempted and the window is not exposed. The possibilities are **:normal**, **:error**, **:permit**, **:expose**, and **:notify**, or a list of messages and message arguments. The default is **:normal**.

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***.

(flavor:method :default-character-style tv:menu) *character-style* *Init Option*

Specifies 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".)

(flavor:method :default-character-style tv:sheet) *character-style* *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)**.

For more information on character styles: See the section "Character Styles".

cp:*default-command-accelerator-echo* *Variable*

Controls whether accelerated commands are echoed (full command name) on the command line when their single-key accelerators are pressed. It is initially bound to **t**.

dw:default-command-top-level *program* &rest *options* &key (*window-wakeup* **#'dw::default-window-wakeup-handler**) &allow-other-keys *Function*

The default command loop function for programs created with **dw:define-program-framework**.

program The program instance. (This argument is supplied by **dw:define-program-framework**).

:window-wakeup
Specifies the function used by the program for handling asynchronous events (for example, when the user presses `REFRESH` or resizes the program frame). For an example:see the section "Handling Asynchronous Window System Events".

In addition to **:window-wakeup**, the following keyword options are available:

:stream Specifies the stream for program command I/O; the default is ***query-io***.

:prompt Specifies the command prompt. The default is a string consisting of the name of your program followed by the word "command" and a colon, that is, "<*Program-name*> command:"

:dispatch-mode
Specifies the Command Processor dispatch mode, one of **:command-only**, **:command-preferred**, **:form-only**, or **:form-preferred**. (See the function **cp:read-command-or-form**.)

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". The default is **:command-preferred**.

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

tv:defaulted-multiple-menu-choose *alist defaults* &optional *near-mode* *Function*

alist is a list of lists of menu items. The form of a menu item can be one of:

```
atom
(something atom)
(something . atom)
(something :value atom)
(something :eval 'atom)
(something :eval atom)
```

where in each case but the last *atom* is the item returned if selected in the menu. (In the last case the value of *atom* is returned.) See the section "The Form of a Menu Item". 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. For an example, see the section "**tv:multiple-menu-choose** Example".

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 sym-

bol other than **nil** (but not a list): See the section "Displaying Prompts in the Input Editor". The default is "Command: ".

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

cp:define-command *name-and-options arguments* &body *body* *Macro*

Defines a Command Processor command. This important macro has many features and options. Its complete definition is contained in the chapter explaining its use, "Managing the Command Processor". See the function **cp:define-command**.

cp:define-command-accelerator *name command-table characters options arglist* &body *body* *Function*

Defines single-key accelerators for Command Processor commands.

name Name for this accelerator.

command-table
Command table in which command and accelerator are included.

characters A character or list of characters (not necessarily more than one) serving as the single-key accelerators. Prefix character sequences are also allowed, for example, `((#\c-x #\c-f))`.

options List of keyword-value pairs. Possible keywords include:

:argument-allowed
Boolean option specifying whether this accelerator is allowed to take numeric arguments (for example, `c-3`). The default depends on whether you provide an *arglist*, **t** if you do, **nil** if you don't.

:activate Boolean option specifying whether the command defined by this accelerator executes immediately when the accelerator is typed; the default is **t**. If **nil**, the command requires confirmation and, possibly, additional args.

:echo Boolean option specifying whether the command defined by this accelerator echoes on the command line as if it were typed. The default is the value supplied to the **:activate** option; this is because in the `:activate nil` case, the command is visible after you are finished editing and need not be repeated.

arglist List of arguments to the accelerated command. If **:argument-allowed** is **nil**, this arglist should be **nil** (no arguments allowed).

If **:argument-allowed** is **t**, the accelerator receives two arguments, *arg-p* and *arg*. *arg-p* means whether or not the user gave an argument to this accelerator; *arg* is the numeric *arg*. *arg-p* has several special keyword values. These are **:sign**, **:digits**, **:control-u**, and **:infinity**. The *arglist* is typically just `(arg-p arg)`, but you can put

anything here you want. This is just so that your *body* can make reference to these symbols under the names you chose.

body A form that returns a command. It can make reference to the symbols bound in *arglist*.

A typical body might be:

```
`(si:com-delete-file ,(list path)))
```

For an overview of **cp:define-command-accelerator** and related facilities: See the section "Managing the Command Processor".

cp:define-command-and-parser *name-and-options arglist parser* &body *body*
Function

Defines a Command Processor command and command line parser.

name-and-options
Either the symbol to be used as the command name or a list whose first element is the name symbol and succeeding elements are alternating keyword-value pairs. To distinguish command names from other kinds of names, we recommend that the prefix *com-* be used; the user-visible command name will not include the prefix.

Permissible keywords are the same as those listed under *name-and-options* in the dictionary entry for **cp:define-command**.

arglist The argument list of the function that implements the body of the command. It is a normal, Common Lisp argument list.

parser A form used to parse the command's arguments. This form has lexical access to the internal functions **cp:read-command-argument**, **cp:read-keyword-arguments**, and **cp:assign-argument-value**. It should use these functions to do the actual reading and assigning of values to command arguments:

cp:read-command-argument *presentation-type &rest options*
A **flet**ted function within **cp:define-command-and-parser**. *presentation-type* is the type of the argument. *options* are all options acceptable in a command argument specification to **cp:define-command**.

cp:read-keyword-arguments *&rest keyword-specs*
A **macrolet**ted macro within **cp:define-command-and-parser**. *keyword-specs* are command argument specifications identical to those you would use if you were writing the command using **cp:define-command**. Even if there are no keyword arguments, the parser should end with **cp:read-keyword-arguments**; any automatically generated keywords (for example, **:output-destination**) can thereby be read.

cp:assign-argument-value *argument-name value*

A **macrolet**ted macro within **cp:define-command-and-parser**. *Argument-name* is a symbol naming a command argument; *value* is its value. Each *argument-name* should correspond to an argument in **arglist** above.

Example:

```
(cp:define-command (com-this-is-a-test
        :command-table 'user)
    ((file 'pathname :default nil :prompt "file")
     &key
     (integer 'integer :default 17
        :mentioned-default 3 :prompt "the number"))
   (loop for i from 0 to integer do
           (print file)))

;;;is equivalent to

(cp:define-command-and-parser (com-this-is-a-test
           :command-table 'user)

        ;; The arglist of the function.
        ;; Note the presence (and need for) the
        ;; default value for INTEGER in the
        ;; argument list.
        (file &key (integer 17))

  ;; The argument parser.  It's just one big PROGN.
  ;; Note that it ends with read-keyword-arguments.
  (progn (cp::assign-argument-value file
           (cp::read-command-argument 'pathname
               :default nil :prompt "file"))
         (cp::read-keyword-arguments
           (integer 'integer :default 17
               :mentioned-default 3 :prompt "the number")))

  ;; The body of the command.
  (loop for i from 1 to integer do (print file)))
```

To see other examples, try **macroexpand**ing some **cp:define-command** definitions; they expand into **cp:define-command-and-parser** definitions.

For an overview of **cp:define-command-and-parser** and related facilities: See the section "Managing Your Program Frame".

dw:define-command-menu-handler *(command-name command-table menu-levels* &key *(:gesture* **:left***) (:documentation* **t***) :tester) arglist* &body *command-form* *Macro*

Defines a menu handler for the command named *command-name* in *command-table* for *menu-levels*. That is, defines a function that executes *command-form* with the arguments in *arglist* when the user clicks the mouse as specified by **:gesture**. *command-form* should return a command, just as **define-presentation-to-command-translator** does.

command-name
: A string, which identifies the item, and which is how it is normally displayed.

command-table
: The command table into which to install the handler. It is normally a string, but can be anything that **cp:find-command-table** understands, including a symbol.

menu-levels
: A list of keywords naming the menu levels in which the handler should appear. The same handler can be in more than one menu level. Normally, of course, menus for both levels would never be on the screen at the same time in this case.

gesture
: A single keyword, or list of keywords, identifying which gestures this handler applies to. A single handler can apply to more than one gesture, and multiple handlers can be defined on different gestures. The possible actions will then naturally be their union.

documentation
: Similar to the documentation for **define-presentation-to-command-translator**. It can be a fixed string or a list of arguments and body to produce such a string. Additionally, it can be **t**, meaning run the *command-form* to produce the desired command and use it for the documentation. This is the default.

tester
: Similar to the tester for **define-presentation-to-command-translator**. It is a function name (symbol) or list of arguments and body to define such a function. The function is run to determine whether the handler is available.

arglist (start with &key):
: Contrary to the case of **define-presentation-to-command-translator**, there is no initial presentation object argument. The keywords passed to as this arglist are **:menu-level** and **:gesture**, specifying how the handler was invoked, and **:window**, giving the menu from which the presentation was taken.

Normally, a command menu handler calls **dw:standard-command-menu-handler**, which takes a command name and arguments as passed to the command form of **dw:define-command-menu-handler** and does the standard actions for two mouse gestures.

For an overview of how command menus work and several examples of the use of **dw:define-command-menu-handler**: See the section "How Command Menus Work".

define-cp-command *name args* &body *body* *Function*

The pre-Genera 7.0 facility for defining Command Processor commands. Currently, the recommended command-definition facility is **cp:define-command**. For an overview of the latter and related facilities: See the section "Managing the Command Processor".

Code that defines commands with the obsolete **define-cp-command** macro still works, but the compiler will suggest that you change the syntax of any old-style argument type specifiers (for example, **:pathname**) to presentation type specifiers (for example, **'pathname**) — in fact, it will make that change for you. **Note**: You can no longer use user-defined old-style type-specifiers. These will not compile.

define-cp-command 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".

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 preced-

ing **&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".

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".

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))
```

define-presentation-action *name* (*from-presentation-type to-presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority exclude-other-handlers blank-area defines-menu*) *arglist* &body *body* *Macro*

Defines a side-effecting mouse handler for performing actions on a displayed pre-

sentation object that are independent of the main body and command loop of an application. The complete description of this macro is containted in the chapter "Programming the Mouse: Writing Mouse Handlers".

define-presentation-to-command-translator *name* *Macro*
(*presentation-type* &key *tester* (*gesture* **:select**) *documentation*
suppress-highlighting (*menu* **t**) (*context-independent* **nil**) *priority*
exclude-other-handlers blank-area do-not-compose) *arglist* &body *body*

Defines a mouse handler that translates from a displayed presentation object into a Command Processor command using that object as input. The complete description of this macro is contained in the chapter "Programming the Mouse: Writing Mouse Handlers".

zwei:define-presentation-to-editor-command-translator *name (type echo-name comtab* &rest *options* &key *:gesture :tester* &allow-other-keys*) (object* &rest *other-args)* &body *body* *Macro*

Returns the list of a function name and argument values that calls an editor command function. The function need not be defined with **zwei:defcommand**. The function should return **nil** if the typeout window should be flushed, or non-**nil** if the typeout window should be left alone.

name	The name of the command.
type	The from-presentation type.
echo-name	A string to echo when the mouse is clicked.
comtab	A symbol whose value is the command table that determines whenter the translator is available. Only if commands in that command table are available is this translator available. This is typically **zwei:*standard-comtab*** or **zwei:*zmacs-comtab*** or it could be your own command table.

The remaining arguments are the same as those of **define-presentation-to-command-translator**. See the function **define-presentation-to-command-translator**.

Example:

```
(defun show-length-of-plist (symbol)
   (zwei:typein-line "~D" (length (symbol-plist symbol))))

(zwei:define-presentation-to-editor-command-translator
                                 show-length-of-plist
                                   (symbol "Plist length"
                                      zwei:*standard-comtab*
                                      :gesture :super-middle)
                                   (symbol)
  '(show-length-of-plist ,symbol))
```

define-presentation-translator *name* (*from-presentation-type to-presentation-type* &key *tester* (*gesture* **:select**) *documentation suppress-highlighting* (*menu* **t**) (*context-independent* **nil**) *priority exclude-other-handlers blank-area do-not-compose*) *arglist* &body *body* *Macro*

Defines a mouse handler that translates from a displayed presentation object of a certain type to a returned presentation object of a different type. The complete description of this macro is contained in chapter "Programming the Mouse: Writing Mouse Handlers".

define-presentation-type *type-name* (*data-arglist . pr-arglist*) &key *parser printer viewspec-choices description describer no-deftype disallow-atomic-type* (*history* **nil** *history-supplied-p*) *expander abbreviation-for choose-displayer accept-values-displayer menu-displayer default-preprocessor history-postprocessor highlighting-box-function presentation-type-arguments presentation-subtypep key-generator key-function do-compiler-warnings map-over-subtypes map-over-supertypes map-over-supertypes-and-subtypes typep with-cache-key* (*data-arguments-are-disjoint* **t**) *Macro*

Defines a presentation type. The full description of this macro is contained in a chapter of its own. See the section "Defining Your Own Presentation Types".

dw:define-program-command *(name program-name* &rest *options* &key *(keyboard-accelerator* **nil***) (menu-accelerator* **nil***) (menu-level* **'(:top-level)***) (menu-documentation* **t***) menu-documentation-include-defaults provide-output-destination-keyword* &allow-other-keys*) arglist* &body *body* *Function*

Defines a Command Processor command for a program created with **dw:define-program-framework** and installs it in the program's command table. The definition generates two internal methods for the program flavor, one to parse the command and one to execute the command. These methods provide lexical access to your program's state variables both in the body of the command definition and in the command argument list; that is, you may use state variables as arguments.

name The name given to the command. To distinguish command names from other kinds of names, we recommend that the prefix `com-` be used, for example `com-exit-program`. The user-visible command does not include the prefix; in the above example, the user-visible command is `Exit Program`.

Like other commands, those you define using **dw:define-program-command** occupy the function namespace.

program-name
The symbol or string naming the program flavor (created by **dw:define-program-framework**) for which the command is being written. This is also the name of the program's command table.

:keyboard-accelerator

Specifies the key used to invoke the command; the default is **nil**. For example, if you are writing an *Exit Program* command, you might wish to specify #\E as the keyboard accelerator.

This option may not be used if **nil** is specified for the **:kbd-accelerator-p** option to the **:command-table** keyword for **dw:define-program-framework**. See the function **dw:define-program-framework**.

:menu-accelerator

Specifies whether the command identifier is displayed in a command menu pane for the program; the default is **nil**.

To make the command available in a menu, supply a value of **t** or a string. **t** causes the user-visible name of the command to be displayed. If you provide a string, it is displayed instead of the user-visible name.

Note that the program frame must include a pane of the **:command-menu** type in order for the command identifier to be displayed. See the function **dw:define-program-framework**.

:menu-level

Specifies the command menu in which the command is to be displayed. You need to use this option explicitly only when more than one command menu pane has been specified in the **dw:define-program-framework** macro for your program. (See the function **dw:define-program-framework**.)

:menu-documentation

Specifies documentation to be displayed in the mouse documentation line when the mouse is over the command in the command menu.

:menu-documentation can be either a string, which is the documentation, or it can be a function that takes **:window**, **:gesture**, **:menu-level**, and **:arguments** as keyword arguments and either returns a string or a command-structure (as in `'(si:com-show-file ,foo)`). If a command-structure is returned it is unparsed to produce the documentation.

Examples:

```
(dw:define-program-framework MD
  :select-key #\!
  :panes ((display :display)
          (menu :command-menu)
          (interactor :interactor))
  :command-definer t)
```

```
(define-MD-command (com-test-1 :menu-accelerator t
                               :menu-documentation
                          "This is the MD command")
    ((integer 'integer))
   (print integer))

(define-MD-command (com-test-2 :menu-accelerator t
                               :menu-documentation
                                  ((&rest ignored)
                                   `(com-test-1 ,3)))
    ()
   ())
```

:menu-documentation-include-defaults

Specifies, when **t**, that the defaults for this command should be presented in the mouse documentation line.

Compare what happens when you move the mouse over Test 3 and Test 4. The mouse documentation for Test Four is "Test 4 7" (or whatever the last integer you typed to the Test 4 command was) while the mouse documentation for Test 3 is just "Test 3".

```
(dw:define-program-framework MDID
  :select-key #\~
  :panes ((display :display)
          (menu :command-menu)
          (interactor :interactor))
  :command-definer t)

(define-MDID-command (com-test-3 :menu-accelerator t)
    ((integer 'integer))
   (print integer))

(define-MDID-command (com-test-4 :menu-accelerator t
              :menu-documentation-include-defaults t)
    ((integer 'integer))
   (print integer))
```

:provide-output-destination-keyword

Boolean option specifying whether to provide the **:output-destination** keyword. The default is **nil** (unlike **cp:define-command**). This option allows the user of the command to redirect the output of the command to a place other than the screen.

Additional keyword options to **dw:define-program-command** are the same as those documented under *name-and-options* in the Dictionary entry for **cp:define-command**. These include **:command-table**, **:explicit-arglist**, **:provide-output-destination-keyword**, and **:values**.

arglist The list of command arguments. Each element of the list is itself a list of the form (*arg-name presentation-type options*) where *arg-name* is the name of the argument; *presentation-type* is the presentation-type of the argument; and *options* are keyword options to the argument.

Permissible options are the same as those documented under *arguments* in the description of **cp:define-command** in "Managing the Command Processor". These include **:documentation**, **:prompt**, **:prompt-mode**, **:default**, **:mentioned-default**, **:when**, **:name**, **:default-type**, **:provide-default**, **:display-default**, and **:confirm**.

For an overview of **dw:define-program-command** and related facilities, see the section "Defining Your Own Program Framework".

dw:define-program-framework *name* &key *pretty-name* Macro
(*command-definer* **nil**) (*command-table* **nil**) (*top-level* **(quote (dw:default-command-top-level))**) (*command-evaluator* **nil**) (*panes* **(quote (dw::main :listener))**) *selected-pane query-io-pane terminal-io-pane label-pane* (*configurations* **nil**) (*state-variables* **nil**) (*selectable* **t**) (*select-key* **nil**) (*system-menu* **nil**) (*size-from-pane* **nil**) (*inherit-from* **(quote (dw::program))**) (*other-defflavor-options* **nil**)

Defines a program framework. The complete definition of this macro is included in its own chapter. See the section "Defining Your Own Program Framework".

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

This function is obsolete. New types should be defined with **define-presentation-type**.

define-prompt-and-read-type defines a new type keyword for **prompt-and-read**, and 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.

define-user-option *(option alist) default [type] [name]* *Function*

(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"**.

define-user-option-alist *name [constructor]* *Function*

(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**.

tv:defwindow-resource *name parameters* &rest *options* *Function*

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** without a mtaching **deallocate-resource** instead of **using-resource** to allocate resources. Default: reusable when not locked and not in use.

tv:defwindow-resource also accepts any of the keywords from **defresource**. **tv:defwindow-resource** handles its own keywords and **defresource**'s **:checker** keyword, and any others it passes to **defresource**.

tv:delayed-redisplay-label-mixin *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.

(flavor:method :delayed-set-label tv:delayed-redisplay-label-mixin) *specification*
Method

Like the **:set-label** method, except that nothing actually happens until an **:update-label** message is sent.

(flavor:method :delayed-set-label dw:margin-mixin) *new-label* *Method*

Provides a new label for a Dynamic Window, but delays the writing of the new label until the **:update-label** message is sent: See the method **(flavor:method :update-label dw:margin-mixin)**. See the method **(flavor:method :set-label dw:margin-mixin)**.

new-label The new label, which can be either a string or **t**, which specifies that the window's name is to be the label, or a list of options to **dw:margin-label**.

For an overview of **(flavor:method :delayed-set-label dw:margin-mixin)** and related facilities: See the section "Window Substrate Facilities".

tv:delaying-screen-management *Function*

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.

(flavor:method :delete-char tv:sheet) &optional (*char-count* **1**) (*unit* **':character**)
Method

Without an argument, deletes the character at the current cursor position. Otherwise, deletes *char-count* units, starting at the current cursor position. Moves 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.

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 "Managing the Command Processor".

(flavor:method :delete-displayed-presentation dw:dynamic-window) *displayed-presentation* *Method*

Deletes an already displayed presentation from a Dynamic Window's output history and display. This method does not immediately affect the display, nor does it remove associated text from the screen; use **dw:erase-displayed-presentation** to do these.

displayed-presentation
The presentation to delete.

For an overview of **(flavor:method :delete-displayed-presentation dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output". Example:

```
(defun delete-displayed-presentation-test ()
 (let ((presentation (dw:with-output-as-presentation (:type 'sys:expression
                                                      :object 'test)
                       (princ "test object"))))
  (format t "~&Click on presentation")
  (read)
  (send *standard-output* :delete-displayed-presentation presentation)
  (format t "~&No no longer sensitive.")
  ))
```

(flavor:method :delete-item tv:text-scroll-window) *item-no* *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.

(flavor:method :delete-line tv:sheet) &optional (*line-count* **1**) (*unit* **':character**) *Method*

Without an argument, delete the line that the cursor is on. Otherwise deletes *line-count* units, starting with the one the cursor is on. Moves 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.

dw:delete-presentation-mouse-handler *name* *Function*

Removes an already defined presentation mouse handler.

name The name of the mouse handler to remove.

For an overview of **dw:delete-presentation-mouse-handler** and related facilities: See the section "Programming the Mouse: Writing Mouse Handlers".

dw:delete-presentation-type *type-name* *Function*

Deletes presentation type *type-name* from the type hierarchy. Note that the system will not let you delete presentation types for which there are mouse-handlers defined.

(flavor:method :delete-string tv:sheet) *string* &optional *(start* **0***) (end* **nil***)* *Method*

Deletes specific strings. It is one of the things to use when dealing with character styles that map to variable-width fonts.

If *string* is a string, excise a region exactly as wide as that string, or a substring specified by *start* and *end*, and closes the gap.

dw:describe-presentation-type *type* &optional (*stream* ***standard-output***) *plural-count* *Function*

Outputs the description of a presentation type provided by the type's definition (**define-presentation-type** macro).

type The presentation type to be described.

stream The output stream; the default is ***standard-output***.

plural-count
Controls whether the description is pluralized. Three values are possible:

nil Do not pluralize the description.

t Pluralize the description.

number Include this number in the pluralization.

Examples:

```
(dw:describe-presentation-type 'integer) ==>an integer

(dw:describe-presentation-type 'integer t t) ==>integers

(dw:describe-presentation-type 'integer t 12) ==>twelve integers

(dw:describe-presentation-type 'integer t 12.2) ==>12.2 integers
```

For an overview of **dw:describe-presentation-type** and related facilities: See the section "Defining Your Own Presentation Types".

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

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.

(flavor:method :deselected-visibility tv:blinker) *Method*

Examines the deselected visibility of the blinker.

(flavor:method :deselected-visibility tv:blinker) *symbol* *Init Option*

Sets the initial deselected visibility. By default, it is **:on**.

dw:*display-ellipsis-help* *Variable*

Controls the presentation of a help message explaining the meaning of a notation such as "`Foo ... (7)`" to indicate 7 possibilities beginning with "`Foo`". By default, the value is **t** which means that this help message continues to be presented every time such a notation is used until the first time such an item is clicked on, at which point the value becomes **nil** and the message is no longer presented. The value may be set to **nil** before this in order to suppress the message at an earlier point (in an init file, for example) or it may be set to **:always** in order to keep it from ever becoming **nil**, regardless of whether such an item is clicked on or not.

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 and of type *type*.

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**.

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.

zl:display-notifications *Function*

Selects a scroll window that displays all notifications received since cold booting. This is the same as SELECT N.

tv:displayed-item-item *mouse-sensitive-item* *Function*

Given a mouse-sensitive item, returns the associated item.

tv:displayed-item-type *mouse-sensitive-item* *Function*

Given a mouse-sensitive item, returns the type of the item.

dw:displayed-presentation-clear-highlighting *displayed-presentation window* &optional (*highlighting-mode* **:underline**) *Function*

Eliminates highlighting of a displayed presentation (see **dw:displayed-presentation-set-highlighting**).

displayed-presentation
The highlighted presentation.

window The window displaying the presentation.

highlighting-mode
The mode in which the displayed presentation is highlighted, either **:underline** (the default for **dw:displayed-presentation-set-highlighting**) or **:inverse-video**.

For an overview of **dw:displayed-presentation-clear-highlighting** and related facilities, see the section "Controlling Location and Other Aspects of Output".

dw:displayed-presentation-highlighting-boxes *displayed-presentation* *Function*

Returns the value of the highlighting-boxes instance variable of the instance *displayed-presentation*.

dw:displayed-presentation-set-highlighting *displayed-presentation window* &optional (*highlighting-mode* **:underline**) *Function*

Highlights a displayed presentation.

displayed-presentation	The presentation to highlight.
window	The window displaying the presentation.
highlighting-mode	Either **:underline** (the default) or **:inverse-video**.

For an overview of **dw:displayed-presentation-set-highlighting** and related facilities, see the section "Controlling Location and Other Aspects of Output". Example:

```
(defun presentation-highlighting-test ()
 (fresh-line)
 (let ((presentations ()))
  (formatting-item-list
   (t :dont-snapshot-variables (presentations))
   (loop for i from 1 to 5 do
     (formatting-cell
      (t :dont-snapshot-variables (presentations))
      (let ((presentation
             (dw:with-output-as-presentation (:object i :type 'integer)
              (format t "~R" i))))
       (when presentation
        (push presentation presentations))))))
  (let ((n 3)
        (highlighted nil))
   (loop
    (when highlighted
     (dw:displayed-presentation-clear-highlighting
      highlighted *standard-output*
      :inverse-video))
    (setq highlighted
          (find n presentations :key #'dw:presentation-object))
    (when highlighted
     (dw:displayed-presentation-set-highlighting
      highlighted *standard-output*
      :inverse-video))
    (sleep 1)
    (setq n (1+ (random 5)))))))
```

dw:displayed-presentation-single-box *displayed-presentation* *Function*

Returns the value of the single-box instance variable of the instance *displayed-presentation*.

: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.

dw:do-redisplay *redisplay-piece* &optional (*stream* ***standard-output***) &key *full-set-cursorpos truncate-p once-only save-cursor-position* *Function*

Causes incremental redisplay from a redisplay object (created by **dw:redisplayer**). It runs the code in the *body* of the redisplayer, doing output to *stream* with respect to the display cache points described under **dw:with-redisplayable-output**.

redisplay-piece The redisplay object.

stream The output stream; the default is ***standard-output***.

:full-set-cursorpos Boolean option specifying whether the cursor will move backwards or sideways, rather than in strict tty style, so that a special stream is necessary; the default is **nil**.

:truncate-p Option specifying whether to do the redisplay with the output stream in truncate mode. With `:truncate-p nil`, the default, the output window rescrolls to update separate parts of the display. With `:truncate-p t`, some updating happens off-screen.

A third value permitted for this option is **:if-necessary**. In this case, **dw:do-redisplay** simulates, if necessary, some cursor motion on behalf of the output stream.

:once-only Specifies when **t** that the redisplay will not be repeated. (This is mostly for internal use by table-formatting facilities.) The default is **nil**.

:save-cursor-position Specifies when **t** that the cursor should be left at its current position rather than moved to the end of the new display. The default is **nil**.

dw:do-redisplay is one of a number of facilities used to do incremental redisplay. For examples, see the file SYS:EXAMPLES;INCREMENTAL-REDISPLAY.LISP.

For an overview of **dw:do-redisplay** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

tv:dolist-noting-progress (*var listform name* &optional (*progress-note-variable* **'tv:*current-progress-note***) (*process* **'sys:current-process**)) &body *body* *Function*

Binds local environment such that the progress of a **dolist** special form is noted by a progress bar displayed in the status line at the bottom of the screen.

var A variable bound to each successive element in *listform* on each successive iteration.

listform The list.

name A string naming the operation being noted. This string is displayed with the progress bar.

progress-note-variable The variable bound to the note object; the default is **tv:*current-progress-note***.

process The process on whose behalf the progress is noted; the default is **sys:current-process**. This is used to determine the precedence of notes.

Example:

```
(defun note-element-printing (list)
  (tv:dolist-noting-progress (element list "Printing elements")
    (print element)
    (sleep 1)))
```

For an overview of **tv:dolist-noting-progress** and related facilities, see the section "Progress Indicator Facilities".

tv:dont-select-with-mouse-mixin *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:dotimes-noting-progress (*var countform name* &optional (*progress-note-variable* **'tv:*current-progress-note***) (*process* **'sys:current-process**)) &body *body* *Function*

Binds local environment such that the progress of a **dotimes** special form is noted by a progress bar displayed in the status line at the bottom of the screen.

var A variable bound to the count (0, 1, 2, and so on) on each successive iteration.

countform The number of iterations.

name A string naming the operation being noted. This string is displayed with the progress bar.

progress-note-variable The variable bound to the note object; the default is **tv:*current-progress-note***.

process The process on whose behalf the progress is noted; this is used to determine the precedence of notes.

Example:

```
(defun note-square-roots (n)
  (tv:dotimes-noting-progress
    (count n "Calculating square roots")
    (sqrt count)))
```

For an overview of **tv:dotimes-noting-progress** and related facilities, see the section "Progress Indicator Facilities".

:draw-1-bit-raster *width height raster from-x from-y to-x to-y &optional (ones-alu :draw) (zeros-alu :erase)* *Generic Function*

Draws a pattern onto the screen. The pattern is replicated as needed, as with **bitblt**. Unlike **bitblt**, which copies bits regardless of any difference in bit depth (element type) between the source array and the screen array, **:draw-1-bit-raster** draws one screen pixel for each source pixel (the source must be a 1-bit array). Bits that are on in the source are drawn using *ones-alu* and bits that are off are drawn using *zeros-alu*. For a 1-bit screen, the result is like **bitblt** would have done with **tv:alu-seta**.

To say it another way, **:draw-1-bit-raster** copies *pixels* from a 1-bit-per-pixel source to the destination, which can be any pixel depth. If the destination is also 1-bit-per-pixel, **:draw-1-bit-raster** is identical to **bitblt**, but if the destination has more bits, **:draw-1-bit-raster** will do the "right" thing where **bitblt** would produce useless results. For detailed information on all the arguments of **:draw-1-bit-raster**:

See the function **bitblt**.

For a color screen, *ones-alu* and *zeros-alu* can be color alus. So, for instance, ones bits might be put out in green and zeros bits in red. Even when drawing in black in white to a color screen, **:draw-1-bit-raster** should be used for drawing stipples, because a whole pixel needs to be drawn black for the on pixels, not just one bit (which is only part of a pixel). Using **:draw-1-bit-raster** rather than **bitblt** is important in programs that run without modification on color screens.

tv:dynamic-item-list-mixin *Flavor*

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), the item list is considered changed and the menu is updated. **:item-list-pointer** is evaluated when the **:choose** message is sent.

tv:dynamic-momentary-menu *Flavor*

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*

A momentary menu with both the **tv:dynamic-item-list-mixin** and the **tv:window-hacking-mixin.**

tv:dynamic-multicolumn-mixin *Flavor*

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.

tv:dynamic-pop-up-abort-on-deexpose-command-menu *Flavor*

A command menu with the **tv:dynamic-pop-up-command-menu** and **tv:abort-on-deexpose** mixins.

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-menu *Flavor*

A pop-up menu with the dynamic item-list mixin.

dw:dynamic-window *Resource*

A resource of Dynamic Windows. The resource is created via **tv:defwindow-resource** with the **:initial-copies** option set to 1 and the **:reuseable-when** option set to **:deactivated**. (For more information on resources generally, see the section "Resources".)

The following keyword options are available when allocating from or using the Dynamic Window resource:

:momentary-p Boolean option specifying whether the window provided is momentary, that is, whether it is deactivated if the mouse cursor is moved off the window. The default is **nil**.

:temporary-p Boolean option specifying whether the window provided is temporary, that is, whether it locks the superior window until it is deactivated. The default is the value of the **:momentary-p** option.

:hysteresis If the **:momentary-p** option is **t**, specifies the distance, in pixels, that the mouse cursor must be from the edge of the window before it is deactivated. The default value is 25.

Note that in order to use these keywords, you must also supply an optional positional argument for the window's superior. In the following example, the superior is **tv:main-screen**, which is also the default if no arguments are supplied.

Example:

```
(defun dw-resource ()
  (using-resource (my-dw dw:dynamic-window tv:main-screen
                         :momentary-p t :hysteresis 15)
    (send my-dw :set-size 500 300)
    (send my-dw :expose)))
```

dw:dynamic-window *Flavor*

The basic Dynamic Window flavor. It provides output-history recording (of displayed presentations) as well as vertical and horizontal scrolling. Dynamic Windows are created in the same manner as static windows, with the **tv:make-window** function.

dw:dynamic-window is built on several component flavors, from which it inherits a large number of init options. These include all init options (about 40) to the basic, non-Dynamic Window flavor, **tv:window**. Below we provide references to these inherited options, but first discuss four that are specific to Dynamic Windows.

:end-of-page-mode Specifies what happens when queued output exceeds the space available in the current viewport of the window. There are four possibilities:

- **:default** Uses the global default for Dynamic Windows set by the Set Screen Options command or the **dw:set-default-end-of-page-mode** function on which the command is based.
- **:scroll** Causes the window to scroll automatically to accommodate the output. The amount by which the window is scrolled is set by the **:scroll-factor** init option to Dynamic Windows.
- **:truncate** Causes scrolling to be the responsibility of the user, who must press the SCROLL key to see more output.
- **:wrap** Causes new output to appear at the top of the window, rather than at the bottom as in the case of **:scroll** or **:truncate**.

:scroll-factor Specifies the amount by which a Dynamic Window is scrolled when the value of its **:end-of-page-mode** init option is **:scroll**. Possible values include an integer (number of lines), ratio (fraction of the screen), or **nil** (use the global default set by the Set Screen Options command or the function **dw:set-default-end-of-page-mode**).

:mouse-blinker-character Specifies the shape of the mouse cursor when it is over the window, for example, **#\mouse:fat-circle**. The default is **#\mouse:nw-arrow**. For a full listing of all the possibilities, see the section "Mouse-Blinker Characters".

:margin-components Specifies a list of the form ((*component-1* [*keys*]) (*component-2* [*keys*]) ... (*component-n* [*keys*])), where *component-x* is one of a set of margin-component flavors and *keys* are zero or more keywords or keyword-value pairs appropriate for the given flavor. Note that the same margin-component flavor can appear more than once in this list. For example, you can have more than one scroll bar.

Available margin-component flavors include the following:

dw:margin-borders Provides a four-sided, black (normal video) border of a specified thickness.

dw:margin-white-borders Provides a four-sided, white border of a specified thickness.

dw:margin-whitespace Provides whitespace of a specified thickness on a specified margin.

dw:margin-drop-shadow-borders Provides a three-pixel-wide black border shadowed on its right and bottom margins by an eight-pixel-wide gray border.

dw:margin-ragged-borders Provides a ragged (wavy) border of a specified thickness.

dw:margin-label Provides a label on the upper or lower margin. By default, the label string is created from the name of the window flavor.

dw:margin-scroll-bar Provides the standard elevator scroll bar on the specified margin.

For more detailed information on these flavors, including allowable keywords, see the respective dictionary entry for each.

The following example illustrates the use of margin-component flavors. Note that the margin is built from the outside in.

```
(defun dynamic-window-margin-example ()
  (let ((test (tv:make-window 'dw:dynamic-window
        :edges-from :mouse
        :margin-components
        '((dw:margin-borders :thickness 1)
          (dw:margin-white-borders :thickness 3)
          (dw:margin-borders :thickness 10)
          (dw:margin-white-borders :thickness 8)
          (dw:margin-borders :thickness 3)
          (dw:margin-whitespace :margin :left :thickness 10)
          (dw:margin-scroll-bar)
          (dw:margin-whitespace :margin :bottom :thickness 7)
          (dw:margin-scroll-bar :margin :bottom)
          (dw:margin-whitespace :margin :left :thickness 10)
          (dw:margin-label :margin :bottom
                           :style (:sans-serif :italic :normal))
          (dw:margin-whitespace :margin :top :thickness 10)
          (dw:margin-whitespace :margin :right :thickness 13))
        :expose-p t)))
    (send test :set-label "Margin Test Window")))
```

The remaining init options to **dw:dynamic-window** are those it shares with **tv:window**. These are documented elsewhere. Below is a list of references and the associated init options.

See the section "Creating a Window".

:blinker-p
:default-character-style
:save-bits
:superior
:activate-p
:expose-p

See the section "Window Attributes for Character Output".

:more-p
:vsp
:reverse-video-p
:deexposed-typeout-action
:deexposed-typein-action
:right-margin-character-flag
:backspace-not-overprinting-flag
:cr-not-newline-flag
:tab-nchars

See the section "Initializing Window Size and Position".

:left **:inside-width**
:x **:inside-height**

:top **:inside-size**
:y **:edges**
:position **:character-width**
:right **:character-height**
:bottom **:integral-p**
:width **:edges-from**
:height **:minimum-height**
:size **:minimum-width**

See the section "Window Borders".

Window borders are comparable to margin components. The two are incompatible: you cannot specify one of these flavors if you specify **:margin-components**.

:borders
:border-margin-width

See the section "Window Labels".

Window labels are also comparable to margin components. The two are incompatible: you cannot specify one of these flavors if you specify **:margin-components**.

:name
:label

See the section "Flavors for Panes and Frames".

:io-buffer

In addition to the large overlap in init options between static and Dynamic Windows, virtually all of the window methods, messages, and functions documented for static windows can also be used with Dynamic Windows. These are too numerous to list individually as we did for the init options; refer to the following sections for more information:

See the section "Window Graying".

See the section "Window Status".

See the section "Activities and Window Selection".

See the section "Creating a Window".

See the section "Character Output to Windows".

See the section "Graphic Output to Windows" (also see the section "Creating Graphic Output").

See the section "Notifications and Progress Indicators".

See the section "Using TV Fonts".

See the section "Handling the Mouse".

See the section "Window Sizes and Positions".

See the section "Window Labels" (Only the **:name** method).

Finally, a number of methods intended exclusively for Dynamic Windows are available. These are included among both Basic Program Output Facilities and window substrate facilities (see the section "Controlling Location and Other Aspects of Output" and see the section "Window Substrate Facilities").

cp:echo-command *command-name arguments* *Function*

Echoes a Command Processor command and its arguments to ***standard-output***. (The echoed command is presented "acceptably", that is, in such a manner that is can subsequently be parsed by **accept**.) The echoed command is displayed in italics, so this is not generally useful. Instead, use the **present** function with type 'cp:command.

command-name The command name (symbol).

arguments A list of command arguments.

For an overview of **cp:echo-command** and related facilities: See the section "Managing Your Program Frame".

dw:echo-presentation-blip *stream blip* &optional (*start-bp* **(send stream :read-location)**) *for-context-type* *Function*

Echoes a presentation blip from the input buffer.

stream The input stream.

blip The presentation blip.

start-bp The position in the input buffer where the presentation blip begins.

for-context-type The input context on whose behalf the presentation blip is echoed. This affects the printing of the blip. For example, the Command Processor uses this option to ensure that echoed command names are preceded by colons when in the `'command-or-form` context.

For an overview of **dw:echo-presentation-blip** and related facilities: See the section "Presentation Input Blip Facilities".

(flavor:method :edges tv:menu) *(left-edge top-edge right-edge bottom-edge)* *Init Option*

Sets various position and size parameters. All the edge parameters are set relative to the outside of the superior window.

(flavor:method :edges tv:sheet) *(left-edge top-edge right-edge bottom-edge)* *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.

(flavor:method :edges tv:sheet) *Method*

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

(flavor:method :edges-from tv:menu) *source* *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.

(flavor:method :edges-from tv:essential-window) *source* *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.

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

encode-universal-time *seconds minutes hours date month year* &optional *timezone*
Function

Given a time in decoded time format, returns the corresponding universal time (plus or minus seconds since midnight, January 1, 1900 GMT).

seconds	An integer between 0 and 59.
minutes	An integer between 0 and 59.
hours	An integer between 0 and 23.
date	An integer between 1 and 31.
month	An integer between 1 and 12.
year	An integer indicating the year A.D. This can be shortened to an integer in the range 0-99, in which case the year is assumed equal to the integer modulo 100 and within 50 years of the current year; for example, in 1986, 36 is assumed to be 1936 and 35 to be 2035.
timezone	Hours west (postive) or east (negative) of GMT; it defaults to the timezone set for the site (for more information: See the section "Specifying a Time Zone for Your Site".) Any real number is allowed, but only numbers of the form *n* or *n.5* can correspond to actual timezones.

Examples:

```
(encode-universal-time 00 00 5 1 9 1986 5) => 2734941600

(encode-universal-time 00 00 5 1 9 86 5) => 2734941600
```

time:encode-universal-time *seconds minutes hours day month year* &optional *timezone*
Function

Converts the decoded time into Universal Time format and returns 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* can be absolute or relative to 1900 (that is, **84** and **1984** both work).

dw:erase-displayed-presentation *displayed-presentation window* &optional *recursive as-single-box (clear-inferiors* **t***)*
Function

Erases the specified *displayed-presentation* from *window* and removes it from the output history. Unless *recursive* is **t**, this causes a **:delete-displayed-presentation** message to be sent to *window*. If *displayed-presentation* has no inferiors or if *as-single-box* is **t**, the whole presentation is erased as a region. If *clear-inferiors* is **t** or if *as-single-box* is true, the inferiors of the presentation are erased as well.

error-output *Variable*

The value is a stream to which error messages should be sent. Normally, this is the same as ***standard-output***, but ***standard-output*** might be bound to a file and ***error-output*** left going to the terminal or a separate file of error messages.

```
(with-open-stream (outstream "myfile" :direction :output)
  (let ((*standard-output* outstream)
        (*error-output* outstream))  ;redirects *error-output* to myfile.lisp
    (fun-likely-to-signal-an-error)) ;capture any error messages in file
                                     ;end of let restores *error-output*, etc.
   ...                               ;more forms
)                                ;end of with-open-file closes file
```

zl:error-output *Variable*

In your new programs, we recommend that you use the variable ***error-output*** which is the Common Lisp equivalent of **zl:error-output**. See ***error-output***.

cp:execute-command *command-name* &rest *command-arguments* *Function*

Invokes a Command Processor command from within a program. See also **cp:build-command**, which **cp:execute-command** makes use of.

command-name Symbol or string naming the command to invoke; if a string, it must be in the command table to which **cp:*command-table*** is currently bound.

command-arguments Positional and keyword arguments to the named command.

Examples:

```
(cp:execute-command "show file" "test-data.text")

(cp:execute-command 'si:com-load-system "unifier"
                    :condition :always :automatic-answer t)
```

For an overview of **cp:execute-command** and related facilities, see the section "Managing the Command Processor".

(flavor:method :execute tv:menu) *item* *Method*

Extracts the value from a chosen item and returns it, or performs a side-effect, or both. It decides what to return based on the item's type. See the section "Types of Menu Items".

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".

: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.

(flavor:method :expose tv:menu) *Method*

Causes a menu to be exposed, that is, displayed on the screen.

(flavor:method :expose-p tv:essential-window) *t-or-nil* *Init Option*

If 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). Note that **:activate-p** and **:expose-p** are arguments in init-options which cannot be specified in the flavor's **:default-init-plist**.

(flavor:method :expose-p tv:menu) *t-or-nil* *Init Option*

When set to **t**, the window is immediately exposed. Otherwise, it must be explicitly exposed with an **:expose** message.

(flavor:method :extra-width tv:choose-variable-values) *arg* *Init Option*

When **:width** is not specified, 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.

(flavor:method :fill-p tv:menu) *t-or-nil* *Init Option*

Specifies whether to use filled format or columnar format.

(flavor:method :fill-p tv:menu) *Method*

Get the menu's fill mode. **t** is returned from **:fill-p** if the menu displays in filled form rather than columnar form. This message is a special case of the **:geometry:set-geometry** messages.

filling-output *(*&optional *stream* &key *:fill-column (:fill-characters* **'(#\Space)***) :after-line-break :after-line-break-initially-too :dont-snapshot-variables)* &body *body* *Function*

Binds local environment such that character output is filled; that is, **filling-output** makes sure that any output done within its body does not split "words" across lines.

"Words" are separated by spaces. When a line is broken to keep from wrapping past the end of a line, the line break is made at a space.

stream The output stream; the default is ***standard-output***.

:fill-column Specifiesthelengthoffilledlines.**:fill-column**isthe cursorpos of the right end of the line. It can be specified in one of the following ways:

integer If the output stream is one whose device units are smaller than single characters (pixels, for example) then if the integer is less than ten, it is interpreted as a number of character spaces; otherwise, if the number is greater than ten, it is interpreted as a number of device units. Note that the requirement that this number be an integer precludes the specification of spacing as a fraction of a character size: use the list method below to get fractional character spacing. (Ten is the number of pixels in a device character.)

string The spacing is the width of the string.

function The spacing is the amount of space the function would consume when called on the stream.

list The list is of the form (*number unit*), where *unit* is one of **:pixel** or **:character**. **'(3 :character)** is different from **(* 3 (send stream char-width))** or just **3**, in that the character width of whatever stream is really used to do the formatting is correctly used. **'(4 :pixel)** is different from just **4** in that it is not subject to the special interpretation of small numbers (< 10) normally applied.

If **:fill-column** is unspecified, line length is determined as follows: If the underlying stream supports the **:visible-cursorpos-limits** message, as do all Dynamic Windows, the right-hand cursorpos limit is used. Otherwise, if the underlying stream supports the **:inside-size** message, the inside size is used. If neither of the two preceding messages are supported, simple character counting is used, and lines are filled to 95 characters in width.

:fill-characters Specifies a character or characters at which to break lines.

```
(progn (terpri)
  (filling-output (t :fill-column '(30. :character)
                     :fill-characters '(#\-))
     (format t
        "this-is-a-test-of-the-emergency-broadcast-system")))
this-is-a-test-of-the-
emergency-broadcast-system
```

:after-line-break Specifies a string to be sent to *stream* after line breaks; the string appears at the beginning of each new line.

:after-line-break-initially-too Boolean option specifying whether the **:after-line-break** text is to be written to *stream* before doing *body*, that is, at the beginning of the first line; the default is **nil**.

:fill-on-spaces Obsolete. Use **:fill-characters** instead.

Example:

```
(defun filling-test ()
  (fresh-line)
  (filling-output (*standard-output*
                   :fill-column 420
                   :after-line-break "Repeat: "
                   :after-line-break-initially-too t)
    (loop for i from 1 to 100
          do
      (format t "Testing ~D " i))))
```

For an overview of **filling-output** and related facilities, see the section "Controlling Line Output".

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 is signalled; this is the default.

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

If *name* is already a command table object, it is returned. So this function is safe to use whenever you have a command table arglist.

For an overview of **cp:find-command-table** and related facilities: See the section "Managing the Command Processor".

dw:find-graph-node *redisplay-helper-stream id* &key (*key* **#'identity**) (*test* **#'eql**) *Function*

Searches for a node object given its symbol and the output stream on which it is to be displayed. The function returns the object if it finds it, **nil** otherwise.

Node objects are created with **formatting-graph-node**. See the function **formatting-graph-node**. Also, see that facility for an example.

redisplay-helper-stream The output stream for the node. This should be the same stream in use by **formatting-graph**.

id A unique identifier for the node. See the function **formatting-graph-node**.

:key Specifies a function applied to a node object before comparison with *id*. The default is the **identity** function.

:test Specifies the function used to compare node objects with the one specified by *id*. The default is **eql**.

For an overview of **dw:find-graph-node** and related facilities, see the section "Presenting Formatted Output".

dw:find-program-window *program-name* &rest *make-window-options* &key *(create-p* **t***) (activate-p* **t***) (selected-ok* **t***) reuse-test console superior program-state-variables* &allow-other-keys *Function*

Returns the window (frame) of a program (created via **dw:define-program-framework**). If no such window is found and **:create-p** is **nil**, then returns **nil**.

program-name The name of the program.

:create-p Specifies whether to create an instance of the program. Possible values are

t, the default: create an instance if one does not exist.
nil: do not create an instance if one does not exist.
:force: create an instance whether or not one exists.

:activate-p Specifies whether to activate an instance of the program window, if it is created.

:selected-ok Boolean option specifying whether to return the program window if it is the currently selected activity; the default is **t**.

:reuse-test Specifies a function to return a non-**nil** value if an existing program window of *program-name* can be reused. The function takes two arguments, the window found and the program name. Typically, a window would be reusable if deexposed or deactivated.

:console Specifies the console on which the program window is to be sought. **dw:find-program-window** returns either a previously selected window of the type *program-name*, or, if there is none such and **:selected-ok** is **t**, the currently selected window of type *program-name*.

:superior Specifies whether to make *superior* be the superior of the program window, if it is created.

:program-state-variables Specifies a list of initializations for the program's state variables. The list is of the form ((<*var-1*> <*val-1*>) (<*var-2*> <*val-2*>) ... (<*var-n*> <*val-n*>)).

If an instance of the program is created, its state variables are initialized according to this specification. If an instance already exists, its state variables are reset according to the specification.

Other keywords permitted are programmer-defined and system init options for the frame. If an instance of the program is created, it is initialized according to the keyword specifications.

For an overview of **dw:find-program-window** and related facilities, see the section "Defining Your Own Program Framework".

dw:find-and-select-program-window *name* &rest *options* *Function*

Returns the window (frame) of a program (created via **dw:define-program-framework**) and selects that window. See the function **dw:find-program-window**.

(flavor:method :finish-typeout si:interactive-stream) &optional *spacing erase?* *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.

tv:flashy-scrolling-mixin *Flavor*

Provides slow scrolling by moving the mouse through margin regions.

(flavor:method :flashy-scrolling-region tv:flashy-scrolling-mixin) *scrolling-region*
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.

(flavor:method :follow-p tv:blinker) *t-or-nil* *Init Option*

Sets whether the blinker follows the cursor; if this option is non-**nil**, it does. By default, this is **nil**, and so the blinker's position gets set explicitly.

zl:font-baseline *font* *Function*

The baseline of this font; a nonnegative integer.

zl:font-blinker-height *font* *Function*

The blinker height of the font.

zl:font-blinker-width *font* *Function*

The blinker width 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 space.

zl:font-char-width-table *font* *Function*

If **nil** 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-chars-exist-table *font* *Function*

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-indexing-table *font* *Function*

If **nil**, all characters are font-raster-width wide. 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 *x*-positions in the font raster.

zl:font-left-kern-table *font* *Function*

If **nil**, 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-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-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.

(flavor:method :force-rescan si:interactive-stream) *Method*

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.

format-cell *object printer* &key *(stream* ***standard-output****) align-x align-y* *Function*

Controls the printing of a table element within a **formatting-table** or **formatting-item-list** macro (see **formatting-table** for an example).

object The table element.

printer The function used to display each element. The function is passed two arguments, the *object* and the output stream. You can have the function write to the stream any information you want included in the table for that element. Typical functions to use are **princ**, **prin1**, and **write-string**.

:stream Specifies the output stream; the default is ***standard-output***.

:align-x Specifies how elements of a column should be aligned. The default **:left**, causes the elements to be flush-left in the column. The other possible values are **:right** (flush-right) and **:center** (centered).

:align-y Specifies how elements of a column should be aligned. The default **:bottom**, causes the bottoms of the elements to be aligned at the bottom of the cell. The other possible values are **:top**, and **:center**.

For an example of the use of **format-cell**: See the function **formatting-table**.

format-graph-from-root *root-object object-printer inferior-producer* &key *(stream* ***standard-output****) (dont-draw-duplicates* **nil***) (key* **#'identity***) (test* **#'eql***) (root-is-sequence* **nil***) (direction* **:after***) (default-drawing-mode* **:line***) (default-drawing-options* **nil***) (cutoff-depth* **nil***) (border* **'(:shape :rectangle)***) (orientation* **dw:*default-graph-orientation****) (balance-evenly* **dw:*default-graph-balance-evenly****) (row-spacing* **dw:*default-graph-row-spacing****) (within-row-spacing* **dw:*default-graph-within-row-spacing****) (column-spacing* **dw:*default-graph-column-spacing****) (within-column-spacing* **dw:*default-graph-within-column-spacing****) (branch-point* **dw:*default-graph-branch-point****) (allow-overlap* **dw:*default-graph-allow-overlap****) *Function*

Constructs and displays a tree graph.

root-object
The root element of the set, from which the tree can be derived.

object-printer A function used to display each tree node. The function is passed the object associated with that node and the stream on which to do output.

inferior-producer A function that knows how to extract the inferiors from a node object. It is passed one argument, the node in question.

:stream Specifies the output stream; the default is ***standard-output***.

:dont-draw-duplicates Boolean option specifying whether items that are duplicated in the tree are drawn only once (with all the reference lines drawn to the same object) or multiple times (once for each occur-

rence in the tree); the default is **nil**. (See the **:test** and **:key** options.)

:key Specifies the function used to extract the node object attribute used for duplicate comparison. The default is **identity**, that is, the object itself.

:test Specifies the test function used for duplicate detection. The default is **eql**.

:root-is-sequence Specifies that the value supplied for *root-object* is a sequence. Each element of the sequence becomes a separate root. (The resulting graphs might not themselves be separate if the **:dont-draw-duplicates** option is **t**.)

:direction
Specifies whether new nodes should be drawn above, below, left, or right of the current node. Possible values are **:after** and **:before**; the default is **:after**.

For `:orientation :horizontal`, **:after** means to the right, **:before** to the left. For `:orientation :vertical`, **:after** means below, **:before** means above.

:default-drawing-mode Specifies the drawing mode used to connect nodes of the tree. The default is **:line**, which connects the nodes with solid lines. Other modes are **:dashed-line**, **:arrow**, **:dashed-arrow**, **:reverse-arrow**, and **:reverse-dashed-arrow**.

:default-drawing-options Specifies one or more drawing function options that may override the default options. These are keyword options such as **:thickness**, **:gray-level**, and the like. The drawing options affect the drawing of the borders as well as the drawing of the connection lines.

See the section "Drawing Function Options".

:cutoff-depth Specifies how many levels of each branch of the tree should be explored. The default is **nil**, which specifies no cutoff.

:border Specifies the shape and thickness, in pixels, of the border drawn around each node. The default is `(:shape :rectangle :thickness 1)`. Other possible shapes are **:circle**, **:oval**, and **:diamond**. **nil** means no border.

Abbreviations:

Full Form	**Abbreviated Form**
`:border (:shape` *xxxx*`)`	`:border` *xxxx*
`:border (:thickness` *n*`)`	`:border` *n*

:orientation Specifies **:vertical** or **:horizontal** orientation for the "parent node to child node" direction of the graph display. The default **dw:*default-graph-orientation*** is initially set to **:vertical**.

:balance-evenly Specifies whether the subtrees of the tree should all be the same size (width or height, depending on **:orientation**), the size of the largest subtree. The default, **dw:*default-graph-balance-evenly***, is initially set to **nil**.

:row-spacing For **:vertical** orientation, specifies the spacing, in pixels, between rows of tree nodes; the default, **dw:*default-graph-row-spacing***, is initially set to `40`.

:within-row-spacing For **:vertical** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default, **dw:*default-graph-within-row-spacing***, is initially set to `20`.

:column-spacing For **:horizontal** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default, **dw:*default-graph-column-spacing*** is initially set to 30.

:within-column-spacing For **:horizontal** orientations, specifies the spacing, in pixels, between rows of tree nodes; the default, **dw:*default-graph-within-column-spacing***, is initially set to `10`.

:branch-point Specifies whether the lines connecting nodes should branch at the parent node (if set to **:at-parent**) or whether they should make a bend somewhere in the space between generations of nodes (if set to **:between-generations**). The default, **dw:*default-graph-branch-point***, is initially set to **:between-generations**. Branching between generations sometimes gives less overlap when not all links are to first generation children or when not all nodes are the same size.

Example:

```
(defun branch-point-test (&optional
                          (branch-point :between-generations))
  (fresh-line)
  (format-graph-from-root '((a bbbbbbbb) (ccc) (d e f))
                          #'prin1
                          #'(lambda
                              (node) (and (consp node) node))
                          :orientation :horizontal
                          :branch-point branch-point))
```

:allow-overlap Specifies whether or not subtrees of different superior nodes can overlap. **dw:*default-graph-allow-overlap***, the default that is intitally **t**, allows overlap. Use **:allow-overlap nil** when you do not need to minimize the amount of space consumed by the graph.

To see the effect of this keyword, evaluate the following form, then evaluate the second form with **:allow-overlap** set to **t** and then **nil**.

```
(defun component-flavors (flavor-name)
            (let* ((fl (flavor:find-flavor flavor-name)))
              (remove flavor-name
                      (cond
                        ((flavor::flavor-components-composed fl)
                         (flavor:flavor-all-components fl))
                        (t (flavor::compose-flavor-components
                             flavor-name))))))
(format-graph-from-root 'tv:minimum-window
                           #'(lambda (thing stream)
                               (present thing 'flavor:flavor
                                        :stream stream))
                           #'cl-user::component-flavors
                             :row-spacing 10
                             :within-row-spacing 10
                             :allow-overlap t)
```

By scrolling horizontally, you will see that in the first case the subtree for **tv:essential-window** overlaps with subtrees of **tv:essential-activate**.

Examples:

```
(defun format-graph-from-root-example-1 ()
  (fresh-line)
  ;; you wouldn't actually bother to write this let, but it makes
  ;; for a clearer example.
  (let ((root (pkg-find-package "hardcopy"))
        (print-function #'princ)
        (inferior-producer #'si:pkg-used-by-list))
    (format-graph-from-root root
                            print-function
                            inferior-producer)))

;;; Try executing the following Show Flavor Tree command first
;;; first on simple flavors like net:object and tv:minimum-window.
;;; More complex flavors let you exercise the horizontal scrolling
;;; capability of Dynamic Windows.

    (defun flavor-components (flavor-name)
      (flavor::flavor-local-components
        (flavor:find-flavor flavor-name)))

    (defun present-flavor (flavor-name
                           &optional (stream *standard-output*))
      (present flavor-name 'flavor:flavor-name :stream stream))
```

```
(cp:define-command (com-show-flavor-tree :command-table "Global")
    ((root-flavor-name 'flavor:flavor-name))
  (fresh-line)
  (format-graph-from-root root-flavor-name
            #'present-flavor #'flavor-components
            :dont-draw-duplicates t :orientation :horizontal))
```

For an overview of **format-graph-from-root** and related facilities, see the section "Presenting Formatted Output".

format-item-list *list* &key (*stream* ***standard-output***) *printer presentation-type (key #'identity)* (*fresh-line* **t**) (*return-at-end* **t**) (*order-columnwise* **t**) (*optimal-number-of-rows* **si:*optimal-number-of-rows***) (*additional-indentation* **2**) (*equalize-column-widths* **nil**) *max-width max-height* *Function*

Displays the elements of a list in a tabular format.

list The list of items to display.

:stream Specifies the output stream; the default is ***standard-output***.

:printer Specifies the function used to display the items; the default printer is **princ**. The function is passed two arguments, an item and the output stream.

This option and the **:presentation-type** option are mutually exclusive.

:presentation-type Specifies the presentation type used to display the items. Items are output via calls to **present** as this type. Items output as presentations can be used as mouse-sensitive input in the proper input context.

This option and the **:printer** option are mutually exclusive.

:key A function of one argument that will extract from an element the part to be printed in place of the whole element. Example:

```
(format-item-list sys:all-processes :key #'si:process-priority)
```

:fresh-line Boolean option specifying whether a **fresh-line** operation should be performed on the output stream before the table is displayed; the default is **t**.

:return-at-end Boolean option specifying whether a newline should be printed on the output stream when the table display is completed; the default is **t**.

:order-columnwise Boolean option specifying whether table items are ordered as a series of columns, the default, or rows.

```
Column-wise ordering:          Row-wise ordering:

     1  4  7                        1  2  3

     2  5  8                        4  5  6

     3  6  9                        7  8  9
```

:optimal-number-of-rows Specifies the number of rows in the table. If the number of rows specified is too small or too large to accommodate the list of items supplied, the appropriate number of rows closest to that specified is used.

:additional-indentation Specifies the number of characters by which the left margin of the table is indented; the default is 2.

:equalize-column-widths Boolean option specifying whether all columns have the same width (that of the widest column); the default is **nil**.

:max-width Specifies the maximum width, in pixels, of the table display.

:max-height Specifies the maximum height, in pixels, of the table display.

For an overview of **format-item-list** and related facilities, see the section "Formatting Tables".

dw:format-output-macro-continuation *(*&key *:name (:warn-p* **t***) :dont-snapshot-variables) var-list* &body *body* *Function*

Performs variable snapshotting. Use this macro in place of **zl:named-lambda** in defining macros that require snapshotting.

```
(defmacro my-formatting-macro ((&optional (stream '*standard-output*)
                                &key dont-snapshot-variables)
                               &body body)
  (dw:format-output-macro-default-stream stream)
  `(my-formatting-macro-helper-function
     ,stream
     (dw:format-output-macro-continuation (:name my-formatting-macro
                                           :dont-snapshot-variables
                                            dont-snapshot-variables)
                                          (,stream)
       . ,body)))

(defun my-formatting-macro-helper-function (continuation xstream)
  (funcall continuation xstream))
```

dw:format-output-macro-default-stream *var* *Function*

Defaults **t** or **nil** to ***standard-output***; for use by output macros. Example:

```
(defmacro my-formatting-macro ((&optional (stream '*standard-output*))
                               &body body)
  (dw:format-output-macro-default-stream stream)
  `(my-formatting-macro-helper-function
     ,stream
     (dw:named-continuation my-formatting-macro (,stream) . ,body)))

(defun my-formatting-macro-helper-function (continuation xstream)
  (funcall continuation xstream))
```

format-sequence-as-table-rows *sequence printer* &rest *options* &key (*stream* ***standard-output***) &allow-other-keys *Function*

Displays the elements in a sequence as a series of table rows.

sequence The sequence to be displayed. Each element of the sequence becomes one row in the resulting table.

printer The function used to display each element. The function is passed two arguments, an element of the sequence and an output stream. You can have the function write to the stream any information you want included in the table row for that item within appropriate **formatting-cell** forms.

options These are the same options as those of **formatting-table**, which see.

Note that each element becomes a row of cells, not a single cell. Use **formatting-item-list** if you want a single cell.

:stream Specifies the output stream; the default is ***standard-output***.

```
(defun format-sequence-as-table-rows-test ()
  (fresh-line)
  (format-sequence-as-table-rows
    sys:all-processes
    #'(lambda (process stream)
        (formatting-cell (stream)
          (present process 'si:process :stream
                   stream))
        (format-cell (si:process-whostate process)
                     #'princ
                     :stream stream))))
```

Additional keyword options available for this function are the same as those to **formatting-table**.

For an overview of **format-sequence-as-table-rows** and related facilities, see the section "Formatting Tables".

format-textual-list *sequence function* &key (*separator* ",") *finally if-two filled after-line-break conjunction* (*stream* ***standard-output***) *Function*

Outputs a sequence of items as a textual list; for example, "1 2 3 4" becomes "1, 2, 3, and 4":

```
(defun simple-list-formatter ()
  (fresh-line)
  (format-textual-list '(1 2 3 4) #'princ :conjunction "and"))
```

sequence The sequence to output.

function The function used to print sequence elements. This should be a function of two arguments: the object to print and the stream to send it to.

:separator Specifies the characters to use to separate elements of a textual list. The default is ", " (comma followed by a space).

:finally Specifies the separator to be used between the next-to-last and last elements of the list. The default is **nil**, meaning use the regular separator (specified by the **:separator** option). A typical value is `" and "`.

:if-two Specifies the separator to use when there are only two elements in the list. A typical value is `" and "`.

:filled Specifies whether the list should be "filled"; the default is **nil**.

A filled list is one containing Newline characters at appropriate points to prevent wrapping of output from right margin to left. Thus, specifying `:filled t` for a long list results in two or more separate lines of output — each of a length less than the width of the output window — rather than one long, wrapped line. Line breaks come between list elements, not within.

Another value permitted for this option is **:before**. This is like **t**, except that in the case where a line break occurs at a **:separator**, the break is made before the separator rather than after.

:after-line-break In `:filled t` mode, specifies the string to insert at the beginning of each new line. This is useful for specifying leading indentation, etc. (See the **:filled** option.)

:conjunction Specifies a string to use in the position between the last two elements. Typical values are `"and"` and `"or"`.

This option is similar to the **:finally** option, but does not affect the separator between the last two elements, unless only two elements

occur. That is, the **:conjunction** option takes care of the two-element case; the **:if-two** option is not necessary if you use this option.

:stream Specifies the output stream; the default is ***standard-output***.

For an overview of **format-textual-list** and related facilities, see the section "Formatting Textual Lists". Example:

```
(format-textual-list sys:area-list #'prin1 :filled t :after-line-break "  ")
```

formatting-cell *(*&optional *stream* &key *align-x align-y dont-snapshot-variables)* &body *body* *Function*

Binds local environment to control the printing of a table element within a **formatting-table** or **formatting-item-list** macro (see the latter facilities for examples).

stream The output stream; the default is ***standard-output***.

:align-x Specifies how elements of a column should be aligned. The default **:left**, causes the elements to be flush-left in the column. The other possible values are **:right** (flush-right) and **:center** (centered).

:align-y Specifies how elements of a column should be aligned. The default **:bottom**, causes the bottoms of the elements to be aligned at the bottom of the cell. The other possible values are **:top**, and **:center**.

:dont-snapshot-variables Specifies whether the free variables within the output macro should be snapshotted. The default of this option, **nil**, specifies that the free variables should be snapshotted. See the section "Snapshotting Variables".

For an overview of **formatting-cell** and related facilities, see the section "Formatting Tables".

formatting-column (&optional *stream* &rest *options*) &body *body* *Function*

Controls column layout within a **formatting-table** macro (see the latter facility for examples).

stream Specifies the output stream; the default is ***standard-output***.

options **:dont-snapshot-variables** is the only option available.

:dont-snapshot-variables Specifies whether the free variables within the output macro should be snapshotted. The default of this option, **nil**, specifies that the free variables should be snapshotted. See the section "Snapshotting Variables".

For an overview of **formatting-column** and related facilities, see the section "Formatting Tables".

formatting-column-headings (&optional *stream* &rest *options*) &body *forms*
Function

Controls the display of column headings within a **formatting-table** macro. If any form in *forms* is just a string, it is treated as

```
(formatting-cell (stream)
        (write-string (string)
                stream))
```

Example:

```
(defun table-with-column-headings
        (&optional (column-one-label "Number"))
  (fresh-line)
  (formatting-table ()
     (formatting-column-headings ()
       (formatting-cell () (write-string column-one-label))
       (formatting-cell () "Square"))
     (loop for i from 1 to 10
           as square = (* i i)
           do
       (formatting-row ()
         (formatting-cell ()
           (princ i))
         (formatting-cell ()
           (princ square))))))
```

stream Specifies the output stream; the default is ***standard-output***.

options The following option is available:

:underline-p Boolean option specifying whether column headings are underlined; the default is **nil**.

Note that it is an error to have more than one **formatting-column-headings** form within a **formatting-table** form. If you need two rows to display a column heading use a form like the following:

```
(defun foo ()
  (fresh-line)
  (formatting-table ()
    (formatting-column-headings () "Parsley" "Sage" "Rose-
mary" "Thyme")))
```

For an overview of **formatting-column-headings** and related facilities, see the section "Formatting Tables".

formatting-graph *(*&optional *stream* &key *(orientation* **dw:*default-graph-orientation***) *(inverted-center* **dw:*default-graph-inverted-center***) *(balance-evenly* **dw:*default-graph-balance-evenly***) *(row-spacing* **dw:*default-graph-row-spacing***) *(within-row-spacing* **dw:*default-graph-within-row-spacing***) *(column-spacing* **dw:*default-graph-column-spacing***) *(within-column-spacing* **dw:*default-graph-within-column-spacing***) *(branch-point* **dw:*default-graph-branch-point***) *(allow-overlap* **dw:*default-graph-allow-overlap***) *(default-drawing-mode* **:line***) default-drawing-options dont-snapshot-variables)* &body *body* *Function*

Binds the local environment to output a graph connecting node objects generated in the body of the macro. The node objects are created by the macro **formatting-graph-node**.

stream The output stream; the default is ***standard-output***.

:orientation Specifies **:vertical** or **:horizontal** orientation for the "parent node to child node" direction of the graph display. The default **dw:*default-graph-orientation*** is initially set to **:vertical**.

:inverted-center Specifies whether the lines connecting nodes are to diverge at the parent node (when set to **nil**) or at the first child node (when set to **t**). The default, **dw:*default-graph-inverted-center*** is initially set to **nil**.

:balance-evenly Specifies whether the subtrees of the tree should all be the same size (width or height, depending on **:orientation**), the size of the largest subtree. The default, **dw:*default-graph-balance-evenly***, is initially set to **nil**.

:row-spacing For **:vertical** orientation, specifies the spacing, in pixels, between rows of tree nodes; the default, **dw:*default-graph-row-spacing***, is initially set to `40`.

:within-row-spacing For **:vertical** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default, **dw:*default-graph-within-row-spacing***, is initially set to `20`.

:column-spacing For **:horizontal** orientation, specifies the spacing, in pixels, between columns of tree nodes; the default, **dw:*default-graph-column-spacing*** is initially set to 30.

:within-column-spacing For **:horizontal** orientation, specifies the spacing, in pixels, between rows of tree nodes; the default, **dw:*default-graph-within-column-spacing***, is initially set to `10`.

:branch-point Specifies whether the lines connecting nodes should branch at the parent node (if set to **:at-parent**) or whether they should make a bend somewhere in the space between generations of nodes (if set to **:between-generations**). The default, **dw:*default-graph-branch-point***, is initially set to **:between-generations**.

Branching between generations sometimes gives less overlap when not all links are to first generation children or when not all nodes are the same size.

Example:

```
(defun branch-point-test (&optional
                          (branch-point :between-generations))
  (fresh-line)
  (format-graph-from-root '((a bbbbbbbb) (ccc) (d e f))
                          #'prin1
                          #'(lambda
                              (node) (and (consp node) node))
                          :orientation :horizontal
                          :branch-point branch-point))
```

:allow-overlap Specifies whether or not subtrees of different superior nodes can overlap. **dw:*default-graph-allow-overlap***, the default that is intitally **t**, allows overlap. Use **:allow-overlap nil** when you do not need to minimize the amount of space consumed by the graph.

To see the effect of this keyword, evaluate the following form, then evaluate the second form with **:allow-overlap** set to **t** and then **nil**.

```
(defun component-flavors (flavor-name)
            (let* ((fl (flavor:find-flavor flavor-name)))
              (remove flavor-name
                      (cond
                        ((flavor::flavor-components-composed fl)
                         (flavor:flavor-all-components fl))
                        (t (flavor::compose-flavor-components
                             flavor-name))))))
(format-graph-from-root 'tv:minimum-window
                           #'(lambda (thing stream)
                               (present thing 'flavor:flavor
                                        :stream stream))
                           #'cl-user::component-flavors
                             :row-spacing 10
                             :within-row-spacing 10
                             :allow-overlap t)
```

By scrolling horizontally, you will see that in the first case the subtree for **tv:essential-window** overlaps with subtrees of **tv:essential-activate**.

:default-drawing-mode Specifies the drawing mode used to connect nodes of the tree. The default is **:line**, which connects the nodes with solid lines. Other modes are **:dashed-line**, **:arrow**, **:dashed-arrow**, **:reverse-arrow**, and **:reverse-dashed-arrow**.

:default-drawing-options Specifies one or more drawing function options that may override the default options. These are keyword options such as **:thickness**, **:gray-level**, and the like. The drawing options affect the drawing of the borders as well as the drawing of the connection lines.

See the section "Drawing Function Options".

:dont-snapshot-variables Specifies whether the free variables within the output macro should be snapshotted. The default of this option, **nil**, specifies that the free variables should be snapshotted. See the section "Snapshotting Variables".

Note: you must supply a form to create borders when you use **formatting-graph**, since you do not get them automatically as you would with **format-graph-from-root**.

Example:

```
(defun simple-graph (stream)
  (fresh-line stream)
  (formatting-graph (stream :orientation :horizontal)
    (let ((node-a (formatting-graph-node (stream)
                    (surrounding-output-with-border
                      (stream :shape :rectangle :thickness 3)
                      (princ 'a stream)))))
      (formatting-graph-node (stream :connections '(:right ,node-a)
                                     :drawing-mode :dashed-line)
        (surrounding-output-with-border
          (stream :shape :rectangle :thickness 3)
          (princ 'b stream))))))
```

If you want to try this example, compile it first. For a more complex example, see the function **formatting-graph-node**.

For an overview of **formatting-graph** and related facilities, see the section "Presenting Formatted Output".

formatting-graph-node (&optional *stream* &key *id connections* (*drawing-mode* **t**)) &body *body* *Function*

Binds local environment to create node objects for use by the **formatting-graph** macro. For an example, see the dictionary entry for the latter facility.

stream The output stream; the default is ***standard-output***.

:id Specifies a unique identifier for the node. A node identifier is used as an argument to **dw:find-graph-node**, see the function **dw:find-graph-node**.

:connections Specifies the connections between this node and one or more other nodes. The specification is a list in the form ((*key node-object-1*) (*key node-object-2*) ... (*key node-object-n*)), where key is one of **:left**, **:right**, **:above**, or **:below**; or one of **:before** or **:after**. If the orientation of the graph is vertical, **:before** is equivalent to **:above**, if the orientation is horizontal, **:before** is equivalent to **:left**. Analogously, **:after** is equivalent to **:below** (vertical) or **:right** (horizontal).

:drawing-mode Specifies the drawing mode used to connect this node with other nodes of the tree. This specification locally overrides the **:default-drawing-mode** specified by **formatting-graph**. Possible modes are **:line**, **:dashed-line**, **:arrow**, **:dashed-arrow**, **:reverse-arrow**, and **:reverse-dashed-arrow**.

:drawing-options Specifies any drawing options that are to override the default drawing options. The drawing options are **:thickness**, **:color**, **:gray-level**, and the like. If **:drawing-options** is **nil**, the default drawing options are used. See the function **formatting-graph**. See the function **format-graph-from-root**. See the section "Drawing Function Options".

Example:

```
(defun graph-1 (list unique-id-p)
  (let ((stream *standard-output*))
    (fresh-line stream)
    (formatting-graph (stream)
      (labels ((do-one (contents &rest connections)
                 (let ((node nil))
                   (when unique-id-p
                     (let ((already-there
                             (dw:find-graph-node stream contents)))
                       (when already-there
                         (dw:connect-graph-nodes
                           stream already-there connections)
                         (setq node already-there))))
                   (unless node
                     (setq node (formatting-graph-node
                                  (stream
                                    :id contents
                                    :connections connections)
                                  (surrounding-output-with-border
                                    (stream)
                                    (prin1 contents stream)))))
                   (when (consp contents)
                     (dolist (sublist contents)
                       (do-one sublist :after node))))))
        (do-one list)))))

(graph-1 '(a (b c) c) nil)
(graph-1 '(a (b c) c) t)
(graph-1 '(#1=(x y z) (w #1#) y) nil)
(graph-1 '(#1=(x y z) (w #1#) y) t)
```

For an overview of **formatting-graph-node** and related facilities, see the section "Presenting Formatted Output".

formatting-item-list (&optional *stream* &key *inter-row-spacing inter-column-spacing row-wise output-row-wise n-rows n-columns inside-width inside-height max-width max-height*) &body *body* *Function*

Binds local environment to output a list of items created in the body of the macro as a table.

Example:

```
(defun formatting-list-example ()
  (fresh-line)
  (formatting-item-list (t :n-columns 3)
    (loop for (p) in si:active-processes
          do
      (when p
        (formatting-cell ()
          (write-string (si:process-name p)))))))
```

stream The output stream; the default ***standard-output***.

:inter-row-spacing Specifiesthenumberofpixelsbetweenrows;thedefault is `0`.

:inter-column-spacing Determinestheamountofspacebetweencolumnsof the table; the default is the width of two spaces. **:inter-column-spacing** can be specified in one of the following ways:

integer If the output stream is one whose device units are smaller than single characters (pixels, for example) then if the integer is less than ten, it is interpreted as a number of character spaces; otherwise, if the number is greater than ten, it is interpreted as a number of device units. Note that the requirement that this number be an integer precludes the specification of spacing as a fraction of a character size: use the list method below to get fractional character spacing. (Ten is the number of pixels in a device character.)

string The spacing is the width of the string.

function The spacing is the amount of space the function would consume when called on the stream.

list The list is of the form (*number unit*), where *unit* is one of **:pixel** or **:character**. **'(3 :character)** is different from **(* 3 (send stream char-width))** or just **3**, in that the character width of whatever stream is really used to do the formatting is correctly used. **'(4 :pixel)** is different from just **4** in that it is not subject to the special interpretation of small numbers (< 10) normally applied.

:row-wise

Boolean option specifying that the table is built by rows, that is, that the each succeeding item in the list be placed in the same row, one column after the previous item (except for line breaks); the default is **t**. **nil** specifies that each item be placed in the same column, one row below the previous item.

:output-row-wise Booleanoptionspecifyingthatthetablebedisplayed row-by-row. The default is **nil**, causing the table to be displayed in the order in which it is constructed (See the **:row-wise** option.) If you specify **:row-wise nil :output-row-wise t**, the graph will be drawn faster, but the row cells will not be in the given order. For example, compare the results of running the following with the list `'(a b c d e f g h i j k l)`, setting **:row-wise** to **t** and then **nil**.

```
(defun what-order (l row-wise)
  (terpri)
  (stack-let ((things (make-array 100 :fill-pointer 0)))
    (formatting-item-list
      (t :row-wise row-wise :output-row-wise t)
      (dolist (x l)
        (formatting-cell ()
          (vector-push-extend x things)
          (princ x))))
    (coerce things 'list)))
```

:n-rows Specifies the number of rows the table should have.

:n-columns Specifies the number of columns the table should have.

:inside-width Specifies the exact width, in pixels, of the table display.

:inside-height Specifies the exact height, in pixels, of the table display.

:max-width Specifies the maximum width, in pixels, of the table display.

:max-height Specifies the maximum height, in pixels, of the table display.

For an overview of **formatting-item-list** and related facilities, see the section "Formatting Tables".

formatting-multiple-columns (&optional *stream* &key *number-of-columns*) &body *body* *Function*

Binds the local environment such that the lines of text generated by the body of the macro are output in a multiple-column format.

stream The output stream; the default is ***standard-output***.

:number-of-columns Specifiesthenumberofcolumnsintowhichtheitems are arranged. If this is unspecified, it uses as many columns as will fit, based on the stream's **:inside-size**.

Example:

```
(defun test-columns (&optional (stream *standard-output*))
  (loop for hundreds from 0 to 100 by 100 do
    (terpri stream)
    (formatting-multiple-columns (stream)
      (loop for j from 1 to 20 do
        (format stream "~d ~r~%" (+ j hundreds) (+ j hundreds))))))
```

Usage note: You should not use **formatting-table** within **formatting-multiple-columns**. Instead, use the **:multiple-columns** option to **formatting-table**, see the function **formatting-table**. Also, **formatting-table** works with redisplay, while **formatting-multiple-columns** does not redisplay.

For an overview of **formatting-multiple-columns** and related facilities, see the section "Formatting Tables".

formatting-row (&optional *stream* &rest *options*) &body *body* *Function*

Controls row layout within a **formatting-table** macro (see the latter facility for examples).

stream The output stream; the default is ***standard-output***.

options There is only one option:

:dont-snapshot-variables Specifies whether the free variables within the output macro should be snapshotted. The default of this option, **nil**, specifies that the free variables should be snapshotted. See the section "Snapshotting Variables".

For an overview of **formatting-row** and related facilities, see the section "Formatting Tables".

formatting-table (&optional *stream* &key *equalize-column-widths extend-width extend-height inter-row-spacing inter-column-spacing multiple-columns* (*multiple-column-inter-column-spacing* **dw::inter-column-spacing**) (*equalize-multiple-column-widths* **nil**) (*output-multiple-columns-row-wise* **nil**)) &body *body* *Function*

Binds local environment to output items in a tabular format.

This macro must be used in conjunction with at least two others. The first, **formatting-row** or **formatting-column**, controls whether items are output as table rows or table columns, respectively. The second, **formatting-cell** or **format-cell**, controls the printing of each item. Contrast the output of the following two examples:

```
(defun row-oriented-table-formatting ()
  (fresh-line)
  (formatting-table ()
    (loop for i from 1 to 10
              as square = (* i i)
          do
      (formatting-row ()
        (formatting-cell ()
          (princ i))
        (formatting-cell ()
          (princ square))))))

(defun column-oriented-table-formatting ()
  (fresh-line)
  (formatting-table ()
    (loop for i from 1 to 10
              as square = (* i i)
          do
      (formatting-column ()
        (format-cell i #'princ)
        (format-cell square #'princ)))))
```

stream The output stream; the default is ***standard-output***.

:equalize-column-widths Boolean option specifying whether all columns have the same width (that of the widest column); the default is **nil**.

:extend-width Specifies whether the spacing of table columns is extended; the default is **nil**. Alternative values are **t**, meaning make use of the full horizontal space available, or a number, indicating the number of pixels over which to extend the table. If the table is already wider than the requested total, then this option makes no difference.

:extend-height Specifies whether the spacing of table rows is extended; the default is **nil**. Alternative values are **t**, meaning make use of the full vertical space available, or a number, indicating the number of pixels over which to extend the table. If the table is already taller than the requested total, then this option makes no difference.

:inter-row-spacing Specifies the minimum number of pixels inserted between rows of the table; the default is 0. This value will be the actual number of pixels inserted unless overridden by the **:extend-height** option.

:inter-column-spacing Determines the amount of space inserted between columns of the table; the default is the width of a space character.

This value can be overriden by the **:extend-width** option. **:inter-column-spacing** can be specified in one of the following ways:

integer If the output stream is one whose device units are smaller than single characters (pixels, for example) then if the integer is less than ten, it is interpreted as a number of character spaces; otherwise, if the number is greater than ten, it is interpreted as a number of device units. Note that the requirement that this number be an integer precludes the specification of spacing as a fraction of a character size: use the list method below to get fractional character spacing. (Ten is the number of pixels in a device character.)

string The spacing is the width of the string.

function The spacing is the amount of space the function would consume when called on the stream.

list The list is of the form (*number unit*), where *unit* is one of **:pixel** or **:character**. **'(3 :character)** is different from **(* 3 (send stream char-width))** or just **3**, in that the character width of whatever stream is really used to do the formatting is correctly used. **'(4 :pixel)** is different from just **4** in that it is not subject to the special interpretation of small numbers (< 10) normally applied.

:multiple-columns Boolean option specifying that table rows be distributed among a series of two or more columns.

For example,

```
(defun multiple-column-example (multiple-columns)
  (fresh-line)
  (formatting-table
    (t :multiple-columns multiple-columns)
     (loop for i from 1 to 4
               as point = (+ 50 (* i i))
         do
       (formatting-row ()
          (formatting-cell ()
             (format t "Set Point #~D:    ~D    "
                     i point))))))
```

```
Set Point #1:    51
Set Point #2:    54
Set Point #3:    59
Set Point #4:    66
```

becomes

```
Set Point #1:    51      Set Point #3:    59
Set Point #2:    54      Set Point #4:    66
```

The arrangement of rows and columns generated is such that the number of columns is maximized, the number of rows is minimized, and the hole, if any, left in the lower right corner of the table is the smallest possible.

:multiple-column-inter-column-spacing Specifies the number of pixels to insert between columns in a multiple-column display (**:multiple-columns** option is **t**). It defaults to the value of the **:inter-column-spacing** option.

:equalize-multiple-column-widths Boolean option specifying whether all columns in a multiple column display (**:multiple-columns** option is **t**) have the same width (that of the widest column); the default is **nil**.

:output-multiple-columns-row-wise Boolean option specifying whether columns in a multiple-column display (**:multiple-columns** option is **t**) are displayed by outputting all the elements in one row followed by all in the next, and so on. The default is **nil**, meaning that the order of display is "column-wise": first all the elements in one column are output, then all the elements in the next, and so on.

The resulting display is the same no matter which way this flag is set; only the order in which the elements are displayed is changed. This affects the order in which calls to **formatting-row** are made within the body of the table-formatting macro. In the default case, calls are made in the order given; in the alternative case, call order is unpredictable. See the section "Snapshotting Variables".

For an overview of **formatting-table** and related facilities, see the section "Formatting Tables".

formatting-textual-list (&optional *stream* &key (*separator* ",") *finally if-two filled after-line-break conjunction*) &body *body* *Function*

Binds local environment to output a sequence of items as a textual list. This macro must be used in conjunction with the **formatting-textual-list-element** macro specifying the printing function.

Example:

```
(defun simple-list-formatting ()
  (fresh-line)
  (formatting-textual-list (t :conjunction "and")
    (loop for i from 1 to 4
          do
      (formatting-textual-list-element ()
        (princ i)))))
```

stream The output stream; the default is ***standard-output***.

:separator Specifies the characters to use to separate elements of a textual list. The default is ", " (comma followed by a space).

:finally Specifies the separator to be used between the next-to-last and last elements of the list. The default is **nil**, meaning use the regular separator (specified by the **:separator** option). A typical value is `" and "`.

:if-two Specifies the separator to use when there are only two elements in the list. A typical value is `" and "`.

:filled Specifies whether the list should be "filled"; the default is **nil**.

A filled list is one containing Newline characters at appropriate points to prevent wrapping of output from right margin to left. Thus, specifying `:filled t` for a long list results in two or more separate lines of output — each of a length less than the width of the output window — rather than one long, wrapped line. Line breaks come between list elements, not within.

Another value permitted for this option is **:before**. This is like **t**, except that in the case where a line break occurs at a **:separator**, the break is made before the separator rather than after.

:after-line-break In `:filled t` mode, specifies the string to insert at the beginning of each new line. This is useful for specifying leading indentation, etc. (See the **:filled** option.)

:conjunction Specifies a string to use in the position between the last two elements. Typical values are `"and"` and `"or"`.

This option is similar to the **:finally** option, but does not affect the separator between the last two elements, unless only two elements occur. That is, the **:conjunction** option takes care of the two-element case; the **:if-two** option is not necessary if you use this option.

For an overview of **formatting-textual-list** and related facilities, see the section "Formatting Textual Lists".

formatting-textual-list-element (&optional *stream*) &body *body* *Function*

Controls the printing of items output as textual list elements within a **formatting-textual-list** macro.

Example:

```
(formatting-textual-list (t :conjunction "and")
  (loop for i from 1 to 4 doing
    (formatting-textual-list-element () (princ i))))
```

stream The output stream; the default is ***standard-output***.

For an overview of **formatting-textual-list-element** and related facilities, see the section "Formatting Text".

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.

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.

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, ty-

peahead 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.

(flavor:method :fresh-line tv:sheet) *Method*

Gets 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.

: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.

(flavor:method :function tv:basic-choose-variable-values) *function* *Init Option*

Specifies the function called when the value of a variable is changed. See the section "The Optional Constraint Function". The default is **nil** (no function).

(flavor:method :function tv:choose-variable-values) *arg* *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".

tv:*function-keys* *Variable*

The value 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**. 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:function-text-scroll-window *Flavor*

Lets you provide a function to print the items in a text scroll window.

(flavor:method :geometry tv:menu) *list* *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".

(flavor:method :geometry 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.

(flavor:method :get-pane tv:basic-constraint-frame) *pane-name* *Method*

Returns the pane (the inferior window itself) that was named by the symbol *pane-name* in the **:panes** specification of this frame.

get-decoded-time *Function*

Returns the current time in decoded time format. The nine values returned are: second (0-59); minute (0-59); hour (0-23); date (1-31); month (1-12); year (A.D.); day-of-week (0[Monday]-6[Sunday]); a flag (**t** or **nil**) indicating whether daylight savings time is in effect; and the timezone (hours west of GMT).

The following example was run at 10:39:18 on Friday, 9/5/86 EDT:

```
(get-decoded-time) =>
18
39
10
5
9
1986
4
T
5
```

dw:get-program-pane *name* &key *(:if-does-not-exist* **:error***)* *Function*

Returns specified pane in a program frame created with **dw:define-program-framework**.

name The name of the pane as specified in the **:panes** option to **dw:define-program-framework**.

:if-does-not-exist Specifies the action to be taken if the specified frame is not found. Possible values are **:error** or **nil**.

For an overview of **dw:get-program-pane** and related facilities, see the section "Defining Your Own Program Framework".

time:get-time *Function*

Gets 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**. Note that you can also get the current time using **zl-user:get-decoded-time**.

get-universal-time *Function*

Returns the current time, in Universal Time form.

(flavor:method :gray-array-for-inferiors tv:gray-deexposed-inferiors-mixin) *gray 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".

(flavor:method :gray-array-for-inferiors 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".

(flavor:method :gray-array-for-unused-areas 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".

(flavor:method :gray-array-for-unused-areas tv:gray-unused-areas-mixin) *gray 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".

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".

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.

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.

(flavor:method :half-period tv:blinker) *n-60ths* *Init Option*

Sets the initial value of the half-period, that is, the time between xor's of the blinker. This defaults to **15**.

(flavor:method :half-period tv:blinker) *Method*

Examines the half-period of the blinker.

(flavor:method :edges-from tv:essential-window) *source* *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.

(flavor:method :handle-asynchronous-character si:interactive-stream) *character* *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.

(flavor:method :handle-mouse 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 out-

side the window. You can add an after daemon to turn off the blinker when the mouse leaves, for example.

dw:handler-applies-in-limited-context-p *context limiting-context-type* *Function*

Intended for use in the **:tester** forms of mouse handlers. It takes a *context* as provided in the **:context** keyword argument to a tester, and a presentation type to use as the *limiting-context-type*. It returns **t** if and only if the presentation type in the context is a subtype of the *limiting-context-type*. Because of caching, it is much faster than using **dw:presentation-subtypep** for this purpose, and it provides the convenience of extracting the presentation type from the context. See the function **dw:presentation-subtypep-cached**.

This function is typically used with translating handlers whose *to-presentation-type* is a subtype of **sys:expression**. For example, a translator from a `.bin` or `ibin` pathname to a `.lisp` pathname may be intended for use only in the pathname input context, not when any Lisp object is acceptable. By putting **dw::handler-applies-to-limited-context-p** in the **:tester** of the handler, the handler can be limited to contexts that are looking for some type of pathname.

Example:

```
(define-presentation-translator source-file-pathname
   (pathname pathname
             :tester ((ignore &key context)
                      (dw:handler-applies-in-limited-context-p
                        context 'pathname)))
   (pathname)
  (send (send (send pathname :generic-pathname) :get
              :qfasl-source-file-unique-id)
        :new-pathname :version :newest))
```

For an overview of **dw:handler-applies-in-limited-context-p** and related facilities: See the section "Programming the Mouse: Writing Mouse Handlers".

(flavor:method :height tv:ibeam-blinker) *n-pixels* *Init Option*

Sets the initial height of the blinker. It defaults to the *line-height* of the window.

(flavor:method :height tv:menu) *arg* *Init Option*

Height in pixels. Includes margins, as opposed to **:inside-height**, which does not include margins.

(flavor:method :height tv:rectangular-blinker) *n-pixels* *Init Option*

Sets 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.

(flavor:method :height tv:sheet) *outside-height* *Init Option*

Specifies the outside height of the window.

dw:help-program-check-for-help-wakeup *blip* *Function*

Checks whether the presentation type of *blip* is **window-wakeup-help**. If so, executes a **throw** to catch tag **return-from-read-command**; otherwise, calls **dw::default-window-wakeup-handler**.

dw:help-program-help *help-program stream string-so-far* &optional *(format-string "") &rest format-args* *Function*

Causes the string, "You are typing a command at the *Program-name Program*.", to be displayed when the user presses the HELP key, and, if used at the top level — that is, if supplied as the value of **:help** in a **user::define-program-framework** form — also displays "For accessing more detailed documentation about the *Program-name Program* itself, click on the *Program-name* command.", where in this last sentence *Program-name* is appropriately.

You can build your own help function using **dw:help-program-help** as in the following example, taken from the Graphic Editor.

```
(defmethod (graphic-editor-help graphic-editor) (stream string-so-far)
  (dw:help-program-help self stream string-so-far "
Click on a command from the menus at the right,
or select a shape to enter from the menu at the bottom.
"))
```

graphic-editor-help is supplied as the value for **:help** as described above. When the user presses the HELP key, the help program displays "You are typing a command at the Graphic-editor Program. Click on a command from the menus at the right, or select a shape to enter from the menu at the bottom."

The arguments for **dw:help-program-help** are:

help-program The name of the program the help facility is for.

stream The stream that *help-program* uses.

string-so-far A string to be displayed after the initial string mentioned above.

(flavor:method :highlighted-items tv:menu-highlighting-mixin) *items* *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**.

(flavor:method :highlighted-values tv:menu-highlighting-mixin) *Method*

Get the list of highlighted items. Refers to the items by value. For instance, if your item-list is an association list, with elements (*string* . *symbol*), this message uses *symbol*. This only works for menu items that can be executed without side-effects, not, for example, the **:eval** and **:funcall** kinds.See the section "**tv:multiple-menu-mixin** Messages".

tv:hollow-rectangular-blinker *Flavor*

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.

(flavor:method :home-cursor tv:sheet) *Method*

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

(flavor:method :home-down tv:sheet) *Method*

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

(flavor:method :hysteresis tv:hysteretic-window-mixin) *n-pixels* *Init Option*

Sets the initial value of the hysteresis, in pixels. It defaults to **25**. (decimal).

(flavor:method :hysteresis tv:hysteretic-window-mixin) *Method*

Examines the hysteresis of the window, in pixels.

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 will not 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.)

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.

indenting-output (*stream indentation*) &body *body* *Function*

Binds local environment to control the insertion of spaces or other characters at the beginning of each newline output to a stream.

stream The output stream. As a special case, **t** and **nil** are abbreviations for ***standard-output***.

indentation What gets inserted at the beginning of each line output to the stream. Four possibilities exist:

integer If the output stream is one whose device units are smaller than single characters (pixels, for example) then if the integer is less than ten, it is interpreted as a number of character spaces; otherwise, if the number is greater than ten, it is interpreted as a number of device units. Note that the requirement that this number be an integer precludes the specification of spacing as a fraction of a character size: use the list method below to get fractional character spacing. (Ten is the number of pixels in a device character.)

string The spacing is the width of the string.

function The spacing is the amount of space the function would consume when called on the stream.

list The list is of the form (*number unit*), where *unit* is one of **:pixel** or **:character**. **'(3 :character)** is different from **(* 3 (send stream char-width))** or just **3**, in that the character width of whatever stream is really used to do the formatting is correctly used. **'(4 :pixel)** is different from just **4** in that it is not subject to the special interpretation of small numbers (< 10) normally applied.

The function should be a function to print a string. It receives one argument, the output stream. Because the system calls this function with other streams, for example, with a dummy stream to determine the space requirements of the output, it should output something of the same size each time it is called.

You should begin the body with `(terpri` *stream*`)`, or equivalent, to position the stream to the indentation initially. That is, it is perfectly valid to indent only subsequent lines.

Examples:

```
(defun simple-indenter ()
  (indenting-output (t 10)
    (loop for i from 1 to 5
          do
      (terpri)
      (format t "This is indented line ~d" i))))
```

The **trace** special form uses **indenting-output** as follows:

```
(flet ((indent (stream)
         (loop for n from 1 below trace-level do
           (write-char (if ... #\| #\sp) stream))))
  (indenting-output (*trace-output* #'indent)
    (terpri *trace-output*)
    ...))
```

For an overview of **indenting-output** and related facilities, see the section "Controlling Line Output". For a related facility, see the function **sys:with-indentation**.

dw:independently-redisplayable-format *stream format-string* &rest *format-args*
Function

Outputs a formatted string such that each format argument is independently redisplayable. (See the function **dw:with-redisplayable-output**.)

stream The output stream; the default is ***standard-output***.

format-string The format-control string. (See the function **format**.)

format-args The format arguments.

The *format-string* is parsed at compile time, resulting in a series of calls to **dw:redisplayable-format** or **format**. Some restrictions result:

- *stream* may not be **nil**, although **format** would permit it.
- **format** commands that need all the **format** arguments, like conditionals, iterations, or gotos, cannot be used.

dw:independently-redisplayable-format is one of a number of facilities used to do incremental redisplay. For examples, see the file SYS:EXAMPLES;INCREMENTAL-REDISPLAY.LISP.

For an overview of **dw:independently-redisplayable-format** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

:inferior-select *sheet* *Message*

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 partici-

pate 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.

(flavor:method :init tv:sheet) *init-plist* *Method*

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

: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**.

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 might want to call it yourself to correct the time if it appears to be inaccurate or wrong. See the function **time:set-local-time**.

(flavor:method :input-editor si:interactive-stream) *read-function* &rest *read-args* *Method*

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

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

Input Editing Options

: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.

: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-1-1
            (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.

: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".

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

: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.

: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".

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

: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.

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

: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.

: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**.

: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**.

: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**.

: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.

: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.

: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.

: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))
```

: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".

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.

: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**.

: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.

: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".

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".

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.

: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.

: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**.

:input-wait Specifies a function testing for some condition while in the input-wait state. If this condition occurs, the **:input-wait-handler** is invoked.

: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.

(flavor:method :insert-char tv:sheet) &optional (*char-count* **1**) (*unit* **':character**) *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.

(flavor:method :insert-item tv:text-scroll-window) *item-no new-item* *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.

(flavor:method :insert-line tv:sheet) &optional (*line-count* **1**) (*unit* **':character**) *Method*

Takes the line containing the cursor and all the lines below it, and moves them down by *line-count* units. A blank space (whose length is variable) 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.

(flavor:method :insert-string tv:sheet) *string* &optional (*start* **0**) (*end* **nil**) (*type-too* **t**) *Method*

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

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

If *type-too* is specified as **nil**, suppress the actual display of the string, and the space that was opened is left blank. **:insert-string**, in this case, uses **:insert-char** to actually make the space.

(flavor:method :inside-edges tv:sheet) *Method*

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

(flavor:method :inside-height tv:menu) *arg* *Init Option*

Specifies the inside height specified in pixels. Excludes margins.

(flavor:method :inside-height tv:sheet) *inside-height* *Init Option*

Specifies the inside height of the window.

(flavor:method :inside-size tv:menu) *(inside-width inside-height)* *Init Option*

Specifies the inside size parameters specified in pixels.

(flavor:method :inside-size tv:sheet) *Method*

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

(flavor:method :inside-size tv:sheet) *(inside-width inside-height)* *Init Option*

Specifies the inside width and height of the window.

(flavor:method :inside-width tv:menu) *arg* *Init Option*

Specifies the inside width of window in pixels.

(flavor:method :inside-width tv:sheet) *inside-width* *Init Option*

Specifies the inside width of the window.

cp:install-command *command-table command-symbol* &optional *command-name* *Function*

Installs a Command Processor command into a command table.

command-table Name (symbol or string) of the command table receiving the new command. If it does not already exist, a command table will be created.

command-symbol The command to install, a symbol.

command-name The name of the command, a string. The default is the value supplied for the **:name** keyword argument in the command definition.

For an overview of **cp:install-command** and related facilities: See the section "Managing the Command Processor".

(flavor:method :integral-p tv:sheet) *t-or-nil* *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.

:interactive *Message*

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.

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.

dw:invalidate-type-handler-tables *Function*

Invalidates presentation mouse handler lookup tables. The next time the tables are accessed, they are updated by this function to reflect any changes in the type hierarchy affecting handler applicability.

This function gets called by the system whenever a new presentation type is defined. You need to call it directly only if your presentation-type definitions change dynamically at runtime, for example, through a global variable in the **:abbreviation-for** option. However, because the updating of the handler lookup tables does not occur in real time, you should avoid such usage.

For an overview of **dw:invalidate-type-handler-tables** and related facilities: See the section "Programming the Mouse: Writing Mouse Handlers".

(flavor:method :io-buffer tv:choose-variable-values-window) *buf* *Init Option*

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.

(flavor:method :io-buffer tv:command-menu) *buf* *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.

(flavor:method :io-buffer tv:command-menu) *Method*

gets the I/O buffer to which a command menu sends a command when an item is chosen.

(flavor:method :io-buffer tv:constraint-frame-with-shared-io-buffer) *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.

(flavor:method :item tv:basic-mouse-sensitive-items) *type item* &rest *format-args* *Method*

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**.

(flavor:method :item si:interactive-stream) *type item* &rest *format-args* *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 **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*.

(flavor:method :item tv:mouse-sensitive-text-scroll-window-without-click) *item type* &optional (*function* **#'prin1**) &rest *print-args* *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.

(flavor:method :item-list tv:menu) *list* *Init Option*

Initializes the item list for a menu. See the section "Types of Menu Items".

(flavor:method :item-list-pointer tv:dynamic-...-menu) *form* *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.)

(flavor:method :item-type-alist tv:basic-mouse-sensitive-items) *alist* *Init Option*

Remembers *alist* as the set of item types allowed in this window. *alist* should be created by **tv:add-typeout-item-type**.

(flavor:method :item-value tv:text-scroll-window) *item-no* *Method*

Returns the item whose number is *item-no*.

(flavor:method :items 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.

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, including **nil**, 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-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-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-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**.

tv:key-state *key-name* *Function*

Returns **t** if the keyboard key named *key-name* is currently pressed, **nil** if it is not.

key-name may be the symbolic name of a modifier key, from the table below, or a character object. 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 pressed 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
```

(flavor:method :label tv:choose-variable-values) *string* *Init Option*

The argument is a string that is the label displayed at the top of the window. The default is "Choose Variable Values".

(flavor:method :label tv:label-mixin) *specification* *Init Option*

Sets 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.

(flavor:method :label tv:menu) *specification* *Init Option*

Specifies the menu's label. The specification is usually a list in the following form:

```
(:string "Foo" :style character-style-specification)
```

tv:label-mixin *Flavor*

Creates the labels in the corners of windows that you often see when using Genera. 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.

(flavor:method :label-size tv:label-mixin) *Method*

Returns the width and height of the area occupied by the label.

cp:*last-command-values* *Variable*

List of values returned by the most recently executed Command Processor command.

For an overview **cp:*last-command-values*** and related facilities, see the section "Managing the Command Processor".

(flavor:method :last-item tv:text-scroll-window) *Method*

Returns the last item in the item list.

time:leap-year-p *year* *Function*

Returns **t** if *year* is a leap year; otherwise returns **nil**. *year* can be absolute or relative to 1900 (that is, **84** and **1984** both work).

(flavor:method :left tv:menu) *arg* *Init Option*

Specifies the left edge of the menu, defined in pixels relative to the outside of the superior window.

(flavor:method :left tv:sheet) *left-edge* *Init Option*

Specifies the x-coordinate of the left edge of the window, relative to the superior's coordinate system.

(flavor:method :left-margin-size tv:sheet) *Method*

Returns the left margin size of the window in pixels.

(flavor:method :line-in si:interactive-stream) &optional *leader* *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 is not 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.

(flavor:method :line-out tv:sheet) *string* &optional *(start* **0***) (end* **nil***)* *Method*

Does 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.

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**.

(flavor:method :list-tyi si:interactive-stream) *Method*

Like **:any-tyi**, except that it returns only blips, never 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).

(flavor:method :listen si:interactive-stream) *Method*

Returns **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.

(flavor:method :listen tv:stream-mixin) *Method*

Returns **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 the characters that have been typed in.

tv:make-blinker *window* &optional *(flavor* **'tv:rectangular-blinker***)* &rest *options* *Function*

Creates and returns a new blinker. The new blinker is associated with the given *window*, and is of the given *flavor*. 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. See the section "General Blinker Operations" for other operations on blinkers. See the section "Specialized Blinkers" for a list of other useful flavors of blinkers.

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 (strings or objects) 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**. Just because accelerators are defined does not mean that non-accelerated command reading is prohibited.

: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 the newly specified init options in the call to **cp:make-command-table**.

:error An error is signalled.

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 "Managing the Command Processor".

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
```

tv:make-sheet-bit-array *window width height* &rest *make-array-options* *Function*

Creates a two-dimensional bit-array useful for bitblting to and from windows. It makes an array whose first dimension is at least *width* but is rounded up so that **bitblt**'s restriction regarding multiples of 32. is met, whose second dimension is *height*, 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.

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

Creates, initializes, and returns 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. Note that **:activate-p** and **:expose-p** are keyword arguments which cannot be specified in the flavor's **:default-init-plist**.

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.

dw:margin-borders *Flavor*

Provides Dynamic Windows with a four-sided, black (or draw-alu color) border.

dw:margin-borders accepts the following init option:

:thickness Specifies the thickness, in pixels, of the border; the default is 1.

For an overview of **dw:margin-borders** and related facilities, see the section "Window Substrate Facilities".

tv:margin-choice-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". 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)
```

(flavor:method :margin-choices tv:choose-variable-values) *list* *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".

(flavor:method :margin-choices tv:choose-variable-values-window) *choice-list* *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". Note that specifying **nil** for this option suppresses the margin-choices entirely.

(flavor:method :margin-choices tv:margin-choice-mixin) *choices* *Init Option*

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.

dw:margin-drop-shadow-borders *Flavor*

Provides Dynamic Windows with a black (normal video) border shadowed on its right and bottom margins by a gray border.

dw:margin-drop-shadow-borders accepts the following init options:

:non-shadow-thickness Specifies the thickness, in pixels, of the black border; the default is `3`.

:outside-margin Specifies the thickness, in pixels, of whitespace surrounding the shadowed and non-shadowed borders of the box; the default is `0`.

:shadow-thickness Specifies the thickness, in pixels, of the gray margins on the right and bottom edges of the window; the default is `8`.

For an overview of **dw:margin-drop-shadow-borders** and related facilities, see the section "Window Substrate Facilities".

dw:margin-label *Flavor*

Specifies Dynamic Window labels.

dw:margin-label accepts the following init options:

:background-gray Specifies a binary array to use as a background pattern for the label.

You can provide your own array via the **tv:make-binary-array** function — for an example, see the dictionary entry for **graphics:draw-pattern** — or use one of the standard, background-gray patterns: **stipples:25%-gray**, **stipples:33%-gray**, **stipples:50%-gray**, or **stipples:75%-gray**.

Note that the specification is for an array object, not its symbol.

:box Specifies whether to enclose the label in a box; the default is **nil**. Other permissible values are **:inside** and **:outside**. If you wish to box the label within a just-specified border, use **:inside**; if you wish to box the label outside of a border about to be specified, use **:outside**.

:box-thickness Specifies the thickness, in pixels, of the line used to draw a box around the label when the **:box** init option is non-**nil**.

:centered-p Boolean option specifying whether the label is left-right centered. The default is **nil**, causing the label to appear on the left side of the margin.

:extend-box-p Boolean option specifying whether the box drawn (when the **:box** option is non-**nil**) extends the full length of the margin or is limited to the length of the label; the default is **t**.

:margin Specifies the margin, **:top** or **:bottom**, on which the label appears; the default is **:bottom**.

:string Specifies the label string. The default string is derived from the name of the window flavor used to make the window instance.

:style Specifies the character style used for writing the label string. The default value is the character-style default for the screen.

:character-style Specifies the character style used for writing the label string. The default value is the character-style default for the screen. **:character-style** and **:style** are different names for the same option (for back-compatibility).

After a window instance is created, you can change its label by using **(flavor:method :set-label dw:margin-mixin)**.

For an overview of **dw:margin-label** and related facilities: See the section "Window Substrate Facilities".

dw:margin-ragged-borders *Flavor*

Provides Dynamic Windows with a ragged (wavy) border to indicate that more output can be viewed by scrolling in the direction indicated. The border is only ragged when there is in fact more output to be viewed; otherwise, it is straight.

dw:margin-ragged-borders accepts the following init options:

:horizontal-too Boolean option specifying whether to provide ragged left and right margins in addition to ragged top and bottom margins; the default is **t**.

:thickness Specifies the thickness, in pixels, of the border; the default is 2.

For an overview of **dw:margin-ragged-borders** and related facilities, see the section "Window Substrate Facilities".

dw:margin-scroll-bar *Flavor*

Provides an "elevator" scroll bar to a Dynamic Window.

dw:margin-scroll-bar accepts the following init options:

:elevator-thickness Specifies the overall width, in pixels, of the scroll bar; the default is `10`.

:history-noun Specifies a string appearing as part of the mouse documentation when the mouse cursor is moved over the scroll bar. The specified string is substituted for the word "history", the default value for this option.

For example, if you specified `"graphic output"`, when the window was created and the user positioned the mouse cursor in the middle of the scroll bar, the mouse documentation line would read, "... Middle: Move to 50% of graphic output..." instead of "... Middle: Move to 50% of history..."

:margin Specifies the margin — **:left**, **:right**, **:top**, or **:bottom** — on which the scroll bar appears; the default is **:left**.

:shaft-whitespace-thickness Specifies the thickness, in pixels, of additional whitespace (normal video) inserted on each side of the scroll bar between it and the neighboring component. The default is `0`, causing the whitespace to be one-pixel-wide on both sides.

:visibility
Specifies when the scroll bar is visible. Three values are permitted:

:normal The scroll bar appears when the flavor is instantiated and remains visible regardless of whether it is needed. This is the default.

:if-requested An empty elevator shaft appears when the flavor is instantiated and after each new output operation to the window. If the user moves the mouse cursor into the scroll bar area, the standard cross-hatched pattern is drawn in the shaft and the scroll bar becomes normally active.

:if-needed The scroll bar does not appear until the output exceeds the window space available for displaying it, that is, until the need for scrolling arises; thereafter it remains visible and normally active. The space needed for drawing the scroll bar is reserved by whitespace (normal video) until the scroll bar appears.

For an overview of **dw:margin-scroll-bar** and related facilities: See the section "Window Substrate Facilities".

tv:margin-scroll-mixin *Flavor*

Provides scrolling by clicking on margin regions.

(flavor:method :margin-scroll-regions tv:margin-scroll-mixin) *regions 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 four 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))
```

tv:margin-scrolling-with-flashy-scrolling-mixin *Flavor*

Provides *More above* and *More below* style window scrolling for a text scroll window.

(flavor:method :margin-space 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)

(flavor:method :margin-space 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.

tv:margin-space-mixin *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.

(flavor:method :margins tv:sheet) *Method*

Returns four values: the sizes of the left, top, right, and bottom margins, respectively.

dw:margin-white-borders *Flavor*

Provides Dynamic Windows with a four-sided, white (or erase-alu color) border.

dw:margin-white-borders accepts the following init option:

:thickness Specifies the thickness, in pixels, of the border; the default is 1.

For an overview of **dw:margin-white-borders** and related facilities, see the section "Window Substrate Facilities".

dw:margin-whitespace *Flavor*

Provides Dynamic Windows with whitespace (or erase-alu color) on a margin.

dw:margin-whitespace accepts the following init options. Note that the **:margin** must be specified or else an error results.

:margin Specifies the margin, one of **:left**, **:right**, **:top**, or **:bottom**.

:thickness Specifies the thickness, in pixels, of the border; the default is 1.

For an overview of **dw:margin-whitespace** and related facilities, see the section "Window Substrate Facilities".

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.

dw:menu-choose *item-list* &key (*prompt* **nil**) (*default* **nil**) (*presentation-type* **nil**) (*printer* **nil**) (*near-mode* **'(:mouse)**) (*superior* **tv:mouse-sheet**) (*center-p* **dw::*default-menu-center-p***) (*character-style* **'(:jess :roman :large)**) (*momentary-p* **t**) (*temporary-p* **dw::momentary-p**) (*alias-for-selected-windows* **nil**) (*minimum-width* **nil**) (*minimum-height* **nil**) *Function*

Constructs a menu from a list of items and returns the value associated with the selected item; also returned are the item and the mouse character that was used to select it.

item-list A quoted list of items to include in the menu display. A menu item can have various forms: See the section "The Form of a Menu Item".

If you wish to control the mouse documentation associated with an item, use the "general list" form and include the **:documentation** menu-item option.

Example:

```
(dw:menu-choose '(("First Choice" :value 1
                   :documentation "Mouse Doc One")
                  ("Second Choice" :value 2
                   :documentation "Mouse Doc Two")))
```

The other available menu-item option is **:style**, specifying the character style of the individual item. This contrasts with the **:character-style** option to **dw:menu-choose** as a whole, which specifies the style for all items. If both are specified, the locally specified **:style** prevails. For an example, see the **:character-style** option; descriptions of this and other options to **dw:menu-choose** follow.

:prompt Specifies a string to use as a title for the menu. The menu title appears at the top of the menu.

:default Specifies an item in the *item-list* that is the currently selected (highlighted) item when the menu is first displayed.

If the item list is a simple one containing symbols, then specify the default item by its symbol as shown below:

Simple example:

```
(dw:menu-choose '(a b c) :default 'a)
```

If the items are themselves lists, then supply the default item in a fashion similar to that shown in the following example. (That is, be sure to specify an *item*, not the item's value. The default you specify must be **eq** to the item.)

```
(setq item-list '(("One" :value 1) ("Two" :value 2)))
(dw:menu-choose item-list :default (first item-list))
```

:presentation-type Specifies the presentation type of the items presented in the menu. This results in the printer for that presentation type being used to display the items in the menu.

Because each item (element) in *item-list* is passed to the presentation type's printer, using this option is, in general, only appropriate when *item-list* is a simple list of objects (as opposed to a "general list"). In this case, you might also consider using **dw:menu-choose-from-set** rather than **dw:menu-choose**.

:printer Specifies a function of two arguments for printing the menu items. The arguments are an object — one element of the *item-list* — and a stream. If both this option and the **:presentation-type** option are specified, the printer used is the one specified by this option, not that of the presentation type.

Example:

```
(dw:menu-choose '((a :value test)
                  (b :value 17))
                :printer
                 #'(lambda (object stream)
                     (format stream
                        "xxx~Axxx" (car object))))
```

The example function pops up a menu displaying two choices, "xxxAxxx" and "xxxBxxx". Clicking on "xxxAxxx" returns TEST; clicking on "xxxBxxx" returns 17.

:near-mode Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)**.

:superior Specifies the window that is the superior of the menu window; the default is **tv:mouse-sheet**.

:center-p Boolean option specifying whether items displayed in the menu are centered, left to right. The default is **nil**, which causes the items to be flush left.

:character-style Specifies the character style for display of menu items. The default is (:jess :roman :large).

If the **:style** option for an individual item is specified, this locally overrides the **:character-style** specified for the menu as a whole, but does not affect other items. The example below illustrates this:

```
(dw:menu-choose '(("First Choice" :value 1
                      :style (nil nil :normal))
                  ("Second Choice" :value 2))
                :character-style
                  '(:serif :bold :very-large))
```

:momentary-p Boolean option specifying whether the menu is momentary or temporary; the default is momentary. If you wish to make the menu temporary, supply a value of **nil** to this option and **t** to the **:temporary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:temporary-p Boolean option specifying whether the menu is temporary or momentary; the default is momentary. If you wish to make the menu temporary, supply a value of **t** to this option and **nil** to the **:momentary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:alias-for-selected-windows Specifies the activity to be returned by the menu in response to the **:alias-for-selected-windows** window message. The default value, **nil**, specifies that the menu should return itself.

Using **:alias-for-selected-windows** allows you to join the menu to another activity, as opposed to having the menu be an activity in its own right. This causes `Function S` to select a different activity when invoked while the menu is popped up, instead of selecting the window under the menu.

For example, if you type

```
(dw:menu-choose-from-set '(a b c) 'symbol
    :alias-for-selected-windows *terminal-io*)
```

from the Lisp Listener, and issue a `FUNCTION S` command while the menu is present, you will select the window that was selected just before the Lisp Listener. If you had taken the default for **:alias-for-selected-windows**, `FUNCTION S` over the window would have selected the Lisp Listener.

:minimum-width Specifies the minimum width of the menu in pixel units. The default, **nil**, causes the width of the window to be only as wide as is necessary to contain the menu items.

:minimum-height Specifies the minimum height of the menu in pixel units. The default, **nil**, causes the width of the window to be only as high as is necessary to contain the menu items.

For an overview of **dw:menu-choose** and related facilities: See the section "Using Presentation Types for Input".

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". 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.

dw:menu-choose-from-copy-of-window-contents *window presentation-type* &key *:prompt :default (:near-mode* **':(:mouse)***) (:superior* **(tv:mouse-default-superior dw::window)***) (:momentary-p* **t***) (:temporary-p* **dw::momentary-p***)* *Function*

Displays a pop-up copy of *window* and chooses an item of *presentation-type* from it. Similar to **dw:menu-choose-from-set**.

window A window, typically, a menu pane of a program frame.

presentation-type The presentation type used to present the objects.

Here is an example taken from the graphic editor:

```
(define-presentation-to-command-translator pop-up-shapes-menu
   (graphic-input:grid-output
     :blank-area t :menu ()
     :gesture :pop-up-shapes-menu       ;:menu
     :documentation "Shapes menu")
   (ignore &key window x y)
  (when (drawing-window-point-p window x y)
    (if dw:*inside-handler-test-phase*
        `(com-create-entity)
        (let* ((frame (send window :superior))
               (shape (dw:menu-choose-from-copy-of-window-contents
                        (send frame :get-pane 'shapes-menu) 'entity-shape
                        :prompt "Create a shape" :superior frame)))
          (if (null shape)
              (signal 'sys:abort)
              `(com-create-entity ,shape))))))
```

:prompt Specifies a string to use as a title for the menu. The menu title appears at the top of the menu.

:default Specifies an item to be the currently selected (highlighted) item when the menu is first displayed.

Examples:

```
(dw:menu-choose-from-set '(a b c) 'symbol :default 'a)

(setq item-list '("One" "Two"))
(dw:menu-choose-from-set item-list 'string
                         :default (first item-list))
```

:near-mode Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method

(flavor:method :expose-near tv:essential-set-edges).

:superior Specifies the window that is the superior of the menu window; the default is **tv:mouse-sheet**.

:momentary-p Boolean option specifying whether the menu is momentary or temporary; the default is momentary. If you wish to make the menu temporary, supply a value of **nil** to this option and **t** to the **:temporary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:temporary-p Boolean option specifying whether the menu is temporary or momentary; the default is momentary. If you wish to make the menu temporary, supply a value of **t** to this option and **nil** to the **:momentary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

dw:menu-choose-from-copy-of-window-contents *window presentation-type* &key *:prompt :default (:near-mode* **'(:mouse)***) (:superior* **(tv:mouse-default-superior dw::window)***) (:momentary-p* **t***) (:temporary-p* **dw::momentary-p***)* *Function*

Displays a pop-up copy of *window* and chooses an item of *presentation-type* from it. Similar to **dw:menu-choose-from-set**.

window A window, typically, a menu pane of a program frame.

presentation-type The presentation type used to present the objects.

Here is an example taken from the graphic editor:

```
(define-presentation-to-command-translator pop-up-shapes-menu
   (graphic-input:grid-output
     :blank-area t :menu ()
     :gesture :pop-up-shapes-menu       ;:menu
     :documentation "Shapes menu")
   (ignore &key window x y)
  (when (drawing-window-point-p window x y)
    (if dw:*inside-handler-test-phase*
        `(com-create-entity)
        (let* ((frame (send window :superior))
               (shape (dw:menu-choose-from-copy-of-window-contents
                        (send frame :get-pane 'shapes-menu) 'entity-shape
                        :prompt "Create a shape" :superior frame)))
          (if (null shape)
              (signal 'sys:abort)
              `(com-create-entity ,shape))))))
```

:prompt Specifies a string to use as a title for the menu. The menu title appears at the top of the menu.

:default Specifies an item to be the currently selected (highlighted) item when the menu is first displayed.

Examples:

```
(dw:menu-choose-from-set '(a b c) 'symbol :default 'a)

(setq item-list '("One" "Two"))
(dw:menu-choose-from-set item-list 'string
                         :default (first item-list))
```

:near-mode Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)**.

:superior Specifies the window that is the superior of the menu window; the default is **tv:mouse-sheet**.

:momentary-p Boolean option specifying whether the menu is momentary or temporary; the default is momentary. If you wish to make the menu temporary, supply a value of **nil** to this option and **t** to the **:temporary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:temporary-p Boolean option specifying whether the menu is temporary or momentary; the default is momentary. If you wish to make the menu temporary, supply a value of **t** to this option and **nil** to the **:momentary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

dw:menu-choose-from-drawer *drawer presentation-type* &key *:prompt :default (:near-mode* **'(:mouse)***) (:superior* **(tv:mouse-default-superior)***) :alias-for-selected-windows (:row-wise* **t***) (:center-p* **dw::*default-menu-center-p***) (:character-style* **dw::*default-menu-character-style***) (:momentary-p* **t***) (:temporary-p* **dw::momentary-p***) (:use-redisplay* **t***) :minimum-width :minimum-height* *Function*

Constructs a menu from a collection of presentations produced by *drawer* of presentation type *presentation-type* and returns the selected object. This function is similar to **dw:menu-choose-from-set**.

drawer A function that returns a set of presentations. It is funcalled (perhaps several times, within **redisplayer**) given *stream* and the keyword arguments *:max-width* and *:max-height*. For example:

```
#'(lambda (stream &rest ignore)
    (loop for item in handler-list
          collect
            (present item presentation-type
                     :stream stream
                     :single-box t
                     :allow-sensitive-inferiors nil
                     :allow-sensitive-raw-text nil)
          do (send stream :tyo #\return)))
```

presentation-type The presentation type used to present the objects (but see the **:printer** option below).

:prompt Specifies a string to use as a title for the menu. The menu title appears at the top of the menu.

:default Specifies an item to be the currently selected (highlighted) item when the menu is first displayed.

Examples:

```
(dw:menu-choose-from-set '(a b c) 'symbol :default 'a)

(setq item-list '("One" "Two"))
(dw:menu-choose-from-set item-list 'string
                         :default (first item-list))
```

:near-mode Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)**.

:superior Specifies the window that is the superior of the menu window; the default is **tv:mouse-sheet**.

:alias-for-selected-windows Specifies the activity to be returned by the menu in response to the **:alias-for-selected-windows** window message. The default value, **nil**, specifies that the menu should return itself.

Using **:alias-for-selected-windows** allows you to join the menu to another activity, as opposed to having the menu be an activity in its own right. This causes `Function S` to select a different activity when invoked while the menu is popped up, instead of selecting the window under the menu.

For example, if you type

```
(dw:menu-choose-from-set '(a b c) 'symbol
    :alias-for-selected-windows *terminal-io*)
```

from the Lisp Listener, and issue a `FUNCTION S` command while the menu is present, you will select the window that was selected just before the Lisp Listener. If you had taken the default for **:alias-for-selected-windows**, `FUNCTION S` over the window would have selected the Lisp Listener.

:row-wise A Boolean specifying when **t**, that the menu items are to be displayed row-wise.

:center-p Boolean option specifying whether items displayed in the menu are centered, left to right. The default is **nil**, which causes the items to be flush left.

:character-style Specifies the character style for display of menu items. The default is (:jess :roman :large).

:momentary-p Boolean option specifying whether the menu is momentary or temporary; the default is momentary. If you wish to make the menu temporary, supply a value of **nil** to this option and **t** to the **:temporary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:temporary-p Boolean option specifying whether the menu is temporary or momentary; the default is momentary. If you wish to make the menu temporary, supply a value of **t** to this option and **nil** to the **:momentary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:use-redisplay A Boolean specifying when **t** that the menu is to be displayed redisplayably.

:minimum-width Specifies the minimum width of the menu in pixel units. The default, **nil**, causes the width of the window to be only as wide as is necessary to contain the menu items.

:minimum-height Specifies the minimum height of the menu in pixel units. The default, **nil**, causes the width of the window to be only as high as is necessary to contain the menu items.

dw:menu-choose-from-set *list presentation-type* &key (*printer* **nil**) (*prompt* **nil**) (*default* **nil**) (*near-mode* **'(:mouse)**) (*superior* **tv:mouse-sheet**) (*center-p* **dw::*default-menu-center-p***) (*character-style* **'(:jess :roman :large)**) (*momentary-p* **t**) (*temporary-p* **dw::momentary-p**) (*alias-for-selected-windows* **nil**) (*minimum-width* **nil**) (*minimum-height* **nil**) *Function*

Constructs a menu from a list of objects of a specified presentation type and returns the selected object.

This function is similar to **dw:menu-choose**, but is intended primarily for presenting a simple list of items in menu format, not items of the "general list" form that **dw:menu-choose** handles.

list The list of objects.

presentation-type The presentation type used to present the objects (but see the **:printer** option below).

Examples:

```
(dw:menu-choose-from-set '(a b c) 'symbol)

(dw:menu-choose-from-set '(#p"sys:site;foo.bar"
                #p"y:>doughty>a.b") 'pathname)

(setq item-list '("One" "Two"))
(dw:menu-choose-from-set item-list 'string)
```

:printer Specifies a function of two arguments for printing menu items. The arguments are an object — one element of *list* — and a stream. If specified, this printer is used for displaying menu items rather than that of the specified *presentation-type*.

Example:

```
(dw:menu-choose-from-set '(#p"sys:site;config.data"
                #p"y:>doty>examples.lisp") 'pathname
                         :printer
                          #'(lambda (object stream)
                              (write-string
                                (send object :name)
                                stream)))
```

The example function creates a menu displaying the choices "CONFIG" and "EXAMPLES". Pathname objects are still returned when clicked on; just the appearance in the menu has changed.

:prompt Specifies a string to use as a title for the menu. The menu title appears at the top of the menu.

:default Specifies an item to be the currently selected (highlighted) item when the menu is first displayed.

Examples:

```
(dw:menu-choose-from-set '(a b c) 'symbol :default 'a)

(setq item-list '("One" "Two"))
(dw:menu-choose-from-set item-list 'string
                         :default (first item-list))
```

:near-mode Specifies where the menu appears. The default makes it appear near the position of the mouse cursor at the time the function is called. For other possibilities: See the method **(flavor:method :expose-near tv:essential-set-edges)**.

:superior Specifies the window that is the superior of the menu window; the default is **tv:mouse-sheet**.

:center-p Boolean option specifying whether items displayed in the menu are centered, left to right. The default is **nil**, which causes the items to be flush left.

:character-style Specifies the character style for display of menu items. The default is (:jess :roman :large).

:momentary-p Boolean option specifying whether the menu is momentary or temporary; the default is momentary. If you wish to make the menu temporary, supply a value of **nil** to this option and **t** to the **:temporary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:temporary-p Boolean option specifying whether the menu is temporary or momentary; the default is momentary. If you wish to make the menu temporary, supply a value of **t** to this option and **nil** to the **:momentary-p** option.

A momentary window is deactivated and returns a value of **nil** if the user moves the mouse cursor off the menu. A temporary menu remains active until the user selects a menu item.

:alias-for-selected-windows Specifies the activity to be returned by the menu in response to the **:alias-for-selected-windows** window message. The default value, **nil**, specifies that the menu should return itself.

Using **:alias-for-selected-windows** allows you to join the menu to another activity, as opposed to having the menu be an activity in its own right. This causes `Function S` to select a different activity when invoked while the menu is popped up, instead of selecting the window under the menu.

For example, if you type

```
(dw:menu-choose-from-set '(a b c) 'symbol
    :alias-for-selected-windows *terminal-io*)
```

from the Lisp Listener, and issue a `FUNCTION S` command while the menu is present, you will select the window that was selected just before the Lisp Listener. If you had taken the default for **:alias-for-selected-windows**, `FUNCTION S` over the window would have selected the Lisp Listener.

:minimum-width Specifies the minimum width of the menu in pixel units. The default, **nil**, causes the width of the window to be only as wide as is necessary to contain the menu items.

:minimum-height Specifies the minimum height of the menu in pixel units. The default, **nil**, causes the width of the window to be only as high as is necessary to contain the menu items.

For an overview of **dw:menu-choose-from-set** and related facilities: See the section "Using Presentation Types for Input".

(flavor:method :menu-highlighted-items tv:menu-highlighting-mixin) *Method*

Get the list of highlighted items.

tv:menu-highlighting-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.

: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.

(flavor:method :minimum-height tv:essential-window) *n-pixels* *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, the console will beep or flash, and the user will be prompted to start over again with a new top-left corner.

(flavor:method :minimum-height tv:menu) *arg* *Init Option*

(flavor:method :minimum-width tv:essential-window) *n-pixels* *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, the console will beep or flash, and the user will be prompted to start over again with a new top-left corner.

(flavor:method :minimum-width tv:menu) *arg* *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.

tv:momentary-menu &optional *(superior* **tv:mouse-sheet***)* *Resource*

A *resource* of momentary menus. **tv:menu-choose** allocates a window from this resource.

tv:momentary-menu *Flavor*

Built on **tv:basic-momentary-menu** mixed with **tv:menu**. See the section "The Flavor Network of **tv:menu**".

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:momentary-multiple-item-list-menu *Flavor*

The instantiable version of the multiple-item-list-menu flavor. It is a mixture of **tv:multiple-item-list-menu-mixin** with **tv:basic-momentary-menu** and other appropriate flavors.

tv:momentary-multiple-menu *Flavor*

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.

tv:momentary-window-hacking-menu *Flavor*

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.

time:month-length *month year* *Function*

Returns the number of days in *month*; you must supply a *year* in case the month is February (which has a different length during leap years). *year* can be absolute or relative to 1900 (that is, **84** and **1984** both work).

time:month-string *month* &optional (*mode* **':long**) *Function*

Returns 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.

(flavor:method :more-p tv:sheet) *t-or-nil* *Init Option*

Initializes whether the window should have `more` processing. It defaults to **t**.

(flavor:method :more-p tv:sheet) *Method*

Returns **t** if `more` processing is enabled; otherwise, return **nil**.

tv:mouse-button-encode *bd* *Function*

Watches the mouse button and determines whether a single-click or double-click is happening. Call this function when a mouse button has been pushed and you want to interpret the push as a click. It returns **nil** if no button is pushed, or an encoded integer giving the click in the usual way.

You call **tv:mouse-button-encode** only when a button has just been pushed; that is, when you see some button down that was not down before. You must pass in the argument, *bd*, which is a bit mask saying which buttons are down now that were not down "before". The form **(boole 2** *old-buttons new-buttons***)** computes this mask.

sys:mouse-buttons &optional *peek* *Function*

Returns 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).

It returns four values:

- An integer representing the state of the mouse buttons. **0** means no buttons were pressed; **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 pressed, the integer returned is 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.

mouse-char-p *char* *Function*

Returns **t** if *char* is a mouse character, **nil** otherwise.

(flavor:method :mouse-click tv:essential-mouse) *buttons x y* *Method*

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".) *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**.

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 to **nil**, disabling double clicking entirely, mouse response time improves slightly in static windows and appreciably in Dynamic Windows. This is the recommended setting.

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-input &optional (*wait-flag* **t**) *Function*

Waits until something happens with the mouse, then returns saying what happened. It returns six values. 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 call this function only 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.

tv:mouse-last-buttons *Variable*

Contains the last setting of the mouse buttons 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 pressed, the integer returned is not the sum of the individual button codes.)

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.

(flavor:method :mouse-moves tv:essential-mouse) *x y* *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.

dw:mouse-char-for-gesture *gesture* *Function*

Returns the mouse character associated with a gesture. You can use this function to assign a new gesture symbol to a mouse character.

gesture An existing or new gesture symbol.

To assign your own symbolic name to a mouse character, use the following form:

`(setf (mouse-char-for-gesture` *gesture*`) #\mouse-`*x*`)`

Conventionally, the *gesture* symbol is a keyword.

For an overview of **dw:mouse-char-for-gesture** and related facilities: See the section "Mouse Gesture Interface Facilities".

For information on mouse characters: See the section "Mouse Characters".

dw:mouse-char-gesture *mouse-char* *Function*

Returns the standard gesture associated with a mouse character.

mouse-char The mouse character (for example, **#\mouse-m**).

For an overview of **dw:mouse-char-gesture** and related facilities: See the section "Mouse Gesture Interface Facilities".

For information on mouse characters: See the section "Mouse Characters".

dw:mouse-char-gestures *mouse-char* *Function*

Returns a list of gestures associated with a mouse character.

mouse-char The mouse character (for example, **#\mouse-m**).

For an overview of **dw:mouse-char-gestures** and related facilities: See the section "Mouse Gesture Interface Facilities".

For information on mouse characters: See the section "Mouse Characters".

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

When sent 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**.

(flavor:method :mouse-sensitive-item tv:mouse-sensitive-text-scroll-window-without-click) *x y* *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: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*

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.

tv:mouse-set-blinker-cursorpos *Function*

Positions the mouse blinker at point (**sys:mouse-x**, **sys:mouse-y**) on **tv:mouse-sheet**.

tv:mouse-sheet *Variable*

The value is the superior window, usually the main screen, that contains the position of the mouse.

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*

Waits until any of the variables **sys:mouse-x**, **sys:mouse-y**, or **tv:mouse-last-buttons** becomes 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 your program 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.

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.

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)**.

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.

tv:mouse-x-scale-array *Variable*

The value 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 arrays are specially wired. 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))
```

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-y-or-n-p *item* *Function*

Takes an *item* as its argument and displays it in a menu. *item* is usually a string. If the user clicks on "Yes" with the mouse button, the value of the item is returned. If the user clicks on "No" with the mouse or moves the mouse out of the menu, **nil** is returned.

tv:mouse-y-scale-array *Variable*

The value 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**.

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:multiple-choose *item-name item-list keyword-alist* &optional *(near-mode* **'(:mouse)***) (maxlines* **20***) sup (blinker-style* **'(nil :bold nil)***)* *Function*

Pops up a multiple-choice window and allows the user to make choices with the mouse. Unless you specifiy otherwise, 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.

See the section "The Multiple Choice Facility". *item-name* is a string of the name of the type of item, for example, /"Buffer/".

item-list is an alist, (*item name choices*). *item* is the item itself, *name* a string of its name, and *choices* a list of possible keywords, either *keyword* or (*keyword default*), where if *default* is non-**nil** the *keyword* is initially on.

keyword-alist is a list of the possible keywords, (*keyword name . implications*). *keyword* is a symbol, the same as in *item-list*'s *choices*. *name* is a string of its name.

implications is a list of on-positive, on-negative, off-positive, and off-negative implications for when the keyword is selected, each one either a list of (other) keywords or **t** for all other keywords. The default for *implications* is **(nil t nil nil)**.

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.

sup is the superior of the menu. By default, this is the sheet superior of the window that the mouse is near, if the *near-mode* is :window; otherwise, it is the mouse-sheet.

blinker-style is a character style specification that indicates how selected items in the menu are to be highlighted. The default is **'(nil :bold nil)**.

tv:multiple-menu *Flavor*

A combination of **tv:multiple-menu-mixin** with **tv:menu**. It must be explicitly deactivated by the user program.

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. For an example, See the section "**tv:multiple-menu-choose** Example".

tv:multiple-menu-choose-menu *Flavor*

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:multiple-menu-choose-menu-mixin *Flavor*

The basic flavor that makes a window exhibit multiple-menu-choose behavior.

tv:multiple-menu-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, 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.

(flavor:method :name tv:menu) *string* *Init Option*

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.

(flavor:method :name tv:sheet) *Method*

Returns the name of the window, which is a string.

(flavor:method :name tv:sheet) *name* *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-for-selection *Message*

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.

(flavor:method :name-style tv:basic-choose-variable-values) *character-style*
Init Option

Specifies the character style in which names of variables are displayed. The default is the system default character style.

dw:named-value-snapshot-continuation *name var-list* &body *body* *Function*

Generates a lexical closure of its *body*, except that it snapshots the current values of lexical variables used free within *body*.

name The internal-function name for the generated lexical closure. This supplies the *X* in names like `(:INTERNAL SOMETHING 2 X)`.

var-list The lambda-list for the generated lexical closure.

dw:named-value-snapshot-continuation can be of use, for example, when collecting closures within an iteration. The following code

```
(defun print-reverse-of-list (list)
  (let ((list-of-closures ()))
    (dolist (x list)
      (push (lambda (stream) (print x stream))
            list-of-closures))
    (dolist (closure list-of-closures)
      (funcall closure *standard-output*))))

(print-reverse-of-list '(1 2 3))
```

would print three occurrences of 3. This is because the first **dolist** might **macroexpand** into something like

```
(let ((x) (temp list))
  (prog nil
   loop (when (null temp) (return))
        (setq x (pop temp))
        (push (lambda (stream) (print x stream))
              list-of-closures)
        (go loop)))
```

where all the `(lambda ...)` share exactly the same binding of `x`. Unfortunately, this means that each time through the loop, `x` — the same `x` in *all* the closures — gets **setq**'d. A way around this is to introduce a new binding for the `x` at the point the closure is produced:

```
(let ((x)
      (temp list))
  (prog nil
        (when (null temp) (return))
        (setq x (pop temp))
        (push (let ((x x))
                (lambda (stream) (print x stream)))
              list-of-closures)))
```

Here the `x` snapshotted by the closure is different for each closure, achieving the desired effect.

dw:named-value-snapshot-continuation processes the *body*, identifying freely-referenced lexical variables which need such snapshotting. It also does special processing for **self** and instance variables referenced within flavor methods. As a result, the above fragment could be written

```
(defun print-reverse-of-list (list)
  (let ((list-of-closures ()))
    (dolist (x list)
      (push (dw:named-value-snapshot-continuation
                writer (stream)
              (print x stream))
            list-of-closures))
    (dolist (closure list-of-closures)
      (funcall closure *standard-output*))))
```

and then `(print-reverse-of-list '(1 2 3))` would correctly print 3, 2, 1.

For an overview of **dw:named-value-snapshot-continuation** and related facilities: See the section "Writing Formatted Output Macros". See the section "Snapshotting Variables".

(flavor:method :near-mode tv:choose-variable-values) *arg* *Init Option*

Specifies where to position the window. The default is the list **(:mouse)**. See the section "Input from Windows".

: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.

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

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

tv:*no-window-alternate-wholine-string* *Variable*

Controls what appears in the status line when switching windows. A program can set **tv:*no-window-alternate-wholine-string*** to indicate what is happening. The default string is `(no window)`.

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

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	Do not perform the **:noise-string-out** operation. This is the default.
:enable	Perform the operation.
:error	Signal an error.

:notification-cell *Message*

Sent to an interactive stream, it 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.

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).

: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))
```

:notification-mode *Message*

Sent to an interactive stream, it 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 does not 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**.

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).

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.

(flavor:method :number-of-items tv:text-scroll-window) *Method*

Returns the number of items in the item list.

tv:note-progress *numerator* &optional (*denominator* **1**) (*note* **tv:*current-progress-note***) *Function*

Notes the progress of an operation by updating the progress bar. This function is only used in the body of the **tv:noting-progress** macro (for examples, look at the dictionary entry for that facility). The progress bar is updated by fractional amounts between 0 and 1.

numerator The numerator of the fraction by which to update the bar.

denominator The denominator of the fraction by which to update the bar; the default is 1.

note The note object (bound to the *variable* supplied to **tv:noting-progress**).

For an overview of **tv:note-progress** and related facilities, see the section "Progress Indicator Facilities".

tv:noting-progress (*name* &optional (*variable* **'tv:*current-progress-note***) (*process* **'sys:current-process**)) &body *body* *Function*

Binds local environment such that the progress of an operation performed within the body of the macro is noted by a progress bar displayed in the status line at the bottom of the screen. The function **tv:note-progress** does the updating of the progress bar.

name A string naming the operation being noted. This string is displayed above the progress bar.

variable The variable bound to the note object; the default is **tv:*current-progress-note***. This variable is an argument to **tv:note-progress**.

process The process on whose behalf the progress is noted; the default is **sys:current-process**. This is used to determine the precedence of notes.

Examples:

```
(tv:noting-progress ("Working Away By Tenths")
  (loop for i from .1 to 1.0 by .1
        do
    (tv:note-progress i)
    (sleep 1)))

(tv:noting-progress ("Working Away By Fifths")
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 1 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 2 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 3 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 4 5)
  (loop for i from 1 to 2 by 1
        do
    (sleep 1))
  (tv:note-progress 5 5)
  (sleep 1))
```

For an overview of **tv:noting-progress** and related facilities, see the section "Progress Indicator Facilities".

tv:pane-mixin *Flavor*

Must be one of the components of the flavor of any window used as a pane of a frame. 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.

(flavor:method :pane-name tv:basic-constraint-frame) *pane* *Method*

Returns the symbol that was used to name *pane* in the **:panes** specification of this frame. If *pane* is not one of the panes, return **nil**.

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

Makes 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**.

(flavor:method :panes tv:basic-constraint-frame) *pane-descriptions* *Init Option*

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.

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*

Interprets *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, 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**.

si:parse-interval-or-never *string* *Function*

Returns an integer if *string* represents an interval, or **nil** if *string* represents "never". *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. 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**).

See the presentation type **time:time-interval**.

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.

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*

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.

: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".

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.

: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**.

peek-char &optional *peek-type input-stream* (*eof-error-p* **t**) *eof-value recursive-p* *Function*

Returns the next character to be read from *input-stream*, without actually removing it from the stream. (This is the default behavior, which can be modified by the *peek-type* argument.)

What **peek-char** does depends on *peek-type*. The effects of *peek-type* are as follows:

Value	Effect
nil	Returns the next character to be read from *input-stream*, without actually removing it from the stream. The next time input is done from *input-stream*, the character will still be there. It is as if you had called **read-char**, then **unread-char** in succession.

t Skips over whitespace characters (but not comments), and then performs the peeking operation on the next character. This is useful for finding the beginning of the next printed representation of a Lisp object. The last character examined (the one that starts an object) is not removed from the input stream.

character object Skips over input characters until a character that is **char=** to that object is found. That character is left in the input stream.

A value of **t** for *input-stream* indicates ***terminal-io***. If *input-stream* is unspecified or **nil**, then ***standard-input*** is used.

The arguments *eof-error-p* and *eof-value* control what happens when the function is called at the end of input-source. If the first argument, *eof-error-p* is **nil**, then nothing is done, otherwise an end-of-file error is signalled, and the value returned is *eof-value*.

The *recursive-p* argument is used to signal that the call to **peek-char** is not at the top level.

```
(list (read-char) (peek-char) (read-char) (read))abcdef
=> (#\a #\b #\b CDEF)

(list (read-char) (peek-char) (read))abcdef
=> (#\a #\b #\b BCDEF)
```

For a discussion of keyboard input differences between CLOE and Genera, see the function **read-char**.

dw:peek-char-for-accept *stream* &optional *hang* *Function*

Returns the next character in the input stream without removing it from the stream. This is equivalent to calling **dw:read-char-for-accept** followed by **dw:unread-char-for-accept**.

stream The input stream.

hang Boolean option specifying whether, if no character is available in the input stream, the function waits until a character is available or returns **nil**. The default is **nil**.

For an overview of **dw:peek-char-for-accept** and related facilities, see the section "Defining Your Own Presentation Types".

(flavor:method :point tv:graphics-mixin) *x y* *Method*

Returns the numerical value of the picture element at the specified coordinates. The result is **0** or **1** on a black-and-white TV. Clipping is performed; if the coordinates are outside the window, the result will be **0**.

tv:pop-up-menu *Flavor*

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". 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:pop-up-multiple-menu-choose-menu *Flavor*

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.

tv:pop-up-multiple-menu-choose-resource *Resource*

A *resource* of multiple-menu-choose menus.

(flavor:method :position tv:menu) *(left-edge top-edge)* *Init Option*

Specifies the left and top edges of the window. All specifications are given with respect to the outside of the superior window.

(flavor:method :position tv:sheet) *Method*

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

(flavor:method :position tv:sheet) *(left-edge top-edge)* *Init Option*

Specifies the x-coordinate of the left edge and the y-coordinate of the top edge of the window.

: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.

tv:prepare-sheet (*sheet*) *body...* *Function*

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

present *object* &optional (*presentation-type* **(type-of dw::object)**) &key (*stream* ***standard-output****) (*acceptably* **nil**) (*sensitive* **t**) *for-context-type (original-type* **nil**) (*form* **nil**) (*location* **nil**) (*single-box* **nil**) (*allow-sensitive-inferiors* **t**) *(allow-sensitive-raw-text* **t**) *Function*

Outputs a presentation object to a stream. The manner in which the object is displayed depends on the presentation type of the object. If the stream supports presentation remembering, the presented object is accessible via the mouse in the appropriate input context; if not, the printed representation is the same, but the object is not mouse-sensitive.

object The object to be presented.

presentation-type The presentation type to be used in presenting the object; the default is the type of the object.

:stream Specifies stream on which the object is presented; the default is ***standard-output****.

:acceptably Specifies when **t** to present the object in such a way that it can later be parsed by **accept**; the default is **nil**. A third possible value, **:very**, is for use with **:for-context-type**. This option is useful when output is to strings or files, but not necessary when output is to Dynamic Windows.

:sensitive
Boolean option specifying whether the presentation is mouse-sensitive; the default is **t**. This option is useful for explicitly preventing mouse sensitivity of objects presented to Dynamic Windows.

:form Specifies a form that can be passed to **setf** to store a new value in place of the current output value. This option and **:location** are mutually exclusive.

The form supplied for this option is used by a predefined, side-effecting mouse handler (available on `c-m-`Right) to modify the contents of structure slots.

:for-context-type Specifies the context in which the presentation is to be presented with **:acceptably :very**. The most often used value is `'((cp:command-or-form :dispatch-mode :form-preferred))`, which causes presentations of the type **cp:command** to be printed with a leading colon.

:original-type The original type supplied, to be passed through in successive recursive calls to **present** (or **present-to-string** or **accept**).

:location Specifies a locative that can be used to store a new value in place of the current output value. This option and **:form** are mutually exclusive.

The locative supplied for this option is used by a predefined, side-effecting mouse handler (available on c-m-Right) to modify the contents of structure slots.

:single-box Specifies that mouse-sensitivity of objects output in a series of inferior calls to this form be indicated by a single, large box for highlighting rather than the sum of all the individual boxes. This option is used mostly with graphic presentations.

:allow-sensitive-inferiors Boolean option specifying whether nested calls to **present** or **dw:with-output-as-presentation** from inside this presentation — for example, when presenting the individual elements of a Lisp list — generate presentation objects. The default is **t**.

For an example, see the function **dw:with-output-as-presentation**.

:allow-sensitive-raw-text Boolean options specifying when **t** that raw text, that is, the individual characters that make up the presentation, are to be mouse-sensitive.

For an overview of **present** and related facilities: See the section "Using Presentation Types for Output".

dw:presentation-blip-case *blip* &body *clauses* *Function*

Dispatches to clauses based on the presentation-type field of a presentation blip.

blip The presentation blip.

clauses The **case** clauses.

This macro is similar to the **case** special form, and could be written as

```
(case (dw:presentation-blip-presentation-type blip)
  <clauses>)
```

but with one exception: comparison of the extracted presentation type with the types used as keys to the *clauses* is based on **dw:presentation-subtypep**, not **eql**.

Normally, you would not use this macro directly. See the function **dw:with-presentation-input-context**.

For an overview of **dw:presentation-blip-case** and related facilities: See the section "Presentation Input Blip Facilities".

dw:presentation-blip-ecase *blip* &body *clauses* *Function*

Dispatches to clauses based on the presentation-type field of a presentation blip.

blip The presentation blip.

clauses The **ecase** clauses.

This macro is similar to the **ecase** special form, and could be written as

```
(ecase (dw:presentation-blip-presentation-type blip)
  <clauses>)
```

but with one exception: comparison of the extracted presentation type with the types used as keys to the *clauses* is based on **dw:presentation-subtypep**, not **eql**.

Normally, you would not use this macro directly. See the function **dw:with-presentation-input-context**.

For an overview of **dw:presentation-blip-ecase** and related facilities: See the section "Presentation Input Blip Facilities".

dw:presentation-blip-mouse-char *presentation-blip* *Function*

Returns the mouse character from a presentation blip.

presentation-blip The presentation blip.

For an overview of **dw:presentation-blip-mouse-char** and related facilities: See the section "Presentation Input Blip Facilities".

dw:presentation-blip-object *presentation-blip* *Function*

Returns the presentation object from a presentation blip.

presentation-blip The presentation blip.

For an overview of **dw:presentation-blip-object** and related facilities: See the section "Presentation Input Blip Facilities".

dw:presentation-blip-options *presentation-blip* *Function*

Returns the options field (a list of keyword-value pairs) of a presentation blip.

presentation-blip The presentation blip.

The options inserted in a presentation blip are obtained from the values returned by translating mouse handlers. A standard blip option is **:activate**, which can be used by a translator to promote or prevent activation of the current field, that is, a return from the current call to **accept**. (See the function **define-presentation-translator**.)

For an overview of **dw:presentation-blip-options** and related facilities: See the section "Presentation Input Blip Facilities".

dw:presentation-blip-p *blip* *Function*

Determines whether a blip is a presentation blip.

blip The blip.

For an overview of **dw:presentation-blip-p** and related facilities: See the section "Presentation Input Blip Facilities".

dw:presentation-blip-presentation-type *presentation-blip* *Function*

Returns the presentation type from a presentation blip.

presentation-blip The presentation blip.

For an overview of **dw:presentation-blip-presentation-type** and related facilities:

See the section "Presentation Input Blip Facilities".

dw:presentation-blip-typep *blip type* *Function*

Determines whether the presentation type of a presentation blip is of a specified type. (The comparison is based on **dw:presentation-subtypep**).

blip The presentation blip.

type The presentation type with which the type of the blip is compared.

For an overview of **dw:presentation-blip-typep** and related facilities: See the section "Presentation Input Blip Facilities".

dw:presentation-equal *presentation-1 presentation-2* *Function*

Determines whether two presentations are "equal", that is, whether they are presenting the same object in the same manner.

presentation-1 The first presentation.

presentation-2 The second presentation.

For an overview of **dw:presentation-equal** and related facilities:

See the section "Defining Your Own Presentation Types".

dw:*presentation-input-context* *Variable*

Bound to the current presentation input context.

An input context is a list of the form `(presentation-type superior-context throw-flag . options)`. Each time a new input context is established, it becomes the new top-level context, enclosing the previous top-level context. (See the function **dw:with-presentation-input-context**.) Thus, there may be a hierarchy of contexts.

For example, if you have a presentation type **microcode-version** whose parser is defined as follows

```
(define-presentation-type microcode-version ()
  :parser ((stream)
           (accept 'integer :stream stream))
  :printer ((object stream)
            (princ object stream)))
```

the call `(accept '((microcode-version)))` results in the following input context:

```
(INTEGER (MICROCODE-VERSION NIL T :INHERIT T) T :INHERIT T)
```

The initial call to **accept** establishes the `MICROCODE-VERSION` context and calls the parser for **microcode-version**. The parser calls **accept** with the presentation type **integer**, and **accept** establishes a new context for `INTEGER`; the new context contains the old context for `MICROCODE-VERSION`.

For an overview of **dw:*presentation-input-context*** and related facilities: See the section "Presentation Input Context Facilities".

dw:presentation-input-context-option *presentation-input-context indicator* *Function*

Extracts the value of the specified option from an input context. The input context options are supplied in the *options* clause to **dw:with-presentation-input-context**.

presentation-input-context Specifies the input context.

indicator Specifies the name of the option to be extracted from the input context.

For an overview of **dw:presentation-input-context-option** and related facilities: See the section "Presentation Input Context Facilities".

dw:presentation-subtypep *subtype supertype* *Function*

Determines whether one presentation type is a subtype of another presentation type and encaches the result of the lookup.

subtype The putative subtype presentation type.

supertype The putative supertype presentation type.

This function is the presentation system equivalent of the Common Lisp function **subtypep**. As does the latter, it returns two values: the first indicates whether the first type is a subtype of the second; the second whether the first result is certain. Three combinations are possible:

t	**t**	*subtype* is definitely a subtype of *supertype*
nil	**t**	*subtype* is definitely not a subtype of *supertype*
nil	**nil**	the relationship could not be determined with certainty

One use of this function is in the **:tester** forms of mouse handlers. Although it is generally more convenient to use **dw:handler-applies-in-limited-context-p**, more complex testers may need **dw:presentation-subtypep**, on which the former is based:

```
(defun handler-applies-in-limited-context-p
       (context limiting-context-type)
  (let ((context-type
          (presentation-input-context-presentation-type
            context)))
    (presentation-subtypep
      context-type limiting-context-type)))
```

See the function **dw:handler-applies-in-limited-context-p**. See the section "Defining Your Own Presentation Types".

For an overview of **dw:presentation-subtypep** and related facilities: See the section "Programming the Mouse: Writing Mouse Handlers".

dw:presentation-subtypep-cached *subtype supertype* *Function*

Obsolete. Use instead the function

dw:presentation-subtypep.

dw:presentation-type-default *presentation-type* *Function*

Returns the current default — the object at the top of the type history — for a presentation type, if the type supports a history; otherwise, it returns **nil**.

presentation-type The presentation type.

Example:

```
(dw:presentation-type-default 'pathname)
==>#P"Y:>reg>saved-mail>ui>defpgm.babyl.newest"
FS:LMFS-PATHNAME
T
```

For an overview of **dw:presentation-type-default** and related facilities:

See the section "Defining Your Own Presentation Types".

dw:presentation-type-name *type* *Function*

Returns the name of the presentation type from a presentation-type specification.

type The type specification.

Example:

```
(dw:presentation-type-name '((pathname) :dont-merge-default nil))
PATHNAME
```

For an overview of **dw:presentation-type-name** and related facilities:

See the section "Defining Your Own Presentation Types".

dw:presentation-type-p *type* *Function*

Returns the presentation type descriptor if its argument is a presentation type, **nil** otherwise. If the argument is a user-defined flavor, the descriptor of that type is returned. If it is a user-defined structure, **t** is returned.

type An object.

Example:

```
(dw:presentation-type-p '((integer 1 10)))
=> #<DW:PRESENTATION-TYPE-DESCRIPTOR INTEGER 42516527>
(dw:presentation-type-p 'dw:dynamic-window)
=> #<DW:PRESENTATION-TYPE-DESCRIPTOR DW:DYNAMIC-WINDOW 42506626>
(defflavor a-flavor () ())
(dw:presentation-type-p 'a-flavor)
=> #<FLAVOR A-FLAVOR 1230276>
```

For an overview of **dw:presentation-type-p** and related facilities:

See the section "Defining Your Own Presentation Types".

present-to-string *object* &optional *(presentation-type* **(type-of dw::object)***)* &key *(original-type* **dw:presentation-type***) (string* **nil***) (index* **nil***) acceptably for-context-type* *Function*

Outputs a presentation object to a string or returns a new string.

object The object to be presented.

presentation-type The presentation type to be used in presenting the object; the default is the type of the object.

:string Specifies a string to which the object is presented. The default is **nil**, causing a new string object to be created and returned as the value.

:index The character position in the string array where the presenting of the object begins; the default is position `0`. Use this option only if the **:string** option is non-**nil**.

:acceptably Specifies when **t** that the object should be presented in such a way that it can later be parsed by **accept**; the default is **nil**. A third value, **:very** is for use with **:for-context-type**.

:for-context-type Specifies the context in which the presentation is to be presented with **:acceptably :very**. The most often used value is `'((cp:command-or-form :dispatch-mode :form-preferred))`, which causes presentations of the type **cp:command** to be printed with a leading colon.

:original-type The original type supplied, to be passed through in successive recursive calls to **present** (or **present-to-string** or **accept**).

For an overview of **present-to-string** and related facilities: See the section "Using Presentation Types for Output".

(flavor:method :primitive-item tv:basic-mouse-sensitive-items) *type item left top right bottom* *Method*

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.

time:print-brief-universal-time *ut* &optional (*stream* **standard-output**) (*ref-ut* **(zl-user:get-universal-time)**) *Function*

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
```

time:print-current-date &optional (*stream* **standard-output**) *Function*

Prints the current time, formatted as in **Friday the twenty-fifth of November, 1988; 3:50:41 pm**, to the specified stream.

time:print-current-time &optional (*stream* **standard-output**) *Function*

Prints the current time, formatted as in **11/25/83 14:50:02**, to the specified stream.

time:print-date *seconds minutes hours day month year day-of-the-week* &optional (*stream* **standard-output**) *Function*

Prints the specified time, formatted as in **Friday the twenty-fifth of November, 1983; 3:50:41 pm**, to the specified stream.

(flavor:method :print-function tv:function-text-scroll-window) *Method*

Returns the window's printing function.

(flavor:method :print-function-arg tv:function-text-scroll-window) *Method*

Returns the object which the window passes as the second argument to the print function.

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**.

: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 (that is, to *self*) to print *item*.

For "mouse-sensitive items" within the "item", send **:item** to *self*.

time:print-time *seconds minutes hours day month year* &optional (*stream* **standard-output**) *Function*

Prints the specified time, formatted as in **11/25/83 14:50:02**, to the specified stream.

time:print-universal-time *ut* &optional (*stream* **standard-output**) *timezone* *Function*

Prints the specified time, formatted as in **11/25/83 14:50:02**, to the specified stream.

time:print-universal-date *ut* &optional (*stream* **standard-output**) *timezone* *Function*

Prints the specified time, formatted as in **Friday the twenty-fifth of November, 1983; 3:50:41 pm**, to the specified stream.

(flavor:method :process tv:process-mixin) (*initial-function* **.** *options*) *Init Option*

initial-function is called in the window's process with the window as its argument. *options* are options to **make-process**.

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.)

dw:*program* *Variable*

Bound to the currently active instance of a program flavor (created via **dw:define-program-framework**).

For an overview of **dw:*program*** and related facilities, see the section "Defining Your Own Program Framework".

dw:program-command-table *program* *Function*

Returns the command table used by an instance of a program flavor (created via **dw:define-program-framework**).

program The program instance. (The currently active program instance can be accessed as the value of **dw:*program***.)

For an overview of **dw:program-command-table** and related facilities, see the section "Defining Your Own Program Framework".

dw:*program-frame* *Variable*

Bound to the program frame associated with the current instance of a program flavor (created via **dw:define-program-framework**).

Use this variable for access to the program frame from a generic function or method of the program flavor, or from a program command definition macro.

Example (for a program flavor named "my-program"):

```
(define-my-program-command (com-enable-secondary-commands
                            :menu-accelerator "More Commands"
                            :menu-level :main)
                           ()
  (send dw:*program-frame* :set-configuration 'secondary))
```

For access to a particular pane of the program frame, send a **:get-pane** message to **dw:*program-frame*** or use **dw:get-program-pane**.

For an overview of **dw:*program-frame*** and related facilities, see the section "Defining Your Own Program Framework".

dw:program-frame *Flavor*

The flavor used by **dw:define-program-framework** for the program frames it creates. **dw:program-frame** is the Dynamic Window equivalent of **tv:constraint-frame-with-shared-io-buffer**, which it incorporates as one of its component flavors; another component flavor is **tv:process-mixin**. Generally, you do not make direct use of this flavor; that you leave up to **dw:define-program-framework**.

Init options, methods, and messages for this flavor include all of those for **tv:constraint-frame-with-shared-io-buffer**:See the section "Frames". The following are additional init options:

:label See the section "Window Labels".

:margin-components See the flavor **dw:dynamic-window**.

:process See the section "Windows and Processes". Normally you automatically get a process running the program's top-level function when you program an application using **dw:define-program-framework**.

:program The name of the program for which this is the program frame.

:query-io-pane Specifies the pane to which ***query-io*** is bound when an instance of the program frame is active.

:size-from-pane Specifies the pane on which to base the size of the program frame.

:terminal-io-pane Specifies the pane to which ***terminal-io*** is bound when an instance of the program frame is active.

For an overview of **dw:program-frame** and related facilities: See the section "Window Substrate Facilities".

dw:program-frame *Resource*

A resource of program frames (of the kind used by **dw:define-program-framework**). The resource is created via **tv:defwindow-resource** with the **:initial-copies** option set to **nil** and the **:reuseable-when** option set to **:deactivated**. (For more information on resources generally: See the section "Resources".)

In addition to the required argument *program-name* and the optional argument *superior* (the frame's superior), the following keyword options are available when allocating from or using the program frame resource:

:temporary-p Boolean option specifying whether the frame provided is temporary, that is, whether it locks the superior window until it is deactivated.

:process The process associated with the frame or **nil**, for no associated process. The default process is that of the program for which this

frame was created (by **dw:define-program-framework**). You can, for example, set this option to **nil** to run a program in you own process.

When using this resource, you must supply the name of the program whose frame is to be provided. In the following example, a Frame-Up Layout Designer frame is specified.

Example:

```
(defun pf-resource ()
  (using-resource (my-pf dw:program-frame 'dw::layout-designer)
    (send my-pf :expose)))
```

: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".

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.

prompt-and-accept *presentation-type-or-args* &optional *format-string* &rest *format-args* *Function*

Prompts for and accepts user input. (This function is similar to **accept**; it differs in that it uses the **format** function for creating the input prompt.)

presentation-type-or-args Presentation type of the input object or, alternatively, a list of keyword-value pairs.

Available keywords are the same as those for **accept** with one exception. This is the **:type** keyword, specifying the presentation type of the input object. If keywords are provided, one of them must be **:type**.

format-string Control string for the **format** function.

format-args Arguments for the format specifiers included in the *format-string*.

Example:

```
(let ((x 5) (y 9))
(prompt-and-accept 'integer "Value for cell ~D ~D" x y))
```

For an overview of **prompt-and-accept** and related facilities: See the section "Using Presentation Types for Input".

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

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 function is obsolete. The function **accept** should be used to perform this operation.

```
(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 **(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 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 termi-

nated 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 chaos-net.
: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 **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 **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). Lowercase 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.

(flavor:method :put-item-in-window tv:text-scroll-window) *item* *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.

(flavor:method :put-last-item-in-window 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.

cp:read-accelerated-command &key (*command-table* **cp:*command-table***) (*stream* ***query-io***) (*help-stream* **stream**) (*echo-stream* **stream**) (*whostate* **nil**) (*prompt* **nil**) (*command-prompt* **cp::*full-command-prompt***) (*full-command-full-rubout* **nil**) (*special-blip-handler* **nil**) (*timeout* **nil**) (*input-wait* **nil**) (*input-wait-handler* **nil**) (*form-p* **nil**) (*handle-clear-input* **nil**) (*catch-accelerator-errors* **t**) (*unknown-accelerator-is-command* **nil**) (*unknown-accelerator-tester* **nil**) (*unknown-accelerator-reader* **nil**) (*unknown-accelerator-reader-prompt* **nil**) (*abort-chars* **nil**) (*suspend-chars* **nil**) (*status* **nil**) (*intercept-function* **nil**) (*window-wakeup* **nil**) *Function*

Reads a Command Processor command input as a single-key accelerator.

The values returned by this function depend on whether a command or form is expected (see the **:form-p** option below). If the caller is expecting a command (**:form-p** is **nil**), the values returned are the command name, command arguments, and a flag. If the caller is expecting a form (**:form-p** is **t**), the values returned are the form and a flag.

Possible values for the returned flag are:

:command A command was read.

:form A form was read.

:accelerator An accelerator character was read.

:timeout A timeout expired.

:status The window's status changed.

:wakeup The window was asynchronously refreshed, selected, exposed, and so on.

:unknown (or **nil**) Something unknown was typed.

cp:read-accelerated-command accepts the following keyword options:

:command-table Specifies the command table containing the accelerator; the default is the current binding of **cp:*command-table***.

:stream Specifies the stream from which to read the command; the default is ***query-io***.

:help-stream Specifies the output stream for help messages; the default is the stream specified by the **:stream** option.

:echo-stream Specifies the stream to which the input command is echoed; the default is the stream specified by the **:stream** option.

To suppress echoing, supply this option with **#'ignore**.

:whostate Specifies a string to appear in the status line in place of `"User Input"`.

:prompt Specifies a string to be used as the prompt, or a prompt option. (See the section "Displaying Prompts in the Input Editor".)

:command-prompt Specifies a string to be used as the prompt if a command is to be read, that is, if the user types ":" or `m-x`. The default is **cp::*full-command-prompt***, which is "Command: ".

:full-command-full-rubout Boolean option specifying whether to return if `CLEAR INPUT` is pressed (or a series `RUBOUT`s back to the prompt) after the command prompt (":" or `m-X`, for example) is typed. The default is **nil**, allowing the continuation of command parsing.

:special-blip-handler Specifies a function called with mouse blips that are not presentation input blips. (See the section "Mouse Blips".)

:timeout Specifies the length of time, in 60ths of a second, after which, if the user types nothing, **cp:read-accelerated-command** returns **:timeout** as the flag and **nil** for the other values.

:input-wait Specifies a function testing for some condition while in the input-wait state. If this condition occurs, the **:input-wait-handler** is invoked.

:input-wait-handler Specifies a function called after a condition satisfying the **:input-wait** function occurs.

:form-p Boolean option specifying whether a form or command is expected; the default is **nil**. If **t**, the function returns an evaluable form rather than the command name and arguments.

:handle-clear-input Boolean option specifying whether **#/clear-input** is treated specially; the default is **nil**. If **t** and the CLEAR INPUT key is pressed, the function clears the input buffer and reprompts.

:catch-accelerator-errors Boolean option specifying whether, when an unknown accelerator character is typed, the function beeps and prints out a warning message; the default is **t**. If **nil**, the error flavor **cp::accelerator-error** is signaled.

:unknown-accelerator-is-command Specifies whether unknown accelerators are dispatched to the **:unknown-accelerator-reader** function.

The default is **nil**. Unknown accelerators that do not pass the **:unknown-accelerator-tester** function give errors (which may or may not get through to the user — see the **:catch-accelerator-errors** option).

If **t**, all unknown accelerators dispatch to the unknown-accelerator reader which should return a command.

A third value permitted for this option is **:alpha**, causing only unknown accelerators that are alphabetic characters to be dispatched to the unknown-accelerator reader.

:unknown-accelerator-tester Specifies a function of one argument, the character typed, which should return something non-**nil** if this particular unknown accelerator is permitted. In this case, **:unknown** is returned as the flag and the value from this function is the first value. If **:form-p** is **nil**, the character is returned as the second value.

:unknown-accelerator-reader Specifies a function of no arguments that should return a form. (The function can call **cp:read-command**, etc., but it should return a form.)

:unknown-accelerator-reader-prompt Specifies a string to use as the prompt in this case, or a prompt option. (See the section "Displaying Prompts in the Input Editor".)

:abort-chars Specifies a list of "abort" characters; the default is **nil**.

If a list of characters is provided and the user types one, **sys:abort** is signalled.

:suspend-chars Specifies a list of "suspend" characters; the default is **nil**.

If a list of characters is provided and the user types one, a **break** loop is entered.

:status Specifies what happens if the window's status changes. Three values are permitted, **:selected**, **:exposed**, and **nil**.

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. This is the default.

:intercept-function Specifies a function of one argument, a character, that gets called on each typed character that is one of **:abort-chars** or **:suspend-chars**.

:window-wakeup Boolean option specifying whether to return **:wakeup** when an asynchronous window system condition like expose, select, or refresh occurs; the default is **nil**.

For an overview of **cp:read-accelerated-command** and related facilities: See the section "Managing Your Program Frame".

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.

(flavor:method :read-bp 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**.

dw:read-char-for-accept *stream* *Function*

Returns the next character in the input stream and removes this character from the stream.

stream The input stream.

The character returned may be a presentation blip character containing information specific to the **accept** input mechanism. Therefore, characters read via **dw:read-char-for-accept** should only be manipulated by the related Dynamic Window input functions. For example, you cannot use **char-equal** to compare a character returned by **dw:read-char-for-accept** with a standard character; you must use **dw:compare-char-for-accept** instead.

For an overview of **dw:read-char-for-accept** and related facilities, see the section "Defining Your Own Presentation Types".

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**) *presentation-context* *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". 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". 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**.

:presentation-context If this is not **nil**, the presentation system is enabled, that is, presentations that are targets of existing mouse handlers will be sensitive.

cp:read-command &optional (*stream* **zl-user:*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 ***cl: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". **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".

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 com-

mand: See the section "Help in the Command Processor". For a general description of how the user enters a command: See the section "Entering a Command".

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".

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". 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 "Managing the Command Processor".

cp:read-command-arguments *command-name* &key *initial-arguments* (*command-table* **cp:*command-table***) (*stream* ***standard-input***) (*prompt* **nil**) *Function*

Prompts for and returns the arguments to a Command Processor command.

command-name The command name (symbol).

:initial-arguments Specifies a list containing zero or more of the initial arguments to the command.

:command-table Specifies the command table containing the command; the default is the current command table (bound to **cp:*command-table***).

:stream Specifies the input stream; the default is ***standard-input***.

:prompt Specifies a string, or a function returning a string, to be used as the prompt for the command arguments. The default value for this option is **nil**, causing the prompt to be derived from the user-visible name of the command.

Example:

```
(cp:read-command-arguments 'si:com-show-file :prompt
                                              "File for viewing")
```

You can apply the command function to the arguments returned in order to execute it.

For an overview of **cp:read-command-arguments** and related facilities: See the section "Managing Your Program Frame".

cp:read-command-or-form &optional *(stream* **zl-user:*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 expression-reader expression-printer (environment* **si:*read-form-environment***)* *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 ***cl: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".

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".

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". The default prompt is the value of **cp::*default-prompt***. The initial default is "Command: ".

The keyword **:exception-chars** is for internal use. It is ignored by **cp:read-command-or-form**.

Possible values of the keywords **:expression-reader** and **:expression-printer** are functions for reading and writing expressions in languages other than Lisp, such as Pascal, Fortran, or C. These are for use by the debugger.

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 "Managing the Command Processor".

(flavor:method :read-cursorpos tv:blinker) *Method*

Returns two values: the *x* and *y* components of the position of the blinker within the inside of the window.

(flavor:method :read-cursorpos tv:sheet) &optional *(units* **':pixel**) *Method*

Return two values: the *x* and *y* coordinates of the cursor position, that is, <*x*, *y*> is the upper left corner of the next character drawn. 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.)

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 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**, but 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 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-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 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)**.

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**.

cp:read-full-command *arg-p ignore* *Function*

This is the m-X (extended) and colon-full-command Command Processor command accelerator, which lets you type extended commands to the single command accelerator reader.

cp:read-full-command is a function that is suitable for use as a command accelerator's function. However, because it is already installed on ":" and m-X in the "Colon Full Command" command-table, the best way to make use of this facility is to have the command tables in your applications that use accelerator characters inherit from "Colon Full Command".

See the function **cp:add-command-accelerator**. For an overview of **cp:read-full-command** and related facilities: See the section "Managing Your Program Frame".

sys:read-interval-or-never &optional *stream or-nil* *Function*

Reads a line of input from *stream* (using **zl:readline**) and calls **si:parse-interval-or-never** on the resulting string.

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 pass-

words 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". 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". 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**.

(flavor:method :read-location si:interactive-stream) *Method*

Returns the value of the scan pointer. This is also defined on noninteractive 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.

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 non-whitespace 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.

dw:read-standard-token *stream* *Function*

Parses string as delimited by activation and blip characters established by **dw:with-accept-activation-chars** and **dw:with-accept-blip-chars**, respectively.

stream The input stream.

For an overview of **dw:read-standard-token** and related facilities, see the section "Defining Your Own Presentation Types".

:receive-notification *Message*

Sent to an interactive stream, it 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**.

tv:rectangular-blinker *Flavor*

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.

(flavor:method :redisplay 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.

(flavor:method :redisplay-variable tv:choose-variable-values-window) *variable* *Method*

Redisplays just the value of the specified variable.

dw:redisplayable-format *stream format-string* &rest *format-args* *Function*

Outputs a formatted string redisplayably. This simply calls **format** within a caching point for incremental redisplay. (See the function **dw:with-redisplayable-output**.) *format-string* is used as the cache-id; the list *format-args* is used as the cache-value.

stream The output stream; the default is ***standard-output***.

format-string The format-control string. (See the function **format**.)

format-args The format arguments.

dw:redisplayable-format is one of a number of facilities used to do incremental redisplay. For examples, see the file SYS:EXAMPLES;INCREMENTAL-REDISPLAY.LISP.

For an overview of **dw:redisplayable-format** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

dw:redisplayable-present *object* &optional *presentation-type* &key (*stream* ***standard-output***) (*unique-id* **nil**) &allow-other-keys *Function*

Presents an object redisplayably. This simply calls **present** within a caching point for incremental redisplay. (See the function **dw:with-redisplayable-output**.) The *object* itself is used as the cache-value.

object The object to present.

presentation-type The presentation type to display the object as; the default is the Lisp object type of the object, that is, **(type-of** *object***)**.

:stream Specifies the output stream; the default is ***standard-output***.

:unique-id Identifies the particular incremental redisplay cache. This may be any object, as long as it is unique with respect to the *id-test* predicate among all such ids in the current incremental redisplay. The default is that there is no id, not that **nil** is the id.

Other keyword options to **dw:redisplayable-present** are the same as those to **present**, to which they are passed: See the function **present**.

dw:redisplayable-present is one of a number of facilities used to do incremental redisplay. For examples, see the file SYS:EXAMPLES;INCREMENTAL-REDISPLAY.LISP.

For an overview of **dw:redisplayable-present** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

dw:redisplayer (&optional *stream*) &body *body* *Function*

Creates a redisplay object out of its *body*, which can be used to do incremental redisplay on *stream*. Provide the redisplay object as the *redisplay-piece* argument to **dw:do-redisplay**: See the function **dw:do-redisplay**.

stream The output stream; the default is ***standard-output***.

dw:redisplayer is one of a number of facilities used to do incremental redisplay. For examples, see the file SYS:EXAMPLES;INCREMENTAL-REDISPLAY.LISP.

For an overview of **dw:redisplayer** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

Note that you cannot currently use any options with this function.

(flavor:method :refresh tv:menu) &optional *type* *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.

(flavor:method :refresh tv:sheet) &optional *type* *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.

(flavor:method :remove-asynchronous-character si:interactive-stream) *character* *Method*

Removes an asynchronous character from the list for the stream.

See the section "Asynchronous Characters".

(flavor:method :remove-highlighted-value tv:menu-highlighting-mixin) *value* *Method*

Removes an item from the list of highlighted items. Refers to the item by value. For instance, if your item-list is an association list, with elements (*string* . *symbol*), this message uses *symbol*. This only works for menu items that can be executed without side-effects, not, for example, the **:eval** and **:funcall** kinds.

See the section "**tv:multiple-menu-mixin** Messages".

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

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	Removes *n-chars* characters from immediately before the scan pointer
string	Removes as many characters as the string contains
nil	Removes 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	Do not perform the **:replace-input** operation. This is the default.
:enable	Perform the operation.
:error	Signal an error.

(flavor:method :replace-item tv:text-scroll-window) *item-no new-item* *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.

: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".

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.

(flavor:method :rescanning-p si:interactive-stream) *Method*

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.

reset-user-options *alist* *Function*

Resets each of the option variables in *alist* to its default initial value.

(flavor:method :reverse-video-p tv:menu) *t-or-nil* *Init Option*

If set to **t**, the menu is displayed in reverse video, that is, white-on-black instead of black-on-white.

(flavor:method :reverse-video-p tv:sheet) *Method*

Returns **nil** normally or **t** if the window displays in white on black rather than black on white. This is separate from the whole screen's inverse video mode (set by FUNCTION C).

(flavor:method :right tv:menu) *right-edge* *Init Option*

Right edge of the window specified in pixels, relative to the outside of the superior window.

(flavor:method :right tv:sheet) *right-edge* *Init Option*

Specifies the x-coordinate of the right edge of the window.

(flavor:method :right-margin-character-flag tv:sheet) *x* *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**.

(flavor:method :right-margin-size tv:sheet) *Method*

Returns the right margin size of the window in pixels.

(flavor:method :rows tv:menu) *n-rows* *Init Option*

Sets the number of rows.

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.

(flavor:method :save-bits tv:sheet) *t-or-nil* *Init Option*

Specifies whether output to the window is written to a bit-save array when the window is deexposed; the default is **nil**. If **t**, the output is redisplayed following re-exposure of the window. The value of this option can also be **:delayed**. For more information on bit-save arrays, see the section "Pixels and Bit-Save Arrays".

(flavor:method :screen tv:menu) *screen* *Init Option*

In a system with multiple screens, sets the screen on which the menu appears.

: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".

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

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

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
```

tv:scroll-parse-item &rest *line-item-spec* *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".

(flavor:method :scroll-to tv:basic-scroll-bar) *number type* *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.

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

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.

tv:*select-keys* *Variable*

As of Genera 7.3 Ivory, the SELECT key uses an internal database rather than **tv:*select-keys***. It is retained for compatibility.

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**. 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.

tv:select-mixin *Flavor*

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

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.

:select-pane *pane* *Message*

Sent 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.

:select-relative *Message*

Sent to a selectable window selects the window relative to its activity, but does not 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.

tv:select-relative-mixin *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**.

:selectable-windows *Message*

Sent to a window, it 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.

(flavor:method :selected-choice-style tv:basic-choose-variable-values) *character-style* *Init Option*

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.

:selected-pane *Message*

Sent to a frame, it returns the selected-pane of the frame. This message is sent by users and received by the system.

(flavor:method :selected-pane tv:basic-constraint-frame) *pane* *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.

tv:selected-window *Variable*

The value is the currently selected window.

(flavor:method :send-all-exposed-panes tv:basic-constraint-frame) *message* &rest *arguments* *Method*

Sends the specified *message* with the specified *arguments* to all of the exposed panes of this frame.

(flavor:method :send-all-panes tv:basic-constraint-frame) *message* &rest *arguments* *Method*

Sends the specified *message* with the specified *arguments* to all of the panes of this frame, including the nonexposed ones.

(flavor:method :send-pane tv:basic-constraint-frame) *pane-name message* &rest *arguments* *Method*

Sends the specified *message* with the specified *arguments* to the pane that was named by the symbol *pane-name* in the **:panes** specification of this frame.

tv:sensitive-item-types *Variable*

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.

(flavor:method :set-border-margin-width tv:borders-mixin) *new-width* *Method*

Sets the value of the border margin width.

(flavor:method :set-borders dw:margin-mixin) *borders* *Method*

Replaces the current borders of a Dynamic Window with simple borders (like those provided by **dw:margin-borders**).

borders The thickness, in pixels, of the new borders; the default is 1.

For an overview of **(flavor:method :set-borders dw:margin-mixin)** and related facilities: See the section "Window Substrate Facilities".

(flavor:method :set-character tv:character-blinker) *nchar* *Method*

Sets the character to display to *nchar*.

(flavor:method :set-configuration tv:basic-constraint-frame) *configuration-name* *Method*

Sets the configuration of the frame to the one named by the symbol *configuration-name*.

(flavor:method :set-cursorpos tv:blinker) *x y* *Method*

Sets 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.

(flavor:method :set-cursorpos tv:sheet) *x y* &optional *(units* **':pixel**) *Method*

Moves the cursor position to the specified coordinates. The units may be specified as with **:read-cursorpos**. If the coordinates are outside the window, move the cursor position to the point nearest to the specified coordinates that is within the window. Sending **nil** for *x* or *y* leaves the current value unmodified.

(flavor:method :set-deexposed-typein-action tv:sheet) *action* *Method*

Sets the deexposed typein action of the window to *action*.

(flavor:method :set-deexposed-typeout-action tv:sheet) *action* *Method*

Sets the deexposed typeout action of the window to *action*.

(flavor:method :set-default-character-style tv:menu) *new-style* *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.

(flavor:method :set-default-character-style tv:sheet) *new-default-style* *Method*

Changes the default character style of the window.

dw:set-default-end-of-page-mode *new-end-of-page-mode* &optional *new-scroll-factor* *Function*

Sets global default for what happens when queued output exceeds the space available in the current viewport of a Dynamic Window.

new-end-of-page-mode The new mode. There are three possibilities:

:scroll Causes the window to scroll automatically to accommodate the output. If you supply this argument, make sure you also supply a numeric value for the *new-scroll-factor* argument.

:truncate Causes scrolling to be the responsibility of the user, who must press the SCROLL key to see more output.

:wrap Causes new output to appear at the top of the window, rather than at the bottom as in the case of **:scroll** or **:truncate**.

new-scroll-factor The amount by which the window is scrolled when the value of the *new-end-of-page-mode* argument is **:scroll**. Permissible values include integers (number of lines) and ratios (fraction of the screen).

For an overview of **dw:set-default-end-of-page-mode** and related facilities: See the section "Window Substrate Facilities".

tv:set-default-window-size *flavor-name superior existing-windows* &rest *options* *Function*

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)
```

(flavor:method :set-deselected-visibility tv:blinker) *new-visibility* *Method*

Changes the deselected visibility of the blinker.

(flavor:method :set-display-item tv:basic-scroll-window) *item* *Method*

Sets the top-level item of the scroll window to *item*.

(flavor:method :set-edges tv:essential-set-edges) *new-left new-top new-right new-bottom* &optional *option* *Method*

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

(flavor:method :set-edges tv:menu) *new-left new-top new-right new-bottom* *Method*

Sets the edges of the window to the four values supplied as arguments, in pixels relative to the superior window.

(flavor:method :set-fill-p tv:menu) *t-or-nil* *Method*

Sets the menu's fill mode. Thus, use **t** to set the fill characteristic. This message is a special case of the **:geometry:set-geometry** messages.

(flavor:method :set-follow-p tv:blinker) *new-follow-p* *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.

(flavor:method :set-geometry tv:menu) &optional *columns rows inside-width inside-height max-width max-height* *Method*

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.

(flavor:method :set-gray-array-for-inferiors tv:gray-deexposed-inferiors-mixin) *gray* *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".

(flavor:method :set-gray-array-for-unused-areas tv:gray-unused-areas-mixin) *gray* *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".

(flavor:method :set-half-period tv:blinker) *new-half-period* *Method*

Changes the half-period of the blinker.

(flavor:method :set-highlighted-items tv:menu-highlighting-mixin) *list* *Method*

Sets the list of items to be highlighted.

(flavor:method :set-highlighted-values tv:menu-highlighting-mixin) *list* *Method*

Sets the list of items to be highlighted. Refers to the items by value. For instance, if your item-list is an association list, with elements (*string* **.** *symbol*), this message uses *symbol*. This only works for menu items that can be executed without side-effects, not, for example, the **:eval** and **:funcall** kinds.

See the section "**tv:multiple-menu-mixin** Messages".

(flavor:method :set-hysteresis tv:hysteretic-window-mixin) *new-hysteresis* *Method*

Sets the hysteresis of the window, in pixels.

(flavor:method :set-inside-size tv:essential-set-edges) *new-inside-width new-inside-height* &optional *option* *Method*

Sets 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.

(flavor:method :set-io-buffer tv:command-menu) *io-buffer* *Method*

Sets the I/O buffer to which a command-menu sends a command when an item is chosen.

(flavor:method :set-item-list tv:menu) *list* *Method*

Sets the item list of a menu.

(flavor:method :set-items tv:text-scroll-window) *new-items* &optional (*new-top-item* **0**) *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.

(flavor:method :set-label tv:label-mixin) *specification* *Method*

Changes some attributes of the label. *specification* can be anything accepted by the **:label** init option. Any attribute that *specification* doesn't mention retains its old value.

(flavor:method :set-label dw:margin-mixin) *label* *Method*

Provides a new label for a Dynamic Window.

label The label string.

The *label* can be specified with any of the options acceptable to **dw:margin-mixin**.

For an overview of **(flavor:method :set-label dw:margin-mixin)** and related facilities: See the section "Window Substrate Facilities".

(flavor:method :set-label tv:menu) *label* *Method*

Sets the label of a menu.

(flavor:method :set-location si:interactive-stream) *location* *Method*

Sets the value of the scan pointer. This is used for complicated parsing and for error recovery.

(flavor:method :set-margin-choices tv:margin-choice-mixin) *choices* *Method*

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.

(flavor:method :set-margin-components dw:margin-mixin) *new-components* *Method*

Replaces the current margin components of a Dynamic Window with a new set of components.

new-components Specifies a list of the form ((*component-1* [*keys*]) (*component-2* [*keys*]) ... (*component-n* [*keys*])), where *component-x* is one of a set of margin-component flavors and *keys* are zero or more keywords or keyword-value pairs appropriate for the given flavor.

For a list of available margin-component flavors and an example, see the flavor **dw:dynamic-window**.

For an overview of **(flavor:method :set-margin-components dw:margin-mixin)** and related facilities: See the section "Window Substrate Facilities".

(flavor:method :set-margin-space tv:margin-space-mixin) *new-space* *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)

(flavor:method :set-more-p tv:sheet) *more-p* *Method*

If *more-p* is **nil**, turns off `**More**` processing; otherwise, turns it on.

(flavor:method :set-mouse-position tv:essential-mouse) *x y* *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**.

For an example showing the use of **:set-mouse-position**, see the function **dw:tracking-mouse**.

(flavor:method :set-name tv:changeable-name-mixin) *new-name* *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.

:set-notification-mode *new-mode* *Message*

Sent 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 does not 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.
:always-pop-up	Like **:pop-up** except that the window is not first given a choice of whether to accept the message.
: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.

:always-ignore Like **:ignore** except that the window is not first given a choice of whether to accept the message.

nil The same as **:pop-up**.

(flavor:method :set-position tv:essential-set-edges) *new-x new-y* &optional *option* *Method*

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

(flavor:method :set-print-function tv:function-text-scroll-window) *function* *Method*

Sets the printing function of the window to *function*.

(flavor:method :set-print-function-arg tv:function-text-scroll-window) *new-function-arg* *Method*

Sets the object which the window passes as the second argument to the print function.

(flavor:method :set-reverse-video-p tv:sheet) *t-or-nil* *Method*

Enable or disable reverse-video display. Changing this mode inverts all of the bits in the window.

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". 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". Give an argument of **nil** to disable graying.

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

(flavor:method :set-sheet tv:blinker) *new-window* *Method*

Sets 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, moves it to the point *(0,0)*.

(flavor:method :set-size tv:essential-set-edges) *new-width new-height* &optional *option* *Method*

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

(flavor:method :set-size tv:rectangular-blinker) *new-width new-height* *Method*

Sets the width and height of the blinker, in pixels.

(flavor:method :set-size-in-characters tv:sheet) *width-spec height-spec* &optional *option* *Method*

Sets the inside size of the window, according to the two specifications, without changing the position of the upper-left corner. *width-spec* and *height-spec* are interpreted the same way as arguments to the **:character-width** and **:character-height** init options, respectively.

(flavor:method :set-status tv:essential-activate) *new-status* *Method*

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

time:set-local-time &optional *new-time* *Function*

Sets 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 time-base becomes unreliable for one reason or another.

(flavor:method :set-truncate-line-out tv:sheet) *new-value* *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.

(flavor:method :set-variables tv:choose-variable-values-window) *item-list* &optional *dont-set-height* *Method*

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.

(flavor:method :set-viewport-position dw:dynamic-window) *new-left new-top*
Method

Scrolls the window to a specified location in the window's output history. Specify the location in terms of absolute window coordinates.

new-left The x-coordinate for the viewport's left edge.

new-top The y-coordinate for the viewport's top edge.

For an overview of **(flavor:method :set-viewport-position dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :set-visibility tv:blinker) *new-visibility* *Method*

Sets 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".

(flavor:method :set-vsp tv:sheet) *new-vsp* *Method*

Sets the value of *vsp* for this window to *new-vsp*.

(flavor:method :setup tv:choose-variable-values-window) *items label function margin-choices* *Method*

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.

(flavor:method :setup tv:multiple-choice) *item-name keyword-alist finishing-choices item-list* &optional *maxlines* *Method*

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".

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.

tv:sheet-following-blinker *window* *Function*

Takes 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:sheet-force-access (*sheet don't-prepare-sheet*) *body...* *Function*

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:show-partially-visible-mixin *Flavor*

If a window has this flavor mixed in, 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.

(flavor:method :size tv:sheet) *(outside-width outside-height)* *Init Option*

Specifies the outside width and height of the window.

(flavor:method :size tv:sheet) *Method*

Returns two values: the outside width and outside height.

(flavor:method :size-in-characters tv:sheet) *Method*

Returns two values: the inside size in characters, and the inside height in lines. The size of the default character style is used.

(flavor:method :special-choices tv:multiple-menu-mixin) *choice-list* *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.

(flavor:method :stack-group tv:basic-choose-variable-values) *sg* *Init 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.

dw:standard-command-menu-handler *command-name* &rest *args* *Function*

Takes *command-name* and arguments *args* as passed to the command form of a **dw:define-command-menu-handler** form, and does the standard actions for two mouse gestures:

:mouse-left If the command has **:confirm** arguments, read them from the keyboard. Otherwise, run the command with all arguments defaulted.

:mouse-right If the command has any arguments at all, read them from an accept-values menu. Otherwise, just run the command.

For an overview of related topic, see the section "How Command Menus Work".

standard-output *Variable*

In the normal Lisp top-level loop, output is sent to whatever stream is the value of ***standard-output***. Many input functions, including **write** and **write-char**, take a stream argument that defaults to ***standard-output***.

```
(print 'foo) = (print 'foo *standard-output*)
```

The variable ***standard-output*** may be set to a file, for example, rather than an interactive stream, thus redirecting subsequent output to the file:

```
(setq outstream
      (open "myfile" :direction :output))  ;opens myfile.lisp
(setq old-standard-out *standard-output*)  ;save old value
(setq *standard-output* outstream)         ;redirects output
(print 'foo)                               ;prints 'foo in myfile.lisp
(setq *standard-output* old-standard-out)  ;restore *standard-output*
```

It is much better, however, to use `let` to temporarily bind the stream:

```
(with-open-file (outstream "myfile" :direction :output)
  (let ((*standard-output* outstream))  ;redirects output
    (print 'foo))                       ;end of let form restores
                                        ; *standard-output*
   ...                                  ;more forms
)                               ;end of with-open-file closes file
```

By setting ***standard-output*** to a synonym-stream of ***terminal-io***, ***standard-output*** can resume writing to the user console.

zl:standard-output *Variable*

In your new programs, we recommend that you use the variable ***standard-output***, which is the Common Lisp equivalent of **zl:standard-output**. See the variable ***standard-output***.

(flavor:method :start-typeout si:interactive-stream) *type* &optional *spacing*
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.

(flavor:method :status tv:essential-activate) *Method*

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

tv:stream-mixin *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.

(flavor:method :string-in si:interactive-stream) *eof string* &optional (*start* **0**) *end*
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.

(flavor:method :string-length tv:sheet) *string* &optional (*start* **0**) (*end* **nil**) (*stop-x* **nil**) *character-style* (*start-x* **0**) (*max-x* **0**) *Method*

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).

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

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.

(flavor:method :string-out tv:sheet) *string* &optional *(start* **0***) (end* **nil***)* *Method*

Types *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.

(flavor:method :string-style tv:basic-choose-variable-values) *character-style*
Init Option

The character style in which items that are just strings (typically heading lines) are displayed. The default is the system default character style.

dw:suggest *completion-string object* *Function*

Adds an element to a completion table being constructed inside a **dw:completing-from-suggestions** macro. **dw:suggest** is not used independently of this macro.

completion-string The completion string, that is, the fully completed string generated from what the user typed in.

object The object associated with the completion string (and to be returned by **dw:completing-from-suggestions**).

For an overview of **dw:suggest** and related facilities, see the section "Defining Your Own Presentation Types".

(flavor:method :superior tv:choose-variable-values) *window* *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***)**.

(flavor:method :superior tv:sheet) *superior* *Init Option*

Makes *superior* the superior window of the window being created.

: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.

surrounding-output-with-border (&optional *stream* &key (*shape* **:rectangle**) (*thickness* **1**) (*margin* **1**) (*pattern* **t**) *(gray-level* **1***) (opaque* **nil***) (filled* **nil**) *alu (label* **nil***) (label-position* **:bottom***) (label-separator-line* **nil***) (label-separator-line-thickness* **1***) (label-alignment* **:left***) (width* **nil**) (*height* **nil**) (*move-cursor* **t**)) &body *body* *Function*

Binds the local environment such that output generated in the body of the macro is enclosed within a border. The border is sized to just surround the output.

stream The output stream; the default is ***standard-output***.

:shape Specifies the shape of the border; the default is **:rectangle**. Other possible shapes are **:circle**, **:oval**, and **:diamond**.

:thickness Specifies the thickness, in pixels, of the border; the default is 1.

:margin Specifies the minimum whitespace, in pixels, between the border and the enclosed output.

:pattern Specifies the pattern to be used in drawing the border.

Example:

```
(defun pattern-test ()
  (fresh-line)
  (surrounding-output-with-border
    (*standard-output* :shape :rectangle
                       :pattern tv:50%-gray)
    (present tv:selected-window 'tv:window)))
```

If the **:filled** option is **t**, the pattern is drawn throughout the rectangular area and XORed with the unfilled values of the area's pixels.

For more information on how to specify patterns, see the section "Texturing".

:gray-level Specifies the black-to-white level of the border as a ratio or decimal fraction between 0 and 1; the default value is 1, that is, 100% black. This option only takes effect if you specify a **:thickness** of 1 or greater.

:opaque A Boolean option specifying whether pixels already being displayed are cleared (before the border is drawn) or left alone; the default is **t**. **:opaque t** means draw pixels that are off in the background color.

:filled Boolean option specifying whether the shape enclosed by the border is filled; the default is **nil**. If **t**, filling occurs by XORing the turned-on and unfilled values of the pixels in the filled area.

If a pattern is specified by the **:pattern** option, filling occurs by XORing the pattern values and unfilled values of the pixels. In general, the best results are achieved by leaving the **:pattern** option unspecified if you intend to fill.

:alu Specifies the drawing mode for drawing drawing the border. Possible values for this option are:

- **:draw** Pixels in the border are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.
- **:erase** Pixels in the border are turned off, regardless of whether some of the pixels were already off.
- **:flip** Pixels in the border are turned on if they were previously off, and off if they were previously on.

:label A string to be included within the border as a label, or a form that prints a label. Note that if you use a form, you must explicitly print the label, for example:

```
(scl:surrounding-output-with-border (nil :label (princ "ID"))
                                    (princ "Stuff"))
```

:label-position Specifies the position of the label within the surrounding box. The possible values are **:top**: put the label above the presented output; and **:bottom**: put it below. In either case, the label is flush left with the output.

:label-separator-line A Boolean option specifying whether to draw a line between the label and the output. The default is **nil**: no line.

:label-separator-line-thickness Specifies the thickness, in pixels, of the line separating the label and the output.

:label-alignment Specifies how the label for the surrounding border is to be aligned. The possibilities are **:left**, **:right**, and **:center**.

:width Specifies the the maximum width, in pixels, of the border; the default (**nil**) places no limit on the maximum.

:height Specifies the the maximum height, in pixels, of the border; the default (**nil**) places no limit on the maximum.

:move-cursor Boolean option specifying whether a newline is performed at the end of *body*. If this is **t**, the cursor is moved to position *x* = *old-x-position* + 0, *y* = *old-y-position* + *height*; if **nil**, then it is moved to *x* = *old-x-position* + *width*, *y* = *old-y-position* + 0.

Example:

```
(defun shape-test (shape fill-p)
  (fresh-line)
  (surrounding-output-with-border
    (*standard-output* :shape shape
                       :filled fill-p)
    (present tv:selected-window 'tv:window)))
```

To see how the differently shaped borders look, try calling the above with the various shape keywords.

For an overview of **surrounding-output-with-border** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :tab-nchars tv:sheet) *n* *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**.

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.

tv:temporary-choose-variable-values-window *Flavor*

A **tv:choose-variable-values-window** that is exposed temporarily. For an explanation of temporary windows, see the section "Temporary Windows".

tv:temporary-multiple-choice-window *Flavor*

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

A resource of temporary multiple-choice windows. It is used by the **tv:multiple-choose** function.

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:text-scroll-window *Flavor*

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.

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".

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"`.

(flavor:method :top tv:menu) *top-edge* *Init Option*

Top edge of the window specified in pixels, relative to the outside of the superior window.

(flavor:method :top tv:sheet) *top-edge* *Init Option*

Specifies the y-coordinate of the top edge of the window.

tv:top-box-label-mixin *Flavor*

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.

(flavor:method :top-item 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.

tv:top-label-mixin *Flavor*

Like **tv:label-mixin** except that the label is placed at the top of the window by default, instead of the bottom.

(flavor:method :top-margin-size tv:sheet) *Method*

Returns the top margin size of the window in pixels.

dw:tracking-mouse *(*&optional *stream* &key *(:whostate* **"Track Mouse"***) :who-line-documentation-string :who-line-more-documentation-string :multiple-window)* &body *clauses* *Function*

Tracks the mouse in the user process. User-supplied routines are executed when mouse events occur, such as position changes and the pressing or releasing of a button. Note that, of the options listed here, there are two groups whose use is mutually exclusive: the presentation set, including **:presentation**, **:presentation-hold**, and **:presentation-click**; and the mouse-motion set, including **:mouse-motion**, **:mouse-motion-hold**, and **:mouse-click**. Use of the presentation set is preferred. Also note that *presentation* can be **nil** sometimes.

stream The output stream; the default is ***standard-output***.

:whostate Specifies the string displayed in the run-state slot of the status line. The default value is `"Track Mouse"`.

:who-line-documentation-string Specifies the mouse documentation string.

:multiple-window All functions are called with their *x* and *y* coordinates relative to the window. This allows you to track the mouse across different panes.

clauses Keyword-value pairs supplying routines (the values) executed when the mouse event indicated by the keyword occurs. Some keywords provide arguments. Available keywords and their arguments are described below:

:presentation (*presentation*) Smallest presentation under mouse, or **nil**; called when the mouse moves.

:presentation-hold (*presentation*) Same as **:presentation**, but used if a mouse button is still down.

:mouse-motion (*x y*) Position of the mouse; called when the mouse moves.

:mouse-motion-feedback (*x y*)

:mouse-motion-hold (*x y*) Same as **:mouse-motion**, but used if a mouse button is still down.

:who-line-documentation-string () Allows dynamic control of mouse documentation line; called whenever anything changes.

:release-mouse () Called when all mouse buttons are up after some were down.

:keyboard (*char*) Called when user presses a keyboard key (rather than clicking).

:presentation-click (*mouse-char presentation*) Called when a mouse button is pressed. *presentation* is the smallest presentation under mouse, or **nil**; *mouse-char* is the mouse-character object corresponding to the mouse gesture used.

:mouse-click (*mouse-char x y*) Called when a mouse button is pressed. Arguments are the mouse position and the mouse-character object corresponding to the mouse gesture used.

To see the macro in action, try the following example:

```
;;; To run this function create two lisp listeners side by side.
;;; In the first Lisp Listener type the form:
;;;     (setq *LL1* *terminal-io*).
;;; Click left on the second Lisp Listener and enter the form:
;;;     (mouse-1 *terminal-io*).

(defvar *LL1*)
```

```
(defun mouse-1 (window)
 (dw:tracking-mouse (window)
  (:who-line-documentation-string
   ()
   (if (zerop (tv:mouse-buttons))
         "Buttons up"
         "Buttons Down"))
  (:release-mouse ()
                       (format *LL1* "~&Mouse key released"))
  (:mouse-motion (x y)
   (format *LL1* "~&Mouse motion(~d,~d)" x y))
  (:mouse-motion-hold (x y)
   (format *LL1* "~&Mouse motion hold(~d,~d)" x y))
  (:mouse-click (button x y)
   (graphics:draw-rectangle x y (+ x 10) (+ y 10))
   (selector button char-mouse-equal
    (#\mouse-left (format *LL1* "~&Left click"))
    (#\mouse-middle-2
     (return-from mouse-1 "~&That's All Folks!"))))))
```

Here is another example, showing how you can effect "rubber-banding" while drawing a line.

```
(defun input-a-line ()
 (multiple-value-bind (start-x start-y)
    (dw:tracking-mouse
     (t :whostate "Pick starting point"
       :who-line-documentation-string "Put start of line here.")
     (:mouse-click (click x y)
     (unless (eql click #\mouse-l)
      (signal 'sys:abort))
     (return (values x y))))
  (let ((old-x nil) (old-y nil))
   (dw:with-output-recording-disabled
    ()
    (dw:tracking-mouse (t :whostate "Pick end point"
                                :who-line-documentation-string
                                "Put other end of line here.")
     (:mouse-motion (x y)
     (when (and old-x old-y)
      (graphics:draw-line start-x start-y old-x old-y :alu :flip))
     (graphics:draw-line start-x start-y x y :alu :flip)
     (setq old-x x old-y y))
     (:mouse-click (click x y)
     (unless (eql click #\mouse-l)
      (signal 'sys:abort))
     (when (and old-x old-y)
      (graphics:draw-line start-x start-y old-x old-y :alu :flip))
     (return (values start-x start-y x y)))))))
```

```
(defun input-a-line ()
  (multiple-value-bind (start-x start-y)
      (dw:tracking-mouse
        (t :whostate "Pick starting point"
           :who-line-documentation-string "Put start of line here.")
        (:mouse-click (click x y)
         (unless (eql click #\mouse-l)
           (signal 'sys:abort))
         (return (values x y))))
    (dw:tracking-mouse (t :whostate "Pick end point"
                          :who-line-documentation-string
                                "Put other end of line here.")
      (:mouse-motion-feedback (x y)
       (graphics:draw-line start-x start-y x y :alu :flip))
      (:mouse-click (click x y)
       (unless (eql click #\mouse-l)
         (signal 'sys:abort))
       (return (values start-x start-y x y))))))
```

This last example shows the use of still lower-level facilities, such as the messages **(flavor:method :visible-cursorpos-limits dw:dynamic-window)** and **(flavor:method :set-mouse-position tv:essential-mouse)** to monitor and control

the position of the mouse cursor. **:visible-cursorpos-limits** is needed because **dw:tracking-mouse** provides the mouse cursor position in terms of absolute window coordinates. Therefore, cursor positioning and output operations performed in conjuction with **dw:tracking-mouse** must also use absolute window coordinates.

```
(defun line-art ()
  ;; Get the limits of the current window viewport
  ;; in absolute window coordinates
  (multiple-value-bind (x1 y1 x2 y2)
      (send *standard-output* :visible-cursorpos-limits)
    (ignore x2 y2)
    ;; Move mouse cursor to relative window
    ;; coordinates (200 200)
    (send *standard-output* :set-mouse-position
          (+ x1 200) (+ y1 200))
    (dw:tracking-mouse (t)
      ;; Draw lines from cursor to relative (200 200)
      (:mouse-motion (x y)
       (graphics:draw-line (+ x1 200) (+ y1 200) x y))
      (:mouse-click (button x y)
       (ignore x y)
       ;; When Mouse-M is clicked, exit and leave
       ;; mouse cursor at relative (0 200)
       (if (char-mouse-equal button #\mouse-m)
           (return-from line-art
             (values
               (send *standard-output* :set-mouse-position
                     (+ x1 0) (+ y1 200))
               (values)))))))))
```

For an overview of **dw:tracking-mouse** and related facilities, see the section "Programming the Mouse: Writing Mouse Handlers".

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.

(flavor:method :truncate-line-out tv:sheet) *Method*

Returns **t** if the window's truncate line out flag is set, or **nil** if it is not.

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*

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.

cp:turn-command-into-form *command arguments* *Function*

Translates a Command Processor command into an evaluable form by quoting non-self-evaluating elements of arguments.

command The command.

arguments The arguments to the command.

For an overview of **cp:turn-command-into-form** and related facilities: See the section "Managing Your Program Frame".

tv:turn-off-sheet-blinkers *window* *Function*

Sets the visibility of all blinkers on *window* to **:off**.

(flavor:method :tyi si:interactive-stream) &optional *eof-action* *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**; 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.

(flavor:method :tyi-no-hang si:interactive-stream) &optional *eof-action* *Method*

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**; in this case, the method pops up a system menu.

(flavor:method :tyo tv:sheet) *ch* *Method*

Type *ch* on the window, as described in "How Windows Display Characters". Basically, type the character *ch* in the current font at the cursor position, and advance the cursor position.

si:*typeout-default* *Variable*

A keyword that determines how the Command Processor prints help messages. Possible values are those acceptable as the first argument to the **:start-typeout** message to interactive streams:

:insert	The help message, like a notification, is inserted before the current input.
:overwrite	The help message is inserted before the current input, but the next time an **:insert** or **:overwrite** operation is done, this message is overwritten. This is the default.
:append	The help message appears after the current input, which is reprinted after the help message.
:temporary	The help message appears after the current input and disappears when you type the next character.
:clear-window	The window is cleared and the help message appears at the top.

For more information: See the method **(flavor:method :start-typeout si:interactive-stream)**.

(flavor:method :typeout-window tv:essential-window-with-typeout-mixin) (*flavor-name* **.** *options*) *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**. You can set to **nil** to suppress the typeout 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: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).

cp:unparse-command *command-name arguments* &optional (*command-table* **cp:*command-table***) (*acceptably* **t**) *Function*

Returns the input string corresponding to a Command Processor command and its arguments. (The string is created via a call to **present-to-string**.)

command-name The command name (symbol).

arguments The list of command arguments.

command-table The command table containing the named command; the default is the current command table.

acceptably Boolean argument passed through to **present-to-string** and specifying whether the output string can subsequently be parsed by **accept** and used for input.

For an overview of **cp:unparse-command** and related facilities: See the section "Managing Your Program Frame".

dw:unread-char-for-accept *char stream* *Function*

Puts a character back into the input stream. This character will be the next one read by a subsequent call to **dw:read-char-for-accept**.

char The character.

stream The input stream.

For an overview of **dw:unread-char-for-accept** and related facilities, see the section "Defining Your Own Presentation Types".

(flavor:method :unselected-choice-style tv:basic-choose-variable-values) *character-style* *Init 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.

(flavor:method :untyi si:interactive-stream) *ch* *Method*

Returns *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**.

(flavor:method :untyi tv:stream-mixin) *ch* *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, decrements the scan pointer. Otherwise, puts *ch* back into the window's I/O buffer.
:read	Decrements the input editor scan pointer.
:tyi	Puts *ch* back into the window's I/O buffer.

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

(flavor:method :update-item-list 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.

(flavor:method :update-label dw:margin-mixin) *Method*

Causes a new label to be written for a Dynamic Window. The label must have previously been created via the **:delayed-set-label** method: See the method **(flavor:method :delayed-set-label dw:margin-mixin)**.

For an overview of **(flavor:method :update-label dw:margin-mixin)** and related facilities: See the section "Window Substrate Facilities".

(flavor:method :value-style tv:basic-choose-variable-values) *character-style* *Init Option*

The character style in which values of variables are displayed. The default is the system default character style.

(flavor:method :variables tv:basic-choose-variable-values) *item-list* *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".

time:verify-date *day month year day-of-the-week* *Function*

Returns **nil** if the day of the week of the date specified by *day*, *month*, and *year* is the same as *day-of-the-week*; otherwise, returns a string that contains a suitable error message. *year* can be absolute or relative to 1900 (that is, **84** and **1984** both work).

(flavor:method :visible-cursorpos-limits dw:dynamic-window) &optional (*unit* **:pixel**) *Method*

Returns the left, top, right, and bottom limits of the current viewport. The limits are returned as absolute window locations.

unit The unit of measure for the viewport limits; the default is **:pixel**. The alternative is **:character**. The character used is the space character in the window's default character style.

For an example showing the use of **:visible-cursorpos-limits**, see the function **dw:tracking-mouse**.

For an overview of **(flavor:method :visible-cursorpos-limits dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :update-label tv:delayed-redisplay-label-mixin) *Method*

Actually does the **:set-label** operation on the *specification* given by the most recent **:delayed-set-label** message.

(flavor:method :visibility tv:blinker) *symbol* *Init Option*

Sets the initial visibility of the blinker. This defaults to **:blink**.

(flavor:method :vsp tv:menu) *n-pixels* *Init Option*

Sets the vertical spacing between lines in the menu. The default is 2 pixels.

(flavor:method :vsp tv:sheet) *n-pixels* *Init Option*

Initializes the window's *vsp*. It defaults to **2**. The *vsp* is the space between lines: below the lowest descender and above the highest ascender.

(flavor:method :vsp tv:sheet) *Method*

Returns the value of *vsp* for this window.

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.

(flavor:method :who-line-documentation-string 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. You can supply two values to obtain two lines of documentation.

tv:*who-line-function-hook* *Variable*

Allows you to add your own functions to display things in the progress note area of the status line. You set this variable to a function. The function is called in several places in the **:update** routine.

Note: Only one function is allowed and the variable must be reset when the hook function is changed.

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.

tv:who-line-mouse-grabbed-more-documentation *Variable*

The value is the string displayed as the message in the second (lower) mouse documentation line when the mouse has been grabbed (as within a **tv:with-mouse-grabbed**). Inside the body of **tv:with-mouse-grabbed** or **tv:with-mouse-usurped**, you should **setq** this variable to an appropriate string. See the variable **tv:who-line-mouse-grabbed-documentation**.

(flavor:method :width tv:choose-variable-values) *arg* *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.

(flavor:method :width tv:menu) *arg* *Init Option*

Specifies the width of the window in pixels.

(flavor:method :width tv:rectangular-blinker) *n-pixels* *Init Option*

Sets 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.

(flavor:method :width tv:sheet) *outside-width* *Init Option*

Specifies the outside width of the window.

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

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-call-relative (*window* &optional *final-action* &rest *final-action-args*) &body *body* *Function*

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-hacking-menu-mixin *Flavor*

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".

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

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**.

tv:window-pane *Flavor*

An instantiable flavor that includes **tv:pane-mixin** and **tv:window**.

tv:window-with-typeout-mixin *Flavor*

Flavor to mix into a superior window to provide an inferior typeout window.

dw:with-accept-activation-chars (*additional-characters* &key *override*) &body *body* *Function*

Binds local environment to establish additional characters to be used as delimiters of input strings. Predefined activation characters are **#\Return** and **#\End**.

additional-characters A list of characters to be used as additional delimiters.

:override Boolean option specifying whether the characters provided in the *additional-characters* argument are the only delimiters used within the body of the macro. If **t**, the provided characters replace the existing set for the dynamic extent of the macro. The default is **nil**, meaning that the supplied characters are added to the existing set of delimiters.

For an overview of **dw:with-accept-activation-chars** and related facilities, see the section "Defining Your Own Presentation Types".

dw:with-accept-blip-chars (*additional-characters* &key *override*) &body *body* *Function*

Binds local environment to establish additional characters to be used as delimiters of input blips. The characters are additional only if a previous, higher-level call to this macro in a nested structure has established an existing set of delimiters; no predefined set exists.

additional-characters A list of characters to be used as additional delimiters.

:override Boolean option specifying whether the characters provided in the *additional-characters* argument are the only delimiters used within the body of the macro. If **t**, the provided characters replace the existing set for the dynamic extent of the macro. The default is **nil**, meaning that the supplied characters are added to the existing set of delimiters.

For an overview of **dw:with-accept-blip-chars** and related facilities, see the section "Defining Your Own Presentation Types".

dw:with-accept-help *options* &body *body* *Function*

Binds local environment to control HELP-key documentation for input to **accept**.

options A list of option specifications. Each specification is itself a list of the form (<*help-option*> <*help-string*>).

help-option The *help-type* or a list of the form (<*help-type*> <*mode-flag*>). Help types are:

:top-level-help Specifies that *help-string* be used instead of the default help documentation provided by **accept**.

:subhelp Specifies that *help-string* be used in addition to the default help documentation provided by **accept**.

Available modes include:

:append Specifies that the current help string be appended to any previous help strings of this type (top-level help or subhelp). This is the default mode.

:override Specifies that the current help string is the help for this help type; no lower-level calls to **dw:with-accept-help** can override this. (**:override** works from the outside in.)

:establish-unless-overridden Specifies that the current help string be the help text for this help unless a higher-level call to **dw:with-accept-help** has already established a help string for this help type in the **:override** mode.

help-string A string or a function returning a string. If a function, it receives two arguments, the stream and the string-so-far.

Examples:

```
(dw:with-accept-help ((:subhelp "This is a test."))
  (accept 'pathname))

==> You are being asked to enter a pathname.   [ACCEPT did this]
    This is a test.                            [You did this]
    Use c-? or c-/ for a list of possibilities.[Completer did this]

(dw:with-accept-help ((:top-level-help "This is a test."))
  (accept 'pathname))

==> This is a test.                            [You did this]
    Use c-? or c-/ for a list of possibilities.[Completer did this]

(dw:with-accept-help (((:subhelp :override) "This is a test."))
  (accept 'pathname))

==> You are being asked to enter a pathname.   [ACCEPT did this]
    This is a test.                            [You did this]
                                               [Completer did
                                                nothing because
                                                you overrode it]

(define-presentation-type test ()
   :parser ((stream)
            (values (dw:with-accept-help
                ((:subhelp "A test is made up of three things:"))
              (dw:completing-from-suggestions ...)))))
```

```
(accept 'test) ==> You are being asked to enter a test.
                   A test is made up of three things:

;;;use function to provide help string
(dw:with-accept-help (((:top-level-help :override)
                       (lambda (stream string-so-far)
                          (format stream "You are typing
                                          a pathname"))))
   ....)
```

For an overview of **dw:with-accept-help** and related facilities, see the section "Defining Your Own Presentation Types".

dw:with-accept-help-if *cond options* &body *body* *Function*

Conditionally binds local environment to control HELP-key documentation for input to **accept**. Similar to **dw:with-accept-help**, but conditional.

cond The condition.

options A list of option specifications. Each specification is itself a list of the form (<*help-option*> <*help-string*>).

help-option The *help-type* or a list of the form (<*help-type*> <*mode-flag*>). Help types are:

:top-level-help Specifies that *help-string* be used instead of the default help documentation provided by **accept**.

:subhelp Specifies that *help-string* be used in addition to the default help documentation provided by **accept**.

Available modes include:

:append Specifies that the current help string be appended to any previous help strings of this type (top-level help or subhelp). This is the default mode.

:override Specifies that the current help string is the help for this help type; no lower-level calls to **dw:with-accept-help** can override this. (**:override** works from the outside in.)

:establish-unless-overridden Specifies that the current help string be the help text for this help unless a higher-level call to **dw:with-accept-help** has already established a help string for this help type in the **:override** mode.

help-string A string or a function returning a string. If a function, it receives two arguments, the stream and the string-so-far.

This macro is equivalent to the following form:

```
(if <cond>
    (dw:with-accept-help <> body)
    body)
```

For examples, see the dictionary entry for **dw:with-accept-help**.

For an overview of **dw:with-accept-help-if** and related facilities, see the section "Defining Your Own Presentation Types".

with-character-face (*face* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Function*

Binds the local environment such that character output is in the specified face.

face The face to be used for character output, for example, **:bold** or **:italic**.

stream Output stream; the default is ***standard-output***.

:bind-line-height Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t** (put the code in an editor buffer first and change the fonts of the strings to **nil**.):

```
(defun line-height-binder (bind)
  (format t "~&First line")
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (terpri)
  (write-string "Another string")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent though subtle; even more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

To see a list of valid character style faces, evaluate the variable **si:*valid-faces***. For more information on character styles, see the section "Character Styles".

For an overview of **with-character-face** and related facilities, see the section "Character Environment Facilities".

with-character-family (*family* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Function*

Binds the local environment such that character output is in the specified family.

style The family to be used for character output, for example, **:serif** or **:jess**.

stream Output stream; the default is ***standard-output***.

:bind-line-height Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t** (put the code in an editor buffer first and change the fonts of the strings to **nil**.):

```
(defun line-height-binder (bind)
  (format t "~&First line")
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (terpri)
  (write-string "Another string")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent though subtle; even more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

To see a list of valid character style families, evaluate the variable **si:*valid-families***. For more information on character styles, see the section "Character Styles".

For an overview of **with-character-family** and related facilities, see the section "Character Environment Facilities".

with-character-size (*size* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Function*

Binds the local environment such that character output is of the specified size.

size The size of character output, for example, **:very-small** or **:very-large**.

stream Output stream; the default is ***standard-output***.

:bind-line-height Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t** (put the code in an editor buffer first and change the fonts of the strings to **nil**.):

```
(defun line-height-binder (bind)
  (format t "~&First line")
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (terpri)
  (write-string "Another string")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent though subtle; even more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

To see a list of valid character style sizes, evaluate the variable **si:*valid-sizes***. For more information on character styles, see the section "Character Styles".

For an overview of **with-character-size** and related facilities, see the section "Character Environment Facilities".

with-character-style (*style* &optional (*stream* **t**) &key *bind-line-height*) &body *body* *Function*

Binds the local environment such that character output is in the specified style.

style List of the form (*:family :face :size*) specifying character style.

stream Output stream; the default is ***standard-output***.

:bind-line-height Boolean option specifying whether the height, in pixels, of the line containing the character output is based on the size of the default character style or of the style specified in the macro. Whether you specify **t** or **nil** (the default) depends on the context of the output. To see the difference, run the following function first with **nil**, then with **t** (put the code in an editor buffer first and change the fonts of the strings to **nil**.):

```
(defun line-height-binder (bind)
  (format t "~&First line")
  (format t "~&Foo")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (write-string "bar"))
  (write-string "baz")
  (terpri)
  (write-string "Another string")
  (with-character-style ('(:fix :roman :very-large) t
        :bind-line-height bind)
    (format t "~&Frob one~%Frob two~%"))
  (format t "Last line"))
```

In this example, the difference is apparent though subtle; even more subtle are the differences produced when only the character family or face is changed, as opposed to its size.

The height of the default character style is determined from the height of the space character.

A character style specifies three style components: family, face, and size. To see lists of valid families, faces, and sizes, evaluate the variables **si:*valid-families***, **si:*valid-faces***, and **si:*valid-sizes***. (The same information is presented in another section; see the section "Available Character Styles".) Note that not all permutations of family, face, and size are legitimate character styles.

A partially specified character style is merged against the default character style for the window. (See the init option **(flavor:method :default-character-style tv:sheet)** and see the section "Merging Character Styles".) Consider the following example (which has to be compiled):

```
(defun character-style-merge ()
  (dw:with-own-coordinates (t)
    (with-character-style ('(nil :bold :large) t)
      (graphics:draw-string "CURRENT DATA" 100 100
                            :alu :flip))))
```

The character style specification in the above example only specifies two components, face and size. The **nil** supplied in the family component slot means that the default family for the window is used. If you wish to keep the defaults for two of the components, then you can use one of the following macros:

with-character-family
with-character-face
with-character-size

You can determine the character style corresponding to a particular TV font by using the **si:backtranslate-font** function. (See the function **si:backtranslate-font**.)

Example:

```
(si:backtranslate-font 'fonts:bigfnt)
#<CHARACTER-STYLE FIX.ROMAN.VERY-LARGE 260250707>
#<STANDARD-CHARACTER-SET 260000540>
0
#<B&W-SCREEN-DISPLAY-DEVICE 260302767>
```

The example shows that **fonts:bigfnt** corresponds to the **(:fix :roman :very-large)** character style.

For more information on character styles, see the section "Character Styles".

For an overview of **with-character-style** and related facilities, see the section "Controlling Character Style".

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

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 function **with-input-editing-options**. See the function **with-input-editing-options-if**.

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 cl:*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))))))))
```

For an overview of this and related functions: See the section "Invoking the Input Editor".

Provides a convenient way of invoking the input editor for use by a reading function, and establishes a context in which input editing should be provided.

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

keyword is reserved for future use.

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

with-input-editing-options *options* &body *body* *Function*

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 an overview of this and related functions: See the section "Invoking the Input Editor"

For a list of input editor options: See the section "Input Editor Options". See the function **with-input-editing-options-if**.

In the following example, the user is prompted for a Lisp expression. One input editor option is specified, and that option specifies a prompt.

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

To write a reading function that invokes the input editor, you should use the **with-input-editing macro**.

with-input-editing-options-if *cond options* &body *body* *Function*

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 an overview of this and related forms:

See the section "Invoking the Input Editor".

For a list of input editor options: See the section "Input Editor Options". See the function **with-input-editing-options**.

tv:with-mouse-and-buttons-grabbed &body *body* *Function*

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* *Function*

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.

tv:with-mouse-grabbed *Function*

A **tv:with-mouse-grabbed** special form has only a body:

```
(tv:with-mouse-grabbed
  form1
  form2)
```

The forms inside are evaluated with the mouse grabbed.

tv:with-mouse-grabbed-on-sheet (&optional (*sheet* **'self**)) &body *body* *Function*

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-usurped *Function*

A **tv:with-mouse-usurped** special form has just a body:

```
(tv:with-mouse-usurped
  form1
  form2)
```

The forms inside are evaluated with the mouse usurped. The system does not handle mouse moves at all when the mouse is usurped; it does not even maintain the mouse *x-y* position.

tv:with-notification-mode (*new-mode* &optional *stream*) &body *body* *Function*

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**.

dw:with-output-as-presentation (&key *stream object type form location single-box* (*allow-sensitive-inferiors* **t**)) &body *body* *Function*

Outputs an object as a presentation object; in effect, allows you to rewrite the printer function (used locally) for a presentation type. The following example illustrates this point:

```
(defun present-this-as-that (this that
        &optional (stream *standard-output*))
  (send stream :clear-history)
  (dw:with-output-as-presentation (:single-box t
        :stream stream :type that :object this)
    (send stream :draw-circle 250 200 25)
    (send stream :draw-circle 270 200 25)))
```

Try calling this function with `"ABC"` as the first argument and `'string` as the second. Now, do `(accept 'string)` and click on the graphic.

Note the `:single-box t` option used in the above example. This is nearly always appropriate when using this macro for graphic presentations.

Following are the keyword arguments recognized by **dw:with-output-as-presentation**. Note that some of them are required.

:stream Specifies stream on which the object is presented; the default is ***standard-output***.

:object Specifies the presentation object of the output presentation. If you do not use this option, then you must supply either the **:form** or **:location** option.

:type Specifies the type of the presentation. You must provide this option.

:form Specifies a form that can be passed to **setf** to store a new value in place of the current output value. This option and **:location** are mutually exclusive.

The form supplied for this option is used by a predefined, side-effecting mouse handler (available on `c-m-`Right) to modify the contents of structure slots.

:location Specifies a locative that can be used to store a new value in place of the current output value. This option and **:form** are mutually exclusive.

The locative supplied for this option is used by a predefined, side-effecting mouse handler (available on `c-m-`Right) to modify the contents of structure slots.

:single-box Specifies that mouse-sensitivity of objects output in a series of inferior calls to this form be indicated by a single, large box for highlighting rather than the sum of all the individual boxes. This option is used mostly with graphic presentations.

:allow-sensitive-inferiors Boolean option specifying whether nested calls to **present** or **dw:with-output-as-presentation** from inside this presentation — for example, when presenting the individual elements of a Lisp list — generate presentation objects. The default is **t**.

Example:

```
(defun sensitive-inferior-test (sensitive-p)
  (let ((fl 'dw:dynamic-window))
     (dw:with-output-as-presentation
       (:object fl
        :type 'sys:flavor-name
        :allow-sensitive-inferiors sensitive-p)
       (format t "The flavor ~S." fl))))
```

Try calling `sensitive-inferiors-test` with **t**, then **nil**. You should find that in the first case both the entire presentation and the flavor name are individually sensitive depending on where you have the mouse cursor; in the latter case, only the entire presentation is sensitive.

For an overview of **dw:with-output-as-presentation** and related facilities; See the section "Using Presentation Types for Output".

(flavor:method :with-output-recording-disabled dw:dynamic-window) *continuation xstream* *Method*

Disables output recording on a specified window for a specified continuation.

continuation The continuation, a function of one argument, the output stream.

xstream The window whose output recording is disabled.

Example:

```
(defun draw-circles (stream)
  (loop repeat 50
        do
    (graphics:draw-circle
      (random 500)
      (random 500) 10 :stream stream)))

(send *standard-output*
      :with-output-recording-disabled
      #'draw-circles *terminal-io*)
```

See also the macros, **dw:with-output-recording-disabled** and **dw:with-output-recording-enabled**.

dw:with-output-to-presentation-recording-string (*stream*) &body *body* *Function*

Binds the local environment to output to a string, the way **with-output-to-string** does, except that the string records presentations resulting from calls to **present** and **dw:with-output-as-presentation**. If the resulting string is subsequently printed (via **princ** or **present**) to a stream supporting presentations, the recorded presentations are re-presented to that stream.

stream The output stream; the default is ***standard-output***.

dw:with-output-to-presentation-recording-string is distinguished from **present-to-string** as follows:

w-o-to-p-r-string	**present-to-string**
Returns a presentation-recording string	Returns an ordinary string
Arbitrary body writing to string	Single object to be presented

Example:

```
(defun test-pr-string ()
  (let ((string (dw:with-output-to-presentation-recording-string
                  (*standard-output*)
                  (dolist (symbol '(butcher baker candlestick-maker))
                    (write-string " ")
                    (present symbol 'symbol)))))
    (princ string)
    (accept 'symbol)))
```

For an overview of **dw:with-output-to-presentation-recording-string** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

dw:with-output-truncation (&optional *stream* &rest *options*) &body *body* *Function*

Binds the local environment to allow textual output to extend beyond the bottom and right borders of the output window.

stream The output stream; the default is ***standard-output***.

To access text extending beyond the margins of the output window, the window needs vertical and horizontal scroll bars. For information on how to equip Dynamic Windows with scroll bars (and other margin components), see the flavor **dw:dynamic-window**.

options Two options are available:

:horizontal Boolean option specifying whether truncation occurs in the horizontal dimension. If you do not specify any option, they both default to **t**. If you specify either **:horizontal** or **:vertical** to be **t**, then the other option defaults to **nil**.

"Truncation" here means that output exceeding the width of the window extends beyond the right margin of the current window viewport; the margin truncates the user's view of the output. If **nil**, the output wraps to the next line.

:vertical Boolean option specifying whether truncation occurs in the vertical dimension. The default, when neither option is specified, is **t**, meaning that output exceeding the

height of the window extends below the bottom margin of the current window viewport. If you specify either **:horizontal** or **:vertical** to be **t**, then the other option defaults to **nil**.

Example:

```
(defun truncation-test (t-or-nil)
  (dw::with-output-truncation (t :horizontal t-or-nil)
    (loop repeat 100 do (write-char #\a))))
```

For an overview of **dw:with-output-truncation** and related facilities, see the section "Presenting Formatted Output".

dw:with-own-coordinates (&optional *stream* &key *left top right bottom* (*clear-window* **t**) (*erase-window* **nil**) (*enable-output-recording* **t**)) &body *body* *Function*

Binds the local environment such that output to a Dynamic Window is in a refreshed area, and the coordinate system is relative to the current viewport, not the window's origin.

dw:with-own-coordinates is only appropriate when you are dealing with absolute constant coordinates. If you are doing anything relative, it is not for you. All messages like **:read-cursorpos** and **:mouse-position** deal in plane coordinates. If you are doing anything relative to those values, the right thing will happen. Do not be put off by the fact that the *y* coordinates keep getting larger as the history gets longer — big numbers are nothing to be afraid of.

Examples:

A valid use:

```
(dw:with-own-coordinates ()
  (graphics:draw-circle 100 100 100))
```

Some things that do not require it.

Cursor relative graphics block:

```
(graphics:with-room-for-graphics ()
  (graphics:draw-triangle 0 0 200 0 150 45))
```

Mouse relative graphics:

```
(tracking-mouse line drawing example above)
```

stream The output stream; the default is ***standard-output***.

:left Specifies the x-coordinate at the beginning of the area to be erased when the **:erase-window** option is **t**.

:top Specifies the y-coordinate at the beginning of the area to be erased when the **:erase-window** option is **t**.

:right Specifies the x-coordinate at the end of the area to be erased when the **:erase-window** option is **t**.

:bottom Specifies the y-coordinate at the end of the area to be erased when the **:erase-window** option is **t**.

:clear-window Boolean option specifying whether the window is scrolled to a clear area before output begins; the default is **t**.

:erase-window Boolean option specifying that the output window be erased before output begins; the default is **nil**.

If this option is **t**, use the **:left**, **:top**, **:right**, and **:bottom** keywords to specify the coordinates of the area to be erased. If no coordinates are specified, they default to the coordinates of the current viewport. Output begins at the top of the erased area.

:enable-output-recording Boolean option specifying whether output is retained in the output history of the window; the default is **t**.

This option is useful with animated graphic presentations that, because of the time required for redisplay, can impede scrolling through a window's history.

dw:with-presentation-input-context (*presentation-type* &rest *options*) (&optional (*blip-var* **'dw::.blip.**)) *non-blip-form* &body *blip-cases* *Function*

Binds local environment to the input context of a specified presentation type. (This essentially establishes mouse sensitivity for that type, and is one of the building blocks for **accept**.) The body (*non-blip-form*) is executed. If no mouse gestures are made by the user during execution of the body, this form returns the value of the *non-blip-form*. If the user clicks on a presentation of an appropriate type, the corresponding *blip-cases* form is executed, with the resulting presentation blip bound as the value of *blip-var*.

presentation-type The presentation type establishing the new input context. This may be a compound type incorporating more than one primitive type.

options Two predefined keyword options are available:

:stream Specifies the input stream; the default is ***standard-input***.

:inherit Boolean option specifying whether to inherit an existing input context or to establish a new root node; the default is **t**.

You may use any additional keywords you want.

blip-var The symbol to bind to the blip generated by clicking on an object of the specified type while in the context.

non-blip-form The body form to execute inside the established input context.

blip-cases A case statement clause list. The keys are presentation types. The clause whose key matches the presentation type of the blip is executed, with the *blip-var* bound to the blip.

The presentation types available for use as keys are limited to the type specified by the *presentation-type* argument or, in the case of a compound presentation type (for example, **or**), the types specified; and the type or types inherited in the case of a nested use of this macro.

For an overview of **dw:with-presentation-input-context** and related facilities: See the section "Presentation Input Context Facilities".

dw:with-presentation-input-editor-context (*stream presentation-type . options*) (&optional (*blip-var* **'dw::.blip.**) *start-loc-var*) *non-blip-form* &body *blip-cases*

Function

Establishes an input context around a call to the input editor to read keyboard input from the user. The body (*non-blip-form*) is executed. If no mouse gestures are made by the user during execution of the body, this form returns the value of the *non-blip-form*. If the user clicks on a presentation of an appropriate type, the resulting presentation blip is bound as the value of *blip-var*; the current location in the input buffer is bound as the value of *start-loc-var*; and the corresponding *blip-cases* form is executed.

accept uses this mechanism to establish an input context for the presentation type being read. This is one of the substrate functions used to build **accept**. Most programs simply want to call **accept**, instead of working at this low level.

stream The input stream; the default is ***standard-input***.

presentation-type The presentation type establishing the new input context. This may be a compound type incorporating more than one primitive type.

options One predefined keyword option is available:

:inherit Boolean option specifying whether to inherit an existing input context or to establish a new root node; the default is **t**.

You may use any additional keywords you want.

blip-var The symbol to bind to the blip generated by clicking on an object of the specified type while in the context.

start-loc-var The symbol to bind to the input buffer location at the time the presentation blip is received.

non-blip-form The body form to execute inside the established input context.

blip-cases A case statement clause list. The keys are presentation types. The clause whose key matches the presentation type of the blip is executed, with the *blip-var* bound to the blip.

The presentation types available for use as keys are limited to the type specified by the *presentation-type* argument or, in the case of a compound presentation type (for example, **or**), the types specified; and the type or types inherited in the case of a nested use of this macro.

This macro is built on **dw:with-presentation-input-context**, to which it is similar:

```
(dw:with-presentation-input-editor-context (stream type)
                                           (blip-var)
    body-form
  blip-clauses)
```

is the same as

```
(with-input-editing (stream)
  (dw:with-presentation-input-context
    (type :stream stream)
    (blip-var)
      body-form
    blip-clauses))
```

For an overview of **dw:with-presentation-input-editor-context** and related facilities: See the section "Presentation Input Context Facilities".

dw:with-presentation-type-arguments (*type-name type*) &body *body* *Function*

Binds local environment such that the arguments in a presentation-type specification are lexically available within the body of the macro.

type-name The name of the presentation type whose arguments are to be used, for example, `pathname`.

type The type specification, for example, `'((pathname) :format :directory :direction :write)`.

The *type-name* argument is known at compile time. It fixes the template for decoding the arguments of the particular *type* specification passed to the macro at runtime.

Example:

```
(define-presentation-type wood ((&key tree grade)
                                &key show-price)
  :printer ((wood stream &key type)
            (format stream "~A [wood ~A, ~A~:[~; ~2D cents~]]"
                    wood tree grade show-price
                                (compute-wood-price type))))

(defun compute-wood-price (presentation-type)
  (dw:with-presentation-type-arguments (wood presentation-type)
    (let ((base-price
            (ecase tree
             (mahogany 69)
             (pine 12)
             (teak 75)))
          (grade-multiplier
            (ecase grade
             (firsts-and-seconds 1.3)
             (firewood .2))))
      (* base-price grade-multiplier))))

(compute-wood-price '((wood :tree teak :grade firewood)
                      :show-price t)) ==>
15.0
```

See the section "Defining Your Own Presentation Types".

For an overview of **dw:with-presentation-type-arguments** and related facilities, see **dw:with-type-decoded**.

dw:with-redisplayable-output (&key *stream cache-value unique-id* (*cache-test* **#'eql**) *copy-cache-value* (*id-test* **#'eql**)) &body *body*

Function

Introduces a caching point for incremental redisplay. If this is used outside the dynamic scope of an incremental redisplay, it has no particular effect. However, when incremental redisplay is occurring, the supplied *cache-value* is compared with the value stored in the cache identified by *unique-id*. If the values differ, the code in *body* runs, and *cache-value* is saved for next time. If the cache values are the same, the code in *body* is not run, because the current output is still valid.

:stream Specifies the output stream; the default is ***standard-output***.

:cache-value Specifies the value to be compared each time against the value stored in the cache.

:unique-id Identifies the particular incremental redisplay cache. This may be any object, as long as it is unique with respect to the *id-test* predicate among all such ids in the current incremental redisplay. The default is that there is no id, not that **nil** is the id.

:cache-test Specifies the test used to compare *cache-value* against the value saved in the cache. The default is **eql**.

:copy-cache-value Boolean option specifying whether to **copy-seq** the cache value before saving it in the cache. Use this, for example, when the cache value is a stack list which must be copied before being stored away somewhere.

:id-test Specifies the test used to locate the cache identified by *unique-id* among the caches used by the current incremental redisplay. The default is **eql**.

dw:with-redisplayable-output is one of a number of facilities used to do incremental redisplay. For examples, see the file SYS:EXAMPLES;INCREMENTAL-REDISPLAY.LISP.

For an overview of **dw:with-redisplayable-output** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

dw:with-replayable-output (&rest *parameters*) &body *body* *Function*

Binds the local environment such that all of the output generated by *body* becomes a single, replayable presentation.

The code in *body* is snapshotted (using **dw:named-value-snapshot-continuation**) so that it can be rerun (replayed) in an altered environment; this results in a new printed representation. The user specifies the new output parameters at runtime via the Edit Viewspecs mouse handler. This handler is invoked by clicking s-sh-Middle on a replayable presentation.

parameters A list of variable specifications in the style of **dw:accept-variable-values**. That is, each item in the list is a list of the form (*variable-name prompt-string presentation-type*).

The parameters are used to construct an **dw:accept-variable-values** menu which pops up in response to the mouse gesture Edit Viewspecs (s-sh-Middle). The values of the variables then be changed by the user, and the presentation rerun with the new values.

Example:

```
;;; Compile and run this code, then Edit Viewspecs by
;;; clicking s-sh-Middle on its output.
(defun wrpo ()
  (fresh-line)
  (let ((style '(:fix :roman :normal))
        (width 50)
        (start 1))
    (dw:with-replayable-output
      ((style "Character style" character-style)
       (width "Width in characters" ((integer 5 120)))
       (start "Starting from" integer))
      (with-character-style (style)
        (let ((fill-width
                (* width (send *standard-output* :char-width))))
          (filling-output (() :fill-column fill-width)
            (loop repeat 50
                  for i from start
                  do (format T " ~r" i))))))))
```

dw:with-replayable-output is similar to **dw:with-output-as-presentation** in the sense that it lets you define a presentation-type printer "on the fly", that is, not as part of a presentation type. In the case of **dw:with-replayable-output**, you are writing a printer that can be modified by the user at runtime, via the Edit Viewspecs handler. This is not the only way to provide users with the ability to alter displayed presentations; you can use the **:viewspec-choices** option to **define-presentation-type** to provide the same capability with regard to all presentations of the defined type. See the section "**:viewspec-choices** Option to **define-presentation-type**".

dw:with-resortable-output is a specialization of **dw:with-replayable-output** for reordering and redisplaying lists: See the function **dw:with-resortable-output**.

For an overview of **dw:with-replayable-output** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

dw:with-resortable-output ((*list key* &key *copy-of*) &rest *sort-clauses*) *other-parameters* &body *body* *Function*

Binds the local environment such that all of the output generated by *body* becomes a single, replayable presentation. The list can be output in one of several orders specified by sorting predicates. Which sorting predicate is used can be specified by users at runtime via the Edit Viewspecs mouse handler, available on `s-sh-Middle`.

list A variable holding the sequence of items for output.

key A variable holding an identifier for selecting which of the *sort-clauses* is used.

:copy-of Specifies a list to be copied and sorted instead of *list*. The value is copied by **copy-seq**. Typically, the value of this option is *list*. Use it when you do not want the order of the original list to be destroyed by sorting.

sort-clauses An **ecase** body, selecting on sort keys (the value of *key*), and returning a sort predicate.

other-parameters Other parameters included in the *parameters* argument passed to **dw:with-replayable-output**: See the function **dw:with-replayable-output**. These and the sorting options appear in the **dw:accept-variable-values** display created by the Edit Viewspecs handler.

To see this macro in action, execute the Command Processor command Show Processes or Show Directory. Both use **dw:with-resortable-output** and produce output resortable via the Edit Viewspecs handler (s-sh-Middle).

Another example:

```
(defun sortable-output ()
  (let ((data (make-array 10))
        (how :alpha)
        (style '(:swiss nil nil)))
    (dotimes (i 10)
      (setf (aref data i)
            (list i (format nil "~r" i))))
    (dw:with-resortable-output
      ;; list and key
      ((data how)
       ;; sort clauses
       (:alpha
         (lambda (x y)
           (string-lessp (second x) (second y))))
       (:length
         (lambda (x y)
           (< (string-length (second x))
              (string-length (second y)))))
       (:number
         (lambda (x y)
           (< (first x) (first y)))))
      ;; other parameter
      ((style "Character style" character-style))
      ;; body
      (with-character-style (style)
        (format t "~&Here come the data, sorted by ~(~a~): " how)
        (format-textual-list data
                             (lambda (x stream)
                               (princ (second x) stream)))))))
```

For an overview of **dw:with-resortable-output** and related facilities: See the section "Displaying Output: Replay, Redisplay, and Formatting".

tv:with-terminal-io-on-typeout-window (*window wait-for-space-p*) &body *body* *Function*

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.

dw:with-type-decoded (*type-name-var* &optional *data-args-var presentation-args-var*) *type* &body *body* *Function*

Binds local environment such that the type-name and, optionally, arguments in a presentation-type specification are bound to variables lexically available within the body of the macro.

type-name-var Symbol to bind the type-name of the presentation type.

data-args-var Symbol to bind to a list of the data arguments of the presentation type.

presentation-args-var Symbol to bind to a list of the presentation arguments of the presentation type.

Example:

```
(defun with-type-decoded-test ()
   (dw:with-type-decoded (type-name data-args pres-args)
                         '((integer 1 10) :base 8
                           :description "Integer between 1 and 10")
     (format t "~2%Type:  ~A
                ~%Data Arguments:  ~A
                ~%Presentation Arguments:  ~A"
                type-name data-args pres-args)))

(with-type-args-test) ==>

Type:  INTEGER

Data Arguments:  (1 10)

Presentation Arguments:  (BASE 8 DESCRIPTION Integer between 1 and 10)
```

See the section "Defining Your Own Presentation Types".

For an overview of **dw:with-type-decoded** and related facilities: See also: **dw:with-presentation-type-arguments**.

with-underlining (&optional *stream* &key (*underline-whitespace* **t**)) &body *body* *Function*

Binds the local environment such that character output is underlined.

stream Output stream; the default is ***standard-output***.

:underline-whitespace Boolean option specifying whether whitespace is underlined (in addition to characters); the default is **t**. If the output you are underlining contains any newline characters, we recommend that you set this option to **nil**.

Example:

```
(defun underline-example ()
  (fresh-line)
  (with-underlining ()
    (princ 12345)
    (sleep 2)
    (princ 56789)))
```

For an overview of **with-underlining** and related facilities: see the section "Controlling Line Output".

(flavor:method :x tv:menu) *arg* *Init Option*

Specifies the left edge of the menu in pixels, relative to the outside of the superior window.

(flavor:method :x tv:sheet) *left-edge* *Init Option*

Specifies the x-coordinate of the left edge of the window.

(flavor:method :x-pos tv:blinker) *x* *Init Option*

Along with the **:y-pos** init option, sets 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.

(flavor:method :x-scroll-position dw:dynamic-window) *Method*

Returns four values:

1. The absolute location of the current viewport's left edge.

2. The viewport's horizontal extent.

3. The window's minimum x-coordinate (typically `0`).

4. The absolute location of the viewport's right edge.

For an overview of **(flavor:method :x-scroll-position dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :x-scroll-to dw:dynamic-window) *position type* *Method*

Scrolls the window to a specified x-coordinate.

position The x-coordinate to scroll to.

type The type of scrolling operation. Three possibilities exist:

:absolute The *position* argument specifies an absolute window location.

:relative The *position* argument specifies a location, in pixels, relative to the current position of the cursor.

:relative-jump The *position* argument specifies a location, in characters, relative to the current position of the cursor. The width of a character in pixels depends on the default character style for the window; the width of the space character is used.

For an overview of **(flavor:method :x-scroll-to dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :y tv:menu) *arg* *Init Option*

Specifies the top edge of the menu in pixels, relative to the outside of the superior window.

(flavor:method :y tv:sheet) *top-edge* *Init Option*

Specifies the y-coordinate of the top edge of the window.

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 one of the characters "y" or "Y" or "SPACE", or **nil** if the answer was one of the characters "n" or "N" or "RUBOUT".

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**.

```
(defun show-directory (&optional dirname)
  (when (not dirname)
    (if (y-or-n-p "Use your home directory? ")
        (setq dirname (user-homedir-pathname))
        (return-from show-directory nil)))
  (dolist (path (directory dirname))
    (format t "~&~A~%" path)))
=> SHOW-DIRECTORY

(show-directory)
Use your home directory? (Y or N) Y
foo.lisp
junk.text
=> NIL
```

zl:y-or-n-p &optional *message* (*query-io* **zl:query-io**) *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 one of the characters "y" or "Y" or "SPACE", or **nil** if the answer was one of the characters "n" or "N" or "RUBOUT".

Asks the user a question whose answer is either "yes" or "no". It types out a message (if supplied), reads a single character (Y or N), and returns **t** if the answer was one of the characters "y" or "Y" or "SPACE", or **nil** if the answer was one of the characters "n" or "N" or "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) **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.
```

(flavor:method :y-pos tv:blinker) *y* *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.

(flavor:method :y-scroll-position dw:dynamic-window) *Method*

Returns four values:

1. The absolute location of the current viewport's top edge.
2. The viewport's vertical extent.
3. The window's minimum y-coordinate (typically 0).
4. The absolute location of the viewport's bottom edge.

For an overview of **(flavor:method :y-scroll-position dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

(flavor:method :y-scroll-to dw:dynamic-window) *position type* *Method*

Scrolls the window to a specified y-coordinate.

position The y-coordinate to scroll to.

type The type of scrolling operation. Three possibilities exist:

:absolute The *position* argument specifies an absolute window location.

:relative The *position* argument specifies a location, in pixels, relative to the current position of the cursor.

:relative-jump The *position* argument specifies a location, in lines, relative to the current position of the cursor. The height of a line in pixels depends on the default character style for the window.

For an overview of **(flavor:method :y-scroll-to dw:dynamic-window)** and related facilities, see the section "Presenting Formatted Output".

cp:yank-and-read-full-command *numeric-arg-p numeric-arg* *Function*

The c-m-Y Command Processor command accelerator. It yanks back the last command typed for editing.

cp:yank-and-read-full-command is a function that is suitable for use as a command-accelerator's function. However, the easiest way to make use of this facility is to have the command tables in your applications that use accelerator characters inherit from "Colon Full Command".

For an overview of **cp:yank-and-read-full-command** and related facilities: See the section "Managing Your Program Frame".

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**, 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** whould be used for unanticipated or important questions, which is why it requires a multiple-action sequence to answer it.

Writes out a message supplied in *format-string*, just as though it were the control-string to *format*, then reads a newline terminated word, which should be either 'yes' or 'no', in upper or lower case.

```
(defvar *the-hash-tables* '())

(defun clear-the-hash-tables ()
  (when (yes-or-no-p "~&Clear all hash tables? ")
    (mapc #'clrhash *the-hash-tables*)))

(push (make-hash-table) *the-hash-tables*)
(push (make-hash-table) *the-hash-tables*)

(clear-the-hash-tables)
Clear all hash tables? yes
→ (#<Hash-Table 32432> #<Hash-Table 32497>)
```

See Also: CLtL 408, **write, y-or-n-p**

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

Asks 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
```

alist-member (&key *alist*) &key *convert-spaces-to-dashes* **nil** *Presentation Type*

Type for accepting or presenting an association list item.

:alist Data option specifying the list of items. The usual form of item is a dotted pair of the print string and its object: ((*String-1* . *object-1*) (*string-2* . *object-2*) ... (*string-n* . *object-n*)).

Alternatively, items can be in the "general list" form. See the section "The Form of a Menu Item". One of the advantages of this form is that documentation for each item can be added that will appear if the user asks for help (presses the HELP key) during an **accept** of this type. Documentation is specified with the **:documentation** keyword. See the examples section of **alist-member**.

Two other keywords are permitted in an item list. The first is **:style**, specifying the character style of the presented item.

The second is **:selected-style**. This keyword may only be used when **alist-member** is part of a **dw:accepting-values** function. It specifies the character style of the item when it is selected, that is, after it has been clicked on. The **:selected-style** defaults to the boldface version of the unselected style.

:convert-spaces-to-dashes Presentation option specifying whether spaces in the print string should be converted to dashes; the default is **nil**. This option can be used to avoid space overloading when **alist-member** is being used with the command processor.

Examples:

```
(accept '((alist-member :alist (("Item 1" . a) ("Item 2" . b)))
          :convert-spaces-to-dashes t)) ==>
Enter Item-1 or Item-2: Item-2
B
((ALIST-MEMBER :ALIST (("Item 1" . A) ("Item 2" . B)))
:CONVERT-SPACES-TO-DASHES T)
(present 'b '((alist-member :alist (("Item 1" . a) ("Item 2" . b)))
              :convert-spaces-to-dashes t)) ==>
Item-2
#<DISPLAYED-PRESENTATION 444272462>
```

```
(defun filter-alist-example ()
  (let ((operator-alist
         '(("Gaussian" :value :gauss
            :documentation "low-pass filter")
           ("Laplacian, HP" :value :lpl-hp
            :documentation "high-pass filter")
           ("Laplacian, ED" :value :lpl-ed
            :documentation "edge detector")
           ("Roberts" :value :rbts
            :documentation "edge detector")
           ("Prewitt, Hz" :value :prw-hz
            :documentation "horizontal edge detector")
           ("Prewitt, Vt" :value :prw-vt
            :documentation "vertical edge detector")
           ("Sobel, Hz" :value :sbl-hz
            :documentation "horizontal edge detector")
           ("Sobel, Vt" :value :sbl-vt
            :documentation "vertical edge detector"))))
    (accept `((alist-member :alist ,operator-alist)
              :description "a 2-dimensional image filter"))))
(filter-alist-example) ==>
Enter a 2-dimensional image filter: HELP
You are being asked to enter a 2-dimensional image filter.

These are the possible 2-dimensional image filters:
Gaussian        low-pass filter
Laplacian, ED   edge detector
Laplacian, HP   high-pass filter
Prewitt, Hz     horizontal edge detector
Prewitt, Vt     vertical edge detector
Roberts         edge detector
Sobel, Hz       horizontal edge detector
Sobel, Vt       vertical edge detector
```

```
Enter a 2-dimensional image filter: Laplacian, HP
:LPL-HP
((ALIST-MEMBER :ALIST
  (("Gaussian" :VALUE :GAUSS :DOCUMENTATION
       "low-pass filter")
   ("Laplacian, HP" :VALUE :LPL-HP :DOCUMENTATION
       "high-pass filter")
   ("Laplacian, ED" :VALUE :LPL-ED :DOCUMENTATION
       "edge detector")
   ("Roberts" :VALUE :RBTS :DOCUMENTATION
       "edge detector")
   ("Prewitt, Hz" :VALUE :PRW-HZ :DOCUMENTATION
       "horizontal edge detector")
   ("Prewitt, Vt" :VALUE :PRW-VT :DOCUMENTATION
       "vertical edge detector")
   ("Sobel, Hz" :VALUE :SBL-HZ :DOCUMENTATION
       "horizontal edge detector")
   ("Sobel, Vt" :VALUE :SBL-VT :DOCUMENTATION
       "vertical edge detector")))
 :DESCRIPTION "a 2-dimensional image filter")
```

Because the prompt generated by **accept** for input of **alist-member** items can sometimes be awkward, you may want to use the meta-presentation argument **:description** to change it. (See the section "The Presentation Type System: an Overview".) This was done in the `(filter-alist-example)` above.

The filter example also demonstrates the advantage of providing an alist of the general list form. The **:documentation** provided in the alist can add much useful information to the display.

A type history is not available for the **alist-member** presentation type.

alist-member is one of a number of types defined in SYS:DYNAMIC-WINDOWS;SEQUENCE-TYPES.LISP. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

and (&rest *types*) *Presentation Type*

Compound type for accepting or presenting an object of two or more presentation types. Typically, the second and subsequent types are derived via the **satisfies** presentation type.

types Data arguments specifying the contributing presentation types.

Examples:

```
(accept '((and sys:expression (satisfies symbolp)))) ==>
Enter the representation of any Lisp
object satisfying SYMBOLP: ramjet
RAMJET
((AND SYS:EXPRESSION
      (SATISFIES SYMBOLP)))

(accept '((and ((integer)) ((satisfies oddp))
                           ((satisfies plusp))))) ==>
Enter an integer satisfying ODDP and
PLUSP [default 9]: 21
21
((AND ((INTEGER))
      ((SATISFIES ODDP))
      ((SATISFIES PLUSP))))
```

The compound presentation type in the first example is equivalent to the **symbol** presentation type and is, in fact, how that type is defined.

and can combine any number of **satisfies** types with an initial, non-**satisfies** type. The second example above shows an initial integer type used with two **satisfies** types to solicit input of odd, positive integers.

Note that the compound type has access to the type history of the initial presentation type, if one exists. However, it does not automatically use the value at the top of the history as the default value in an **accept** function. Rather, it uses the item most recently added to the type history that also satisfies the **satisfies** function(s).

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

boolean *Presentation Type*

Type for accepting or presenting a yes-or-no answer, where "yes" is **t** and "no" is **nil**.

Examples:

```
(accept '((boolean))) ==>
Enter Yes or No: No
NIL
((BOOLEAN))

(present t '((boolean))) ==>Yes
#<DISPLAYED-PRESENTATION 444300153>
```

A type history is not available for the **boolean** presentation type.

boolean is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

See also the **inverted-boolean** presentation type.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

zwei:buffer &key (*create-p :if-forced*) *Presentation Type*

Type for accepting or presenting Zmacs editor buffers.

:create-p Presentation option specifying whether to create the buffer entered in response to an **accept** prompt if it does not already exist.

The default is **:if-forced**. This gives the user the option of changing the input or creating the new buffer by terminating with CONTROL-RETURN, rather than just creating the buffer as in the case of `:create-p t`.

Examples:

```
(accept '((zwei:buffer))) ==>
Enter an editor buffer
[default ui-dict15.sar >sys>doc>uims Q:]: HELP ==>
You are being asked to enter an editor buffer.

These are the possible editor buffers:
  *Buffer-1*
  *Definitions-1*
  doc-29-55.lisp >sys>doc>patch>doc-29 Q:
  miscui2.sar >sys>doc>miscui Q:
  standard-presentation-types.lisp >sys>dynamic-windows Q:
  ui-dict15.sar >sys>doc>uims Q:

Enter an editor buffer
[default ui-dict15.sar >sys>doc>uims Q:]: *Buffer-1*
#<NON-FILE-BUFFER "*Buffer-1*" 47700004>

(accept '((zwei:buffer) :create-p t)) ==>
Enter an editor buffer
[default ui-dict15.sar >sys>doc>uims Q:]: foo.test
#<NON-FILE-BUFFER "foo.test" 47700567>
((ZWEI:BUFFER) :CREATE-P T)

(present (zwei:make-buffer 'zwei:non-file-buffer)
         '((zwei:buffer))) ==>*Buffer-2*
#<DISPLAYED-PRESENTATION 274672153>
```

The **zwei:buffer** presentation type uses the special variable **zwei:*buffer-history*** to provide its type history.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

character *Presentation Type*

Type for accepting or presenting single characters. When reading, this is just a single character without termination.

Examples:

```
(accept '((character)) ==>
Enter a character: R
#\R
((CHARACTER))

(accept '((character)) ==>
Enter a character: r
#\r
((CHARACTER))

(accept '((character)) ==>
Enter a character: %
#\%
((CHARACTER))

(accept '((character)) ==>
Enter a character: 3
#\3

(present #\, '((character))) ==>,
#<DISPLAYED-PRESENTATION 445346702>
((CHARACTER))
```

Use the **character** presentation type for normal, editable character input. To accept characters that would be mistaken as input-editor commands, for example **#\C-B**, use **dw:out-of-band-character** instead.

There is no type history for the **character** presentation type.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

character-face-or-style *(*&key *device (against-default si:*standard-default-character-style*))* &key *for-attribute-list* *Presentation Type*

Type for accepting either a fully specified character style, or just the face component. (The device argument, although implemented as a keyword, is required.)

:device Data option specifying the device for the character style. There are four possibilities: **si:*b&w-screen***, **lgp:*lgp-printer***, **lgp:*lgp2-printer***, and **dmp1:*dmp1-printer***. This is normally gotten with the **:display-device-type** message to a stream.

:against-default Data option specifying a character style against which the input character style is merged. This is used to limit possibilities since the merge result must be valid. See the section "Merging Character Styles".

:for-attribute-list Presentation option specifying whether the character style should be presented in list form, for example, `(:fix :bold :normal)`. The default is **nil**. Supply a value of **t** when presenting a character style for inclusion in the attribute list of file.

Examples:

```
 (accept '((character-face-or-style
:device (send *terminal-io* :display-device-type))) ==>
Enter a character face or style: BOLD
#<CHARACTER-STYLE NIL.BOLD.NIL 155157247>
((CHARACTER-FACE-OR-STYLE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604>))

(accept '((character-face-or-style
:device ,si:*b&w-screen*))) ==>
Enter a character face or style: DUTCH.ROMAN.NORMAL
#<CHARACTER-STYLE DUTCH.ROMAN.NORMAL 154174235>
((CHARACTER-FACE-OR-STYLE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604>))
```

The **character-face-or-style** presentation type does not support a type history.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

character-style (&key *against-default*) &key *for-attribute-list* *Presentation Type*

Type for accepting or presenting character styles.

:against-default Data option specifying a character style against which the input character style is merged. This is used to limit possibilities since the merge result must be valid. See the section "Merging Character Styles".

:for-attribute-list Presentation option specifying whether the character style should be presented in list form, for example, `(:fix :bold :normal)`. The default is **nil**. Supply a value of **t** when presenting a character style for inclusion in the attribute list of file.

When accepting a character style, the user is prompted for the family, face, and size, in that order. The first two entries must be terminated by a period, the last by RETURN or END.

Examples:

```
(accept '((character-style))) ==>
Enter a valid character style: SWISS.BOLD.LARGE
#<CHARACTER-STYLE SWISS.BOLD.LARGE 264231477>

(present (si:parse-character-style '(:swiss :bold :large))) ==>
SWISS.BOLD.LARGE
#<DISPLAYED-PRESENTATION 425221252>
```

The **character-style** presentation type supports a type history.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

character-style-for-device *(*&key *device* (*against-default si:*standard-default-character-style**) (*allow-relative* **t**) (*allow-device-font nil*)) &key *for-attribute-list* (*provide-subhelp t*) *Presentation Type*

Type for accepting or presenting character styles for a specified device. (The device argument, although implemented as a keyword, is required.)

:device Data option specifying the device for the character style. There are four possibilities: **si:*b&w-screen***, **lgp:*lgp-printer***, **lgp:*lgp2-printer***, and **dmp1:*dmp1-printer***. This is normally gotten with the **:display-device-type** message to a stream.

:against-default Data option specifying a character style against which the input character style is merged. This is used to limit possibilities since the merge result must be valid. See the section "Merging Character Styles".

:allow-relative Data option specifying whether relative style specifications, such as **smaller** or **larger**, are permitted. See the section "Merging Character Styles".

:allow-device-font Data option specifying whether a device font is permitted; the default is **nil**.

For more information about device fonts, see the section "Mapping a Character Style to a Font".

:for-attribute-list Presentation option specifying whether the character style should be presented in list form, for example, (:fix :bold :normal). The default is **nil**. Supply a value of **t** when presenting a character style for inclusion in the attribute list of file.

:provide-subhelp Presentation option specifying whether to provide a HELP display; the default is **t**. Disable this if a higher-level call provides help.

Examples:

```
(accept '((character-style-for-device
:device ,si:*b&w-screen*))) ==>
Enter a character style: FIX.BOLD.TINY
#<CHARACTER-STYLE FIX.BOLD.TINY 154222436>
((CHARACTER-STYLE-FOR-DEVICE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604>))

(accept '((character-style-for-device
:device ,lgp:*lgp2-printer* :allow-relative t))) ==>
Enter a character style [default FIX.BOLD.TINY]: SWISS.BOLD.SAME
#<CHARACTER-STYLE SWISS.BOLD.SAME 15212221>
((CHARACTER-STYLE-FOR-DEVICE :DEVICE
#<LGP2-DISPLAY-DEVICE 154173651> :ALLOW-RELATIVE T))

(accept '((character-style-for-device
:device ,si:*b&w-screen* :allow-device-font t))) ==>
Enter a character style: DEVICE-FONT.BIGFNT
#<CHARACTER-STYLE DEVICE-FONT.BIGFNT.NORMAL 14251534>
((CHARACTER-STYLE-FOR-DEVICE :DEVICE
#<B&W-SCREEN-DISPLAY-DEVICE 154221604> :ALLOW-DEVICE-FONT T))
```

character-style-for-device is a subtype of **character-style**, from which it inherits a type history.

For an overview of presentation types and related facilitie,: see the section "Using Presentation Types".

sys:code-fragment *Presentation Type*

Type for accepting or presenting pieces of Lisp code. This presentation type is a subtype of **sys:form**, and intended primarily for accessing code fragments in editor buffers. The following example, the definition of a translating mouse handler for editor commands, uses **sys:code-fragment** as the *from-presentation-type* argument:

```
(zwei:define-presentation-to-editor-command-translator
        typeout-menu-arglist-from-buffer
        (sys:code-fragment "Arglist" *standard-comtab*
         :gesture :hyper-meta-middle)
                              (function-spec)
  (when (and (sys:validate-function-spec function-spec)
             (fdefinedp function-spec))
    '(typeout-menu-arglist ,function-spec)))
```

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

cp:command *(&key command-table *command-table* command-table-p) &key wait-for-activation t* *Presentation Type*

Type for accepting or presenting a command processor command.

:command-table Data option specifying the command table in which to find the command. The default, ***command-table***, is bound to the command table currently in use. See the section "Managing Command Tables".

:wait-for-activation Presentation option specifying, when **nil** that the command should be executed when the presentation is accepted. With the default, **t**, the user is required to enter an activation character to execute the command.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

fs:directory-pathname &key *(default-version :newest) (default-type nil) (default-name nil) dont-merge-default (direction :read) (format :normal)* *Presentation Type*

Type for accepting or presenting a directory pathname. This is a directory as a file, that is, something like `>a>b.directory`, as opposed to `>a>b`.

This presentation type can be useful if you need to distinguish unequivocally between directory pathname presentations and file pathname presentations. For example, if you can arrange for the availability to your users of some **fs:directory-pathname** presentations, then mouse handlers performing directory-related functions can be defined that do not have to test whether a given **pathname** presentation is a directory pathname, or extract directory objects from **pathname** presentations.

fs:directory-pathname is a subtype of the **pathname** presentation type, from which it inherits a printer, parser, and type history. It also takes the same keyword arguments, as follows:

:default-version Presentation option specifying the default version number of an accepted file. The default value for this option is **:newest**, the newest file version.

:default-type Presentation option specifying the default file type, for example, "lisp", "text", "data", and so on. The default value for this option is **nil**.

:default-name Presentation option specifying the default file name. The default value for this option is **nil**.

:dont-merge-default Presentation option specifying whether to prevent merging of a partially specified pathname entered by the user against the default pathname. The default value for this option is **nil**, meaning that merging occurs when appropriate; that is, parts of the pathname not entered by the user are supplied from the default.

Suppression of merging against the default and providing a different default (against which merging may or may not be enabled) are different issues. To deal with the latter, use the **:default** option to **accept**. (See the function **accept**.) An example follows:

```
(accept '((pathname) :default-type nil)
        :default (send (fs:default-pathname)
                       :new-pathname :type nil
                       :version :newest))
```

:direction
Presentation option specifying either **:read** (the default) or **:write**. The value supplied is passed through to **fs:complete-pathname** and affects completion behavior. (See the function **fs:complete-pathname**.)

Use the default (**:read**) if the user is likely to enter the pathname of an already existing file when prompted by **accept**, **:write** otherwise.

:format Presentation option specifying the output format of the pathname. There are four choices:

:normal For example, `S:>mb>dw-pgms>fancy-windows.lisp`. This is the default format.

:directory For example, `>mb>dw-pgms>`. The host, file name, and file type are not displayed.

:dired For example, `fancy-windows.lisp`. Only the file name and type are displayed.

:editor For example, `fancy-windows.lisp >mb>dw-pgms S`. The display format is that used by Zmacs.

For examples illustrating the use of these keywords in pathname presentations, see the presentation type **pathname**.

fs:directory-pathname is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

Type for accepting or presenting dynamic window objects.

Examples:

```
  (accept '((dw:dynamic-window)))
You are being asked to enter a dynamic window.

These are the possible dynamic windows:
  Dynamic ... (19)
  Terminal 1

Enter a dynamic window [default Terminal 1]: Terminal 1
#<NVT-WINDOW Terminal 1 700372 deexposed>
((DW:DYNAMIC-WINDOW))

(present (tv:make-window 'dw:dynamic-window)
         '((dw:dynamic-window)))Dynamic Window 2
#<DW::DISPLAYED-PRESENTATION 650404277>
```

The **dw:dynamic-window** presentation type supports a type history.

dw:dynamic-window is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

sys:expression &rest *options* *Presentation Type*

Type for accepting or presenting expressions. An expression is the readable, printed representation of a Lisp object. The expression is not evaluated.

options Presentation options controlling the generation of the printed representation. They are listed in the following table, along with the special variables providing each option with its default value. (Note that these options are the same as those available for the Common Lisp function **write**.)

Option	*Special Variable*
:escape	***print-escape***
:pretty	***print-pretty***
:abbreviate-quote	***print-abbreviate-quote***
:radix	***print-radix***
:base	***print-base***
:circle	***print-circle***
:level	***print-level***
:length	***print-length***
:case	***print-case***
:gensym	***print-gensym***
:array	***print-array***
:readably	***print-readably***
:array-length	***print-array-length***
:string-length	***print-string-length***
:bit-vector-length	***print-bit-vector-length***
:structure-contents	***print-structure-contents***

The special variables are documented together in another section: See the section "Output Functions". Consult the documentation for the individual variables to find out what they do and what values they can have. These values are the same that can be supplied with the corresponding presentation options to **sys:expression**.

Examples:

```
(accept '((sys:expression))) ==>
Enter the representation of any Lisp object
[default (ACCEPT '((SYS:EXPRESSION)))]: setq
SETQ
SYS:EXPRESSION

(accept '((sys:expression))) ==>
Enter the representation of any Lisp object
[default (ACCEPT '((SYS:EXPRESSION)))]: (+ 33 900)
(+ 33 900)
SYS:EXPRESSION

(present (net:find-object-named :network "DNA")
'((sys:expression))) ==>#<DNA-NETWORK DNA 13702517>
#<DISPLAYED-PRESENTATION 275045641>
```

```
(accept '((sys:expression))) ==>
Enter the representation of any Lisp object
[default (ACCEPT '((SYS:EXPRESSION)))]:
'#<DISPLAYED-PRESENTATION 275045641>
'#<DISPLAYED-PRESENTATION 275045641>
SYS:FORM
```

The **sys:expression** type occupies a unique position in the data type hierarchy, namely, the highest spot but for one, that occupied by **t**. This means that, with a few exceptions, **sys:expression** is supertype to all other Symbolics Common Lisp types.

For all data types not explicitly defined as presentation types (via **define-presentation-type**), **sys:expression** serves as the access point to the presentation system. It provides these types with a parser, printer, and type history. In fact, it provides one or more of these functions to many defined presentation types as well.

sys:expression's history includes all previously accepted Lisp objects. This is why, in the **accept** examples above, the default is always `(ACCEPT '((SYS:EXPRESSION)))`; this expression is the most recently accepted one.

When accessed by other types, **sys:expression**'s type history is pruned to objects of the accessing type. For example, **number** and types descended from **number** do not maintain their own type histories. When a previously accepted value is needed to provide, say, a default value in an **accept** of an **integer**, the expression history is pruned to integer objects of which the most recently accepted is used as the default.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

sys:flavor-name *Presentation Type*

Type for accepting or presenting symbols that name flavors.

Examples:

```
(accept '((sys:flavor-name))) ==>
Enter a flavor name: DW:PROGRAM-FRAME
DW:PROGRAM-FRAME
((SYS:FLAVOR-NAME))

(present 'dw:margin-mixin '((sys:flavor-name))) ==>DW:MARGIN-MIXIN
#<DISPLAYED-PRESENTATION 275147735>
```

```
(accept '((sys:flavor-name))) ==>
Enter a flavor name [default DW:PROGRAM-FRAME]: DW:MARGIN-MIXIN
DW:MARGIN-MIXIN
((SYS:FLAVOR-NAME))
```

The **sys:flavor-name** presentation type supports a type history and completion.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

sys:font *Presentation Type*

Type for accepting or presenting loaded fonts.

Examples:

```
(accept '((sys:font))) ==>
Enter a loaded font: HELP ==>
You are being asked to enter a loaded font.

There are 87 possible loaded fonts.  Do you want to see them all?
(Y or N) Yes.
These are the possible loaded fonts:
  5X5         DUTCH20B   HL14I    MEDFNTBI         TR12BI
  BIGFNT      DUTCH20BI  HL18     MEDFNTI          TR12I
  BIGFNTB     DUTCH20I   HL18B    MOUSE            TR14
  BIGFNTBI    EINY7      HL18BI   NARROW           TR14B
  BIGFNTI     EUREX12I   HL18I    SWISS12-CCAPS    TR14I
  BOXFONT     EUREX24I   HL8      SWISS12B-CCAPS   TR18
  CPTFONT     HIPPO12    HL8B     SWISS20          TR18B
  CPTFONTB    HL10       HL8BI    SWISS20B         TR8
  CPTFONTBI   HL10B      HL8I     SWISS20BI        TR8B
  CPTFONTC    HL10BI     JESS13   SWISS20I         TR8BI
  CPTFONTCB   HL10I      JESS13B  SYMBOL12         TR8I
  CPTFONTCC   HL12       JESS13I  TINY             TVFONT
  CPTFONTI    HL12B      JESS14   TR10             TVFONTB
  DUTCH14     HL12BI     JESS14B  TR10B            TVFONTBI
  DUTCH14B    HL12I      JESS14I  TR10BI           TVFONTI
  DUTCH14BI   HL14       MATH12   TR10I
  DUTCH14I    HL14B      MEDFNT   TR12
  DUTCH20     HL14BI     MEDFNTB  TR12B

Enter a loaded font: DUTCH20
#<FONT DUTCH20 260074563>
SYS:FONT
```

```
(accept '((sys:font))) ==>
Enter a loaded font [default DUTCH20]: SWISS20
#<FONT SWISS20 260160676>
((SYS:FONT))

(present (si:get-font si:*b&w-screen* si:*standard-character-set*
'(:jess :roman :normal))) ==>JESS13
#<DISPLAYED-PRESENTATION 440305757>
```

The **sys:font** presentation type supports a type history.

sys:font is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

sys:form &key (*environment* **si:*read-form-environment*** *environment-p*) (*expression-reader* **nil**) (*expression-printer* **nil**) (*edit-trivial-errors-p* **zl:*read-form-edit-trivial-errors-p***) *Presentation Type*

Type for accepting or presenting Lisp forms.

:environment Presentation option specifying the lexical environment of an input form. (For more on environments: See the section "Lexical Environment Objects and Arguments".)

:edit-trivial-errors Specifies, when **t**, that two kinds of errors should be checked for: If a symbol is read, check whether the symbol is bound. If a list whose first element is a symbol is read, check whether the symbol has a function definition. If an unbound symbol or undefined function is encountered, the parser 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**.

Possible values of the keywords **:expression-reader** and **:expression-printer** are functions for reading and writing expressions in languages other than Lisp, such as Pascal, Fortran, or C. These are for use by the debugger.

```
(accept '((sys:form))) ==>
Enter A Lisp expression to be evaluated
[default (ACCEPT '((SYS:FORM)))]: (symbolp t)
(SYMBOLP T)
((SYS:FORM))

(present '(symbolp t) '((sys:form))) ==>(SYMBOLP T)
#<DISPLAYED-PRESENTATION 275141170>
```

```
Command: (SYMBOLP T)
T
```

Presented forms are evaluable. In the above examples, run in the command-or-form context, the *(SYMBOLP T)* form was entered to the `Command:` prompt by clicking left on the output of the preceding **present** function. This form was immediately evaluated. Contrast this behavior with that of **sys:expression** presentations; presented forms are quoted and not evaluable directly.

The **sys:form** presentation type inherits its printer and type history from **sys:expression**.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

sys:function-spec (&key *defined-p*) &key (*partial-completers* '(**#/space**))
Presentation Type

Type for accepting or presenting valid function specs. (For information on function specs, see the section "Function Specs".) Note that a valid function spec is anything that can be parsed to **record-source-file-name**. It is not restricted to those objects actually defined as functions, except in the **:defined-p t** case.

:defined-p Data option restricting function specs to those that are defined; possible values are **t**, **nil**, or **:any**, which last means any kind of definition. The default is **nil**.

:partial-completers Presentation option specifying a list of characters to be used as completers of function-spec tokens during input; the default list is (**#/space**).

Examples:

```
(present '+ '((sys:function-spec))) ==>+
#<DISPLAYED-PRESENTATION 275374421>

(accept '((sys:function-spec))) ==>
Enter a valid function spec: +
+
((SYS:FUNCTION-SPEC))

(accept '((sys:function-spec))) ==>
Enter a valid function spec [default +]: (:PROPERTY alpha bravo)
(:PROPERTY ALPHA BRAVO)
((SYS:FUNCTION-SPEC))
```

```
(accept '((sys:function-spec :defined-p t))) ==>
Enter a defined function spec: (:PROPERTY alpha bravo)
(:PROPERTY ALPHA BRAVO) is not a defined function spec.
Type RUBOUT to correct your input. [Abort]

(defun (:property alpha bravo) () 1) ==>
(:PROPERTY ALPHA BRAVO)

(accept '((sys:function-spec :defined-p t))) ==>
Enter a defined function spec
[default (:PROPERTY ALPHA BRAVO)]: (:PROPERTY ALPHA BRAVO)
(:PROPERTY ALPHA BRAVO)
((SYS:FUNCTION-SPEC :DEFINED-P T))
```

The **sys:function-spec** presentation type supports a type history. For an overview of presentation types and related facilities, see the section "Using Presentation Types".

sys:generic-function-name &key *show-compatible-message* *Presentation Type*

Type for accepting or presenting function specs for generic functions.

:show-compatible-message Presentation option specifying whether to also print, if defined, the name of the compatible message for the generic function. (Compatible messages are specified by an option to **defgeneric**, see the section "Defining a Compatible Message for a Generic Function".)

Examples:

```
(accept '((sys:generic-function-name))) ==>
Enter a generic function name: HELP
You are being asked to enter a generic function name.

There are 11630 possible generic function names.
Do you want to see them all? (Y or N) No. [Thanks, anyway.]

Enter a generic function name: DW:DO-REDISPLAY
DW:DO-REDISPLAY
((SYS:GENERIC-FUNCTION-NAME))

(present 'sys:print-self '((sys:generic-function-name))) ==>
SYS:PRINT-SELF
#<DISPLAYED-PRESENTATION 275755254>
```

```
(present 'sys:print-self '((sys:generic-function-name)
:show-compatible-message t)) ==>SYS:PRINT-SELF (:PRINT-SELF)
#<DISPLAYED-PRESENTATION 275755527>
```

The **sys:generic-function-name** presentation type supports a type history and completion.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

net:host *Presentation Type*

Type for accepting or presenting a network host.

Examples:

```
(accept '((net:host))) ==>
Enter the name of a host: Harpagornis
#<LISPM-HOST HARPAGORNIS 53344734>
((NET:HOST))

(accept '((net:host))) ==>
Enter the name of a host [default HARPAGORNIS]: laurent
#<MSDOS-HOST YVES-ST-LAURENT 533601167>
((NET:HOST))

(present (si:parse-host "owl") '((net:host))) ==>OWL
#<DISPLAYED-PRESENTATION 275435731>

(accept '((net:host))) ==>
Enter the name of a host [default YVES-ST-LAURENT]: OWL
#<LISPM-HOST OWL 13707365>
((NET:HOST))
```

The **net:host** presentation type has its own parser and type history; it inherits its printer via **net:object**, to which it is subtype, from **sys:expression**.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

instance (&optional (*flavor* '*)) *Presentation Type*

Type for accepting or presenting flavor instances.

flavor Data argument specifying what flavor this is an instance of; the default leaves the flavor unspecified.

Examples:

```
(present (tv:make-window 'dw:dynamic-window) 'instance) ==>
Dynamic Window 1
#<DW::DISPLAYED-PRESENTATION #<DYNAMIC-WI... INSTANCE 420070634>

(accept '((instance))) ==>
Enter the Lisp representation of an instance [default Dynamic Window 1]:
#<DYNAMIC-WINDOW Dynamic Window 1 3132740 deactivated>
((INSTANCE))
```

The **instance** presentation type inherits its printer and parser functions — as well as a type history — from the **sys:expression** presentation type. Thus, in the first **accept** function above, the prompt says to "Enter the representation of any Lisp object". We override this by providing our own prompt in the second call to **accept**.

In the first **accept** form, the entered *Dynamic Window 1* is in italics because it was entered via a mouse click on the presentation created by the **present** function. If we had tried to type in "dynamic window 1", **accept** would have returned the object DYNAMIC when the first space character was typed.

instance is not a presentation type that you are likely to need for writing end-user interfaces to applications. A number of Common Lisp presentation types are in this category, for example, **structure** and **hash-table**. Like **instance**, all inherit their parser, printer, and type history from **sys:expression**. And, as in the case of **instance**, when **sys:expression**'s type history is accessed to provide, for example, a default value in an **accept** function, the history is "pruned" to objects only of the sought-after type. Thus, in the second **accept** function above, not any Lisp object is offered as a default, but an **instance** object.

All flavors are subtype to the **instance** presentation type. Similarly, all structures are subtype to the **structure** type. The two types are thereby important for links they provide to the presentation-type system for flavors and structures, respectively.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

integer (&optional (*range-low* '*) (*range-high* '*)) &key (*base 10*) *Presentation Type*

Type for accepting or presenting integers.

range-low Data argument specifying a lower limit for integer objects. The default is no lower limit. As in Common Lisp type specifications, * stands for an unspecified subsidiary item.

range-high Data argument specifying an upper limit for integer objects. The default is no upper limit. As in Common Lisp type specifications, * stands for an unspecified subsidiary item.

:base Presentation option specifying the base used for integer presentations; the default is `10`.

Examples:

```
(accept '((integer 0 100))) ==>
Enter an integer greater than or equal to 0
and less than or equal to 100: 0
0
((INTEGER 0 100))

(accept '((integer (0) (100)))) ==>
Enter an integer greater than 0 and less than 100: 1
1
((INTEGER (0) (100)))

(present 10 '((integer) :base 8)) ==>12
#<DISPLAYED-PRESENTATION 445411244>

(accept '((integer 0 100)))
Enter an integer greater than or equal to 0
and less than or equal to 100: 12
10
((INTEGER) :BASE 8)

(accept '((integer 0 100) :base 8)) ==>
Enter an octal integer greater than or equal to 0
and less than or equal to 144: 12
10
((INTEGER) :BASE 8)

(present 50 '((integer 0 100))) ==>50
#<DISPLAYED-PRESENTATION 445430232>

(accept '((integer)))
Enter an integer [default 8]: 50
50
((INTEGER 0 100))

(accept '((integer))) ==>
Enter an integer [default 5]: 50
50
((INTEGER 0 100))
```

When specifying range limits, if the limits are provided without enclosing parentheses, they are inclusive; with parentheses, exclusive. Contrast the first two **present** functions.

The 12 input to the second and third **accept** functions above was entered by clicking on the output of the first **present** function. Note that, regardless of the base used for the integer presentation, the object returned remains the same.

Note also in the second and third **accept**s that the data type returned is the one entered, an integer, not a range-restricted integer, even though the functions restricted the range of acceptable integers. Contrast this with the final **present-accept** pair: the object presented as a range-restricted integer is entered to a non-restricted integer accepting function; the object's data type (subtype, actually) is retained.

Finally, note that the **integer** presentation type supports a type history (inherited from **sys:expression**), the source of the default value offered in the last **accept** function, but that range-restricted integer types do not.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

inverted-boolean *Presentation Type*

Type for accepting or presenting a yes-or-no answer, where "yes" is **nil** and "no" is **t**. Use it when the sense of the internal action is inverted from the user sense.

Examples:

```
(accept '((inverted-boolean))) ==>
Enter Yes or No: No
T
((INVERTED-BOOLEAN))

(present t '((inverted-boolean))) ==>No
#<DISPLAYED-PRESENTATION 444312267>
```

A type history is not available for the **inverted-boolean** presentation type.

inverted-boolean is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

See also the **boolean** presentation type.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

keyword *Presentation Type*

Type for accepting or presenting keywords.

Examples:

```
(accept '((keyword))) ==>
Enter a keyword: orientation
:ORIENTATION
((KEYWORD))

(accept '((keyword))) ==>
Enter a keyword [default ORIENTATION]: :sojac
:|:SOJAC|
((KEYWORD))

(accept '((keyword)))
Enter a keyword: 1492
:|1492|
((KEYWORD))

(present :orientation '((keyword))) ==>ORIENTATION
#<DISPLAYED-PRESENTATION 454276732>
```

keyword inherits its printer and type history from the **sys:expression** presentation type.

keyword is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

net:local-host *Presentation Type*

Type for accepting or presenting the local host. The local host is accepted and presented as "Local". This is useful to use with **or** and **net:host** to obtain output of host objects with the special syntax for a local host.

Examples:

```
(accept '((si:local-host))) ==>
Enter a local host: Local
#<LISPM-HOST OYSTERCATCHER 13702373>
((SI:LOCAL-HOST))

(present net:*local-host* '((si:local-host))) ==>Local
#<DISPLAYED-PRESENTATION 275456200>

(accept '((si:local-host))) ==>
Enter a local host [default Local]: Local
#<LISPM-HOST OYSTERCATCHER 13702373>
((SI:LOCAL-HOST))
```

The **net:local-host** presentation type is subtype to the **net:host** type, but has its

own parser and printer. It inherits a type history from the latter, but prunes it to occurrences of "Local".

net:local-host is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

neti:local-network *Presentation Type*

Type for accepting or presenting local network objects. (A local network is one to which the current machine is connected.)

Examples:

```
(accept '((neti:local-network))) ==>
Enter a local network: HELP
You are being asked to enter a local network.

These are the possible local networks:
  CHAOS
  FBAND
  INTERNET

Enter a local network: INTERNET
#<INTERNET-NETWORK INTERNET 13700021>
((NETI:LOCAL-NETWORK))

(present (car neti:*local-networks*)
         '((neti:local-network))) ==>FBAND
#<DISPLAYED-PRESENTATION 275517001>

(accept '((neti:local-network)))
Enter a local network [default INTERNET]: FBAND
#<FBAND-NETWORK FBAND 261216753>
((NETI:LOCAL-NETWORK))
```

The **neti:local-network** presentation type supports its own type history.

neti:local-network is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

member (&rest *elements*) *Presentation Type*

Type for accepting or presenting one of a series of objects. The printed representations of the objects must be unique, that is, no two representations can be **string-equal**.

elements The series of objects. These objects are data arguments for this presentation type.

Examples:

```
(accept '((member New York Stock Exchange))) ==>
Enter New, York, Stock, or Exchange: York
YORK
((MEMBER NEW YORK STOCK EXCHANGE))

(accept `((member ,(pathname "y:>pgm>ui-1.lisp")
                  ,(pathname "y:>pgm>ui-2.lisp")
                  ,(pathname "y:>pgm>ui-3.lisp")))) ==>
Enter Y:>pgm>ui-1.lisp, Y:>pgm>ui-2.lisp,
or Y:>pgm>ui-3.lisp: Y:>pgm>ui-2.lisp
#P"Y:>pgm>ui-2.lisp"
((MEMBER #P"Y:>pgm>ui-1.lisp" #P"Y:>pgm>ui-2.lisp"
#P"Y:>pgm>ui-3.lisp"))
```

Because the prompt generated by **accept** for input of **member** objects can sometimes be awkward, you may want to use the meta-presentation argument **:description** to change the prompt. (See the section "The Presentation Type System: an Overview".)

The **member** presentation type works differently from the **member** function in how it determines group membership. The presentation type merely checks to see if the printed representation of an object is the same as one of its elements. The function bases membership decisions on **eql**.

There is no type history for the **member** presentation type.

The **dw:member-sequence** presentation type is similar to **member**, except that it takes a single argument instead of a series of arguments. See the presentation type **dw:member-sequence**.

For an overview of presentation types and related facilities, see the section "Presentation Substrate Facilities".

dw:member-sequence (*sequence*) *Presentation Type*

Type for accepting or presenting one of a series of objects. The printed representations of the objects must be unique, that is, no two representations can be **string-equal**.

sequence Data argument specifying a sequence containing the objects.

Examples:

```
(accept '((dw:member-sequence
             (Kierkegaard Heidegger Buber Barth)))) ==>
Enter Kierkegaard, Heidegger, Buber, or Barth: Heidegger
HEIDEGGER
((DW:MEMBER-SEQUENCE (KIERKEGAARD HEIDEGGER BUBAR BARTH)))

(setq adenosine-list '("AMP" "ADP" "ATP"))
("AMP" "ADP" "ATP")

(accept `((dw:member-sequence ,adenosine-list)))
Enter AMP, ADP, or ATP: ATP
"ATP"
((DW:MEMBER-SEQUENCE ("AMP" "ADP" "ATP")))
```

Because the prompt generated by **accept** for input of **dw:member-sequence** objects can sometimes be awkward, you may want to use the meta-presentation argument **:description** to change it. (See the section "The Presentation Type System: an Overview".)

dw:member-sequence is similar to the **member** presentation type, except that it takes a single argument instead of a series of arguments. See the presentation type **member**.

The **dw:member-sequence** presentation type does not support a type history.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

neti:namespace *Presentation Type*

Type for accepting or presenting namespace objects.

Examples:

```
(present net:*namespace* '((neti:namespace))) ==>SCRC
#<DISPLAYED-PRESENTATION 275467554>

(accept '((neti:namespace)))
Enter a namespace: SCRC
#<NAMESPACE SCRC 13700207>
((NETI:NAMESPACE))

(accept '((neti:namespace)))
Enter a namespace [default SCRC]: SCRC
#<NAMESPACE SCRC 13700207>
((NETI:NAMESPACE))
```

Through flavor inheritance, the **neti:namespace** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For presentations of namespace classes, as opposed to the namespace objects themselves, use the **net:namespace-class** presentation type.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

net:namespace-class *Presentation Type*

Type for accepting or presenting namespace classes, of which there are currently seven:

:file-system
:user
:printer
:network
:host
:site
:namespace

Examples:

```
(accept '((net:namespace-class))) ==>
Enter a namespace class: printer
:PRINTER
((NET:NAMESPACE-CLASS))

(accept '((net:namespace-class))) ==>
Enter a namespace class: Namespace
:NAMESPACE
((NET:NAMESPACE-CLASS))

(present :site '((net:namespace-class))) ==>Site
#<DISPLAYED-PRESENTATION 275427546>
```

The **net:namespace-class** presentation type is based on the **dw:member-sequence** type. Neither supports a type history.

For presentations of namespace objects, as opposed to namespace classes, use the **net:namespace** presentation type.

net:namespace-class is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

net:network *Presentation Type*

Type for accepting or presenting network objects.

Examples:

```
(present (net:find-object-named
          :network "DNA") '((net:network))) ==>DNA
#<DISPLAYED-PRESENTATION 275510033>

(accept '((net:network))) ==>
Enter a network: DNA
#<DNA-NETWORK DNA 13702517>
((NET:NETWORK))
```

Through flavor inheritance, the **net:network** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

dw:no-type *Presentation Type*

Bogus presentation type for use with mouse handlers. **dw:no-type** is used to ensure that handlers intended to be active only over blank areas of a window are not active over presentations. See the section "**:blank-area** Option to Mouse Handlers".

For an overview of **dw:no-type** and related facilities, see the section "Using Presentation Types".

not *(inverted-type)* *Presentation Type*

Type for modifying another presentation type. There is no parser or printer for this type; it can only be used as part of a compound type — specifically, only as part of an **and** presentation type.

inverted-type Data argument specifying the presentation type to qualify.

There is no type history for the **not** presentation type. Example:

```
(accept '((and number ((not integer)))))
```

For an overview of presentation types and related facilities, see the section "Presentation Substrate Facilities".

null *Presentation Type*

Type for accepting or presenting a null object (**nil**). The **null** type is necessary because no parser or printer can be defined for **nil**.

Null objects are presented as "None". They can be accepted by pressing RETURN to the **accept** function prompt, or clicking on a previously presented "None".

Examples:

```
(present nil '((null))) ==>None
#<DISPLAYED-PRESENTATION 454227454>

(present nil) ==>None
#<DISPLAYED-PRESENTATION 454227707>

(accept '((null))) ==>
Enter a null value: <RETURN>
NIL
((NULL))

(accept '((null))) ==>
Enter a null value: None
NIL
NULL
```

The most common use of **null** is as part of an **or** compound presentation type. For such a combination, use the **null-or-type** presentation type.

For an overview of presentation types and related facilities, see the section "Presentation Substrate Facilities".

null-or-type (*presentation-type*) *Presentation Type*

Compound type for accepting or presenting either **nil** or an object of a specified presentation type. **nil** is accepted or presented as "None".

presentation-type Data argument specifying a presentation type.

Examples:

```
(accept '((null-or-type number))) ==>
Enter a null or type: 2.2
2.2
((NULL-OR-TYPE NUMBER))

(accept '((null-or-type number))
        :prompt "Enter a number or \"None\"") ==>
Enter a number or "None" [default 2.2]: None
NIL
((NULL-OR-TYPE NUMBER))

(present nil '((null-or-type number))) ==>None
#<DISPLAYED-PRESENTATION 444713264>
```

If the type specified in the **null-or-type** presentation type supports a type history, this history is used. This is the source of the default value shown in the second call to **accept** above.

null-or-type is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

number (&optional *range-low range-high*) &key (*base 10*) *Presentation Type*

Type for accepting or presenting numbers.

range-low Data argument specifying a lower limit for number objects. The default is no lower limit.

range-high Data argument specifying an upper limit for number objects. The default is no upper limit.

:base Presentation option specifying the base used for integer presentations; the default is `10`.

Examples:

```
(accept 'number)
Enter a number: 23
23
(NUMBER)

(accept '(number :base 10)) ==>
Enter a decimal number: 12
12
(NUMBER :BASE 10)

(accept '((number 0 10) :base 2)) ==>
Enter a binary number greater than or equal to 0
and less than or equal to 1010: 111
7
((NUMBER 0 10) :BASE 2)

(accept '((number 0 10) :base 2)) ==>
Enter a binary number greater than or equal to 0
and less than or equal to 1010: 2
2
((NUMBER 0 10) :BASE 2)
```

When specifying range limits, if the limits are provided without enclosing parentheses, they are inclusive; with parentheses, exclusive.

Unlike the **integer** presentation type, number does not check input for violation of the **:base** specification. Thus, in the final **accept** function above, a `2` is entered and returned even though binary numbers are sought.

number is supertype to all other numeric presentation types. See the section "Types of Numbers". It provides the family with its printer and parser functions. As with other Common Lisp types, **number** is subtype to **sys:expression**, from which it inherits a type history.

For an overview of presentation types and related facilities, see the section "Presentation Substrate Facilities".

net:object *Presentation Type*

Type for accepting or presenting network objects.

Examples:

```
(accept '((net:object))) ==>
Enter a namespace object: (Class) HELP==>
You are being asked to enter a namespace object.

These are the possible namespace classes:
  File-System  Printer
  Host         Site
  Namespace    User
  Network

Enter a namespace object: User JO
#<USER JO 13731243>
((NET:OBJECT))

(accept '((net:object))) ==>
Enter a namespace object [default JO]: Host OYSTERCATCHER
#<LISPM-HOST OYSTERCATCHER 13702373>
((NET:OBJECT))

(present (net:find-object-named :network "chaos")
'((net:object))) ==>CHAOS
#<DISPLAYED-PRESENTATION 275037261>

(accept '((net:object))) ==>
Enter a namespace object [default OYSTERCATCHER]: CHAOS
#<CHAOS-NETWORK CHAOS 13700033>
CHAOS:CHAOS-NETWORK
```

When accepting **net:object** input, the user is first prompted for the class of the object. The possible classes, from File-System to User, are listed in the help display shown in the first example above. After entering the class of net object, the user should type a space and then the name of the object itself.

The **net:object** presentation type is built on a flavor of the same name. It inherits its printer and type history from the **sys:expression** presentation type. It is, in turn, supertype to several other network-related types:

net:host
net:local-host
neti:namespace

net:network
neti:site
net:user

When you wish handle a particular class of network object, as opposed to any object, one of the above presentation types might be more suitable than **net:object**.

net:object is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

or (&rest *types*) *Presentation Type*

Compound type for accepting objects as one of two or more possible presentation types. (Presenting objects as **or** types is not useful.)

types Data arguments specifying the possible presentation types.

Examples:

```
(present 'some-symbol) ==>SOME-SYMBOL
#<DISPLAYED-PRESENTATION 274336643>

(present "some-string") ==>some-string
#<DISPLAYED-PRESENTATION 274337201>

(accept '((or symbol string))) ==>
Enter a symbol or a string: SOME-SYMBOL
SOME-SYMBOL
SYMBOL

(accept '((or symbol string))) ==>
Enter a symbol or a string [default SOME-SYMBOL]: some-string
"some-string"
STRING
```

Some tips on the use of **or**: If you give it to **accept** directly or use it in a **cp:define-command**, remember to specify a type for the default using the **:default-type** keyword. **or** is useful for automatically writing token rescanning multiple syntax parsers for your own presentation type. Use it in an **:expander**. See the section "**:expander** Option to **define-presentation-type**". The types **null-or-type**, **token-or-type**, and **type-or-string** are provided for the common cases.

The **or** presentation type has access to the **sys:expression** type history. The value provided as a default in an **accept** of an **or** type is the most recently accepted object whose presentation type is one of the possible *types*.

For an overview of presentation types and related facilities, see the section "Presentation Substrate Facilities".

dw:out-of-band-character (&rest *chars*) *Presentation Type*

Type for accepting characters that would normally be interpreted as input editor commands, such as the shifted characters c-B or c-E.

chars Data arguments specifying the shifted characters.

Examples:

```
(accept '((dw:out-of-band-character #\c-F #\m-Scroll #\m-C))) ==>
Enter one of the characters c-F, m-SCROLL, or m-sh-C: m-SCROLL
#\m-Scroll
((DW:OUT-OF-BAND-CHARACTER #\c-F #\m-Scroll #\m-C))

(accept '((dw:out-of-band-character #\c-F #\m-SCROLL #\m-C))) ==>
Enter one of the characters c-F, m-SCROLL, or m-sh-C
[default Meta-Scroll]: c-F
#\c-F
((DW:OUT-OF-BAND-CHARACTER #\c-F #\m-Scroll #\m-C))
```

dw:out-of-band-character is subtype to the **character** presentation type, from which it inherits its printer and type history. The type history is pruned to include only previously accepted out-of-band characters.

To accept or present ordinary characters, use **character**. See the presentation type **character**.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

package *Presentation Type*

Type for accepting or presenting packages.

Examples:

```
(present (find-package 'dynamic-windows) '((package))) ==>
DYNAMIC-WINDOWS
#<DISPLAYED-PRESENTATION 274353464>

(accept '((package))) ==>
Enter a package: DYNAMIC-WINDOWS
#<Package DYNAMIC-WINDOWS 45652740>
((PACKAGE))

(accept '((package))) ==>
Enter a package [default DYNAMIC-WINDOWS]: SCL
#<Package SYMBOLICS-COMMON-LISP 46405507>
((PACKAGE))
```

The **package** presentation type supports a type history.

package is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Presentation Substrate Facilities".

pathname &key (*default-version :newest*) (*default-type nil*) (*default-name nil*) *dont-merge-default* (*direction :read*) (*format :normal*) *Presentation Type*

Type for accepting or presenting pathnames.

:default-version Presentation option specifying the default version number of an accepted file. The default value for this option is **:newest**, the newest file version.

:default-type Presentation option specifying the default file type, for example, "lisp", "text", "data", and so on. The default value for this option is **nil**.

:default-name Presentation option specifying the default file name. The default value for this option is **nil**.

:dont-merge-default Presentation option specifying whether to prevent merging of a partially specified pathname entered by the user against the default pathname. The default value for this option is **nil**, meaning that merging occurs when appropriate; that is, parts of the pathname not entered by the user are supplied from the default.

Suppression of merging against the default and providing a different default (against which merging may or may not be enabled) are different issues. To deal with the latter, use the **:default** option to **accept**. (See the function **accept**.) An example follows:

```
(accept '((pathname) :default-type nil)
        :default (send (fs:default-pathname)
                       :new-pathname :type nil
                       :version :newest))
```

:direction

Presentation option specifying either **:read** (the default) or **:write**. The value supplied is passed through to **fs:complete-pathname** and affects completion behavior. (See the function **fs:complete-pathname**.)

Use the default (**:read**) if the user is likely to enter the pathname of an already existing file when prompted by **accept**, **:write** otherwise.

:format Presentation option specifying the output format of the pathname. There are four choices:

:normal For example, `S:>mb>dw-pgms>fancy-windows.lisp`. This is the default format.

:directory For example, `>mb>dw-pgms>`. The host, file name, and file type are not displayed.

:dired For example, `fancy-windows.lisp`. Only the file name and type are displayed.

:editor For example, `fancy-windows.lisp >mb>dw-pgms S`. The display format is that used by Zmacs.

Examples:

```
(present #p"y:>yosemite-s>gold.text") ==>Y:>yosemite-s>gold.text
#<DISPLAYED-PRESENTATION 274370245>

(present #p"y:>yosemite-s>gold.text" '((pathname)
                                       :format :editor)) ==>
gold.text >yosemite-s Y:
#<DISPLAYED-PRESENTATION 274370523>

(accept '((pathname))) ==>
Enter the pathname of a file: gold.text >yosemite-s Y:
#P"Y:>yosemite-s>gold.text"
((PATHNAME) :FORMAT :EDITOR)

(accept '((pathname) :default-version 1)) ==>
Enter the pathname of a file
[default Y:>yosemite-s>gold.text]: silver
#P"Y:>yosemite-s>silver.text.1"
FS:LMFS-PATHNAME

(accept '((pathname) :default-type "data"
                     :default-name "the-rabbit")) ==>
Enter the pathname of a file
[default Y:>yosemite-s>silver.text.1]: Y:>yosemite-s>
#P"Y:>yosemite-s>the-rabbit.data.newest"
FS:LMFS-PATHNAME

(accept '((pathname) :dont-merge-default t)) ==>
Enter the pathname of a file
[default Y:>yosemite-s>the-rabbit.data]: other-varmints
#P"Y:other-varmints"
FS:LMFS-PATHNAME
```

```
(accept '((pathname))) ==>
Enter the pathname of a file
[default Y:>other-varmints]: VIXEN:/b-bunny/y-s.data
#P"VIXEN:/b-bunny/y-s.data"
FS:UNIX42-PATHNAME
```

The **pathname** presentation type supports a type history.

pathname is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

Two subtypes to **pathname** are included among the documented predefined presentation types:

- **fs:directory-pathname**
- **fs:wildcard-pathname**

For an overview of presentation types and related facilities, see the section "Presentation Substrate Facilities".

sys:printer *Presentation Type*

Type for accepting or presenting printers, that is, hardcopy output devices.

Examples:

```
(accept '((sys:printer)))
Enter a printer [default Symbolics Paradigm]: Symbolics Paradigm
#<LGP2-PRINTER PARADIGM 13701250>
SYS:PRINTER

(present (net:find-object-named :printer "Asahi")
         '((sys:printer))) ==>Asahi Shimbun
#<DISPLAYED-PRESENTATION 275641455>
```

The **sys:printer** presentation type supports a type history.

sys:printer is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

neti:protocol-name (&key *service*) *Presentation Type*

Type for accepting or presenting names of network protocols.

Examples:

```
(accept '((neti:protocol-name))) ==>
Enter a network protocol: Domain-Simple
:DOMAIN-SIMPLE
((NETI:PROTOCOL-NAME))

(present :converse '((neti:protocol-name))) ==>CONVERSE
#<DISPLAYED-PRESENTATION 275603433>

(present (car neti:*protocol-list*)
         '((neti:protocol-name))) ==>MANDELBROT
#<DISPLAYED-PRESENTATION 275607026>
```

There is no type history for the **neti:protocol-name** presentation type.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

dw:raw-text *Presentation Type*

Type providing access to the individual characters from which all textual presentations are constructed. This type is for the exclusive use of mouse handlers, usually as the *from-presentation-type* argument. (For more on handlers, see the section "Mouse Handler Facilities".) You cannot use it to accept or present text or characters.

The following example is the source code for a translating mouse handler defined on **dw:raw-text**, translating it to an internal presentation type, **dw::character-style-family**:

```
(define-presentation-translator
        si:characters-character-style-family
        (dw:raw-text dw::character-style-family) (bp)
  (when (< (second bp) (string-length (first bp)))
    (let ((char (aref (first bp) (second bp))))
      (si:cs-family (si:char-style char)))))
```

zwei:bp is a presentation type inheriting from **dw:raw-text**, and used for accessing text characters in editor buffers.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

satisfies (*satisfies-function*) *Presentation Type*

Type for forming compound presentation types using the presentation type **and** by specializing an initial type with the argument *satisfies-function*, a predicate that returns **t** or **nil** when applied to the object in question. The **satisfies** type is used *only* as part of an **and** presentation type. For example,

```
(accept '((and character ((satisfies digit-char-p)))
                                    :description "a digit"))
```

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

sequence (&optional (*type* '*)) &key (*sequence-delimiter* #\,) (*echo-space t*)
Presentation Type

Type for accepting or presenting one or more objects of a specified presentation type.

type Presentation type for the objects in the sequence. The specified type is a data argument to the **sequence** presentation type.

The default *type* argument, '*, results in the use of the **t** presentation type. Because **t** has no parser and uses **princ** as its printer, not supplying the *type* argument when you use the **sequence** presentation type does not produce useful results.

:sequence-delimiter Presentation option specifying the character used to delimit items in the sequence; the default is the comma character, #\,.

When accepting objects in an enumerated sequence, the user must enter the sequence-delimiter character between items.

:echo-space Presentation option specifying whether to echo a space character after the comma (or other **:sequence-delimiter** character) is typed; the default is **t**.

:element-default *object*Specifies that *object* is to be used as the default when doing a recursive **accept** of the first element of the sequence. This is different than specifying the default for the sequence, since you might want the sequence default to be empty, and yet you might want to specify that if the user decides to type an element, that element should be parsed against a particular default. For example:

```
(accept '((sequence pathname)) :default '())
Enter pathnames of files: tennis
  ;completes to "ACME:tennis"
=> (#P"ACME:tennis")
   ((SEQUENCE-ENUMERATED FS:LMFS-PATHNAME))

(accept '((sequence pathname)
          :element-default #P"S:>Joe>bowling.text")
        :default '())
Enter pathnames of files: golf
  ;completes to "ACME:>Joe>golf.text.newest"
=> (#P"ACME:>Joe>golf.text.newest")
   ((SEQUENCE-ENUMERATED FS:LMFS-PATHNAME))
```

Although not a subtype, **sequence** can be regarded as a specialized version of the **sequence-enumerated** presentation type. Instead of specifying a series of presentation types as in the case of **sequence-enumerated**, you specify only one type for the entire series of objects. In fact, when objects are entered individually to an **accept** of a **sequence**, the types of the objects, although identical, are enumerated. Observe this behavior in the first example below.

Examples:

```
(accept '((sequence package))) ==>
Enter one or more packages
[default SYMBOLICS-COMMON-LISP]: SCL, DW, TV, SCT
(#<Package SYMBOLICS-COMMON-LISP 46405507>
#<Package DYNAMIC-WINDOWS 45652740>
#<Package TV 46031453>
#<Package SYSTEM-CONSTRUCTION-TOOL 46366410>)
((SEQUENCE-ENUMERATED PACKAGE PACKAGE PACKAGE PACKAGE))

(present '(0 16 32 64) '((sequence ((integer) :base 16))))
#<DISPLAYED-PRESENTATION 274631670>

(accept '((integer))) ==>
Enter an integer: 40
64
((INTEGER) :BASE 16)

(accept '((sequence integer))) ==>
Enter one or more integers: 0, 10, 20, 40
(0 16 32 64)
((SEQUENCE ((INTEGER) :BASE 16)))
```

Note that when you have presented a **sequence** of objects, the objects are subsequently acceptable as input either as individual objects or as the **sequence**. This is shown by the last three examples above. We **present** a series of integers, and subsequently click on one of them (40) to enter it to an **accept** or an **integer**; and then click on the entire sequence to give it to an **accept** of and **integer** sequence.

The syntax for use of presentation type arguments with **sequence** is shown in the next example.

```
(accept '((sequence sys:printer) :sequence-delimiter #\;)
```

The **sequence** presentation type has access to the type history supported, if any, by the specified **type**.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

sequence-enumerated (&rest *data-types*) &key (*sequence-delimiter* "") (*echo-space* **t**)
Presentation Type

Compound type for accepting or presenting a sequence of objects, each of a specified presentation type.

data-types
The presentation types of the objects. These are the data arguments to the **sequence-enumerated** presentation type.

:sequence-delimiter Presentation option specifying the character used to delimit items in the sequence; the default is the comma character, ",".

When accepting objects in an enumerated sequence, the user must enter the sequence-delimiter character between items.

:echo-space Presentation option specifying whether to echo a space character after the comma (or other **:sequence-delimiter** character) is typed; the default is **t**.

:element-defaults '(*object1 object2 ...*) Specifies an enumerated sequence of *objects* to be used one by one as the default for each recursive **accept** involved in accepting elements of the enumerated sequence. This is different than specifying the default for the sequence, since you might want the sequence default to be empty, and yet you might want to specify that *if* the user decides to type an element, that element should be parsed against a particular default. For example:

```
(accept '((sequence-enumerated pathname pathname)
          :element-defaults (#P"ACME:>JDoe>file.text"
                             #P"ACME:>Fred>file.text"))
        :provide-default nil)

Enter the pathnames of two files: a, b
  ;completes to "ACME:>JDoe>a.text.newest, ACME:>Fred>b.text.newest"
=> (#P"ACME:>JDoe>a.text.newest" #P"ACME:>Fred>b.text.newest")
   ((SEQUENCE-ENUMERATED FS:LMFS-PATHNAME FS:LMFS-PATHNAME))
```

Examples:

```
(accept '((sequence-enumerated (integer 1 10)
                sys:form string))) ==>
Enter an integer greater than or equal to 1 and less
than or equal to 10, A Lisp expression to be evaluated,
and a string: 5, (setq alpha "bravo"), "Not very useful"
(5 (SETQ ALPHA "bravo") "Not very useful")
((SEQUENCE-ENUMERATED (INTEGER 1 10) SYS:FORM STRING))

(present `(,(pathname "y:>ui.lisp") telson
           ,(find-package "dynamic-windows"))
         '((sequence-enumerated pathname symbol package))) ==>
Y:>ui.lisp, TELSON, and DYNAMIC-WINDOWS
#<DISPLAYED-PRESENTATION 444476230>
```

The **sequence-enumerated** presentation type does not support a type history.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

tv:sheet *Presentation Type*

Type for accepting or presenting window objects.

Examples:

```
(accept '((tv:sheet))) ==>
Enter a window [default Graphic Editor 1]: HELP ==>
You are being asked to enter a window.

These are the possible windows:

  Command ... (3)   Dynamic ... (19)  Mode Line Window 4   Zmail ... (4)
  Converse          Fsmaint ... (2)   Terminal 1           Zwei ... (3)
  Converse Frame 1  Graphic Editor 1  Typein ... (3)       Who ... (8)
  Dex ... (6)       Main ... (2)      Zmacs ... (3)

Enter a window [default Dex Frame 1]: Graphic Editor 1
#<PROGRAM-FRAME Graphic Editor 1 701735 deexposed>
((TV:SHEET))

(present (tv:make-window 'dw:dynamic-window)
         '((tv:sheet)))Dynamic Window 1
#<DW::DISPLAYED-PRESENTATION 650364063>
```

The **tv:sheet** presentation type supports a type history.

tv:sheet is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

neti:site *Presentation Type*

Type for accepting or presenting site objects.

Examples:

```
(present net:*local-site* '((neti:site))) ==>SCRC
#<DISPLAYED-PRESENTATION 275626405>

(accept '((neti:site))) ==>
Enter a site: SCRC
#<SITE SCRC 13700014>
((NETI:SITE))
```

Through flavor inheritance, the **neti:site** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

sys:stack-frame *Presentation Type*

Type for accepting or presenting stack frames. This presentation type is intended primarily for use by the debugger and debugging functions.

The following example shows entry into the debugger from an editor typeout window. The debugger was entered because **oddp** was called with no arguments. The frame, **ODDP**, containing the error is at the top of the stack.

```
Command: (oddp) ==>
Trap: The function ODDP was called with too few arguments.
```

ODDP:
--Missing args:--
Arg 0 (INTEGER)
s-A, RESUME: Supply the missing arguments.
s-B: Retry the FUNCALL-N-RETURN instruction
s-C, ABORT: Return to Breakpoint ZMACS in Editor Typeout Window
s-D: Editor Top Level
s-E: Restart process ZMACS-WINDOWS
→ *Eval (program):* (setq stk-frm (accept '((sys:stack-frame)))) ==>
Enter a stack frame: ODDP
(#<DTP-LOCATIVE 52700741> . #<TOO-FEW-ARGUMENTS-TRAP 44070612>)
→ *Eval (program):* (present stk-frm '((sys:stack-frame))) ==>ODDP
#<DISPLAYED-PRESENTATION 276024201>
→ Abort *Abort*
Return to Breakpoint ZMACS in Editor Typeout Window

sys:stack-frame does not support a type history.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

string *(*&optional *size)* &key *delimiters* *Presentation Type*

size An integer specifying the length of the string.

Type for accepting or presenting strings.

:delimiters Presentation option specifying a list of characters serving as string delimiters (terminators) during input of strings to **accept**. The default delimiters are **#\return** and **#\end**.

Examples:

```
(accept '((string))) ==>
Enter a string: "Morgan the Pirate"
"Morgan the Pirate"
((STRING))

(accept '((string) :delimiters (#\line))) ==>
Enter a string (end with LINE)
[default Morgan the Pirate]: Several species
of small, furry
creatures gathered
together in a cave ...
```

```
"Several species
of small, furry
creatures gathered
together in a cave ..."
((STRING) :DELIMITERS (#\Line))

(present "Another whimsical string") ==>Another whimsical string
#<DISPLAYED-PRESENTATION 274760165>

(accept '((string)))
Enter a string: Another whimsical string
"Another whimsical string"
STRING
```

The **string** presentation type supports a type history.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

subset (&rest *keywords*) *Presentation Type*

Type for accepting or presenting zero or more objects from a group of keyword identifiers.

keywords The set of keywords. These are data arguments to the **subset** presentation type.

Examples:

```
 (accept '((subset :mercenaria :mya :mytilus))) ==>
Enter a subset of the identifiers MERCENARIA,
MYA, and MYTILUS: Mercenaria, Mytilus
(:MERCENARIA :MYTILUS)
((SUBSET :MERCENARIA :MYA :MYTILUS))

(present '(:mya) '((subset :mercenaria :mya :mytilus))) ==>MYA
#<DISPLAYED-PRESENTATION 444621057>
```

When accepting input of this type, the user must separate identifiers with commas. If input is terminated without any identifiers having been entered, **accept** returns **nil**.

A type history is not available for the **subset** presentation type.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

symbol *Presentation Type*

Type for accepting or presenting symbols.

Examples:

```
(accept '((symbol)))
Enter a symbol: RNA
RNA
((SYMBOL))

(accept '((symbol)))
Enter a symbol [default RNA]: DNA
DNA
((SYMBOL))

(present 't-RNA)
#<DISPLAYED-PRESENTATION 274753204>

(accept '((symbol)))
Enter a symbol [default RNA]: T-RNA
T-RNA
SYMBOL
```

The **symbol** presentation type inherits its parser, printer, and type history from the **sys:expression** presentation type.

To accept or present symbol names as opposed to symbol objects, use the **symbol-name** presentation type.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

symbol-name *Presentation Type*

Type for accepting or presenting a symbol name, that is, the print name of a symbol. (For accepting or presenting symbol objects, use the **symbol** presentation type.)

Examples:

```
(accept '((symbol-name)))
Enter a symbol name: T-M-S
"T-M-S"
((SYMBOL-NAME))

(present "T-M-S" '((symbol-name))) ==>T-M-S
#<DISPLAYED-PRESENTATION 444645436>
```

The **symbol-name** presentation type inherits its printer and type history from the **string** presentation type.

symbol-name is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

sct:system ((&key (*patchable-only* **nil**))) *Presentation Type*

Type for accepting or presenting systems.

:patchable-only Data option restricting systems to those that are patchable; the default is **nil**.

Examples:

```
(accept '((sct:system))) ==>
Enter a system: Zmail
#<SCT:SYSTEM ZMAIL 520001430>
((SCT:SYSTEM))

(accept '((sct:system :patchable-only t))) ==>
Enter a system: Documentation Database
#<SYSTEM DOC 261374510>
((SCT:SYSTEM :PATCHABLE-ONLY T))

(present (sct:find-system-named 'extended-help)
        '((sct:system))) ==>Extended Help
#<DISPLAYED-PRESENTATION 274651506>

(present (car sct:*all-systems*) '((sct:system))) ==>System
#<DISPLAYED-PRESENTATION 274641244>
```

The **sct:system** presentation type supports a type history.

sct:system is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

sct:system-version *Presentation Type*

Type for accepting or presenting a system version designator. Three kinds of designators are permitted:

- a positive, non-zero integer
- one of the special keywords **:released**, **:latest**, or **:newest**
- an arbitrary keyword

Examples:

```
(accept '((sct:system-version))) ==>
Enter a version designator: 2
2
((SCT:SYSTEM-VERSION))

(accept '((sct:system-version))) ==>
Enter a version designator: Released
:RELEASED
((SCT:SYSTEM-VERSION))

(accept '((sct:system-version))) ==>
Enter a version designator: arbitrary
:ARBITRARY
((SCT:SYSTEM-VERSION))

(present :newest '((sct:system-version))) ==>Newest
#<DISPLAYED-PRESENTATION 274677471>
```

The **sct:system-version** presentation type does not support a type history.

sct:system-version is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

t *Presentation Type*

Type that is supertype to all other presentation types.

t occupies a necessary spot (the top) in the type hierarchy, and is important for that reason. However, it has no parser and cannot be used with **accept**. Moreover, objects presented as **t** presentations are not mouse-sensitive in any input context.

One of the key uses for the **t** type is in mouse handlers, as the *from-presentation-type* or *to-presentation-type*. If the former, it means that the handler in question is potentially applicable to any type of presentation; if the latter, it means that the handler is potentially applicable in any input context. See the section "Mouse Handler Concepts".

For an overview of presentation types and related facilities: See the section "Using Presentation Types".

time:time-interval *Presentation Type*

Type for accepting or presenting intervals of time. Internally, time intervals are in seconds; externally, in seconds, minutes, hours, days, weeks, and years. **nil** is represented as "never".

Examples:

```
(accept '((time:time-interval))) ==>
Enter a time interval: 1 second
1
((TIME:TIME-INTERVAL))

(accept '((time:time-interval))) ==>
Enter a time interval [default 1 second]: 1 minute
60
((TIME:TIME-INTERVAL))

(accept '((time:time-interval))) ==>
Enter a time interval [default 1 minute]: 1 hour
3600
((TIME:TIME-INTERVAL))

(present 3661 '((time:time-interval))) ==>1 hour 1 minute 1 second
#<DISPLAYED-PRESENTATION 276047342>

(present nil '((time:time-interval))) ==>never
#<DISPLAYED-PRESENTATION 276047575>
```

Note that time intervals are specified with integers only.

The **time:time-interval** presentation type supports a type history.

time:time-interval is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types". See the function **si:parse-interval-or-never**.

time:time-interval-60ths *Presentation Type*

Type for accepting or presenting intervals of time. Internally, time intervals are in 60ths of a second; externally, in seconds, minutes, hours, days, weeks, and years. **nil** is represented as "never".

Examples:

```
(accept '((time:time-interval-60ths))) ==>
Enter a time interval 60ths: 1 second
60
((TIME:TIME-INTERVAL-60THS))

(accept '((time:time-interval-60ths))) ==>
Enter a time interval 60ths [default 1 second]: 1 minute
3600
((TIME:TIME-INTERVAL-60THS))
```

```
(accept '((time:time-interval-60ths))) ==>
Enter a time interval 60ths [default 1 minute]: 1 hour
216000
((TIME:TIME-INTERVAL-60THS))

(present 3661 '((time:time-interval-60ths))) ==>1 minute 1 second
#<DISPLAYED-PRESENTATION 276061445>

(present 30 '((time:time-interval-60ths))) ==>0 seconds
#<DISPLAYED-PRESENTATION 276062366>

(present 31 '((time:time-interval-60ths))) ==>1 second
#<DISPLAYED-PRESENTATION 276062621>

(present nil '((time:time-interval-60ths))) ==>never
#<DISPLAYED-PRESENTATION 276061700>
```

Note that time intervals are specified with integers only; also, that they are rounded to the nearest second when presented.

The **time:time-interval-60ths** presentation type supports a type history.

time:time-interval-60ths is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

time:timezone &key *force-numeric-p* *Presentation Type*

Type for accepting or presenting timezones.

Timezones are represented externally either by commonly accepted abbreviations, for example, "EST" (for Eastern Standard Time), or by a signed digit string, for example, "-0500". The sign of the digit string indicates the location of the timezone relative to Greenwich; positive means east, negative west.

Internally, timezones are represented by numbers in the form *n.0* or *n.5*. Note that the sign of the externally displayed digit string is opposite to that of the number used internally. The printed digit string "-0530", for example, corresponds to an internal representation of `5.5`.

:force-numeric-p Presentation option specifying whether a timezone is presented only by a signed digit string. The default is **nil**; this causes the timezone's unique abbreviation, if there is one, to be printed. If a unique abbreviation is not available, the digit string is printed regardless of the value supplied for this option.

Examples:

```
(accept '((time:timezone))) ==>
Enter a defined timezone symbol or an hour offset from GMT
such as +0500 (east of GMT) or -0330 (west of GMT): EST
5
((TIME:TIMEZONE))

(accept '((time:timezone))) ==>
Enter a defined timezone symbol or an hour offset from GMT
such as +0500 (east of GMT) or -0330 (west of GMT): -0500
5
((TIME:TIMEZONE))

(present 5 '((time:timezone))) ==>EDT
#<DISPLAYED-PRESENTATION 274454265>

(present 5 '((time:timezone) :force-numeric-p t)) ==>-0400
#<DISPLAYED-PRESENTATION 274454520>
```

Note in the last two examples, created in July, that the displayed presentations reflect daylight savings time. At sites in 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 and digit string. For other timezones, the switch must be made manually. In either case, **time:timezone** presentations display the current setting for daylight savings time. For more information, see the section "Specifying a Time Zone for Your Site".

The **time:timezone** presentation type does not support a type history.

time:timezone is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

token-or-type (*special-tokens otherwise-type*) *Presentation Type*

Compound type for accepting or presenting a special token — for example "None", "Any", "All" — or an object of a specified type.

special-tokens Data argument specifying a list of tokens. The list is an alist: each item is a dotted pair of a print string and its object: ((*String-1* . *object-1*) (*string-2* . *object-2*) ... (*string-n* . *object-n*))

otherwise-type Data argument specifying the presentation type to use for accepting or presenting objects other than listed tokens.

Examples:

```
(defun try-it ()
  (accept '((token-or-type (("either" . :either)
                            ("neither" . :neither)
                            ("both" . :both))
                           ((subset  :fixed-wing :rotary-wing))))
                           :prompt
"Enter \"fixed-wing\", \"rotary-wing\", \"either\", \"neither\", or \"both\""))

(try-it)==>
Enter "fixed-wing", "rotary-wing", "either", "neither", or "both": Fixed-Wing
(:FIXED-WING)
SUBSET

(try-it)==>
Enter "fixed-wing", "rotary-wing", "either", "neither", or "both": neither
:NEITHER
((ALIST-MEMBER :ALIST (("either" . :EITHER) ("neither" . :NEITHER) ("both" . :BOTH)))
 :DESCRIPTION "special token")
```

Here is an example of a common idiom in the system:

```
(cp:define-command (com-filename-example :command-table "GLOBAL"
                                         :provide-output-destination-keyword nil)
    ((pathnames '((token-or-type (("All" :all))
                                 ((sequence pathname))))
                :default :all))
  (present pathnames))
```

If the presentation type specified by *otherwise-type* supports a type history, the history is available for objects of that type.

token-or-type is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

type-or-string (*presentation-type*) &key *reject-null-string string-if-quoted*
Presentation Type

Compound type for accepting or presenting objects of a specified type or strings.

presentation-type Data argument specifying the presentation type to use for accepting or presenting objects which are not strings.

:reject-null-string Takes a boolean value. A non-**null** value allows you to give no default value, but still not allow a null (string) as input. A **null** value means that this presentation type refuses a null string only if there is a non-**null** default. This keyword enables you to control whether or not a null string is allowed as input separately from the default value.

:string-if-quoted Takes a boolean value, which controls whether the following is parsed as a *type* or as a string when explicit quotes are present.

((type-or-string *type*) :string-if-quoted *string-if-quoted*)

The default, if **:string-if-quoted** is not supplied, or is **nil**, is to always try to parse it first as a type and then as a string if that fails. However, if **:string-if-quoted** is **t**, then explicit quoting (with doublequotes) will force the object to be parsed as a string.

Examples:

```
(accept '((type-or-string net:user)))
Enter a user: JWALKER
#<USER JWALKER 6434203>
SI:USER

(accept '((type-or-string net:user))
        :default (dw:presentation-type-default 'net:user)
Enter a user [default JWALKER]: JBIRD
"JBIRD"
STRING

(present 'JWALKER '((type-or-string net:user))) ==>JWALKER
#<DISPLAYED-PRESENTATION 445112577>

(present "JWALKER" '((type-or-string net:user))) ==>JWALKER
#<DISPLAYED-PRESENTATION 445105072>
```

Here is an example of **:string-if-quoted**:

```
;;; the following treats "3" as an integer
(accept '(type-or-string integer))

;;; the following treats "3" as a string
(accept '((type-or-string integer) :string-if-quoted t)
```

Although the type specified by *presentation-type* might support a type history, accepting a **type-or-string** does not automatically display the default; you have to provide one to **accept** yourself. This is illustrated in the second **accept** form above.

Note in the **present** examples that the objects presented have the same printed representation. The first, however, is an **net:user** object, the second a string object. Each will only be mouse-sensitive in the appropriate input context.

type-or-string is one of a number of types defined in sys:dynamic-windows;standard-presentation-types.lisp. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

time:universal-time &key *base-time past-p must-have timezone long-date brief (replace-relative-time* **nil***) present-based* *Presentation Type*

Type for accepting or presenting universal times. (Universal time is measured in seconds elapsed since midnight, Jan 1, 1900, GMT.)

When accepting universal times, a large variety of input formats are possible. For more information and examples, see the section "Reading Dates and Times".

The following keyword options, all presentation arguments, are available. The first three — **:base-time**, **:past-p**, and **:must-have** — affect the input of universal times. The remaining options, — **:timezone**, **:long-date**, **:brief**, **:replace-relative-time**, and **:present-based** — affect their output. The discussion of each includes examples.

:base-time Presentation option specifying a base time from which defaults are taken for unspecified components when accepting a universal time.

The base time is specified as the number of seconds since midnight, January 1, 1986 (that is, 1/01/00 00:00:00). In the following example, the base time is midnight, January 1, 1986.

Example:

```
(accept '((time:universal-time) :base-time 2713928400
          :description "a date in 1986"))) ==>
Enter a date in 1986: 3/2 ==>3/02/86 00:00:00
2719112400
((TIME:UNIVERSAL-TIME) :BASE-TIME 2713928400)
```

:past-p Presentation option specifying whether partially specified times default to the nearest corresponding universal time in the past or future; the default is **nil**.

The following examples were created in 7/86.

Examples:

```
(accept '((time:universal-time))) ==>
Enter a universal time: 3/2 ==>3/02/87 00:00:00
2750648400
((TIME:UNIVERSAL-TIME))

(accept '((time:universal-time))) ==>
Enter a universal time
[default 3/02/87 00:00:00]: 8/2 ==>8/02/86 00:00:00
2732328000
((TIME:UNIVERSAL-TIME))
```

```
(accept '((time:universal-time) :past-p t)) ==>
Enter a universal time in the past
[default 8/02/86 00:00:00]: 3/2 ==>3/02/86 00:00:00
2719112400
((TIME:UNIVERSAL-TIME) :PAST-P T)

(accept '((time:universal-time) :past-p t)) ==>
Enter a universal time in the past
[default 3/02/86 00:00:00]: 8/2 ==>8/02/85 00:00:00
2700792000
((TIME:UNIVERSAL-TIME) :PAST-P T)
```

:must-have Presentation option specifying that the year field or second field or both must be explicitly entered when accepting a universal time. The required fields are provided as a list of keywords.

Example:

```
(accept '((time:universal-time) :must-have (:year year))) ==>
Enter a universal time, year is required
[default 7/07/86 19:19:00]: 12/12 ==>
no year supplied
Type RUBOUT to correct your input.
Enter a universal time, year is required
[default 7/07/86 19:19:00]: 12/12/47 00:00:00
1512968400
((TIME:UNIVERSAL-TIME) :MUST-HAVE (YEAR))
```

:timezone Presentation option specifying the timezone used when presenting universal times. **time:*timezone*** provides the default value.

Supply the value as a number (either *n* or *n.5*): 0 specifies Greenwich Mean Time; positive numbers timezones to the west of Greenwich; negative numbers timezones to the east. (For more on timezone representations, see the presentation type **time:timezone**.)

Examples:

```
(present 123456789 '((time:universal-time)
                     :timezone -5)) ==>12/1/03 02:33:09
#<DISPLAYED-PRESENTATION 274337427>

(present 123456789 '((time:universal-time)
                     :timezone 0)) ==>11/30/03 21:33:09
#<DISPLAYED-PRESENTATION 274340115>
```

```
(present 123456789 '((time:universal-time)
                     :timezone 5)) ==>11/30/03 16:33:09
#<DISPLAYED-PRESENTATION 274337662>

(present 123456789 '((time:universal-time)
                     :timezone 5.5)) ==>11/30/03 16:03:09
#<DISPLAYED-PRESENTATION 274345125>
```

:long-date Presentation option specifying that the date be presented as in the following example when presenting universal times;

```
(present 123456789 '((time:universal-time)
                     :long-date t)) ==>
Monday the thirtieth of November, 1903; 4:33:09 pm
#<DISPLAYED-PRESENTATION 274353534>
```

:brief Presentation option specifying whether presented times should be printed briefly, that is, without the seconds field. Contrast the following two examples:

```
(present (time:get-universal-time)
  '((time:universal-time))) ==>7/07/86 14:55:35
#<DISPLAYED-PRESENTATION 274421523>

(present (time:get-universal-time)
  '((time:universal-time) :brief t)) ==>7/7/86 14:55
#<DISPLAYED-PRESENTATION 274421756>
```

:replace-relative-time Presentation option specifying that if a relative time, for example, `3 minutes from now`, is supplied, it should be replaced with the actual time in the output history. The default is **nil**, since usually if you are giving a relative time and re-run that command, you want the same relative offset from the current time, not the time the command was previously run.

:present-based Presentation option specifying that missing components in the supplied time default to the beginning of the smallest unsupplied unit of time. For example, `5:00 pm` means 5:00 pm today, whether it is before or after 5:00 pm. `Thursday` means Thursday of this week, whether it is currently Monday or Friday. The default is **nil**, meaning that missing components default to the future.

For example, if it is Wednesday, December 27, and you specify Monday:

```
(accept '(time:universal-time))
Enter a universal time [default 12/28/89 10:00:00]: Monday
=>   1/01/90 00:00:00
     2840158800
     (TIME:UNIVERSAL-TIME)

(accept '((time:universal-time) :present-based t))
Enter a universal time [default 1/01/90 00:00:00]: Monday
=>   12/25/89 00:00:00
     2839554000
     ((TIME:UNIVERSAL-TIME) :PRESENT-BASED T)
```

time:universal-time is one of a number of types defined in SYS:DYNAMIC-WINDOWS;PRESENTATION-TYPES.LISP. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

net:user *Presentation Type*

Type for accepting or presenting user objects.

Examples:

```
(present si:*user* '((si:user))) ==>REG
#<DISPLAYED-PRESENTATION 275633757>

(accept '((si:user))) ==>
Enter a user: REG
#<USER REG 13730364>
((SI:USER))
```

Through flavor inheritance, the **net:user** presentation type is subtype to the **net:object** type, from which it inherits a type history. The history inherited includes all accepted objects of the **net:object** type; that is, no pruning of the history occurs.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

fs:wildcard-pathname &key (*default-version :newest*) (*default-type nil*) (*default-name nil*) *dont-merge-default* (*direction :read*) (*format :normal*) *Presentation Type*

Type for accepting or presenting pathnames that include wildcard characters.

This presentation type can be useful if you need to distinguish unequivocally between pathname presentations that include wildcard characters (asterisks) and other file pathname presentations. For example, if you can arrange for the availability to your users of some **fs:wildcard-pathname** presentations, then mouse handlers performing functions specifically on pathnames containing wildcards can be defined

that do not have to test whether a given **pathname** presentation includes a wildcard character.

fs:wildcard-pathname is a subtype of the **pathname** presentation type, from which it inherits a printer, parser, and type history. It also takes the same keyword arguments, as follows:

:default-version Presentation option specifying the default version number of an accepted file. The default value for this option is **:newest**, the newest file version.

:default-type Presentation option specifying the default file type, for example, "lisp", "text", "data", and so on. The default value for this option is **nil**.

:default-name Presentation option specifying the default file name. The default value for this option is **nil**.

:dont-merge-default Presentation option specifying whether to prevent merging of a partially specified pathname entered by the user against the default pathname. The default value for this option is **nil**, meaning that merging occurs when appropriate; that is, parts of the pathname not entered by the user are supplied from the default.

Suppression of merging against the default and providing a different default (against which merging may or may not be enabled) are different issues. To deal with the latter, use the **:default** option to **accept**. (See the function **accept**.) An example follows:

```
(accept '((pathname) :default-type nil)
        :default (send (fs:default-pathname)
                       :new-pathname :type nil
                       :version :newest))
```

:direction
Presentation option specifying either **:read** (the default) or **:write**. The value supplied is passed through to **fs:complete-pathname** and affects completion behavior. (See the function **fs:complete-pathname**.)

Use the default (**:read**) if the user is likely to enter the pathname of an already existing file when prompted by **accept**, **:write** otherwise.

:format Presentation option specifying the output format of the pathname. There are four choices:

:normal For example, `S:>mb>dw-pgms>fancy-windows.lisp`. This is the default format.

:directory For example, `>mb>dw-pgms>`. The host, file name, and file type are not displayed.

:dired For example, `fancy-windows.lisp`. Only the file name and type are displayed.

:editor For example, `fancy-windows.lisp >mb>dw-pgms S`. The display format is that used by Zmacs.

For examples illustrating the use of these keywords in pathname presentations, see the presentation type **pathname**.

fs:wildcard-pathname is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

tv:window *Presentation Type*

Type for accepting or presenting old-style (static) window objects.

Examples:

```
  (present (tv:make-window 'tv:window) '((tv:window))) ==>
Window 2
#<DW::DISPLAYED-PRESENTATION #<WINDOW Win... ((TV:WINDOW)) 420205722>

  (accept '((tv:window)))
Enter an old-style window: Window 2
#<WINDOW Window 2 3133400 deactivated>
((TV:WINDOW))
```

The **tv:window** presentation type supports a type history.

tv:window is one of a number of types defined in `sys:dynamic-windows;standard-presentation-types.lisp`. See that file for the source code.

For an overview of presentation types and related facilities, see the section "Using Presentation Types".

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-rectangle *width height x y alu sheet-or-raster* *Function*

This is analogous to the **:draw-rectangle** message to **tv:stream-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**.

graphics:2pi *Variable*

An approximation to 2π in double floating-point format.

For an overview of **graphics:2pi** and related functions: See the section "Other Basic Facilities for Graphic Output".

:alu *Option*

Specifies the drawing mode for a drawing function. Possible values for this option are:

:draw Pixels specified by the drawing function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the drawing function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the drawing function are turned on if they were previously off, and off if they were previously on.

Additionally, numeric and color-extended alu operations are valid values for this option. Whether "on" means white or black depends on the whether or not the display window is in inverse video mode: if inverse video is not in effect, on means white.

graphics:angle-between-angles-p *theta theta-1 theta-2* *Function*

Returns a Boolean value: **t** if *theta* is between *theta-1* and *theta-2* considered clockwise; otherwise **nil**.

Example:

```
(graphics:angle-between-angles-p (/ graphics:2pi 4) (/ graphics:2pi 8) 0)

T
```

graphics:basic-pattern *Flavor*

The basis for the graphics pattern drawing protocol. It has one required method: **graphics:pattern-call-with-drawing-parameters**.

graphics:binary-decode-graphics-from-array-into-function *array* &rest *args* *Function*

array An array with elements of type **(unsigned-byte 8)** produced by the function **graphics:binary-encode-graphics-to-array**, which contains the encoded version of a graphics function.

args The optional keyword **:compile**. When **t**, the function returned is compiled.

Returns a function of one argument, *stream*. When called, this function produces the same graphics output that the originally encoded function produces. See the section "Other Advanced Facilities for Graphic Output".

graphics:binary-encode-graphics-to-array *function* *Function*

function A graphics function of one argument, *stream*.

Returns an array containing an encoded version of *function*. See the section "Other Advanced Facilities for Graphic Output".

graphics:build-graphics-transform &key *(:rotation* **0***) (:scale* **1***) (:scale-x* **1***) (:scale-y* **1***) :translation :transform* *Function*

Creates a coordinate transformation matrix based on the values of the keyword arguments given. The transformation is a result of composing coordinate system transforms for scaling, rotation, and then translation, in that order. The graphics dictionary contains separate entries for this function's keyword arguments:

:rotation
:scale
:scale-x
:scale-y
:translation
:transform

For more information on coordinate transformation matrices: See the section "Advanced Transformation Facilities".

graphics:build-multiple-point-transform *points* *Function*

Takes two, four, or six points (4, 8, or 12 numbers) and returns a transform matrix that converts the figure described by the first half of the points into the figure described by the second half.

Example:

```
(graphics:build-multiple-point-transform '(0 0 0 4 10 10 10 20)) ==>
(5/2 0 0 5/2 10 10)
```

graphics:building-graphics-transform *(*&optional *stream)* &body *body* *Function*

Returns a coordinate transformation matrix to affect the stream *stream* based on the results of the graphics transformation functions — as, for example, **graphics:graphics-translate**, **graphics:graphics-rotate**, or **graphics:graphics-**

scale — included in *body*, in the order in which they are specified. Use this to compose several operations in a specific order, rather than **graphics:build-graphics-transform**, as the latter uses a canonical order, not the order of the arglist.

Example:

```
(graphics:building-graphics-transform ()
(graphics:graphics-translate 0 10)
(graphics:graphics-scale 5 2)
(graphics:graphics-rotate (/ graphics:2pi 4))) ==>
(0 2 -5 0 0 10)
(graphics:build-graphics-transform :rotation (/ graphics:2pi 4)
                                   :scale-x 5 :scale-y 2
                                   :translation (0 10)) ==>
(graphics:build-graphics-transform :rotation (/ graphics:2pi 4)
                                   :scale-x 5 :scale-y 2
                                   :translation (0 10)) ==>
(0 5 -2 0 0 10)
```

graphics:close-path &key (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **#(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a straight-line segment from the current position of the graphics cursor to the beginning of the current path segment and ends that segment. The beginning of the current path segment is specified by the last **graphics:set-current-position** operation.

This function is intended primarily for path building, that is, within path-drawing functions supplied to **graphics:draw-path** or **graphics:with-clipping-path**. For example uses: See the function **graphics:draw-path**.

The listed keyword options are common to all drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:close-path** and related functions: See the section "Path-Drawing Functions".

graphics:compose-transforms *transform additional-transform* *Function*

transform

A list of the essential six elements of a two-dimensional homogeneous graphics transformation matrix describing a combination of scaling, rotation, and translation.

additional-transform A second such list.

Destructively modifies *transform* so that it contains the essential elements of a new graphics transformation matrix that is the result of the matrix dot product of the matrix for *additional-transform* and the matrix for *transform*.

The transformation effected by the result of **graphics:compose-transforms** is equivalent to the result obtained by first effecting the transformation represented by *transform* and then transforming the result by applying *additional-transform*. See the section "Advanced Transformation Facilities".

Compare the following three examples:

```
(defun nested ()
 (graphics:with-room-for-graphics (t 100)
  (graphics:draw-line 0 0 0 100)
  (graphics:draw-line 0 0 100 0)
  (graphics:with-graphics-scale (t 2)
   (graphics:with-graphics-translation (t 10 20)
    (graphics:draw-rectangle 0 10 20 0)))))

(defun composed ()
 (graphics:with-room-for-graphics (t 100)
  (graphics:draw-line 0 0 0 100)
  (graphics:draw-line 0 0 100 0)
  (graphics:with-graphics-transform (t
        (graphics:compose-transforms
           (graphics:build-graphics-transform :scale 2)
           (graphics:build-graphics-transform :translation '(10 20))))
    (graphics:draw-rectangle 0 10 20 0))))

(defun composed-backward ()
 (graphics:with-room-for-graphics (t 100)
  (graphics:draw-line 0 0 0 100)
  (graphics:draw-line 0 0 100 0)
  (graphics:with-graphics-transform (t
        (graphics:compose-transforms
          (graphics:build-graphics-transform :translation '(10 20))
          (graphics:build-graphics-transform :scale 2)))
    (graphics:draw-rectangle 0 10 20 0))))
```

graphics:compute-cubic-spline *px py z* &optional *cx cy (c1* **:relaxed**) *(c2* **graphics::c1**) *p1-prime-x p1-prime-y pn-prime-x pn-prime-y* *Function*

px A list of the *x* coordinates of the points through which the cubic spline is to pass.

py A list of the *y* coordinates of the points through which the cubic spline is to pass.

z The number of intermediate points to be calculated.

cx An array to be filled in with the computed *x* coordinates. Its length should be (+ *N* (* *z* (- *N* 1))), where *N* is the number of points specified in *px* and *py*.

cy An array to be filled in with the computed *y* coordinates. Its length should be (+ *N* (* *z* (- *N* 1))), where *N* is the number of points.

c1 The initial end condition, one of **:relaxed**, **:clamped**, **:cylic**, or **:anticyclic**. The default is **:relaxed**.

c2 The final end condition, one of **:relaxed** or **:clamped**. The default for the final end condition to be the same as the initial end condition, that is, **graphics::c1**.

p1-prime-x The *x* value of the initial slope if the initial end condition is **:clamped**.

p1-prime-y The *y* value of the initial slope if the initial end condition is **:clamped**.

pn-prime-x The *x* value of the final slope if the final end condition is **:clamped**.

pn-prime-y The *y* value of the final slope if the final end condition is **:clamped**.

Where a list of points is specified by the lists, *px* and *py*, **graphics:compute-cubic-spline** computes *z* additional points between each pair of specified points. The *z* points lie on a cubic spline that passes through the specified points and conform to the end conditions specified by *c1*, *c2*, and the remaining optional arguments. The function returns three values, *cx*, *cy*, and *N*. *cx* and *cy* are lists containing the coordinates of the original points together with the computed points inserted, and *N* is the number of points originally specified. The caller can plot lines between successive points of *cx* and *cy* to draw a smooth curve through the given points. For an explanation of the end conditions: See the function **graphics:draw-cubic-spline**.

graphics:compute-cubic-spline-points *(points* &key *(start-relaxation* **:relaxed***) (end-relaxation* **graphics::start-relaxation***) start-slope-dx start-slope-dy end-slope-dx end-slope-dy (number-of-samples* **20***))* *Function*

Like **graphics:compute-cubic-spline**, except that instead of two lists of *x* and *y* coordinates of points through which to draw the spline, **graphics:compute-cubic-spline-points** takes a single list of alternating *x* and *y* values for the points. Instead of returning two arrays, it returns a single list in a format like that of the input. The remaining positional arguments of **graphics:compute-cubic-spline** are provided by keyword arguments for **graphics:compute-cubic-spline-points**.

graphics:current-position &key (*stream* ***standard-output***) *Function*

Returns the current position of the graphics cursor in the user coordinate system. Note that the value of current position may change without the cursor moving if the stream transform changes.

:stream Specifies the output stream; the default is ***standard-output***.

This facility is useful with drawing functions that explictly use the graphics cursor. Such functions include **graphics:draw-bezier-curve-to**, **graphics:draw-circular-arc-to**, **graphics:draw-line-to**, and other facilities commonly used for creating path-drawing functions. For examples of path-drawing functions: See the function **graphics:draw-path**.

For an overview of **graphics:current-position** and related functions: See the section "Drawing Functions".

:dash-pattern *Option*

Specifies a sequence, usually a vector, controlling the dash pattern of a drawing function. If no pattern is specified, the default dashes are 10 pixels long and separated by spaces of 10 pixels. The vector must contain an even number of elements or you will get an error.

The following example draws a line as a series of dashes, alternating in length between 16 and 8 pixels, with intervening spaces of 4 pixels. Note that these lengths result from applying a scaling factor of 4, implemented by the **:scale** and **:scale-dashes** keywords.

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-line 25 25 100 25
                      :dashed t :scale 4
                      :scale-thickness nil
                      :scale-dashes t
                      :dash-pattern #(4 1 2 1)))
```

This option is not operable if the **:dashed** option is **nil**.

:dashed *Option*

Boolean option specifying whether lines are drawn as a series of dashes by a drawing function; the default is **nil**.

This option is not operable if the function draws a filled (`:filled t`) figure.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-rectangle 0 0 100 100 :filled nil :dashed t))
```

graphics:decompose-transform *transform* *Function*

transform
: A list of the essential six elements of a two-dimensional homogeneous graphics transformation matrix describing a combination of scaling, rotation, and translation.

Returns six values that describe the three basic coordinate-system transformations specified by *transform*. *Transform* is regarded as having been composed of a translation, followed by a rotation, followed by a scaling. This composition has the effect on the coordinates of any given point of first scaling those coordinates, then rotating them, and finally translating them. For an explanation: See the section "The Transformation Matrix".

The values returned are:

- The angle of the rotation in radians
- The scaling factor for the x dimension
- The scaling factor for the y dimension
- The x offset for translation
- The y offset for translation
- The x skew

```
(setq translate-trnsfm '(1 0 0 1 50 50))
                                    ;x offset = y offset = 50
(setq rotate-trnsfm '(0 1 -1 0 0 0))
                                    ;rotate pi/4 radians
(setq scale-trnsfm '(4 0 0 4 0 0))
                                    ;scale x = scale y = 4
(setq scale-then-rotate
  (graphics:compose-transforms scale-trnsfm rotate-trnsfm))
(setq composite-trnsfm
  (graphics:compose-transforms scale-then-rotate translate-trnsfm)))
(graphics:decompose-transform composite-trnsfm)
```

tv:*default-arrow-length* *Variable*

Bound to the default value of the ratio of the length of the arrowhead to the thickness of the arrow shaft of arrows drawn by **graphics:draw-arrow**. Initially set to 10.

tv:*default-arrow-width* *Variable*

Bound to the default value of the ratio of the base width of the arrowhead to the thickness of the arrow shaft of arrows drawn by **graphics:draw-arrow**. Initially set to 5.

graphics:defstipple *name (height width) (*&key *:pretty-name :gray-level :tv-gray) patterns* *Function*

Defines a stipple array named *name*. *patterns* is a list of integers, typically using #b, in which case there are *height* integers, each of which is *width* binary digits long. The keywords **:gray-level** and **:tv-gray** specify when **t** that the resulting stipple be added to the lists **graphics::*gray-level-arrays*** or **tv:*gray-arrays***, respectively.

The system defined stipples have a named array leader using the following defstruct:

```
(defstruct (stipple-array :named-array-leader
                          (:print-function print-stipple-array)
                          (:constructor make-stipple-array)
                          (:constructor-make-array-keywords (dimensions '(1 32))
                                                            (type 'sys:art-1b)))
  (name nil)
  (gray-level nil)
  (x-phase nil)
  )
```

From this definition, the following constructor functions are derived: **graphics:make-stipple-array**, **graphics:stipple-array**, **graphics:stipple-array-gray-level**, **graphics:stipple-array-name**, **graphics:stipple-array-repeat-size**, and **graphics:stipple-array-x-phase**.

Example:

```
(graphics:defstipple stipples:33%-gray (3 3) (:tv-gray t) (:gray-level t))
```

graphics:device-pattern *Flavor*

The flavor for device-dependent patterns. It has one method, **graphics:pattern-call-with-drawing-parameters**, which can be called with **:pattern self** to invoke a device-specific drawing protocol.

:draw-1-bit-raster *width height raster from-x from-y to-x to-y &optional (ones-alu :draw) (zeros-alu :erase)* *Generic Function*

Draws a pattern onto the screen. The pattern is replicated as needed, as with **bitblt**. Unlike **bitblt**, which copies bits regardless of any difference in bit depth (element type) between the source array and the screen array, **:draw-1-bit-raster** draws one screen pixel for each source pixel (the source must be a 1-bit array). Bits that are on in the source are drawn using *ones-alu* and bits that are off are drawn using *zeros-alu*. For a 1-bit screen, the result is like **bitblt** would have done with **tv:alu-seta**.

To say it another way, **:draw-1-bit-raster** copies *pixels* from a 1-bit-per-pixel source to the destination, which can be any pixel depth. If the destination is also 1-bit-per-pixel, **:draw-1-bit-raster** is identical to **bitblt**, but if the destination has more bits, **:draw-1-bit-raster** will do the "right" thing where **bitblt** would produce useless results. For detailed information on all the arguments of **:draw-1-bit-raster**:

See the function **bitblt**.

For a color screen, *ones-alu* and *zeros-alu* can be color alus. So, for instance, ones bits might be put out in green and zeros bits in red. Even when drawing in black in white to a color screen, **:draw-1-bit-raster** should be used for drawing stipples, because a whole pixel needs to be drawn black for the on pixels, not just one bit (which is only part of a pixel). Using **:draw-1-bit-raster** rather than **bitblt** is important in programs that run without modification on color screens.

graphics:draw-arrow *from-x from-y to-x to-y* &rest *args* &key (*arrow-head-length* **tv:*default-arrow-length***) (*arrow-base-width* **tv:*default-arrow-width***) (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws an arrow.

from-x The x-coordinate for the base of the arrow.

from-y The y-coordinate for the base of the arrow.

to-x The x-coordinate for the tip of the arrow.

to-y The y-coordinate for the tip of the arrow.

Of the listed keyword options, **:arrow-head-length** and **:arrow-base-width** are unique to **graphics:draw-arrow** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:arrow-head-length Specifies the ratio of the length of the arrowhead to the thickness of the arrow shaft. The default is the value of

tv:*default-arrow-length*, initially set to `10`; thus, if thickness is `1` pixel, arrowhead length is `10` pixels, if thickness is `2`, the length is `20`, and so on.

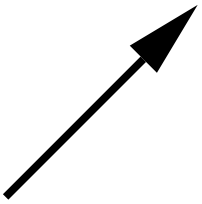
```
(graphics:with-room-for-graphics (t 100)
  (graphics:graphics-translate 20 0)
  (graphics:draw-arrow 0 0 0 50 :arrow-head-length 10)
  (graphics:graphics-translate 20 0)
  (graphics:draw-arrow 0 0 0 50 :arrow-head-length 20)
  (graphics:graphics-translate 20 0)
  (graphics:draw-arrow 0 0 0 50 :thickness 2 :arrow-head-length 10))
```

:arrow-base-width Specifies the ratio of the base width of the arrowhead to the thickness of the arrow shaft. The default is the value of **tv:*default-arrow-width***, initially set to `5`; thus, if thickness is `1` pixel, arrowhead width is `5` pixels, if thickness is `2`, the width is `10`, and so on.

```
(graphics:with-room-for-graphics (t 100)
  (graphics:graphics-translate 20 0)
  (graphics:draw-arrow 0 0 0 50 :arrow-base-width 5)
  (graphics:graphics-translate 20 0)
  (graphics:draw-arrow 0 0 0 50 :arrow-base-width 10)
  (graphics:graphics-translate 20 0)
  (graphics:draw-arrow 0 0 0 50 :thickness 2 :arrow-base-width 5))
```

For an overview of **graphics:draw-arrow** and related functions: See the section "Drawing Functions". Example:

```
(graphics:with-room-for-graphics (t 100)
   (graphics:draw-arrow 0 0 100 100 :thickness 4))
```



graphics:draw-bezier-curve *start-x start-y end-x end-y control-1-x control-1-y control-2-x control-2-y* &key (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-*

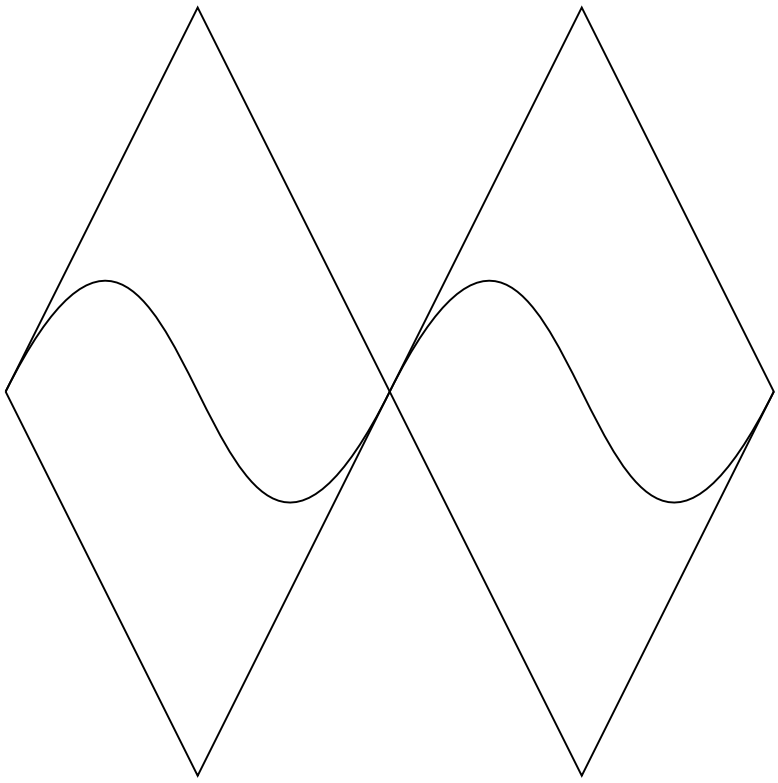
level **1**) *(opaque* **t***)* (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t***)* (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a Bezier parameterization of a cubic curve.

The curve passes through (*start-x*, *start-y*) and (*end-x*, *end-y*). The vector from (*start-x*, *start-y*) to (*control-1-x*, *control-1-y*) determines the starting slope of the curve; its length determines the next derivative. A similar relation exists between (*end-x*, *end-y*) and (*control-2-x*, *control-2-y*). The curve is bounded by the quadrilateral determined by the four points.

The following example approximates a couple of sine wave curves and shows the bounding quadrilaterals for each:

```
(graphics:with-room-for-graphics (t 400)
  (graphics:with-graphics-translation (t 0 200)
    (graphics:draw-bezier-curve 0 0 200 0 100 200 100 -200)
    (graphics:draw-lines '(0 0 100 200 200 0 100 -200 0 0)))
  (graphics:with-graphics-translation (t 200 200)
    (graphics:draw-bezier-curve 0 0 200 0 100 200 100 -200)
    (graphics:draw-lines '(0 0 100 200 200 0 100 -200 0 0))))
```



The listed keyword options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-bezier-curve** and related functions: See the section "Drawing Functions".

graphics:draw-bezier-curve-to *px4 py4 px2 py2 px3 py3* &key (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **#(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a Bezier parameterization of a cubic parabola from the current position of the graphics cursor to a specified end point (*px4*, *py4*). The vector from the current cursor position to (*px2*, *py2*) determines the starting slope of the curve; its length determines the next derivative. A similar relation exists between (*px4*, *py4*) and (*px3*, *py3*). The curve is bounded by the quadrilateral determined by the four points. When done, the graphics cursor is moved to the end point.

This function is intended primarily for path building, that is, within path-drawing functions supplied to **graphics:draw-path** or **graphics:with-clipping-path**. For an example: See the function **graphics:draw-path**. Also: See the function **graphics:draw-bezier-curve**.

The listed keyword options are common to all drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-bezier-curve-to** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
 (graphics:drawing-path ()
  (graphics:set-current-position 0 0)
  (graphics:draw-bezier-curve-to 50 50 25 75 40 0)
  (graphics:draw-line-to 50 0)
  (graphics:close-path)))
```

(flavor:method :draw-char tv:sheet) *char x-bitpos y-bitpos* &optional (*alu* **tv:char-aluf**) *Method*

Displays *char* with its upper left corner at coordinates (*x-bitpos, y-bitpos*).

graphics:draw-circle *center-x center-y radius* &key *(:inner-radius* **0***) (:start-angle* **0***) (:end-angle* **graphics:2pi***) (:alu* **:draw***) (:filled* **t***) :color (:gray-level* **1***) :tile :stipple :return-presentation :pattern (:opaque* **t***) :mask (:mask-x* **0***) (:mask-y* **0***) :thickness (:scale-thickness* **t***) (:line-end-shape* **:butt***) (:line-joint-shape* **:miter***) :dashed :dash-pattern (:initial-dash-phase* **0***) (:draw-partial-dashes* **t***) :scale-dashes (:stream* ***standard-**

output*) *(:rotation* **0***) :clockwise :join-to-path (:scale* **1***) (:scale-x* **1***) (:scale-y* **1***) :translation :transform* *Function*

Draws a circle. The circle may end up looking like an ellipse on the screen if the current transformation matrix does not scale x and y uniformly.

center-x The x-coordinate for the center of the circle.

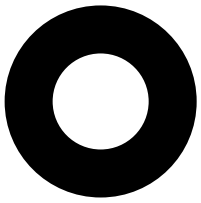
center-y The y-coordinate for the center of the circle.

radius The radius of the circle.

Of the listed keyword options, **:inner-radius**, **:start-angle**, **:end-angle**, and **:clockwise** are unique to **graphics:draw-circle** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:inner-radius Specifies the inner radius of a circular ring figure; the default is `0`.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-circle 50 50 50 :inner-radius 25))
```



:start-angle Specifies the angle in radians at which drawing of the circle begins; the default is `0`. This argument is interpreted the same way that the argument returned by the **atan** function is interpreted. Since in a window increasing *y* is down the screen, this means that angles with positive sines also point down the screen. (This is not the way the **:draw-circular-arc** message interprets angles.) To avoid confusion, use a local coordinate system oriented in the direction you prefer, such as with **graphics:with-room-for-graphics**.

Used in conjunction with the **:end-angle** option, arbitrary circular arcs or wedges can be drawn. The following example draws a filled semicircle starting at 90 degrees and ending at 270 degrees:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-circle 100 100 50
                        :start-angle (* pi 1/2)
                        :end-angle (* pi 3/2)))
```

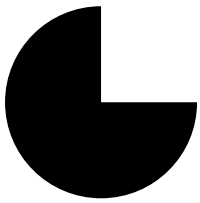
:end-angle Specifies the angle in radians at which drawing of the circle ends; the default is **graphics:2pi**. Refer to the **:start-angle** option for a note about the orientation of angles and for an example.

Used in conjunction with the **:start-angle** option, arbitrary circular arcs or wedges can be drawn. For an example, see the **:start-angle** option.

:clockwise Boolean option specifying whether the circle is drawn in a clockwise or counterclockwise direction. The default is **nil**, counterclockwise.

When **graphics:draw-circle** is being used as a standalone drawing function, this option only has a visible effect when less than a full circle is drawn, that is, when the **:start-angle** or **:end-angle** option is specified to some non-default value. The following example illustrates this:

```
(defun clockwise-circle (t-or-nil)
  (graphics:with-room-for-graphics (t 200)
    (graphics:draw-circle 100 100 50
                          :end-angle (* .5 pi)
                          :clockwise t-or-nil)))
```



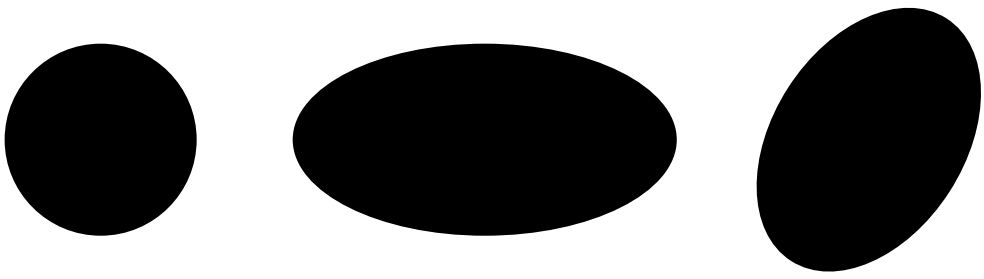
```
(clockwise-circle t)
(clockwise-circle nil)
```

When **graphics:draw-circle** is being used as a path-drawing function inside **graphics:draw-path**, this option affects path winding rules. For more information: See the function **graphics:draw-path**. Note that if you wish to include an arc as part of a drawing-path outline, you should use the **graphics:draw-ellipse** function and, in particular, that function's **:join-to-path** option, instead of **graphics:draw-circle**.

For an overview of **graphics:draw-circle** and related functions: See the section "Drawing Functions". For information on how circles are drawn and how to obtain the appearance you prefer: See the section "Scan Conversion".

See the macro **graphics:with-coordinate-mode**.

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-circle 50 50 50)
  (graphics:draw-circle 0 0 50 :translation '(250 50) :scale-x 2)
  (graphics:draw-circle 0 0 50 :translation '(450 50) :scale-x 1.5 :rotation 1))
```



(flavor:method :draw-circle tv:graphics-mixin) *center-x center-y radius* &optional *alu* *Method*

Draw the outline of a circle specified by its center and radius.

graphics:draw-circle-driver *center-x center-y radius slice-function* *Function*

Scan-converts the given circle, that is, computes the coordinates of the pixels that lie in the circle on a two-dimensional raster grid, and calls *slice-function* to draw these pixels. See the section "Graphics Drivers".

center-x The x-coordinate for the center of the circle.

center-y The y-coordinate for the center of the circle.

radius The radius of the circle.

slice-function A function specifying how a rectangular slice of the circle is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the slice. This function must take four arguments: *width*, the width of the slice; *height*, its height; and *x* and *y*, the coordinates of the slice's location. A typical slice function is

```
#'(lambda (width height x y)
(send *standard-output* :draw-rectangle
        width height x y :draw))
```

(flavor:method :draw-circular-arc tv:graphics-mixin) *center-x center-y radius start-theta end-theta* &optional *(alu* **tv:char-aluf***)* *Method*

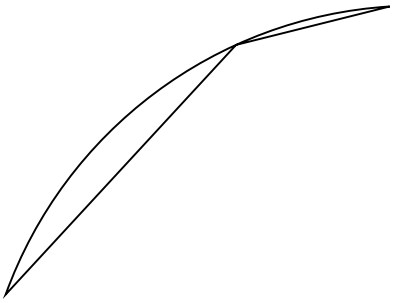
Draws a circular arc for the circle centered at *center-x*, *center-y* with radius *radius*. It draws the part of the circle swept counterclockwise from the starting angle to the finishing angle. The angles are assumed to be in radians and are reduced mod 2pi before drawing. For example, drawing from pi/4 to -pi/4 draws a "C". The same "C" appears when you draw from pi/4 to 7pi/4. Note: the interpretation of *start-theta* and *end-theta* are different for this message than it is for the **graphics:draw-circle** *start-angle* and *end-angle*: the angles are measured as they appear, not as a call to **zl:atan** would return if you gave it the position of the relevant points, since the positive *y*-direction is down.

For **tv:alu-xor**, the behavior with respect to points that would fall on the same pixel is not defined.

graphics:draw-circular-arc-through-point-to *to-x to-y through-x through-y* &key *(alu* **:draw***) (pattern* **nil***) (stipple* **nil***) (tile* **nil***) (color* **nil***) (gray-level* **1***) (opaque* **t***) (mask* **nil***) (mask-x* **0***) (mask-y* **0***) (thickness* **0***) (scale-thickness* **t***) (line-end-shape* **:butt***) (line-joint-shape* **:miter***) (dashed* **nil***) (dash-pattern* **#(10 10)** *) (initial-dash-phase* **0***) (draw-partial-dashes* **t***) (scale-dashes* **nil***) (stream* ***standard-output****) (return-presentation* **nil***) (rotation* **0***) (scale* **1***) (scale-x* **1***) (scale-y* **1***) (translation* **nil***) (transform* **nil***)* *Function*

Draws a circular arc through three points. The first point is the current position. The final position, specified by (*to-x to-y*), is the second point. The third point, specified by *(through-x through-y)*, also lies on the circle.

```
(graphics:with-room-for-graphics (t 200)
  (graphics:drawing-path (t :filled nil)
    (graphics:with-graphics-translation (t 100 50)
        (graphics:set-current-position 0 0)
        (graphics:draw-circular-arc-through-point-to 100 100 20 80)
        (graphics:draw-line-to 20 80)
        (graphics:close-path))))
```



This function is intended primarily for path building, that is, within path-drawing functions supplied to **graphics:draw-path** or **graphics:with-clipping-path**. For examples of path-drawing functions: See the function **graphics:draw-path**.

The listed keyword options are common to all drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-circular-arc-through-point-to** and related functions: See the section "Drawing Functions".

graphics:draw-circular-arc-to *to-x to-y tangent-intersection-x tangent-intersection-y radius* &key (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **#(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a circular arc, with a specified *radius*, tangent to two lines. The first tangent passes through the current position of the graphics cursor and the point specified by (*tangent-intersection-x tangent-intersection-y*). The second passes through the intersection point and (*to-x to-y*). The two tangents, articulating at (*tangent-intersection-x tangent-intersection-y*), form the "sharpened" version of the arc you wish to draw.

The arc is drawn in the direction of (*to-x to-y*), between the two tangent intersection points, that is, the points where the arc intersects the tangent lines. If the starting position of the graphics cursor is different than the first tangent intersection point, then a straight line segment is drawn from the current cursor position

to the starting point of the arc. When done, the graphics cursor is moved to the end of the arc. The function returns four values: *tangent-point-x1, tangent-point-y1, tangent-point-x2,* and *tangent-point-y2*, the coordinates of the points of tangency.

The following example draws arcs providing two rounded corners, one convex and one concave:

```
(graphics:with-room-for-graphics (t 300)
  (graphics:drawing-path ()
    (graphics:with-graphics-translation (t 100 50)
        (graphics:set-current-position 0 0)
        (graphics:draw-circular-arc-to 200 200 0 200 50)
        (graphics:draw-line-to 200 200)
        (graphics:close-path)
        (graphics:set-current-position 0 0)
        (graphics:draw-line-to 150 0)
        (graphics:draw-circular-arc-to 200 50 150 50 50)
        (graphics:draw-line-to 200 200)
        (graphics:close-path))))
```

This function is intended primarily for path building, that is, within path-drawing functions supplied to **graphics:draw-path** or **graphics:with-clipping-path**. For examples of path-drawing functions: See the function **graphics:draw-path**.

The listed keyword options are common to all drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-circular-arc-to** and related functions: See the section "Drawing Functions".

graphics:draw-circular-arc-to-compute-points *from-x from-y to-x to-y tangent-intersection-x tangent-intersection-y radius* *Function*

from-x The *x* coordinate of the first tangent line.

from-y The *y* coordinate of the first tangent line.

to-x The *x* coordinate of the second tangent line.

to-y The *y* coordinate of the second tangent line.

tangent-intersection-x The *x* coordinate of the intersection of the tangent lines.

tangent-intersection-y The *y* coordinate of the intersection of the tangent lines.

radius The radius of the circle to be drawn.

Where a circular arc is to be drawn tangent to two given lines with a specified radius, **graphics:draw-circular-arc-to-compute-points** computes and returns nine values:

center-x The *x* coordinate of center of the circle.

center-y The *y* coordinate of center of the circle.

theta-1 The starting angle of circular arc to be drawn.

theta-2 The ending angle of circular arc to be drawn.

clockwise A Boolean specifying, when **t**, that the arc is to be drawn in a clockwise direction from *theta-1* to *theta-2* or, when **nil**, conterclockwise.

tangent-point-x1 The *x* coordinate of the beginning of the arc.

tangent-point-y1 The *y* coordinate of the beginning of the arc.

tangent-point-x2 The *x* coordinate of the end of the arc.

tangent-point-y2 The *y* coordinate of the end of the arc.

theta-1, theta-2, and *clockwise* are calculated such that the arc drawn is never greater than a semicircle (> (- *theta-2 theta-1*) pi) and the direction is the one specified.

graphics:draw-circular-ring-driver *center-x center-y inner-radius outer-radius slice-function* *Function*

Scan-converts the circular ring, that is, computes the coordinates of the pixels that lie between an outer circle specified by *major-radius* and an inner circle specified by *minor-radius* on a two-dimensional raster grid, and calls *slice-function* to draw these pixels. See the section "Graphics Drivers".

center-x The x-coordinate for the center of the circular ring.

center-y The y-coordinate for the center of the circular ring.

major-radius The outer radius of the circular ring.

minor-radius The inner radius of the circular ring.

slice-function A function specifying how a rectangular slice of the circular ring is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the slice. This function must take four arguments: *width*, the width of the slice; *height*, its height; and *x* and *y*, the coordinates of the slice's location. A typical slice function is

```
#'(lambda (width height x y)
(send *standard-output* :draw-rectangle width height x y :draw))
```

(flavor:method :draw-closed-curve tv:graphics-mixin) *x-array y-array* &optional *end (alu* **tv:char-aluf***)* *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.

graphics:draw-conic-section *from-x from-y to-x to-y tangent-x tangent-y* &key *:shape (:alu* **:draw***) :pattern :stipple :tile :color (:gray-level* **1***) (:opaque* **t***) :mask (:mask-x* **0***) (:mask-y* **0***) :thickness (:scale-thickness* **t***) (:line-end-shape* **:butt***) (:line-joint-shape* **:miter***) :dashed (:dash-pattern* **#<ART-Q-2 550031730>***) (:initial-dash-phase* **0***) (:draw-partial-dashes* **t***) :scale-dashes (:stream* ***standard-output****) :return-presentation (:rotation* **0***) (:scale* **1***) (:scale-x* **1***) (:scale-y* **1***) :translation :transform*
Function

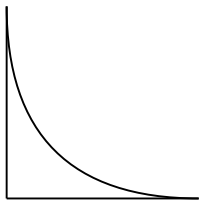
Draws a conic section through two points, <*from-x from-y*> and <*to-x*, *to-y*>. The section is tangent to the two lines whose intersection is <*tangent-x*, *tangent-y*>. The first tangent passes through <*from-x from-y*> and the point <*tangent-x*, *tangent-y*>. The second tangent passes through <*tangent-x*, *tangent-y*> and <*to-x*, *to-y*>. The two tangents, articulating at <*tangent-x*, *tangent-y*>, form the "sharpened" version of the section you wish to draw. Parametrically, the curve drawn is the Bezier quadratic curve *t*^2***P2**+2*s***t*(1-*t*)***Pc**+(1-*t*)^2***P1**, where *t* is the parameter, **P1** and **P2** are the two endpoints, **Pc** is the intersection of the tangents, and *s* is related to the *shape* parameter.

shape The eccentricity of the conic section from a parabola: if *shape*=1, the curve is a parabola; if *shape*<1, the curve is an ellipse; if *shape*>1, the curve is a hyperbola. The default of *shape* specifies the curve of the least eccentricity that will satisfy the constraints. If you take the default, the section will always be an elliptical arc. If

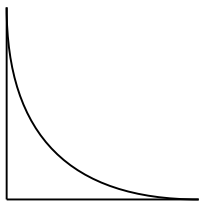
the tangents meet at a right angle, the axes of the ellipse will be parallel to the tangents, and, additionally, if the magnitudes of the tangent lines are equal, the section will be a circular arc. If you specify *shape* to be less than or equal to 1, your bounding triangle can be any shape; however, keep in mind that if you specify *shape*>1, the angle at <*tangent-x*, *tangent-y*> has to be large enough so that its secant is at least equal to *shape*.

The section is drawn in the direction of (*to-x to-y*). When done, the graphics cursor is moved to the end of the section.

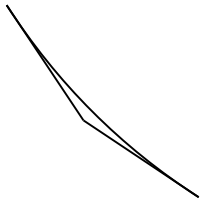
```
(defun conic-example (from-x from-y
                      to-x to-y tan-x tan-y &optional (shape nil))
 (graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-translation (t 50 50)
   (graphics:draw-line tan-x tan-y to-x to-y)
   (graphics:draw-line tan-x tan-y from-x from-y)
   (graphics:draw-conic-section from-x from-y
                                to-x to-y tan-x tan-y :shape shape))))
(conic-example 0 100 100 0 0 0)
```



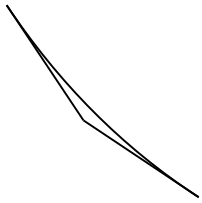
```
(conic-example 0 100 100 0 0 0 1)
```



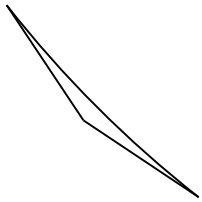
```
(conic-example 0 100 100 0 40 40)
```



```
(conic-example 0 100 100 0 40 40 1)
```



```
(conic-example 0 100 100 0 40 40 1.1)
```



graphics:draw-conic-section-to *to-x to-y tangent-x tangent-y* &key *shape* (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **#(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**)

Function

Draws a conic section through two points, the current graphics cursor position and <*to-x*, *to-y*>. The section is tangent to the two lines whose intersection is <*tangent-x*, *tangent-y*>. The first tangent passes through the current position of the graphics cursor and the point <*tangent-x*, *tangent-y*>. The second tangent passes through <*tangent-x*, *tangent-y*> and <*to-x*, *to-y*>. The two tangents, articulating at

<*tangent-x*, *tangent-y*>, form the "sharpened" version of the section you wish to draw. Parametrically, the curve drawn is the Bezier quadratic curve *t*^2***P2***+2*s***t*(1-*t*)****Pc***+(1-*t*)^2****P1***, where *t* is the parameter, **P1** and **P2** are the two endpoints, **Pc** is the intersection of the tangents, and *s* is related to the *shape* parameter.

shape The eccentricity of the conic section from a parabola: if *shape*=1, the curve is a parabola; if *shape*<1, the curve is an ellipse; if *shape*>1, the curve is a hyperbola. The default of *shape* specifies the curve of the least eccentricity that will satisfy the constraints. If you take the default, the section will always be an elliptical arc. If the tangents meet at a right angle, the axes of the ellipse will be parallel to the tangents, and, additionally, if the magnitudes of the tangent lines are equal, the section will be a circular arc. If you specify *shape* to be less than or equal to 1, your bounding triangle can be any shape; however, keep in mind that if you specify *shape*>1, the angle at <*tangent-x*, *tangent-y*> has to be large enough so that its secant is at least equal to *shape*.

The section is drawn in the direction of (*to-x to-y*). When done, the graphics cursor is moved to the end of the section.

This function is intended primarily for path building, that is, within path-drawing functions supplied to **graphics:draw-path** or **graphics:with-clipping-path**. For examples of path-drawing functions: See the function **graphics:draw-path**.

With the exception of **:shape**, documented above, the listed keyword options are common to all drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-conic-section-to** and related functions: See the section "Path-Drawing Functions".

graphics:draw-cubic-spline *points* &key (*start-relaxation* **:relaxed**) (*end-relaxation* **graphics::start-relaxation**) *start-slope-dx start-slope-dy end-slope-dx end-slope-dy* (*number-of-samples* **20**) &allow-other-keys (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a cubic spline through a series of points.

points A list of points in the form (*x1 y1 x2 y2 ... xn yn*).

Of the listed keyword options, **:start-relaxation**, **:end-relaxation**, **:start-slope-dx**, **:start-slope-dy**, **:end-slope-dx**, **:end-slope-dy**, and **:number-of-samples** are unique to **graphics:draw-cubic-spline** and documented below. The **:number-of-samples** documentation includes an example. The remaining options are common to other

drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:start-relaxation Determines the boundary condition (derivative) of the curve at its starting point (*x1, y1*). Four values are possible:

:relaxed The derivative is set to a value that continues the trend of the curve established by neighboring points. This is the default.

:cyclic If you specify this value, then the value of **:end-relaxation** must also be **:cyclic**. This forces the derivatives at the two end-points of the curve to be equal. When the starting and ending points are equal, this results in a smooth, continuous curve. This — that is, the specification of identical start and end points — is the normal way to make use of the cyclic option. If the starting and ending points specified by the user are *not* equal, the drawing function adds the beginning point on to the end of the list of points it uses, resulting in a closed curve in which a straight line connects the starting and ending point.

:anti-cyclic If you specify this value, then the value of **:end-relaxation** must also be **:anti-cyclic**. This forces the derivatives at the two end-points of the curve to be equal in magnitude but opposite in sign. When the starting and ending points are equal, this causes the curve to come to a point. This — that is, the specification of identical start and end points — is the normal way to make use of the anti-cyclic option. If the starting and ending points specified by the user are *not* equal, the drawing function adds the beginning point on to the end of the list of points it uses, resulting in a closed curve in which a straight line connects the starting and ending point.

:clamped Clamps the derivative to the values specified by the **:start-slope-dx**, **:start-slope-dy**, **:end-slope-dx**, and **:end-slope-dy** options.

```
(defun spline-relaxation (start-relaxation)
  (graphics:with-room-for-graphics (t 50)
    (graphics:draw-cubic-spline '(0 20 20 40 40 20 20 0 0 20)
                                :number-of-samples 5
                                :start-relaxation start-relaxation)))

(spline-relaxation :cyclic)
(spline-relaxation :anti-cyclic)
```

:end-relaxation Determines the boundary condition (derivative) of the curve at its ending point (*xn, yn*). It defaults to the value of **:start-relaxation**. Four values are possible:

:relaxed The derivative is set to a value that continues the trend of the curve established by neighboring points.

:cyclic If you specify this value, then the value of **:start-relaxation** must also be **:cyclic**. This forces the derivatives at the two end-points of the curve to be equal. When the starting and ending points are equal, this results in a smooth, continuous curve. This — that is, the specification of identical start and end points — is the normal way to make use of the cyclic option.

:anti-cyclic If you specify this value, then the value of **:start-relaxation** must also be **:anti-cyclic**. This forces the derivatives at the two end-points of the curve to be equal in magnitude but opposite in sign. When the starting and ending points are equal, this causes the curve to come to a point. This — that is, the specification of identical start and end points — is the normal way to make use of the anti-cyclic option.

:clamped Clamps the derivative to the values specified by **:start-slope-dx**, **:start-slope-dy**, **:end-slope-dx**, and **:end-slope-dy** options.

:start-slope-dx Specifies the *dx* component of the derivative at the starting point (*x1, y1*) of the curve. If you wish to specify this value, then you must supply `:start-relaxation :clamped` and specify the remaining three slope values.

:start-slope-dy Specifies the *dy* component of the derivative at the starting point (*x1, y1*) of the curve. If you wish to specify this value, then you must supply `:start-relaxation :clamped` and specify the remaining three slope values.

:end-slope-dx Specifies the *dx* component of the derivative at the ending point (*xn, yn*) of the curve. If you wish to specify this value, then you must supply `:end-relaxation :clamped` and specify the remaining three slope values.

:end-slope-dy Specifies the *dy* component of the derivative at the ending point (*xn, yn*) of the curve. If you wish to specify this value, then you must supply `:end-relaxation :clamped` and specify the remaining three slope values.

:number-of-samples The number of intermediate points generated between each pair of points specified in the *points* argument. The default is

20.

Because the curve is drawn through each intermediate point, the more such points the smoother the curve, but the longer it takes to get drawn. This is illustrated by the following example:

```
(defun cubic-spline-example (n)
 (graphics:with-room-for-graphics (t 450)
  (graphics:with-graphics-translation (t 200 200)
    (let ((points-1 (list 0 50 25 90 40 70 30 0 50 70
                          70 80 70 50 100 90 100 40 110 0
                          130 30 150 40 130 20 145 0 160 20
                          145 35 170 40 190 40 190 0 200 30
                          220 20 240 40 250 20 260 0 270 20))
          (points-2 (list 160 -200 60 -80 -40 0 -70 100 40 190
                          145 130 144 129 200 200 300 200 330 60
                          230 -80 160 -200)))
      (graphics:draw-cubic-spline points-1 :number-of-samples 3
                                  :thickness 4)
      (graphics:draw-cubic-spline points-2
                                  :start-relaxation :anti-cyclic
                                  :number-of-samples n
                                  :thickness 4)))))
(cubic-spline-example 1)
(cubic-spline-example 5)
```

A reasonable **:number-of-samples** for the inner spline was determined, by trial and error, to be 3. For the outer spline, 4 or 5 seems about right, but try a range of values and note the trade-off between smoothness and speed. (For best results, REFRESH the screen each time you run this example.)

For an overview of **graphics:draw-cubic-spline** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 50)
  (graphics:draw-cubic-spline '(0 0 20 20 30 0) :number-of-samples 5))
```

(flavor:method :draw-cubic-spline tv:graphics-mixin) *px py z* &optional *curve-width alu c1 c2 p1-prime-x p1-prime-y pn-prime-x pn-prime-y* *Method*

Draws a cubic spline curve that passes through a sequence of points. The arrays *px* and *py* hold the *x* and *y* coordinates of the sequence of points; the number of points is determined from the active length of *px*. Through each successive pair of points, a parametric cubic curve is drawn with the **:draw-curve** 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 Makes the derivative zero at this end.

:clamped Allows the caller to specify the derivative. The arguments *p1-prime-x* and *p1-prime-y* specify the derivative at the starting point, and are only used if *c1* is **:clamped**; likewise, *pn-prime-x* and *pn-prime-y* specify the derivative at the ending point, and are only used if *c2* is **:clamped**.

:cyclic Makes the derivative at the starting point and the ending point be equal. If *c1* is **:cyclic** then *c2* is ignored. To draw a closed curve through *n* points, in addition to using **:cyclic**, you must pass in *px* and *py* with one more than *n* entries, since you must pass in the first point twice, once at the beginning and once at the end.

:anti-cyclic Makes the derivative at the starting point be the negative of the derivative at the ending point. If *c1* is **:anticyclic** then *c2* is ignored.

(flavor:method :draw-curve tv:graphics-mixin) *x-array y-array* &optional *end alu*
Method

Draws a sequence of connected line segments. The *x* and *y* coordinates of the points at the ends of the segments are in the arrays *x-array* and *y-array*. The points between line segments are drawn exactly once and the point at the end of the last line is not drawn at all; this is especially useful when *alu* is **tv:alu-xor**. The number of line segments drawn is 1 less than the length of the arrays, unless a **nil** is found in one of the arrays first in which case the lines stop being drawn. If *end* is specified it is used in place of the actual length of the arrays.

(flavor:method :draw-dashed-line tv:graphics-mixin) *from-x from-y to-x to-y* &optional *(alu* **tv:char-aluf***) (dash-spacing* **20***) space-literally-p (offset* **0***) dash-length*
Method

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
 (dw:with-own-coordinates (CL:*standard-output*)
  (send CL:*standard-output*
        ':draw-dashed-line 0 0 200. 200. tv:alu-ior 25. t 0 10.)
  (send CL:*standard-output*
        ':draw-dashed-line 0 0 200. 200. tv:alu-ior 25. t 15. 5.)))
```

This gives you alternating long and short dashes. Because the default value, **nil**, for *space-literally-p* changes the spacing, this technique does not work well when *space-literally-p* is **nil**.

graphics:draw-ellipse *center-x center-y x-radius y-radius* &key (*inner-x-radius* **0**) (*inner-y-radius* **(/ (* graphics::inner-x-radius graphics::y-radius) graphics::x-radius)**) (*start-angle* **0**) (*end-angle* **graphics:2pi**) (*clockwise* **nil**) (*join-to-path* **nil**) (*alu* **:draw**) (*pattern* **nil**) (*filled* **t**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws an ellipse, with

center-x The horizontal center of the ellipse.

center-y The vertical center of the ellipse.

x-radius The length of one of the ellipse's semi-axes; in the unrotated figure this axis is oriented horizontally.

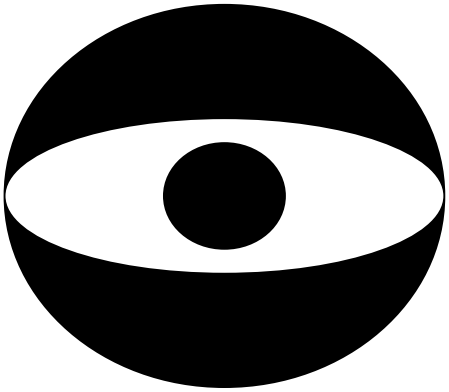
y-radius The length of the other semi-axis of the ellipse; in the unrotated figure this axis is oriented vertically.

Of the listed keyword options, **:inner-x-radius**, **:inner-y-radius**, **:start-angle**, **:end-angle**, **:clockwise**, and **:join-to-path** are unique to **graphics:draw-ellipse** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:inner-x-radius Specifies the inner horizontal (unrotated) semi-axis of an elliptical ring figure; the default is `0`.

Example:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-translation (t 200 100)
    (graphics:draw-ellipse 0 0 115 100
                           :inner-x-radius 114
                           :inner-y-radius 40)
    (graphics:draw-ellipse 0 0 32 28)))
```



:inner-y-radius Specifies the inner vertical (unrotated) semi-axis of an elliptical ring figure. The default value is calculated so that the ratio of the two inner semi-axes equals the ratio of the two outer semi-axes.

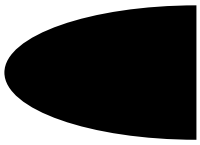
For an example, see the **:inner-x-radius** option.

:start-angle Specifies the angle in radians at which drawing of the ellipse begins; the default is `0`. This argument is interpreted the same way that the argument returned by the **atan** function is interpreted. Since in a window increasing *y* is down the screen, this means that angles with positive sines also point down the screen. (This is not the way the **:draw-circular-arc** message interprets an-

gles.) To avoid confusion, use a local coordinate system oriented in the direction you prefer, such as with **graphics:with-room-for-graphics**.

Used in conjunction with the **:end-angle** option, arbitrary elliptical wedges can be drawn. The following example draws a semi-ellipse starting at 90 degrees and ending at 370 degrees:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-ellipse 200 100 100 35
                         :start-angle (* .5 pi)
                         :end-angle (* 1.5 pi)))
```



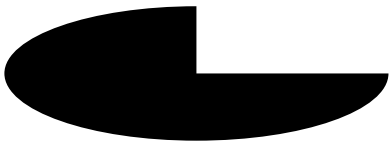
:end-angle Specifies the angle in radians at which drawing of the circle ends; the default is **graphics:2pi**. Refer to the **:start-angle** option for a note about angle orientation.

Used in conjunction with the **:start-angle** option, arbitrary elliptical wedges can be drawn. For an example, see the **:start-angle** option.

:clockwise Boolean option specifying whether the ellipse is drawn in a clockwise or counterclockwise direction. The default is **nil**, counterclockwise.

When **graphics:draw-ellipse** is being used as a standalone drawing function, this option only has a visible effect when less than a full ellipse is drawn, that is, when the **:start-angle** or **:end-angle** option is specified to some non-default value. The following example illustrates this:

```
(defun clockwise-ellipse (t-or-nil)
 (graphics:with-room-for-graphics (t 200)
  (graphics:draw-ellipse 200 100 100 35
                                  :end-angle (* .5 pi)
                                  :clockwise t-or-nil)))


(clockwise-ellipse t)
```



```
(clockwise-ellipse nil)
```

When **graphics:draw-ellipse** is being used as a path-drawing function inside **graphics:draw-path**, this option affects path-winding rules. For more information: See the function **graphics:draw-path**. See also the **:join-to-path** option below.

:join-to-path Specify this option t when you are making an elliptical arc part of a path outline, that is, using it in a **graphics:draw-path** function. This is to ensure that the arc joins the path properly to allow for filling without gaps.

For an overview of **graphics:draw-ellipse** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-ellipse 100 50 40 20 :filled nil)
 (graphics:draw-ellipse 0 0 40 20
                        :translation '(300 50) :rotation (* pi 1/4)))
```

graphics:draw-ellipse-driver *center-x center-y x-radius y-radius slice-function*
Function

center-x The *x*-coordinate of the horizontal center of the ellipse.

center-y The *y*-coordinate of the vertical center of the ellipse.

x-radius The length of the ellipse's horizontal semi-axis.

y-radius The length of the vertical semi-axis of the ellipse.

slice-function A function specifying how a rectangular slice of the ellipse is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the slice. This function must take four arguments: *width*, the width of the slice; *height*, its height; and *x* and *y*, the coordinates of the slice's location. A typical slice function is

```
#'(lambda (width height x y)
(send *standard-output* :draw-rectangle width height x y :draw))
```

Scan-converts the given ellipse, that is, computes the coordinates of the pixels that lie in the ellipse on a two-dimensional raster grid, and calls *slice-function* to draw these pixels. See the section "Graphics Drivers".

graphics:draw-elliptical-ring-driver *center-x center-y inner-x-radius inner-y-radius outer-x-radius outer-y-radius slice-function* *Function*

center-x The *x*-coordinate of the horizontal center of the elliptical ring.

center-y The *y*-coordinate of the vertical center of the elliptical ring.

inner-x-radius The length of the elliptical ring's inner horizontal semi-axis.

inner-y-radius The length of the vertical inner semi-axis of the elliptical ring.

outer-x-radius The length of the elliptical ring's outer horizontal semi-axis.

outer-y-radius The length of the vertical outer semi-axis of the elliptical ring.

slice-function A function specifying how a rectangular slice of the elliptical ring is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the slice. This function must take four arguments: *width*, the width of the slice; *height*, its height; and *x* and *y*, the coordinates of the slice's location. A typical slice function is

```
#'(lambda (width height x y)
(send *standard-output* :draw-rectangle width height x y :draw))
```

Scan-converts the given elliptical ring, that is, computes the coordinates of the pixels that lie on the elliptical ring on a two-dimensional raster grid, and calls *slice-function* to draw these pixels. See the section "Graphics Drivers".

(flavor:method :draw-filled-in-circle tv:graphics-mixin) *center-x center-y radius* &optional *alu* *Method*

Draws a filled-in circle specified by its center and radius. The actual figure produced is one pixel wider than *radius*.

(flavor:method :draw-filled-in-sector tv:graphics-mixin) *center-x center-y radius theta-1 theta-2* &optional *alu* *Method*

Draws a "triangular" section of a filled-in circle, bounded by an arc of the circle and the two radii at *theta-1* and *theta-2*. These angles are in radians; an angle of zero is the positive-X direction, and angles increase counter-clockwise. Note: the interpretation of *start-theta* and *end-theta* are different for this message than it is for the **graphics:draw-circle** *start-angle* and *end-angle*. Also, the figure is not quite the same s the relevant portion produced by **:draw-filled-in-circle**.

graphics:draw-glyph *index font x y* &key (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a figure referenced by a font array. The orientation of the glyph is unaffected by transforms.

index	The index into the font array.
font	The font.
x	The x-coordinate where the glyph is drawn (left edge).
y	The y-coordinate where the glyph is drawn (top edge).

Example:

```
(graphics:with-room-for-graphics (t 20)
  (graphics:draw-glyph (sys:char-subindex #\mouse:fat-double-horizontal-arrow)
                       fonts:mouse 200 10))
```

To see the elements of a font, use the Show Font command. To see the list of loaded fonts, press the HELP key to the Show Font command. For more information on fonts, including information on how to create your own: See the section "Font Editor".

Note that **graphics:draw-glyph** accepts relatively few of the keywords permitted by other drawing functions. All are documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-glyph** and related functions: See the section "Drawing Functions".

graphics:draw-image *image left top* &key (*image-left* **0**) (*image-top* **0**) (*image-right* **nil**) (*image-bottom* **nil**) (*copy-image* **nil**) (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a bit array as a graphics image. The orientation of the image is affected by transforms.

image The bit array. This must be of "correct width" — that is, the width in bits must be a multiple of 32.

left The x-coordinate where drawing of the image begins.

top The y-coordinate where drawing of the image begins.

Of the listed keyword options, **:image-left**, **:image-top**, **:image-right**, **:image-bottom**, and **:copy-image** are unique to **graphics:draw-image** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:image-left The first column in the bit array to be included in the output image.

:image-top The first row in the bit array to be included in the output image.

:image-right The last column in the bit array to be included in the output image. It defaults to the width of the image plus left.

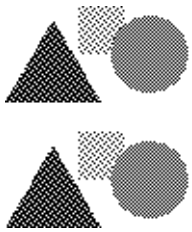
:image-bottom The last row in the bit array to be included in the output image.

:copy-image Boolean option specifying whether to make a copy of the *image* argument (bit array) and use the copy for drawing the image as opposed to using the original. The default is **nil**, meaning that the original is used.

Specify `:copy-image t` if the bit array is re-used in your code but you want the image output by this particular operation to remain constant, that is, to always appear the same when the window is scrolled backwards and the image is redrawn.

Example:

```
(defun draw-logo (&optional (stream *standard-output*)
                  &key (scale 50))
  (graphics:with-graphics-scale (stream scale)
    (graphics:draw-regular-polygon .75 .5 1.25 .5 4
                                   :stream stream :gray-level .25)
    (graphics:draw-regular-polygon 0 0 1 0 3
                                   :stream stream :gray-level .75)
    (graphics:draw-circle 1.5 .5 .4 :stream stream
                                   :gray-level .5 :opaque t)))

(graphics:with-room-for-graphics (t 50)
  (draw-logo))

(setq logo (graphics:with-output-to-bitmap (t) (draw-logo)))

(graphics:with-room-for-graphics (t 50)
  (graphics:draw-image logo 0 0))
```



For an overview of **graphics:draw-image** and related functions: See the section "Drawing Functions".

graphics:draw-line *start-x start-y end-x end-y* &key (*draw-end-point* **t**) (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**)

Function

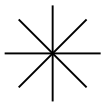
Draws a line.

start-x	The x-coordinate of the starting point.
start-y	The y-coordinate of the starting point.
end-x	The x-coordinate of the ending point.
end-y	The y-coordinate of the ending point.

All of the options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-line** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
  (graphics:with-graphics-translation (t 50 50)
    (dotimes (r 8)
      (graphics:draw-line 0 0 25 0 :rotation (* graphics:2pi (/ r 8))))))
```



(flavor:method :draw-line tv:graphics-mixin) *x1 y1 x2 y2* &optional *alu (draw-end-point* **t***)* *Method*

Draws 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. Note: **:draw-line** does not always clip properly. If correct clipping is important, use the function **graphics:draw-line**. See the section "Thin-Line Clipping".

graphics:draw-line-driver *x1 y1 x2 y2 draw-end-point slice-function* *Function*

Scan-converts the given line, that is, computes the coordinates of the pixels that lie near the line on a two-dimensional raster grid, and calls *slice-function* to draw these pixels. See the section "Graphics Drivers".

x1 The *x*-coordinate of the starting point of the line. This must be an integer.

y1 The *y*-coordinate of the starting point of the line. This must be an integer.

x2 The *x*-coordinate of the ending point of the line. This must be an integer.

y2 The *y*-coordinate of the ending point of the line. This must be an integer.

draw-end-point A Boolean option specifying whether the end point of the line should be drawn. If *draw-end-point* is **t**, the pixel at <*x2,y2*> will be drawn; otherwise, it will not.

slice-function A function specifying how a rectangular slice of the line is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the slice. This function must take four arguments: *width*, the width of the slice; *height*, its height; and *x* and *y*, the coordinates of the slice's location. A typical slice function is

```
#'(lambda
   (width height x y)
   (send *standard-output* :draw-rectangle width
          height
          x y :draw))
```

graphics:draw-line-to *end-x end-y* &key (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **#(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a line from the current position of the graphics cursor to a specified point. When done, the graphics cursor is moved to the point.

end-x The x-coordinate of the end-point.

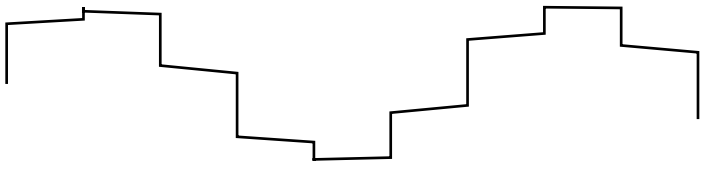
end-y The y-coordinate of the end-point.

This function is intended primarily for path building, that is, within path-drawing functions supplied to **graphics:draw-path** or **graphics:with-clipping-path**. For example uses: See the function **graphics:draw-path**.

The listed keyword options are common to all drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-line-to** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
 (graphics:with-graphics-translation (t 50 50)
  (graphics:with-graphics-scale (t 40)
   (graphics:drawing-path (t :filled nil :scale-thickness nil)
    (graphics:set-current-position 0 0)
    (loop for r below 10 by 1/10 do
      (graphics:draw-line-to r (sin r)))))))
```



graphics:draw-lines *points* &key (*closed* **nil**) (*join-to-path* **nil**) (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

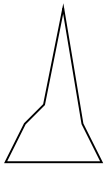
Draws a connected series of line segments.

points A sequence of points in the form (*x1 y1 x2 y2 ... xn yn*).

Of the listed keyword options, **:closed** and **:join-path-to** are documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:closed Boolean option specifying whether the points are to form a closed figure, that is, whether to draw a line connecting the last point specified with the first; the default is **nil**.

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-lines '(0 0 10 20 20 30 30 80 40 20 50 0) :closed t))
```



:join-to-path Specify this option **t** when you are making the line series part of a path outline, that is, using it in a **graphics:draw-path** function. This is to ensure that the multiple-line segment joins the path properly to allow for filling without gaps.

For an overview of **graphics:draw-lines** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-lines '(0 0 10 20 20 30 30 80 40 20 50 0) :thickness 2))
```

(flavor:method :draw-lines tv:graphics-mixin) *alu x0 y0 x1 y1 ... xn yn* *Method*

Draws *n* lines on the screen, the first with endpoints (*x0, y0*) and (*x1, y1*), the second with endpoints (*x1, y1*) and (*x2, y2*), and so on. The points between lines are drawn exactly once and the last endpoint, at (*xn*, *yn*), is not drawn.

graphics:draw-oval *center-x center-y x-radius y-radius* &rest *args* &key *(:filled* **t***)* &allow-other-keys *Function*

Draws an oval, that is, a "race-track" shape, centered on (*center-x center-y*): if *x-radius* or *y-radius* is 0, draws a circle with the specified non-zero radius; otherwise, draws the figure that results from drawing a rectangle with dimensions *x-radius* and *y-radius* and then replacing the two short sides with semicircular arc of appro-

priate size. Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-oval 100 50 40 20 :filled nil))
```

:draw-partial-dashes *Option*

Boolean option specifying whether a partial dash is drawn at the end of a dashed line so that it reaches its specified end-point. The default is **t**: dashes are drawn with the specified numbers of pixels on and off until the endpoint is reached, at which point drawing stops wherever in the pattern you happen to be.

If you specify **nil** for this option, the drawing routine will adjust the spacing of the dashes so that the lines ends on a "dash." In the simple case — that is, with only a single pair of numbers in the dash pattern — a dash is a solid line of (on) pixels, so both ends of such a line are drawn. For example, try these:

```
(graphics:with-room-for-graphics (t 10)
  (graphics:draw-line 0 3 200 3 :dashed t :dash-pattern #(20 15)
                              :draw-partial-dashes t)
  (graphics:draw-line 0 -3 200 -3 :dashed t
                              :dash-pattern #(20 15) :draw-partial-dashes nil)
  (graphics:draw-line 200 -3 200 3))
```

```
(graphics:with-room-for-graphics (t 250)
  (let ((zoom 5))
    (dolist (partial '(t nil))
      (graphics:with-graphics-translation (t 0 (if partial (* 25 zoom) 0))
        (dotimes (i 20)
          (let ((y (* (- 19 i) zoom)))
            (graphics:draw-line 0 y (* i 4 zoom) y
                                :dashed T
                                :dash-pattern #(20 15)
                                :draw-partial-dashes partial)
            (graphics:draw-line 0 (- y 1) (* i 4 zoom) (- y 1)))))))))
```

For more complicated dash patterns, a dash is considered to be a solid line somewhere in the pattern: you will have to experiment to determine the exact result of using the option.

This option is not operable if the **:dashed** option to the drawing function is **nil**.

Some hardcopy devices, most notably PostScript printers, cannot adjust the spacing of the dashes; that is, they will draw partial dashes even if you specify **:draw-partial-dashes nil**.

graphics:draw-path *path-function* &key (*winding-rule* **:non-zero**) (*alu* **:draw**) (*pattern* **nil**) (*filled* **t**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **#(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a fillable figure whose outline is specified by a user-supplied path function.

path-function A drawing function creating the outline of a figure, that is, a path. This path can be arbitrarily complex, contain straight- and curved-line segments, and include more than one closed subpath.

Of the listed keyword options, **:winding-rule** is unique to **graphics:draw-path** and documented below. The remaining options are common to all drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:winding-rule Controls filling of the region outlined by the path-drawing function. Two values are possible:

:non-zero A point is within the area to be filled if a ray from the point to infinity crosses an unequal number of left-to-right and right-to-left path segments.

:odd-even A point is within the area to be filled if a ray from the point to infinity crosses the path an odd number of times.

The **:non-zero** rule is generally the more robust in terms of filling complex shapes completely. The **:odd-even** rule is useful mostly for special effects. See the `path-drawing-examples` function in the "Examples" section below for winding-rule effects on the filling of a star-shaped path.

Note that the direction in which ellipses and circles are drawn, when part of a path-drawing function, affects filling using the **:non-zero** rule. This is demonstrated in the `overlapping-circles` function below.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:drawing-path ()
  (graphics:draw-circle 50 50 40 :filled nil)
  (graphics:draw-circle 50 50 20 :filled nil))
 (graphics:graphics-translate 100 0)
 (graphics:drawing-path (t :winding-rule :odd-even)
  (graphics:draw-circle 50 50 40 :filled nil)
  (graphics:draw-circle 50 50 20 :filled nil))
 (graphics:graphics-translate 100 0)
 (graphics:drawing-path ()
  (graphics:draw-circle 50 50 40 :filled nil)
  (graphics:draw-circle 50 50 20 :filled nil :clockwise t))
 (graphics:graphics-translate 100 0)
 (graphics:drawing-path (t :winding-rule :odd-even)
  (graphics:draw-circle 50 50 40 :filled nil)
  (graphics:draw-circle 50 50 20 :filled nil :clockwise t)))
```

Examples:

```
;;; The following three functions are "drawers" to be
;;; called by the function "path-drawing-examples".

(defun star-drawer (*standard-output*)
  (graphics:set-current-position 0 0)
  (dotimes (i 4)
    (graphics:draw-line-to 1 0)
    (graphics:graphics-origin-to-current-position)
    (graphics:graphics-rotate (float (* -4/5 pi) 0.0)))
  (graphics:close-path))
(defun bz-drawer (*standard-output*)
  (graphics:set-current-position 0 0)
  (graphics:draw-bezier-curve-to 1 1 1/2 3/2 3/4 -1/2)
  (graphics:draw-line-to 1 -1)
```

```
  (graphics:close-path))
(defun widget-drawer (*standard-output*)
  (graphics:graphics-scale 1/10)
  (graphics:set-current-position 1 0)
  (graphics:draw-line-to 2 0)
  (graphics:draw-line-to 2 1)
  (graphics:draw-line-to 7 1)
  (graphics:draw-line-to 7 0)
  (graphics:draw-line-to 8 0)
  (graphics:draw-line-to 8 3)
  (graphics:draw-line-to 7 3)
  (graphics:draw-line-to 7 2)
  (graphics:draw-line-to 5 2)
  (graphics:draw-line-to 5 8)
  (graphics:draw-line-to 4 8)
  (graphics:draw-line-to 4 2)
  (graphics:draw-line-to 2 2)
  (graphics:draw-line-to 2 3)
  (graphics:draw-line-to 1 3)
  (graphics:close-path))


;;; This function applies graphics:draw-path to one
;;; of the drawers (above drawing functions).  You
;;; specify which drawer with the :drawer keyword,
;;; one of #'star-drawer (the default), #'bz-drawer,
;;; or #'widget-drawer.  You can also provide any
;;; other keywords recognized by graphics:draw-path;
;;; for example, try :filled t and :winding-rule
;;; :odd-even (versus. :non-zero) on the star-drawer.
;;; (For another winding rule example, see the
;;; overlapping-circles function, below.)

(defun path-drawing-examples (&rest args
                              &key (scale 100)
                              (rotation 0)
                              (drawer #'star-drawer)
                              &allow-other-keys)
  (graphics:with-room-for-graphics (t 200)
    (graphics:with-graphics-translation (t 100 100)
      (graphics:with-graphics-scale (t scale)
        (graphics:with-graphics-rotation (t rotation)
          (si:with-rem-keywords
            (some-args args
                       '(:scale :drawer))
            (apply #'graphics:draw-path
                   drawer some-args)))))))
```

```
;;; How circles and ellipses are drawn, that is,
;;; whether clockwise or counterclockwise, affects
;;; path filling via the :non-zero winding rule.

(defun overlapping-circles (clockwise-p winding-rule)
  (graphics:with-room-for-graphics (t 200)
    (graphics:draw-path
      (lambda (s)
        (graphics:draw-circle 100 100 75
                              :stream s
                              :filled nil
                              :clockwise
                              clockwise-p)
        (graphics:draw-circle 200 100 75
                              :stream s
                              :filled nil))
      :translation '(100 0)
      :filled t
      :gray-level 4/5
      :winding-rule winding-rule)))
```

For an overview of **graphics:draw-path** and related functions: See the section "Drawing Functions".

graphics:draw-pattern *left top pattern* &key *(:stream* ***standard-output****) (:alu* **:draw***) :right :bottom (:pattern-left* **0***) (:pattern-top* **0***) :copy-pattern* *Function*

Included only for compatibility with Genera 7.1. If you are writing new code, use the function **graphics:draw-image** for drawing images and the **:stipple** option to **graphics:draw-rectangle** for repeating patterns.

graphics:draw-point *x y* &key (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*stream* ***standard-output****) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a point. The single-pixel size of the point is unaffected by transforms. Note that if you have a lot of points to draw to a dynamic window, it is better to draw them to a bitmap first and then to the screen, since this will help reduce the size of the output history.

```
(multiple-value-bind (bitmap x y)
   (graphics:with-output-to-bitmap ()
     (dotimes (i 100)
      (graphics:draw-point (random 50) (random 50) :alu :flip)))
  (graphics:with-room-for-graphics (t 60)
    (graphics:draw-image bitmap (+ x 50) (- y 50) :scale-y -1)))
```

x The point's x-coordinate.

y The point's y-coordinate.

All of the options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-point** and related functions: See the section "Drawing Functions".

(flavor:method :draw-point tv:graphics-mixin) *x y* &optional *alu value* *Method*

Draws *value* into the picture element at the specified coordinates, combining it with the previous contents according to the specified *alu* function (*value* is the first argument to the operation, and the previous contents is the second argument.) *value* should be **0** or **1** on a black-and-white TV. Clipping is performed; that is, this message will have no effect if the coordinates are outside the window. *value* defaults to -1, that is, a number with all ones.

graphics:draw-polygon *points* &key (*points-are-convex-p* **nil**) &allow-other-keys (*alu* **:draw**) (*pattern* **nil**) (*filled* **t**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a polygon connecting a set of points.

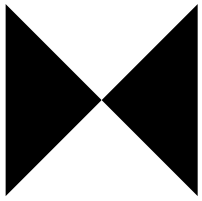
points A sequence of points in the form (*x1 y1 x2 y2 ... xn yn*); these form the points of the polygon.

Of the listed keyword options, **:points-are-convex-p** is unique to **graphics:draw-polygon** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:points-are-convex-p Boolean option specifying whether the *points* describe a convex polygon; the default is **nil**. If **t**, an algorithm more efficient for drawing convex polygons, as opposed to any polygon, is used.

For an overview of **graphics:draw-polygon** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-polygon '(0 100 100 0 100 100 0 0)))
```



graphics:draw-rectangle *left top right bottom* &key (*alu* **:draw**) (*pattern* **nil**) (*filled* **t**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a rectangle.

left	The x-coordinate of the left side of the rectangle.
top	The y-coordinate of the top side of the rectangle.
right	The x-coordinate of the right side of the rectangle.
bottom	The y-coordinate of the bottom side of the rectangle.

The listed keyword options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-rectangle** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-rectangle 60 30 140 70 :filled nil)
  (graphics:draw-rectangle -40 -20 40 20 :translation '(300 50) :rotation (* pi 1/4)))
```

(flavor:method :draw-rectangle tv:sheet) *width height x y* &optional *alu* *Method*

Draws a filled-in rectangle with dimensions *width* by *height* on the window with its upper left corner at coordinates (*x, y*).

graphics:draw-regular-polygon *start-x start-y end-x end-y number-of-sides* &key (*handedness* **:left**) &allow-other-keys (*alu* **:draw**) (*pattern* **nil**) (*filled* **t**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Given the starting and ending coordinates for a single side and the number of sides, draws a regular polygon.

start-x The x-coordinate of the starting point for side 1.

start-y The y-coordinate of the starting point for side 1.

end-x The x-coordinate of the ending point for side 1.

end-y The y-coordinate of the ending point for side 1.

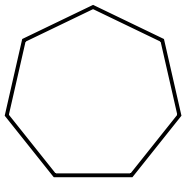
number-of-sides The total number of sides.

Of the listed keyword options, **:handedness** is unique to **graphics:draw-regular-polygon** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

:handedness Specifieswhetherthepolygonisdrawntothe**:left**or **:right** of side 1. The default is **:left**, meaning that, if you were located at (*start-x start-y*) and facing (*end-x end-y*), the polygon would be drawn to your left.

For an overview of **graphics:draw-regular-polygon** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-regular-polygon 100 0 140 0 7 :filled nil :thickness 2))
```



(flavor:method :draw-regular-polygon tv:graphics-mixin) *x1 y1 x2 y2 n* &optional *alu* *Method*

Draws a filled-in, closed, convex, regular polygon of **(abs** *n***)** sides, where the line from *(x1, y1)* to *(x2, y2)* is one of the sides. If *n* is positive, 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").

graphics:draw-string *string x y* &key *(:attachment-y* **:baseline***) (:attachment-x* **:left***) (:toward-x* **(1+ graphics::start-x)***) (:toward-y* **graphics::start-y***) :stretch-p :character-style :record-as-text (:alu* **:draw***) :pattern :stipple :tile :color (:gray-level* **1***) (:opaque* **t***) :mask (:mask-x* **0***) (:mask-y* **0***) (:stream* ***standard-output****) :return-presentation (:rotation* **0***) (:scale* **1***) (:scale-x* **1***) (:scale-y* **1***) :translation :transform* *Function*

Draws a character string.

string The string.

x The x-coordinate where drawing of the string begins (see the **:attachment-x** option below).

y The y-coordinate where drawing of the string begins (see the **:attachment-y** option below).

Of the listed keyword options, **:attachment-x**, **:attachment-y**, **:toward-x**, **:toward-y**, **:stretch-p**, and **:character-style** are unique to **graphics:draw-string** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

Note that coordinate system options (**:rotation**, **:scale**, and so on) affect the position of the character string (baseline) but not the size or orientation of the individual characters. Character size is controlled by the **:character-style** option and the orientation of the individual glyphs is always upright. If you want the character string, including glyphs, to respond similarly to other graphic images, use the **graphics:draw-string-image** function.

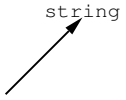
:attachment-x Specifiesthestringattachmentpointtothexcoordinate:

:left The left edge of the first character is positioned at *x*. This is the default.

:right The right edge of the last character is positioned at *x*.

:center The horizontal center of the string is positioned at *x*.

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-arrow 0 0 40 40)
  (graphics:draw-string "string" 40 40 :attachment-x :center))
```



:attachment-y Specifiesthestringattachmentpointtotheycoordinate:

:baseline The baseline of the string is positioned at *y*. This is the default.

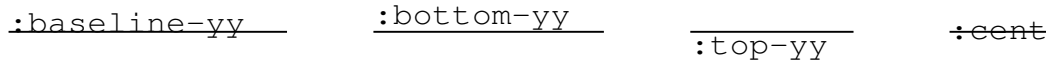
:bottom The bottom of the string is positioned at *y*.

:top The top of the string is positioned at *y*.

:center The vertical center of the string is positioned at *y*.

The following example illustrates the differences among these possibilities:

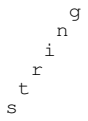
```
(graphics:with-room-for-graphics (t 150)
  (graphics:draw-string ":baseline-yy" 10 50
                        :attachment-y :baseline
                        :character-style
                        '(nil nil :very-large))
  (graphics:draw-line 10 50 155 50)
  (graphics:draw-string ":bottom-yy" 200 50
                        :attachment-y :bottom
                        :character-style
                        '(nil :roman :very-large))
  (graphics:draw-line 200 50 320 50)
  (graphics:draw-string ":top-yy" 365 50
                        :attachment-y :top
                        :character-style
                        '(nil nil :very-large))
  (graphics:draw-line 365 50 450 50)
  (graphics:draw-string ":center-yy" 500 50
                        :attachment-y :center
                        :character-style
                        '(nil nil :very-large))
  (graphics:draw-line 500 50 620 50))
```



:toward-x The x-coordinate toward which the string is drawn. The default value is one greater than the starting x-coordinate, meaning that the string is drawn to the right; its deviation from the horizontal is determined by the **:toward-y** option.

:toward-y The y-coordinate toward which the string is drawn. The default value is equal to the starting y-coordinate, meaning that the string is drawn horizontally.

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-string "string" 0 0 :toward-x 100 :toward-y 50))
```



:stretch-p Boolean option specifying whether the characters are spaced evenly between the starting (*start-x start-y*) and ending (**:toward-x** <*end-x*> **:toward-y** <*end-y*>) coordinates.

If the space provided is greater than that required by the default spacing between characters of the given style, then additional spacing is inserted; the string is stretched. If the space is less than that required by the default spacing, space is eliminated; the string is compressed.

The default is **nil**, meaning that the default spacing for the character style in question is used, regardless of the distance between the starting and ending coordinates.

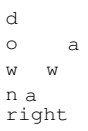
```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-string "normal" 0 40 :toward-x 200)
 (graphics:draw-string "stretched" 0 20 :toward-x 200 :stretch-p t))
```

:character-style Specifies a character style for the string. Merging against the default character style for the output stream is supported.

```
(graphics:with-room-for-graphics (t 30)
  (graphics:draw-string "string" 10 10 :character-style '(:dutch nil nil)))
```

For an overview of **graphics:draw-string** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-string "right" 0 0)
 (graphics:draw-string "down" 0 50 :toward-x 0 :toward-y 0)
 (graphics:draw-string "away" 10 10 :toward-x 50 :toward-y 50 :stretch-p t))
```



(flavor:method :draw-string tv:graphics-mixin) *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**) *Method*

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 "Character Environment Facilities".

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.

graphics:draw-string-image *string x y* &key (*attachment-y* **:baseline**) (*attachment-x* **:left**) (*character-style* **nil**) (*character-size* **nil**) *string-width* (*scale-down-allowed* **t**) (*alu* **:draw**) (*pattern* **nil**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

Draws a character string as a graphics image. This enables the string to be manipulated in the same manner that other images can be manipulated. For example, the string can be scaled or rotated as a unit. This allows you to draw slanted or tilted character strings. This is unlike strings generated by **graphics:draw-string**, in which only the position of the baseline is affected by coordinate system changes; the character glyphs remain the same. Note that **graphics:draw-string-image** draws the image upside down relative to the character glyphs. You can use **graphics:with-room-for-graphics** around **graphics:draw-string-image** or transform coordinates to get the character glyphs oriented the way you want it.

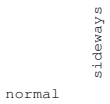
string The string.

x The x-coordinate where drawing of the string begins (see the **:attachment-x** option below).

y The y-coordinate where drawing of the string begins (see the **:attachment-y** option below).

Of the listed keyword options, **:attachment-x**, **:attachment-y**, **:character-style**, **:character-size**, **:string-width**, and **:scale-down-allowed** are unique to **graphics:draw-string-image** and documented below. The remaining options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions". Example:

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-string-image "normal" 0 0)
 (graphics:draw-string-image "sideways" 10 0
                 :translation '(50 0) :rotation (/ pi 2)))
```



:attachment-x Specifies the string attachment point to the *x* coordinate:

:left The left edge of the first character is positioned at *x*. This is the default.

:right The right edge of the last character is positioned at *x*.

:center The horizontal center of the string is positioned at *x*.

The position in the string is as viewed in the user coordinate system, so if the string is being rotated, it may not be that point of the string on the screen.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-arrow 0 0 40 0 :rotation (/ pi 4))
 (graphics:draw-string-image "string" 40 0
               :rotation (/ pi 4) :attachment-x :center))
```

:attachment-y Specifies the string attachment point to the *y* coordinate, one of the following:

:baseline The baseline of the string is positioned at *y*. This is the default.

:bottom The bottom of the string is positioned at *y*.

:top The top of the string is positioned at *y*.

:center The vertical center of the string is positioned at *y*.

For an example showing the differences among the different attachment points: See the function **graphics:draw-string**. In particular, look at the **:attachment-y** option.

:character-style Specifies the character style for the string. Merging against the default character style for the output stream is supported.

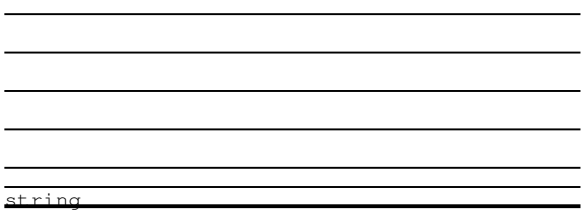
If the **:character-size** option is specified, it overrides the size component of the **:character-style** specification.

:character-size Specifies a number controlling character size. The default is **nil**, meaning that character size is determined by the size component of the output character style.

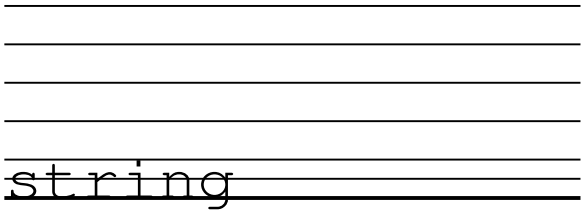
If you use this option, the number specified is scaled and a font chosen with the same family and face as the output character style, but with a size as close as possible to that desired. If you supply this option with a large number and none of the predefined fonts are big enough, then the largest font available is scaled up, pixel by pixel, to achieve the desired size.

To see the effects of this option, try calling the following with the string of your choice and various numbers in the 10 to 100 range (you are not limited to this range, however). The horizontal lines are 20 pixels apart, except for the first three which are 10 pixels apart.

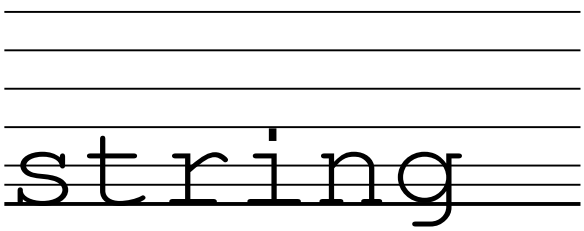
```
(defun string-size (string size)
  (graphics:with-room-for-graphics (t 100)
    (graphics:draw-string-image string 0 0
                                :character-size size)
    (graphics:draw-line 0 0 300 0 :thickness 2)
    (graphics:draw-line 0 10 300 10)
    (graphics:draw-line 0 20 300 20)
    (graphics:draw-line 0 40 300 40)
    (graphics:draw-line 0 60 300 60)
    (graphics:draw-line 0 80 300 80)
    (graphics:draw-line 0 100 300 100)))

(string-size "string" 10)
```




```
(string-size "string" 30)
```

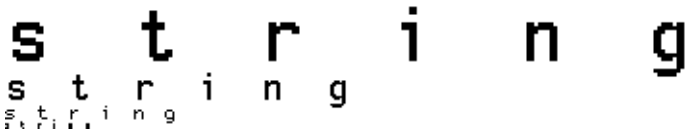



```
(string-size "string" 60)
```



:string-width Specifies the final width of the string drawn. The image is scaled if necessary so that this width is attained. The spacing within characters is adjusted to make up for differences between the font sizes available and the desired size. When used in conjunction with the **:character-size** option, a string can be drawn of exactly the desired size using the closest available font.

```
(graphics:with-room-for-graphics (t 100)
 (loop for scale in '(1/2 1 2 4) do
   (graphics:draw-string-image "string" 0 10
                               :scale scale
                               :string-width 100
                               :character-size 10)))
```



:scale-down-allowed A Boolean option specifying when true, the default, that the string image may be scaled down, that is, drawn with a

scale factor less than one, if necessary to accommodate the requirement of **:string-width**. If **:scale-down-allowed** is **t**, the resulting string will probabaly not be readable is **:string-width** is small; if it is **nil**, the string will exceed the size constraint.

For an overview of **graphics:draw-string-image** and related functions: See the section "Drawing Functions".

graphics:draw-triangle *x1 y1 x2 y2 x3 y3* &key (*alu* **:draw**) (*pattern* **nil**) (*filled* **t**) (*stipple* **nil**) (*tile* **nil**) (*color* **nil**) (*gray-level* **1**) (*opaque* **t**) (*mask* **nil**) (*mask-x* **0**) (*mask-y* **0**) (*thickness* **0**) (*scale-thickness* **t**) (*line-end-shape* **:butt**) (*line-joint-shape* **:miter**) (*dashed* **nil**) (*dash-pattern* **'(10 10)**) (*initial-dash-phase* **0**) (*draw-partial-dashes* **t**) (*scale-dashes* **nil**) (*stream* ***standard-output***) (*return-presentation* **nil**) (*rotation* **0**) (*scale* **1**) (*scale-x* **1**) (*scale-y* **1**) (*translation* **nil**) (*transform* **nil**) *Function*

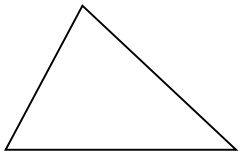
Draws a triangle.

x1	The x-coordinate of the first point of the triangle.
y1	The y-coordinate of the first point of the triangle.
x2	The x-coordinate of the second point of the triangle.
y2	The y-coordinate of the second point of the triangle.
x3	The x-coordinate of the third point of the triangle.
y3	The y-coordinate of the third point of the triangle.

The listed keyword options are common to other drawing functions and documented separately: See the section "Keyword Options to Drawing Functions".

For an overview of **graphics:draw-triangle** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-triangle 0 0 120 0 40 75 :filled nil))
```



(flavor:method :draw-triangle tv:graphics-mixin) *x1 y1 x2 y2 x3 y3* &optional *alu* *Method*

Draws a filled-in triangle with its corners at (*x1, y1*), (*x2, y2*), and (*x3, y3*).

graphics:draw-triangle-driver *x1 y1 x2 y2 x3 y3 slice-function* *Function*

Scan-converts the given triangle, that is, computes the coordinates of the pixels that lie in the triangle on a two-dimensional raster grid, and calls *slice-function* to draw these pixels. See the section "Graphics Drivers".

x1 The *x*-coordinate of the first vertex of the triangle. This must be an integer.

y1 The *y*-coordinate of the first vertex of the triangle. This must be an integer.

x2 The *x*-coordinate of the second vertex of the triangle. This must be an integer.

y2 The *y*-coordinate of the second vertex of the triangle. This must be an integer.

x3 The *x*-coordinate of the third vertex of the triangle. This must be an integer.

y3 The *y*-coordinate of the third vertex of the triangle. This must be an integer.

slice-function A function specifying how a rectangular slice of the triangle is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the slice. This function must take four arguments: *width*, the width of the slice; *height*, its height; and *x* and *y*, the coordinates of the slice's location. A typical slice function is

```
#'(lambda (width height x y)
(send *standard-output* :draw-rectangle width height x y :draw))
```

graphics:draw-unfilled-circle-driver *center-x center-y radius point-function* &optional (*separate-quadrants* **nil**) *Function*

Scan-converts the outline of the circle, that is, computes the coordinates of the pixels that lie near the outline of the circle on a two-dimensional raster grid, and calls *point-function* to draw these pixels. See the section "Graphics Drivers".

center-x The x-coordinate for the center of the circle.

center-y The y-coordinate for the center of the circle.

radius The radius of the circle.

point-function A function specifying how a point on the circle is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the point. This function must take two arguments, *x* and *y*, the point's coordinates. A point function could be, for example,

```
(graphics:draw-point x y)
```

separate-quadrants A Boolean option specifying whether all the points in a quadrant should be drawn before the another quadrant is started. When *separate-quadrants* is **t**, each quadrant's points are drawn in turn, making it possible for the points to be a connected set of line segments. This is necessary if the circle is to be part of a path. When the default, **nil**, is taken, the unfilled circle is drawn with slices, possibly overlapping the axes, which are converted to points, by calling draw-circular-ring-driver with appropriate radii.

graphics:draw-unfilled-ellipse-driver *center-x center-y x-radius y-radius point-function* &optional (*separate-quadrants* **nil**) *Function*

Scan-converts the outline of the ellipse, that is, computes the coordinates of the pixels that lie near the outline of the ellipse on a two-dimensional raster grid, and calls *point-function* to draw these pixels. See the section "Graphics Drivers".

center-x The x-coordinate for the horizontal center of the ellipse.

center-y The y-coordinate for the vertical center of the ellipse.

x-radius The length of one of the ellipse's semi-axes; in the unrotated figure this axis is oriented horizontally.

y-radius The length of the other semi-axis of the ellipse; in the unrotated figure this axis is oriented vertically.

point-function A function specifying how a point on the ellipse is to be drawn on a raster device and possibly specifying any other operations to be performed in conjunction with drawing the point. This function must take two arguments, *x* and *y*, the point's coordinates. A point function could be, for example,

```
(send *standard-output* :draw-line x y (1+ x) (1+ y) :draw nil)
```

separate-quadrants A Boolean option specifying whether all the points in a quadrant should be drawn before the another quadrant is started. When *separate-quadrants* is **t**, each quadrant's points are drawn in turn, making it possible for the points to be a connected set of line segments. This is necessary if the circle is to be part of a path. When the default, **nil**, is taken, the unfilled circle is drawn by calling draw-circular-ring-driver with appropriate radii.

(flavor:method :draw-wide-curve tv:graphics-mixin) *x-array y-array width* &optional *end alu* *Method*

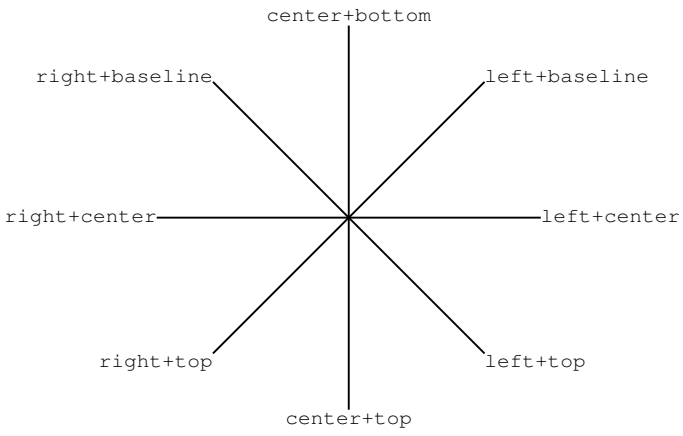
Like **:draw-curve**, but *width* is how wide to make the lines.

graphics:drawing-path (&optional *stream* &rest *draw-path-args*) &body *body* *Function*

Draws a fillable figure whose outline is specified by the functions in *body*, which are executed within an implicit **progn** form. Collects and returns any values produced by these functions. The arguments in *draw-path-args* include any of the keyword arguments accepted by the **graphics:draw-path** function.

Here is an example in which the **progn** and value-return features are used, though not in drawing a fillable figure:

```
(graphics:with-room-for-graphics (t 240)
  (graphics:with-graphics-translation (t 300 100)
    (loop for (angle xa ya) in '((0 :left :center)
                                 (1/4 :left :baseline)
                                 (1/2 :center :bottom)
                                 (3/4 :right :baseline)
                                 (1 :right :center)
                                 (5/4 :right :top)
                                 (3/2 :center :top)
                                 (7/4 :left :top))
          do
      (multiple-value-bind (x y)
          (graphics:drawing-path (t :filled nil)
            (graphics:set-current-position 0 0)
            (graphics:draw-line-to 100 0 :rotation (* pi angle))
            (graphics:current-position))
        (let ((string (format nil "~(~A+~A~)" xa ya)))
          (graphics:draw-string string x y
                                :attachment-x xa :attachment-y ya))))))
```



graphics:erase-graphics-presentation *presentation* &key (*stream* ***standard-output***) (*redisplay-overlapping-presentations* **t**) *Function*

Erases a graphics presentation. Graphic presentations are created with **graphics::with-output-as-graphic-presentation**.

presentation The presentation to erase.

:stream Specifies the output stream; the default is ***standard-output***.

:redisplay-overlapping-presentations Boolean option specifying whether presentations overlapping the erased presentation are redisplayed; the default is **t**.

For an example: See the function **graphics:with-output-as-graphics-presentation**.

For an overview of **graphics:erase-graphics-presentation** and related functions: See the section "Other Basic Facilities for Graphic Output".

graphics:erase-rectangle *left top right bottom* &key (*stream* ***standard-output***) *Function*

Clears a rectangular area of a graphics display.

left The x-coordinate of the left side of the rectangular area.

top The y-coordinate of the top side of the rectangular area.

right The x-coordinate of the bottom side of the rectangular area.

bottom The y-coordinate of the bottom side of the rectangular area.

:stream Specifies the output stream; the default is ***standard-output***.

For an overview of **graphics:with-room-for-graphics** and related functions: See the section "Other Basic Facilities for Graphic Output".

:filled *Option*

Boolean option specifying whether all pixels within the figure created by a drawing function are turned on, or only the outline pixels; the default is **t** (filled).

tv:graphics-mixin *Flavor*

A flavor mixed into almost all windows. It provides basic graphics capabilities.

graphics:graphics-origin-to-current-position &key (*stream* ***standard-output***) *Function*

Moves the graphics origin (0, 0) to the current position of the graphics cursor.

:stream Specifies the output stream; the default is ***standard-output***.

This facility is useful with drawing functions that explictly use the graphics cursor. Such functions include **graphics:draw-bezier-curve-to**, **graphics:draw-**

circular-arc-to, **graphics:draw-line-to**, and other facilities commonly used for creating path-drawing functions. For examples of path-drawing functions: See the function **graphics:draw-path**.

For an overview of **graphics:graphics-origin-to-current-position** and related functions: See the section "Drawing Functions".

```
(graphics:with-room-for-graphics (t 100)
 (graphics:with-graphics-translation (t 50 50)
  (graphics:with-graphics-scale (t 50)
   (graphics:drawing-path ()
    (graphics:set-current-position 0 0)
    (dotimes (i 4)
     (graphics:draw-line-to 1 0)
     (graphics:graphics-origin-to-current-position)
     (graphics:graphics-rotate (* -4/5 pi)))
    (graphics:close-path)))))
```

graphics:graphics-rotate *theta* &key (*stream* ***standard-output***) *Function*

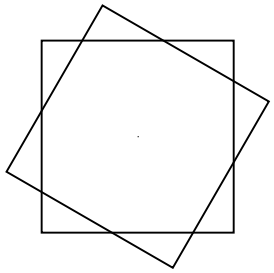
Modifies the graphics transformation matrix to rotate graphics output; rotation is about the local origin. (For an explanation of the graphics tranformation matrix: See the function **graphics:with-graphics-transform**.)

theta A number specifying the rotation in radians.

:stream Specifies the output stream; the default is ***standard-output***.

Example:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-translation (t 100 100)
    (graphics:draw-point 0 0)
    (graphics:graphics-rotate (* 1/3 pi))
    (graphics:draw-rectangle -50 50 50 -50 :filled nil)
    (graphics:graphics-rotate (* -1/3 pi))
    (graphics:draw-rectangle -50 50 50 -50 :filled nil)))
```



This and two related functions (**graphics:graphics-translate** and **graphics:graphics-scale**) are intended primarily for use within path-drawing functions supplied to **graphics:draw-path** and **graphics:with-clipping-path** or within graphics encapsulating macros such as **graphics:with-room-for-graphics** and **graphics:with-graphics-transform**. For examples: See the function **graphics:draw-path**. In other contexts, **graphics:with-graphics-rotation** is generally more appropriate.

For an overview of **graphics:graphics-rotate** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

graphics:graphics-scale *x-scale* &optional (*y-scale* **graphics::x-scale**) &key (*stream* ***standard-output***) *Function*

Modifies the graphics transformation matrix to apply a scaling factor to graphics output. (For an explanation of the graphics tranformation matrix: See the function **graphics:with-graphics-transform**.)

x-scale A number specifying the x scaling factor.

y-scale A number specifying the y scaling factor (if different than *x-scale*).

:stream Specifies the output stream; the default is ***standard-output***.

Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:graphics-scale 5 5)
  (graphics:draw-rectangle 10 20 20 10)
  (graphics:graphics-scale 1/2 1/2)
  (graphics:draw-rectangle 10 20 20 10 :gray-level .5))
```

This and two related functions (**graphics:graphics-translate** and **graphics:graphics-rotate**) are intended primarily for use within path-drawing functions supplied to **graphics:draw-path** and **graphics:with-clipping-path** or within graphics encapsulating macros such as **graphics:with-room-for-graphics** and **graphics:with-graphics-transform**. For examples: See the function **graphics:draw-path**. In other contexts, **graphics:with-graphics-scale** is generally more appropriate.

For an overview of **graphics:graphics-scale** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

graphics:graphics-stream-p *stream* *Function*

Returns a Boolean value: **t** if *stream* supports the generic graphics protocol; otherwise **nil**. Use this predicate to determine whether to use a graphical or a textual representation of an object.

graphics:graphics-transform *transform* &key (*stream* ***standard-output***) *Function*

Modifies the graphics transformation matrix move the origin of the graphics coordinate system in accordance with the translation, rotation, and scaling specified by *transform*. (For an explanation of the graphics tranformation matrix: See the function **graphics:with-graphics-transform**.)

transform A list of the essential six elements of a two-dimensional homogeneous graphics transformation matrix describing a combination of scaling, rotation, and translation.

:stream Specifies the output stream; the default is ***standard-output***.

Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:graphics-translate 40 40)
  (graphics:draw-rectangle 0 40 40 0)
  (graphics:graphics-transform '(2 0 0 1 10 10))
  (graphics:draw-rectangle 0 40 40 0 :gray-level .5))
```

This and related functions such as (**graphics:graphics-scale** and **graphics:graphics-rotate**) are intended primarily for use within path-drawing functions supplied to **graphics:draw-path** and **graphics:with-clipping-path** or within graphics encapsulating macros such as **graphics:with-room-for-graphics** and **graphics:with-graphics-transform**. For examples: See the function **graphics:draw-path**. In other contexts, **graphics:with-graphics-transform** is generally more appropriate.

For an overview of **graphics:graphics-transform** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

graphics:graphics-translate *delta-x delta-y* &key (*stream* ***standard-output***)
Function

Modifies the graphics transformation matrix to offset the origin of the graphics coordinate system. (For an explanation of the graphics tranformation matrix: See the function **graphics:with-graphics-transform**.)

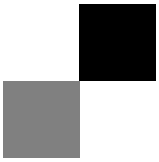
delta-x A number specifying the x offset.

delta-y A number specifying the y offset.

:stream Specifies the output stream; the default is ***standard-output***.

Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:graphics-translate 40 40)
  (graphics:draw-rectangle 0 40 40 0)
  (graphics:graphics-translate -40 -40)
  (graphics:draw-rectangle 0 40 40 0 :gray-level .5))
```



This and two related functions (**graphics:graphics-scale** and **graphics:graphics-rotate**) are intended primarily for use within path-drawing functions supplied to **graphics:draw-path** and **graphics:with-clipping-path** or within graphics encapsulating macros such as **graphics:with-room-for-graphics** and **graphics:with-graphics-transform**. For examples: See the function **graphics:draw-path**. In other contexts, **graphics:with-graphics-translation** is generally more appropriate.

For an overview of **graphics:graphics-translate** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

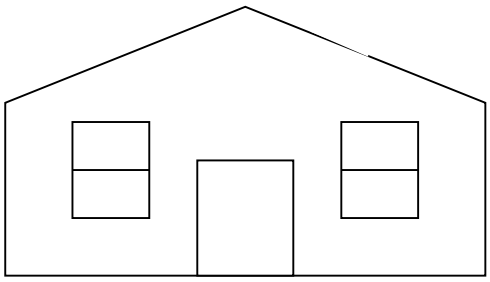
:gray-level *Option*

Specifies the black-to-white level of the graphic as a ratio or decimal fraction between 0 and 1; the default value is 1. On 1-bit devices, gray levels are simulated by stippling.

Example:

```
(defun gray-level-example ()
  (graphics:with-room-for-graphics (t 200)
    (graphics:draw-polygon
      '(100 10 350 10 350 100 225 150 100 100 100 10)
      :gray-level 1/4
      :convex-p t)
    (graphics:draw-polygon
      '(100 10 350 10 350 100 225 150 100 100 100 10)
      :filled nil
      :convex-p t)
    (graphics:draw-polygon
      '(259 137 259 165 289 165 289 124 259 137)
      :gray-level 3/4
      :convex-p t)
    (graphics:draw-rectangle 200 70 250 10
                             :gray-level 1/2
                             :opaque t)
    (graphics:draw-rectangle 200 70 250 10
                             :filled nil)
    (graphics:draw-rectangle 135 90 175 40
                             :gray-level 1/20
                             :opaque t)
    (graphics:draw-rectangle 135 90 175 40
                             :filled nil)
    (graphics:draw-line 135 65 175 65)
    (graphics:draw-rectangle 275 90 315 40
                             :gray-level 1/20
                             :opaque t)
    (graphics:draw-rectangle 275 90 315 40
                             :filled nil)
    (graphics:draw-line 275 65 315 65)))
```

```
(gray-level-example)
```



graphics:gray-level-stipple *gray-level* *Function*

Returns a stipple array that approximates *gray-level*, a number between 0 and 1.

Example:

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-circle 50 50 50 :stipple (graphics:gray-level-stipple .5)))
```

graphics:*identity-transform* *Variable*

The list (1 0 0 1 0 0), which is the representation of the two-dimensional homogeneous transform matrix:

```
1 0 0
0 1 0
0 0 1
```

Applying the identity transform to any set of coordinates results in the same set of coordinates. Composing the identity transform with any other transform *t* results in *t*. See the section "Coordinate System Facilities".

:initial-dash-phase *Option*

Specifies the offset, in pixels, of the start of the first dash from the starting point of the line; the default value is `0`.

This option is not operable if the **:dashed** option to the drawing function is **nil**.

graphics:invert-transform *transform* &optional *(into-transform* **(graphics:make-graphics-transform)***)* *Function*

transform

A list of the essential six elements of a two-dimensional homoge-

neous graphics transformation matrix describing a combination of scaling, rotation, and translation.

into-transform Another such list that is to become the result of the inversion.

Returns or optionally stores into *into-transform* the result of calculating the inverse of the matrix represented by *transform*. Application of the transform resulting from **graphics:invert-transform** to a point originally moved by applying *transform* will restore the point to its original location. See the section "Coordinate System Facilities".

:line-end-shape *Option*

Specifies the shape for the ends of lines drawn by a drawing function, one of **:butt**, **:square**, **:round**, or **:no-end-point**; the default is **:butt**.

To see the differences among end shapes, try evaluating the following:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-line 200 40 200 140
                      :line-end-shape :butt
                      :thickness 20)
  (graphics:draw-line 240 40 240 140
                      :line-end-shape :no-end-point
                      :thickness 20)
  (graphics:draw-line 280 40 280 140
                      :line-end-shape :square
                      :thickness 20)
  (graphics:draw-line 320 40 320 140
                      :line-end-shape :round
                      :thickness 20)
  (graphics:draw-line 190 30 330 30)
  (graphics:draw-line 190 150 330 150))
```

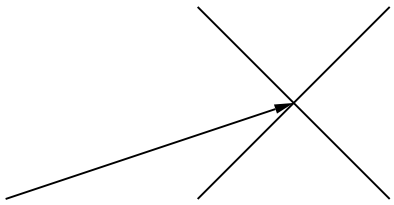
The vertical lines on the left have **:butt**ed ends; they do not extend beyond the y-coordinates (`40` and `140`) given for the line. The lines on the right have **:square**d and **:round**ed ends. These lines extend, because of the **:thickness** parameter (`20`), 10 pixels above and below the top and bottom line coordinates.

This option is not operable if the function draws a filled (`:filled t`) figure.

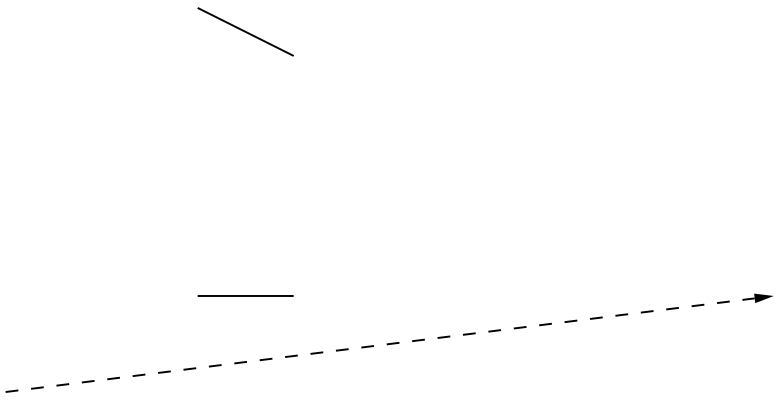
graphics:line-intersection *x1 y1 x2 y2 x3 y3 x4 y4* &optional *(interval* **:closed***)*
Function

Returns the intersection point of the two lines specified by *<x1 y1>* to *<x2 y2>* and *<x3 y3>* to *<x4 y4>*, or **nil** if the lines do not intersect. *interval* can be **t** to indicate you want the intersection point wherever it is, **:open** to indicate the endpoints do not count as matching, or **:closed** (default) to include endpoints.

```
(defun show-line-intersection (x1 y1 x2 y2 x3 y3 x4 y4)
 (graphics:with-room-for-graphics ()
  (graphics:draw-line x1 y1 x2 y2)
  (graphics:draw-line x3 y3 x4 y4)
  (block done
   (multiple-value-bind (x y)
      (graphics:line-intersection x1 y1 x2 y2 x3 y3 x4 y4)
    (when (and x y)
     (return-from done
      (graphics:draw-arrow 0 0 x y))))
   (multiple-value-bind (x y)
      (graphics:line-intersection x1 y1 x2 y2 x3 y3 x4 y4 t)
    (when (and x y)
     (return-from done
      (graphics:draw-arrow 0 0 x y :dashed t)))))))

(show-line-intersection 100 0 200 100 100 100 200 0)
```



```
(show-line-intersection 100 50 150 50 100 200 150 175)
```

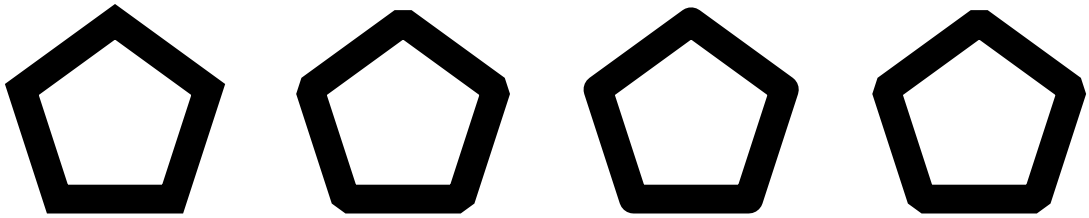


:line-joint-shape *Option*

Specifies the shape of joints between line segments of closed, unfilled figures, when the **:thickness** option to a drawing function is greater than 1. The possible shapes are **:miter**, **:bevel**, **:round**, and **:none**; the default is **:miter**.

The following example draws four pentagons illustrating the differences among, from left to right, the **:miter**, **:bevel**, **:round**, and **:none** joint shapes:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-regular-polygon 40 40 100 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :miter)
  (graphics:draw-regular-polygon 190 40 250 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :bevel)
  (graphics:draw-regular-polygon 340 40 400 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :round)
  (graphics:draw-regular-polygon 490 40 550 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :none))
```



This option is not operable if the function draws a filled (`:filled t`) figure.

Some hardcopy devices may not support all line joint shapes. Notably, PostScript printers treat **:none** as **:bevel**.

graphics:make-contrasting-pattern *index-number number-of-indices* *Function*

Returns a pattern instance that draws a distinct (recognizably different) pattern for adjacent indices. This is done with a few colors if the output stream supports color and with a few stipples otherwise.

Example:

```
(defun pie-chart ()
 (let ((nitems 4)
       (total 2.0))
   (graphics:with-room-for-graphics (t 300)
     (graphics:with-graphics-translation (t 350 150)
       (loop for (fraction . rest) on '(1.0 .7 .1 .2)
         for angle = (* pi 1/2) then nangle
         as nangle = (if rest (- angle
                       (* graphics:2pi (/ fraction total)))(* pi 5/2))
              for item-no from 0
              do
        (graphics:draw-circle 0 0 50 :start-angle angle
              :end-angle nangle :clockwise t
              :pattern
                (graphics:make-contrasting-pattern item-no nitems))
         (graphics:draw-circle 0 0 50  :start-angle angle
                                       :end-angle nangle
                                       :clockwise t
                                       :filled nil :thickness 2)
          (graphics:draw-line 0 0 50 0 :rotation angle
                                       :thickness 2))))))
```

graphics:make-device-conditional-pattern *device-alist* *Function*

Creates a pattern instance that draws in a way that is conditionalized by the output stream. Each *device-alist* element is **(type . drawing-args)**. The types are:

:color	A stream that can draw in color.
:postscript	A stream that goes to a postscript interpreter.
:window	A window stream.
otherwise	(the symbol **otherwise**) Anything.

The first element that matches is used, which means that drawing is done with its **drawing-args**, which is a set of keyword/value pairs acceptable to **graphics:with-drawing-state**. Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-rectangle 0 75 90 0 :pattern
            (graphics:make-device-conditional-pattern
               `((:color :color :magenta)
                 (otherwise :stipple ,stipples:horizontal-dashes)))))
```

graphics:make-graphics-transform &key *:r11 :r12 :r21 :r22 :tx :ty* *Function*

The **defstruct** constructor for a transform matrix: it returns the transform matrix specified by the keyword arguments. See the section "Advanced Transformation Facilities".

graphics:make-identity-transform *Function*

Creates and returns the list (1 0 0 1 0 0).

graphics:make-raster-array-with-correct-width *width height* &rest *args* &key *(:element-type* **t***)* &allow-other-keys *Function*

width The minimum width of the raster array in pixels.

height The height of the raster array.

args A list containing the remaining arguments, which can include the keyword argument **:element-type** and other keys, which are the possible options for **make-array**.

:element-type Specifies the element type of the raster array. One of (**unsigned-byte** *n*), **fixnum**, **character**, **string-char**, or **boolean**.

See the section "Common Lisp Array Element Types".

Calls **make-raster-array** with the *width* argument adjusted if necessary to ensure that it is an acceptable width for a raster array (that is, it is a multiple of 32 so it can be used by **bitblt**).

graphics:make-simple-pattern &rest *drawing-args* *Function*

Returns a pattern instance which does the drawing as if *drawing-args* had been passed in place of :pattern <instance>. The possible drawing arguments are the same as those acceptable to **graphics:with-drawing-state**. The most useful keywords to include are **:alu**, **:pattern**, **:stipple**, **:tile**, **:gray-level**, **:color**, and **:opaque**.

```
(graphics:with-room-for-graphics (t 100)
       (graphics:draw-circle 50 50 50 :pattern
            (graphics:make-simple-pattern
                          :stipple stipples:weave8 :gray-level .5))
```

graphics:make-two-color-stipple *stipple ones-color zeros-color* *Function*

Returns a pattern instance that draws the stipple pattern *stipple* in the two colors specified by *ones-color* and *zeros-color*. If the output stream does not support color, the stipple is drawn in black-and-white in the normal way.

Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:draw-circle 50 50 50 :pattern
                        (graphics:make-two-color-stipple
                          stipples:filled-diamonds
                          :red :yellow)))
```

graphics:map-points *function points* *Function*

function A function that accepts three arguments: *x, y,* and *last-point-p*. This last argument is a Boolean which indicates that this is the last point to be processed.

points A sequence, that is, a list or an array, of *x* and *y* coordinates of a collection of points, for example, (*x1 y1 x2 y2 x3 y3 x4 y4*).

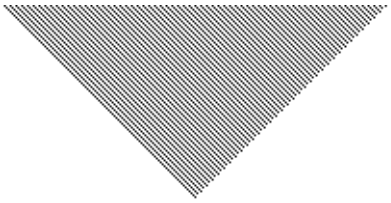
Sequentially calls *function* on each coordinate pair in the sequence.

:mask *Option*

Specifies a bitmap of points affected by the drawing operation; the default is **nil**, that is, no bitmap.

Use this option to mask out portions of the output graphic that you do not want displayed. For example, the following form outputs a rectangle, but does so through a triangular mask, so that only a triangle gets displayed:

```
(graphics:with-room-for-graphics (t 200)
  (multiple-value-bind (bitmap x y)
       (graphics:with-output-to-bitmap ()
         (graphics:draw-triangle 100 50 200 150 300 50))
    (graphics:draw-rectangle 0 200 400 0
                             :gray-level .33
                             :mask bitmap :mask-x x :mask-y (+ 100 y))))
```



:mask-x *Option*

Specifies the *x*-coordinate, with respect to the local graphics origin, of the lower lefthand corner of the position of the rectangular bitmap region specified by the **:mask** option.

:mask-y *Option*

Specifies the *y*-coordinate, with respect to the local graphics origin, of the lower lefthand corner of the position of the rectangular bitmap region specified by the **:mask** option.

:opaque *Option*

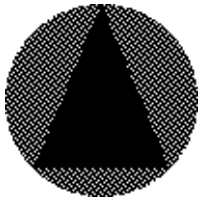
Boolean option specifying whether pixels in the source pattern of a drawing function are cleared (before the graphic is output) or left alone; the default is **t**. **:opaque t** means draw pixels that are off in the background color. The result of **:opaque t** is not influenced by the **:alu** supplied for the pixels that are on. If you draw something over an existing figure using **:opaque t :alu :flip**, the pixels that were originally on are inverted, and pixels that were originally off are cleared (set to the background color).

To see the effect of this option, try calling the following function with **t** and then **nil**.

```
(defun opaque-example (t-or-nil)
  (graphics:with-room-for-graphics (t 200)
    (graphics:draw-circle 200 100 50
                          :pattern tv:75%-gray)
    (graphics:draw-triangle 165 65 235 65 200 150
                            :pattern tv:25%-gray
                            :opaque t-or-nil)))
```

```
(opaque-example t)
```

```
(opaque-example nil)
```



:pattern *Option*

Specifies a pattern to be drawn within the figure created by a drawing function. **:pattern** is the most general control of the "looks" of thie inside of a filled shape. The default is **nil**. The **:pattern** drawing option can be a(n)

- **stipple** This is identical to **:stipple** <*value*>.
- **color** This is identical to **:color** <*value*>.
- **instance** The case of interest here.

Several functions are provided for making instances to pass as the pattern. See the function **graphics:make-simple-pattern**. See the function **graphics:make-two-color-stipple**. See the function **graphics:make-contrasting-pattern**. See the function **graphics:make-device-conditional-pattern**. Additionally, there are three layered protocols for implementing one's own instances to be passed as **:pattern**. See the section "Texturing".

```
(defun pattern-example1 ()
  (graphics:with-room-for-graphics (t 200)
    (let ((bitmap (tv:make-binary-gray 8 8
                    '(#b00000000    ; The picture of
                      #b00001000    ; what you want
                      #b00111000    ; the bit pattern
                      #b00001000    ; displayed to
                      #b00001000    ; look like, in
                      #b00001000    ; this case the
                      #b00001000    ; number 1.
                      #b00111110))))
          (graphics:draw-circle 200 100 50
                                :pattern bitmap))))

(defun pattern-example2 ()
  (let ((bitmap (tv:with-output-to-bitmap (t)
                  (graphics:draw-string
                    "symbolics" 0 0
                    :character-style
                    '(:swiss :bold-italic :large)))))
    (graphics:with-room-for-graphics (t 250)
      (graphics:draw-triangle 100 25 400 25 250 250
                              :pattern bitmap))))
(pattern-example1)
(pattern-example2)
```

graphics:pattern-call-with-drawing-parameters *pattern function stream drawing-state* *Generic Function*

As implemented by *pattern*, which should be built upon **graphics:basic-pattern**, should call *function* with keyword arguments such as would be acceptable to **graphics:with-drawing-state**. It can examine *stream* and *drawing-state* to determine what arguments to supply. See the section "Texturing". Example:

```
(defflavor my-pattern () (graphics:basic-pattern))

(defmethod (graphics:pattern-call-with-drawing-parameters
             my-pattern)
           (function stream drawing-state)
  (ignore drawing-state)
  (if (color:color-stream-p stream)
      (funcall function :color :magenta)
      (funcall function :stipple stipples:hearts :gray-level .75)))

(compile-flavor-methods my-pattern)

(graphics:with-room-for-graphics ()
  (graphics:draw-rectangle 0 0 200 100 :pattern (make-instance
                                                  'my-pattern)))
```

Note that this example could have been done in a data-driven way rather than in this procedural way by using **graphics:make-device-conditional-pattern**.

graphics:pattern-compute-raster-source-pattern *pattern source-so-far ones-alu zeros-alu temporary-p stream drawing-state* *Generic Function*

Returns *updated-source*, *ones-alu*, *zeros-alu*, and *temporary-p*; fills in the *source-so-far* with *pattern*; and updates the alus with *ones-alu* and *zeros-alu*. If *temporary-p* is true, the temporary raster sheet is deallocated. Pattern is to be drawn on *stream* with *drawing-state*. Methods that need to exercise fuller control over the individual slices in the shape should return **self** as the source. See the section "Texturing". Example:

```
(defflavor my-pattern-2 () (graphics:raster-device-pattern))
```

```
(defmethod (graphics:pattern-compute-raster-source-pattern my-pattern-2)
           (source-so-far ones-alu zeros-alu temporary-p ignore ignore)
  ;; Get a raster we can modify.
  (unless temporary-p
    (multiple-value-bind (width height)
       (if source-so-far
           (multiple-value-bind (width height)
               (decode-raster-array source-so-far)
             (values (lcm width 32) height))
           (values 32 1))
      (let ((source (with-stack-list (dims height width)
                      (tv:allocate-temp-sheet-raster-and-header
                                           dims :type 'tv:art-1b))))
        (if source-so-far
            (bitblt boole-1 width height source-so-far 0 0 source 0 0)
            (bitblt boole-set width height source 0 0 source 0 0))
        (setq source-so-far source
              temporary-p t))))
  (multiple-value-bind (width height)
      (decode-raster-array source-so-far)
    (bitblt boole-xor width height stipples:vertical-lines 0 0
                                                  source-so-far 0 0))
  (values source-so-far ones-alu zeros-alu temporary-p))

(compile-flavor-methods my-pattern-2)

(graphics:with-room-for-graphics ()
  (let ((pattern (make-instance 'my-pattern-2)))
    (graphics:draw-rectangle 0 0 200 100 :pattern pattern)
    (graphics:draw-rectangle 0 100 200 200
                         :stipple stipples:hearts :pattern pattern)))
```

graphics:pattern-draw-raster-slice *pattern width height x y ones-alu zeros-alu stream drawing-state* *Generic Function*

pattern should use some of the primitive raster drawing messages (such as **:draw-rectangle** or **:draw-1-bit-raster**) to output an appropriately patterned slice to *stream* whose upper-left corner is at *x,y* and whose size is *width,height*. See the section "Texturing". Example:

```
(defflavor random-pattern
        (phase)
        (graphics:raster-slice-device-pattern
                                  lgp:postscript-device-pattern)
  (:constructor make-random-pattern (phase)))
```

```
(defmethod (graphics:pattern-draw-raster-slice random-pattern)
           (width height x y ones-alu zeros-alu stream ignore)
  (let ((raster (graphics:make-raster-array-with-correct-width
                                          width height
                                         :element-type 'bit)))
    (loop for real-y from y
          for y from 0 below height
          do
      (loop repeat (floor (* width (mod real-y phase)) phase)
            do
        (setf (raster-aref raster (random width) y) 1)))
    (send stream :draw-1-bit-raster width height
                         raster 0 0 x y ones-alu zeros-alu)))

(defmethod (lgp:pattern-output-postscript-code random-pattern)
           (device-stream filled ignore ignore)
  (format device-stream
           "8 0 {pop pop 1 rand and} setscreen .25 setgray ")
  (write-string (if filled "fill" "stroke") device-stream))

(compile-flavor-methods random-pattern)

(graphics:with-room-for-graphics (t 100)
   (graphics:draw-triangle 0 0 50 100 100 0
                           :pattern (make-random-pattern 8)))
```

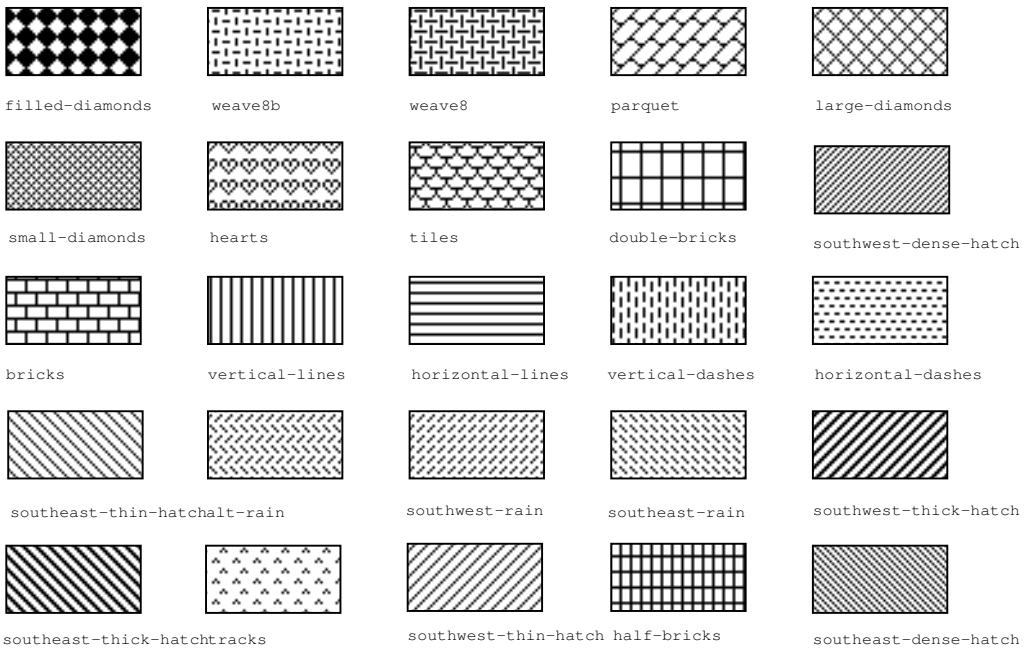
lgp:pattern-output-postscript-code *pattern device-stream filled stream drawing-state*
Generic Function

As implemented by *pattern*, which should be built upon **lgp:postscript-device-pattern**, should send PostScript code to *device-stream*, which will be a thin character output stream. The current path will be the figure being drawn. *filled* is **t** if the outline is to be filled and **nil** if it is to be stroked. The implementation can examine *stream* and *drawing-state* to determine what to do. See the section "Texturing".

graphics:*pattern-stipple-arrays* *Variable*

A list of those stipple arrays that are patterns, as opposed to gray levels.

See the variable **graphics:*stipple-arrays***.



(flavor:method :point tv:graphics-mixin) *x y* *Method*

Returns the numerical value of the picture element at the specified coordinates. The result is **0** or **1** on a black-and-white TV. Clipping is performed; if the coordinates are outside the window, the result will be **0**.

lgp:postscript-device-pattern *Flavor*

The flavor for patterns on postscript devices. Built on **graphics:raster-device-pattern**, it has the required method, **lgp:pattern-output-postscript-code**.

graphics:raster-graphics-mixin *Flavor*

A flavor mixin to windows and other raster devices that implements graphics by means of primitive slice functions.

graphics:read-encoded-graphics-as-characters *stream* *Function*

Returns an array with elements of type **(unsigned-byte 8)**, which contains an encoded version of a graphics function.

See the function **graphics:binary-encode-graphics-to-array**.See the section "Other Advanced Facilities for Graphic Output".

stream A character stream to which an encoded graphics function has been written with the function **graphics:write-encoded-graphics-as-characters**.

graphics:replacing-graphics-presentation (*stream presentation &rest args*) &body *body* *Function*

Replaces a graphic presentation with the graphic output generated in *body*. The new output will also be a graphics presentation and is returned as such. The replacing operation is done in such a way that flicker is minimized, making this facility useful for simple animations. Returns two values, the old display, which is a bitmap stream, and the pattern array generated by the body.

stream The output stream.

presentation The graphics presentation to be replaced.

args Optional keyword arguments, including **:pattern-array**, the array that holds the pattern array generated by the body, and **:bitmap-stream**, a specially allocated raster array that contains the old display.

The following example illustrates the use of the **:bitmap-stream** and **:pattern-array** options to **graphics:replacing-graphics-presentation**:

```
(defun animation-example ()
  (graphics:with-room-for-graphics (t 150)
    (graphics:with-output-to-bitmap-stream (bitmap-stream
                          :for-stream *standard-output*)
      (let ((old-output nil)
            (old-pattern nil))
        (loop for i to 25
              as x = (+ 100 (* i 16))
              do
          (multiple-value-setq (old-output old-pattern)
            (graphics:replacing-graphics-presentation (t old-output
                        :pattern-array old-pattern
                        :bitmap-stream bitmap-stream)
              (graphics:with-graphics-translation (t x 100)
                (graphics:draw-circle 0 0 20)
                (let ((angle (/ (* pi i) 25)))
                  (graphics:draw-triangle 20 0 40 0 40 10
                            :rotation angle)
                  (graphics:draw-circle 0 0 30
                      :inner-radius 20 :start-angle angle)))))))))

(animation-example)
```

Use of the options as illustrated saves the cost of allocation of arrays and bitmap streams each time around the loop. For an overview of **graphics:replacing-graphics-presentation** and related functions:

See the section "Other Basic Facilities for Graphic Output".

:return-presentation *Option*

Boolean option specifying whether a drawing function should return the newly created graphic as a presentation object; the default is **nil**.

Use this option when you wish to manipulate the output of a single drawing function as a presentation. If you want to manipulate the collective output of a series of drawing functions as a single presentation, use the **graphics:with-output-as-graphics-presentation** macro instead.

Some facilities are provided for handling graphic presentations, in particular, **graphics:erase-graphics-presentation** and **graphics:replacing-graphics-presentation**. The following example uses the former in conjunction with the **:return-presentation** keyword:

```
(graphics:with-room-for-graphics (t 100)
 (graphics:with-graphics-scale (t 50)
  (let ((presentations
         (list
          (graphics:draw-regular-polygon .75 .5 1.25 .5 4
            :gray-level .25 :return-presentation t)
          (graphics:draw-regular-polygon .25 0 1.25 0 3
            :gray-level .75 :return-presentation t)
          (graphics:draw-ellipse 1.5 .5 .4 .4
            :gray-level .5 :return-presentation t))))
    (sleep 2)
    (graphics:erase-graphics-presentation (first presentations)))))
```

:rotation *Option*

Specifies the rotation of the graphic in plus or minus radians; the default is `0`. The axis of rotation is the local origin (0, 0).

In the following example, the origin is first established at the lower, left corner of the graphic display area created by **graphics:with-room-for-graphics**, then translated 300 pixels to the right and 15 pixels up by the **graphics:with-graphics-translation** macro. In this coordinate system, the arrow is rotated counter-clockwise about the origin at an offset of 10 pixels.

```
(defun rotating-arrow ()
  (graphics:with-room-for-graphics (t 225)
    (graphics:with-graphics-translation (t 300 15)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5)
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .125))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .25))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .375))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .5))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .625))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .75))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .875))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation pi))))

(rotating-arrow)
```

graphics:saving-graphics-transform *(*&optional *stream)* &body *body* *Function*

Perform *body* while *stream* has a modifiable copy of its current transformation matrix. That is, functions like **graphics:graphics-rotate** can be called from within **graphics:saving-graphics-transform** without permanent effect.

:scale *Option*

Specifies a number applied as a scaling factor to the x and y parameters of a drawing function; the default is 1.

:scale does not affect the line thickness parameter. To do so, use the **:scale-thickness** keyword.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-circle 1 1 1 :scale 20))
```

:scale-dashes *Option*

Boolean option specifying whether to scale dashes when the **:dashed** option to a drawing function is **t** and a scaling factor is specified (via the **:scale** keyword: See the option **:scale**.). The default is **nil**.

For an example: See the option **:dash-pattern**.

This option is not operable if the **:dashed** option is **nil**.

scale-float *float integer* *Function*

Computes and returns (* *float* $2^{integer}$).

Although the same result can be obtained by using exponentiation and multiplication, the use of **scale-float** can be much more efficient and avoids the intermediate overflow and underflow if the final result is representable.

Examples:

```
(scale-float .5 2) => 2.0
(scale-float .5 3) => 4.0
(scale-float .5 4) => 8.0
(scale-float .75 2) => 3.0
```

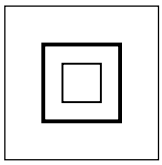
For a table of related items, see the section "Functions that Decompose and Construct Floating-point Numbers".

:scale-thickness *Option*

Boolean option specifying whether to scale thickness, as well as other linear dimensions, when a scaling factor is specified (via the **:scale** keyword to a drawing function: See the option **:scale**.). The default is **nil**.

This option is not operable if the function draws a filled (`:filled t`) figure.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:with-graphics-translation (t 100 50)
  (graphics:draw-rectangle -10 -10 10 10 :filled nil :thickness 1)
  (graphics:draw-rectangle -10 -10 10 10 :filled nil :thickness 1 :scale 2)
  (graphics:draw-rectangle -10 -10 10 10 :filled nil :thickness 1 :scale 4
                                :scale-thickness nil)))
```



graphics:sector-wide-p *start-angle end-angle* *Function*

Returns a Boolean value that is **t** if the angle from *start-angle* to *end-angle* is no more than a quarter circle in the clockwise direction; otherwise **nil**.

```
(graphics:sector-wide-p (* pi 1/4) (* pi 3/8))

NIL
```

graphics:set-current-position *new-x new-y* &key (*stream* ***standard-output***) (*explicit* **t**) *Function*

Moves the graphics cursor to a specified position.

new-x The new x-coordinate.

new-y The new y-coordinate.

:stream Specifies the output stream; the default is ***standard-output***.

:explicit Boolean option specifying whether this movement opens up a new portion of a path for **graphics:draw-path**; the default is **t**. Specify **nil** for operations that do output and then move the cursor.

This facility is useful with drawing functions that explictly use the graphics cursor. Such functions include **graphics:draw-bezier-curve-to**, **graphics:draw-circular-arc-to**, **graphics:draw-line-to**, and other facilities commonly used for creating path-drawing functions. For examples of path-drawing functions: See the function **graphics:draw-path**.

For an overview of **graphics:set-current-position** and related functions: See the section "Drawing Functions".

graphics:standard-graphics-mixin *Flavor*

A flavor mixin defining required methods of graphics primitives. It should be included in all streams that support graphics.

graphics:*stipple-arrays* *Variable*

A list of predefined stipple arrays that can be specified for the graphics **:stipple** option. These are the names of the arrays in the list:

stipples:large-diamonds
stipples:small-diamonds
stipples:filled-diamonds
stipples:weave8b
stipples:weave8
stipples:parquet
stipples:diagonals
stipples:small-diamonds
stipples:hearts
stipples:tiles
stipples:double-bricks
stipples:half-bricks
stipples:bricks
stipples:vertical-lines
stipples:horizontal-lines
stipples:vertical-dashes
stipples:horizontal-dashes
stipples:tracks
stipples:alt-rain
stipples:southwest-rain
stipples:southeast-rain
stipples:southwest-thick-hatch
stipples:southeast-thick-hatch
stipples:southwest-thin-hatch
stipples:southeast-thin-hatch
stipples:southwest-dense-hatch
stipples:southeast-dense-hatch
stipples:5.5%-gray
stipples:6%-gray
stipples:7%-gray
stipples:8%-gray
stipples:9%-gray
stipples:10%-gray
stipples:12%-gray
stipples:hes-gray
stipples:33%-gray
stipples:75%-gray
stipples:25%-gray
stipples:50%-gray

:stream *Option*

Specifies the output stream for a drawing function; the default is ***standard-output***.

graphics:stream-transform *stream* *Function*

stream Any graphics stream, that is, any stream that has a flavor component of **graphics:standard-graphics-mixin**.

Returns a list whose elements are the current transform matrix of *stream*. If the coordinate context is established with one of the **graphics:with-graphics** ... macros, the value returned is a stack list. See the section "Coordinate System Facilities".

graphics:stream-transform-point *x y stream* *Function*

Returns the result of transforming the point <*x y*> by the inverse of the current transformation matrix of *stream*, that is, it returns the coordinates the point would have if the current transformation matrix were not applied.

graphics:stream-untransform-point *x y stream* *Function*

Returns the result of transforming the point <*x y*> by the current transformation matrix of *stream*.

:scale-x *Option*

Specifies a number applied as a scaling factor to the x parameters of a drawing function; the default is 1.

```
(graphics:with-room-for-graphics (t 100)
```

```
 (graphics:draw-circle 50 50 20 :scale-x 4))
```

:scale-y *Option*

Specifies a number applied as a scaling factor to the y parameters of a drawing function; the default is 1.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-circle 100 100 50 :scale-y 1/2))
```

:thickness *Option*

Specifies the thickness, in pixels, of the line or lines drawn by a drawing function. The default is device-dependent: for the screen, it is 0, which specifies the minimum thickness for that device; for the lgp2 and lgp3, it is 1. The thinnest possible thickness is 0, which also specifies that the line is to be positioned roughly. For accurate positioning, specify a thickness of 1.

This option is not operable if the function draws a filled (`:filled t`) figure.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-triangle 0 0 50 0 25 25 :filled nil :thickness 2)
 (graphics:draw-triangle 100 0 150 0 125 25 :filled nil :thickness 4)
 (graphics:draw-triangle 200 0 250 0 225 25 :filled nil :thickness 8))
```

:transform *Option*

Specifies a list of six elements in the graphics transformation matrix applied to the local coordinate system used for a drawing function. The element order is *a*, *b*, *c*, *d*, *x*, and *y* (see below). The default is **nil**, resulting in no transformation.

Arbitrary transformation of coordinates is effected by multiplication of coordinate vectors by a transformation matrix. For coordinates in two-dimensional space, a 3 x 3 transformation matrix is used,

```
a   b   0

c   d   0

x   y   1
```

of which the elements in the third column are constant. Thus, six elements, effectively control the transformation as follows:

- **Scaling** in the x and y dimensions is controlled by elements *a* and *d*, respectively. Values of `1` for these elements result in no scaling.
- **Translation** in the x and y dimensions is controlled by elements *x* and *y*, respectively. Values of `0` for these elements result in no translation.
- **Rotation** about the origin is controlled by elements *a*, *b*, *c*, and *d*. Counterclockwise rotation by an angle α is effected by *a* = cosα, *b* = sinα, *c* = -sinα, and *d* = cosα. A value of `0` for *b* and *c* results in no rotation (α = 0).

Example:

```
;;; This example scales the size of the rectangle by a
;;; factor of 4 and translates its position by 100 in
;;; the x direction and 50 in the y direction.

(graphics:with-room-for-graphics (t 200)
  (graphics:draw-rectangle 0 20 20 0
                           :transform '(4 0 0 4 100 50)))
```

graphics:transform-distance *x y transform* *Function*

Returns the product of the matrix multiplication of the vector <*x, y*> and the transformation matrix of *transform*. The values returned are the transformed values of the *x* and *y* distances. See the section "Coordinate System Facilities".

x A number representing a distance along the *x* direction.

y A number representing a distance along the *y* direction.

transform A list of the essential six elements of a two-dimensional homogeneous graphics transformation matrix describing a combination of scaling, rotation, and translation.

graphics:transform-point *x y transform* *Function*

Returns the product of the matrix multiplication of the vector <*x, y*> and the transformation matrix of *transform*. The values returned are the transformed coordinates of the point. See the section "Coordinate System Facilities".

x A number representing the x-coordinate of a point.

y A number representing the y-coordinate of a point.

transform A list of the essential six elements of a two-dimensional homogeneous graphics transformation matrix describing a combination of scaling, rotation, and translation.

:translation *Option*

Specifies x and y offsets relative to the local origin (0, 0) used for a drawing function. The offsets are specified as a list, (*x y*). The offset units depend on the output device; if it is the terminal screen, the units are pixels.

Using this option has the effect of moving the local origin to the new position specified by *x* and *y*. The default is **nil**, no translation.

```
;;; Contrast this with giving 50 50 as the center, which would have caused it to
;;; be rotated.
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-ellipse 0 0 40 20 :translation '(50 50) :rotation (/ pi 4)))
```

graphics:untransform-distance *x y transform* *Function*

Returns the product of the matrix multiplication of the vector <*x, y*> and the inverse of the transformation matrix represented by *transform*. The values returned are the original values of *x* and *y*; that is, if the given distance <*x, y*> is the result of having undergone transform *transform*, then **graphics:untransform-distance** returns the coordinates of the original distance.

Another way to look at the result is to consider that *transform* is composed of a set of transformations of a point: a translation, followed by a rotation, followed by a scaling. **graphics:untransform-distance** performs these same transformations in reverse, restoring the transformed distance to its original values. See the section "Coordinate System Facilities".

x A number representing a distance along the *x* direction.

y A number representing a distance along the *y* direction.

transform A list of the essential six elements of a two-dimensional homogeneous graphics transformation matrix describing a combination of scaling, rotation, and translation.

graphics:untransform-point *x y transform* *Function*

Returns the product of the matrix multiplication of the vector <*x, y*> and the inverse of the transformation matrix represented by *transform*. The values returned are the original coordinates of the point <*x, y*>; that is, if the given point <*x, y*> is at its current location as a result of having undergone transform *transform* from some original location, then **graphics:untransform-point** returns the coordinates of that original location.

Another way to look at the result is to consider that *transform* is composed of a set of transformations of a point: a translation, followed by a rotation, followed by a scaling. **graphics:untransform-point** performs these same transformations in reverse, restoring the transformed point to its original coordinates. See the section "Coordinate System Facilities".

x A number representing the x-coordinate of a point.

y A number representing the y-coordinate of a point.

transform A list of the essential six elements of a two-dimensional homogeneous graphics transformation matrix describing a combination of scaling, rotation, and translation.

graphics:untransform-window-points *stream* &rest *points* *Macro*

points A list of numbers representing pairs of *x*- and *y*-coordinates of points.

Modifies a list containing the products of the matrix multiplication of each vector <*x, y*> and the inverse of the transformation matrix of *stream*. The new values in the list are the transformed coordinates of the points.

See the macro **graphics:transform-window-points**.See the section "Coordinate System Facilities".

graphics:with-clipping-from-output (*stream* &body *clipping-region-body*) &body *output-body* *Function*

Binds the local environment to apply a clipping region to the output of drawing functions in the body. The clipping region limits output from the drawing functions to points within the region. The shape of the clipping region is defined by a drawing function.

stream The output stream.

clipping-region-body A drawing function outlining the clipping region. This can be any function generating a closed figure as output, including functions based on **graphics:draw-path** for complex figures. Output should be to *stream*.

graphics:with-clipping-path is a facility closely related to **graphics:with-clipping-from-output**. You use it to apply clipping regions defined by path-drawing functions, rather than by drawing functions as in the case of **graphics:with-clipping-from-output**. For an example illustrating the use of the two facilities: See the function **graphics:with-clipping-path**.

For an overview of **graphics:with-clipping-from-output** and related functions: See the section "Drawing Functions".

graphics:with-clipping-mask *(stream mask* &rest *mask-args)* &body *body* *Macro*

Performs the graphics output specified by *body* on *stream* using *mask* as a clipping region. This is similar to using a drawing function with **:mask** *mask*. *mask-args* are `&key (left 0) (top 0) right bottom`.

graphics:with-clipping-path (*stream path-function* &rest *path-filling-args*) &body *body* *Function*

Binds the local environment to apply a clipping region to the output of drawing functions in the body. The clipping region limits output from the drawing functions to points within the region. The shape of the clipping region is defined by a path-drawing function.

stream The output stream.

path-function A path-drawing function outlining the clipping region. The path can be arbitrarily complex, contain straight- and curved-line segments, and include more than one closed subpath. (For several examples of path functions: See the function **graphics:draw-path**.)

graphics:with-clipping-from-output is a facility closely related to **graphics:with-clipping-path**. You use it to apply clipping regions defined by drawing functions, rather than by path-drawing functions as in the case of **graphics:with-clipping-path**. The following example shows the use of both facilities to achieve, in this case, the same graphic effect.

```
;;; Star-shaped path drawer used by clipping-example
```

```
(defun star-drawer (*standard-output*)
  (graphics:set-current-position 0 0)
  (dotimes (i 4)
    (graphics:draw-line-to 1 0)
    (graphics:graphics-origin-to-current-position)
    (graphics:graphics-rotate (float (* -4/5 pi) 0.0)))
  (graphics:close-path))

;;; This function draws four figures:
;;; 1) a star
;;; 2) a circle
;;; 3) a circle clipped by a star-shaped path,
;;;     demonstrating graphics:with-clipping-path,
;;; 4) a star clipped by a circular path,
;;;     demonstrating the closely related macro
;;;     graphics:with-clipping-from-output.

(defun clipping-example ()
  (graphics:with-room-for-graphics (t 100)
    (graphics:with-graphics-translation (t 100 50)
      (graphics:with-graphics-scale (t 50)
          (graphics:draw-path #'star-drawer)))
    (graphics:with-graphics-translation (t 200 50)
        (graphics:with-graphics-scale (t 50)
          (graphics:draw-circle 0 0 1/2)))
    (graphics:with-graphics-translation (t 300 50)
        (graphics:with-graphics-scale (t 50)
          (graphics:with-clipping-path (t #'star-drawer)
            (graphics:draw-circle 0 0 1/2))))
    (graphics:with-graphics-translation (t 400 50)
        (graphics:with-graphics-scale (t 50)
          (graphics::with-clipping-from-output
            (t (graphics:draw-circle 0 0 1/2))
            (graphics:draw-path #'star-drawer))))))

(clipping-example)
```

For an overview of **graphics:with-clipping-path** and related functions: See the section "Drawing Functions".

graphics:with-coordinate-mode *(stream mode)* &body *body* *Macro*

Binds the local environment such that the figures produced within it are drawn with the specified coordinate mode.

stream The output stream.

mode Specifies the coordinate mode to be used, which is one of:

:exact This is the default. Figures are drawn exactly according to the coordinates specified. Use this mode if it is important that figures tile correctly or if you require that shapes with fractional coordinates not be rounded to integer shapes.

:integer The coordinates specified to the drawing function are rounded to integer values and special, faster integer drawing methods are used. Use this mode when speed is important in drawing a filled figure or one with thick lines and exactness is of little importance.

:center Figures are drawn so that they are centered around a whole pixel. For example, a circle with specified raduis *r* and center <*x*, *y*> would be drawn with actual center <*x*+1,*y*+1> and radius *r*+1/2. Use this mode when you want small circles to appear symmetrical about a single pixel and the circles need not align with other shapes.

See the section "Scan Conversion".

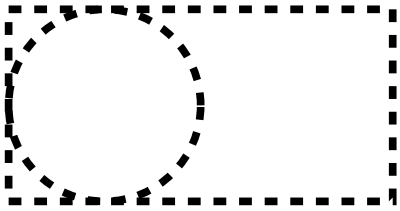
graphics:with-drawing-state *(stream* &rest *args)* &body *body* *Macro*

Binds the local environment such that the drawing-state arguments in *args* are in effect during the execution of *body* with the output being sent to *stream*. These arguments can include any of the drawing keyword arguments other than the transform arguments. These are:

:thickness **:scale-dashes**
:scale-thickness **:alu**
:line-end-shape **:pattern**
:line-joint-shape **:stipple**
:dashed **:tile**
:dash-pattern **:gray-level**
:initial-dash-phase **:color**
:draw-partial-dashes **:opaque**

Example:

```
(graphics:with-room-for-graphics (t 100)
 (graphics:with-drawing-state (t :thickness 4 :dashed t)
  (graphics:draw-rectangle 0 100 200 0 :filled nil)
  (graphics:draw-circle 50 50 50 :filled nil)))
```



graphics:with-graphics-identity-transform (&optional *stream*) &body *body* *Function*

Binds the local environment such that the graphics transformation matrix is the identity matrix. Graphics output from functions in the body is subject only to those coordinate system changes that are implemented by drawing-function keywords. (See the section "Keyword Options Affecting the Coordinate System".)

stream The output stream.

Examples:

```
;;; Note that these will draw at the beginning of the window, because
;;; the with-room-for-graphics is defeated.
(graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-transform
    (*standard-output* '(4 0 0 4 100 50))
    (graphics:with-graphics-identity-transform (t)
      (graphics:draw-rectangle 0 20 20 0))))

(graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-transform
    (*standard-output* '(4 0 0 4 100 50))
    (graphics:with-graphics-identity-transform (t)
      (graphics:draw-rectangle 0 20 20 0
                               :transform
                               '(4 0 0 4 100 50)))))
```

For an overview of **graphics:with-graphics-identity-transform** and related functions: See the section "Coordinate System Facilities".

graphics:with-graphics-rotation (*stream theta*) &body *body* *Function*

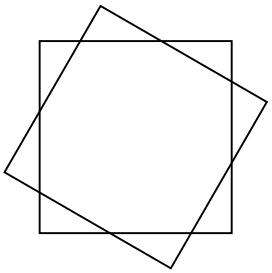
Binds the local environment such that the graphics transformation matrix is modified to rotate graphics output. (For an explanation of the graphics tranformation matrix: See the function **graphics:with-graphics-transform**.)

stream The output stream.

theta A number specifying the rotation in radians.

Example:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-translation (t 100 100)
    (graphics:with-graphics-rotation (t (* 1/3 pi))
      (graphics:draw-rectangle -50 50 50 -50 :filled nil))
    (graphics:draw-rectangle -50 50 50 -50 :filled nil)))
```



For an overview of **graphics:with-graphics-rotation** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

graphics:with-graphics-scale (*stream scale* &optional (*y-scale* **graphics::scale**)) &body *body* *Function*

Binds the local environment such that the graphics transformation matrix is modified to apply a scaling factor to graphics output. (For an explanation of the graphics tranformation matrix: See the function **graphics:with-graphics-transform**.)

stream The output stream.

scale A number specifying the scaling factor.

y-scale A number specifying the y scaling factor (if different from *scale*).

Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:with-graphics-scale (t 5)
    (graphics:draw-rectangle 10 10 20 20))
  (graphics:draw-rectangle 10 10 20 20 :gray-level .5))
```

For an overview of **graphics:with-graphics-scale** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

graphics:with-graphics-subroutine *(*&optional *(stream* **'*standard-output*** *)* &rest *special-variable-arguments)* &body *body* *Function*

Improves the density of binary encoding.

stream The graphic output stream

special-variable-arguments Arguments for special variables accessed within *body*.

graphics:with-graphics-transform (*stream transform*) &body *body* *Function*

Binds the local environment such that the graphics transformation matrix is composed with *transform*.

stream The output stream.

transform Specifies a list of six elements in the graphics transformation matrix applied to the local coordinate system used for a drawing function. The element order is *a*, *b*, *c*, *d*, *e*, and *f* (see below).

Arbitrary transformation of coordinates is effected by multiplication of coordinate vectors by a transformation matrix. For coordinates in two-dimensional space, a 3 x 3 transformation matrix is used,

```
a   b   0

c   d   0

e   f   1
```

of which the elements in the third column are constant. Thus, six elements, effectively control the transformation as follows:

- **Scaling** in the x and y dimensions is controlled by elements *a* and *d*, respectively. Values of `1` for these elements result in no scaling.

- **Translation** in the x and y dimensions is controlled by elements *e* and *f*, respectively. Values of `0` for these elements result in no translation.

- **Rotation** about the origin is controlled by elements *a*, *b*, *c*, and *d*. Counterclockwise rotation by an angle α is effected by *a* = cosα, *b* = sinα, *c* = -sinα, and *d* = cosα. A value of `0` for *b* and *c* results in no rotation (α = 0).

Example:

```
;;; This example scales the size of the rectangle by a
;;; factor of 4 and translates its position by 100 in
;;; the x direction and 50 in the y direction.

(graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-transform (t '(4 0 0 4 100 50))
    (graphics:draw-rectangle 0 20 20 0)))
```

For an overview of **graphics:with-graphics-transform** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

graphics:with-graphics-translation (*stream delta-x delta-y*) &body *body* *Function*

Binds the local environment such that the graphics transformation matrix is modified to offset the origin of the graphics coordinate system. (For an explanation of the graphics tranformation matrix: See the function **graphics:with-graphics-transform**.)

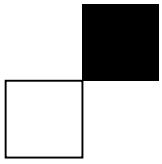
stream The output stream.

delta-x A number specifying the x offset.

delta-y A number specifying the y offset.

Example:

```
(graphics:with-room-for-graphics (t 100)
  (graphics:with-graphics-translation (t 40 40)
    (graphics:draw-rectangle 0 40 40 0))
  (graphics:draw-rectangle 0 40 40 0 :filled nil))
```



For an overview of **graphics:with-graphics-translation** and related functions: See the section "Coordinate System Facilities". Also: See the section "Keyword Options Affecting the Coordinate System".

graphics:with-output-as-graphics-presentation (&optional *stream* &rest *args*) &body *body* *Function*

Binds the local environment such that graphics output generated in the body is returned as a presentation.

stream The output stream.

This macro is the graphics equivalent of **dw:with-output-as-presentation** (on which it is based). It is useful with functions for manipulating graphics presentations, namely, **graphics:erase-graphics-presentation** and **graphics:replacing-graphics-presentation**. The following example shows the use of all three facilities:

```
(defun replacing-example ()
  (graphics:with-room-for-graphics (t 200)
    (let (poly)
      (setq poly (graphics:with-output-as-graphics-presentation ()
                   (graphics:draw-polygon
                     '(100 10 350 10 350 100 225 150 100 100 100 10)
                     :filled t
                     :gray-level .25
                     :convex-p t)))
      (sleep 1)
      (setq poly (graphics:replacing-graphics-presentation (t poly)
                   (graphics:draw-polygon
                     '(100 10 350 10 350 100 225 150 100 100 100 10)
                     :filled t
                     :gray-level 1
                     :convex-p t)))
      (sleep 1)
      (graphics:erase-graphics-presentation poly))))
(replacing-example)
```

For an overview of **graphics:with-output-as-graphics-presentation** and related functions: See the section "Other Basic Facilities for Graphic Output".

tv:bitmap-stream-copy-bitmap *stream* *Function*

Makes a copy of the bitmap or raster associated with a stream and returns five values:

- The bitmap
- Left top
- Right top
- Left bottom

- Right bottom

tv:with-output-to-bitmap *(*&optional *stream* &key *:for-stream :graphics-transform)* &body *body* *Function*

stream The stream to which to return the bitmap.

:for-stream The stream for which the bitmap is intended.

:graphics-transform An optional transform to be applied.

Returns a raster array and positions containing the image output by *body*.

```
(defun bitmap-example (&optional (stream *standard-output*))
  (graphics:with-room-for-graphics ()
    (graphics:draw-triangle 0 0 200 0 50 50
        :tile (tv:with-output-to-bitmap (bstream
                    :for-stream stream)
                (graphics:draw-circle 0 0 10
                    :gray-level .25 :stream bstream)
                (graphics:draw-regular-polygon 8 0 16 0 6
                      :gray-level .75
                    :stream bstream)))))
```

tv:with-output-to-bitmap-stream (*bitmap-stream* &rest *args* &key (*for-stream* **nil**) &allow-other-keys) &body *body* *Function*

bitmap-stream A stream that is a raster array intended to hold the image generated by *body*.

args **:for-stream**, the stream for which the bitmap is intended, and, optionally, **:graphics-transform**, an optional transform to be applied.

Binds *bitmap-stream* to a specially allocated stream that accepts the graphic output during execution of *body*. At any time, the **:bitmap-and-edges** message to this stream returns the current image.

graphics:with-physical-device-scale *(stream scale unit)* &body *body* *Macro*

Scales the output produced by *body* on *stream* so that it really is the size specified on the output device used. *scale* is a number and *unit* is one of **:inch**, **:centimeter**, or **:mica**.

Note that a scale factor is applied that assumes that the current scale is 1. Therefore, you can still zoom a picture drawn using this by surrounding it with a **graphics:with-graphics-scale**.

```
(graphics:with-room-for-graphics ()
 (graphics:with-physical-device-scale (t 1 :centimeter)
  (graphics:draw-circle 1 1/2 1/2)))
```

graphics:with-room-for-graphics (&optional *stream height*) &body *body* *Function*

Binds the local environment to establish a Cartesian coordinate system for doing graphics output. The origin <0, 0> of the local coordinate system is in the lower left corner of the area created. After graphic output is completed, the cursor is positioned past (immediately below) this origin. The bottom of the vertical block allocated is at this location -- that is, just below point <0, 0> -- not necessarily at the bottom of the output done. If your drawing extends in the negative *y* direction, then you should use **graphics:with-graphics-translation** to position it within the allocated space. This works for the screen, the lgp2, and the lgp3, even though their coordinate systems are really not the same.

stream The output stream. If this argument is **t** or not supplied, it defaults to ***standard-output***.

height A number specifying the vertical space, in pixels, created for graphics output. If this argument is not supplied, the height is computed from the height of the output to be done.

Example:

```
;;; Try evaluating the following forms.  Note the orientation
;;; of the lines drawn and the position of the cursor when done.

(multiple-value-bind (x1 y1 x2 y2)
    (send *standard-output* :visible-cursorpos-limits)
  (ignore x1 x2 y2)
  (graphics:draw-line 0 (+ 0 y1) 200 (+ 200 y1)))

(graphics:with-room-for-graphics (t 200)
  (graphics:draw-line 0 0 200 200))

(graphics:with-room-for-graphics (t 200)
  (graphics:draw-line 0 0 200 -200))

(graphics:with-room-for-graphics (t 200)
  (graphics:with-graphics-translation (t 0 200)
     (graphics:draw-line 0 0 200 -200)))
```

For an overview of **graphics:with-room-for-graphics** and related functions: See the section "Coordinate System Facilities".

graphics:write-encoded-graphics-as-characters *array stream* *Function*

Translates each element of *array* into a character, possibly preceded by a code character, and writes the result to *stream*. See the section "Other Advanced Facilities for Graphic Output".

array An array with elements of type **(unsigned-byte 8)** produced by the function **graphics:binary-encode-graphics-to-array**, which contains the encoded version of a graphics function.

stream Any output stream.

Standard Keyword Options to Drawing Functions

:stream *Option*

Specifies the output stream for a drawing function; the default is ***standard-output***.

:alu *Option*

Specifies the drawing mode for a drawing function. Possible values for this option are:

:draw Pixels specified by the drawing function are turned on, regardless of whether some of the pixels were already on. This is the default drawing mode.

:erase Pixels specified by the drawing function are turned off, regardless of whether some of the pixels were already off.

:flip Pixels specified by the drawing function are turned on if they were previously off, and off if they were previously on.

Additionally, numeric and color-extended alu operations are valid values for this option. Whether "on" means white or black depends on the whether or not the display window is in inverse video mode: if inverse video is not in effect, on means white.

:stipple *Option*

Specifies a two-dimensional one-bit array. In normal, **:alu :draw**, mode, a 1 in the array specifies that the corresponding pixel is on; 0, off. The origin of the array is aligned with the coordinate origin of the output display.

Predefined stipple patterns are in **graphics:*stipple-arrays***.

You can create your own stipple patterns using the Stipple Editor which, if it is loaded, is invoked by pressing `SELECT |`. You can also generate a stipple pattern by using the **tv:with-output-to-bitmap** macro.

:tile *Option*

Specifies a two-dimensional array of *n*-bit values. The *n* bits of each entry in the array specify the color of the corresponding pixel. The origin of the array is aligned with the coordinate origin of the output display. You can also generate a tiling pattern by using the **tv:with-output-to-bitmap** macro.

:color *Option*

Specifies a color to be used for output on a device that supports color. On devices that do not support color, a gray-level pattern appropriate to the intensity of the specified color is displayed instead. The possible choices are: **:black**, **:red**, **:green**, **:blue**, **:cyan**, **:yellow**, **:magenta**, and **:white**.

:filled *Option*

Boolean option specifying whether all pixels within the figure created by a drawing function are turned on, or only the outline pixels; the default is **t** (filled).

:pattern *Option*

Specifies a pattern to be drawn within the figure created by a drawing function. **:pattern** is the most general control of the "looks" of thie inside of a filled shape. The default is **nil**. The **:pattern** drawing option can be a(n)

- **stipple** This is identical to **:stipple** <*value*>.
- **color** This is identical to **:color** <*value*>.
- **instance** The case of interest here.

Several functions are provided for making instances to pass as the pattern. See the function **graphics:make-simple-pattern**. See the function **graphics:make-two-color-stipple**. See the function **graphics:make-contrasting-pattern**. See the function **graphics:make-device-conditional-pattern**. Additionally, there are three layered protocols for implementing one's own instances to be passed as **:pattern**. See the section "Texturing".

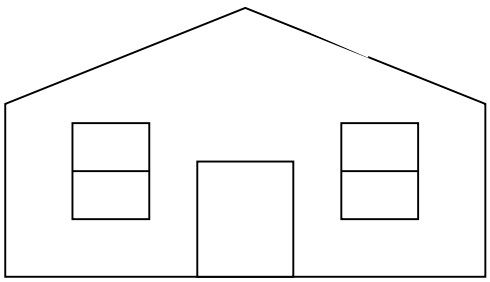
```
(defun pattern-example1 ()
  (graphics:with-room-for-graphics (t 200)
    (let ((bitmap (tv:make-binary-gray 8 8
                    '(#b00000000    ; The picture of
                      #b00001000    ; what you want
                      #b00111000    ; the bit pattern
                      #b00001000    ; displayed to
                      #b00001000    ; look like, in
                      #b00001000    ; this case the
                      #b00001000    ; number 1.
                      #b00111110))))
          (graphics:draw-circle 200 100 50
                                :pattern bitmap))))

(defun pattern-example2 ()
  (let ((bitmap (tv:with-output-to-bitmap (t)
                  (graphics:draw-string
                    "symbolics" 0 0
                    :character-style
                    '(:swiss :bold-italic :large)))))
    (graphics:with-room-for-graphics (t 250)
      (graphics:draw-triangle 100 25 400 25 250 250
                              :pattern bitmap))))
(pattern-example1)
(pattern-example2)
```

:gray-level *Option*

Specifies the black-to-white level of the graphic as a ratio or decimal fraction between 0 and 1; the default value is 1. On 1-bit devices, gray levels are simulated by stippling.

Example:

```
(defun gray-level-example ()
  (graphics:with-room-for-graphics (t 200)
    (graphics:draw-polygon
      '(100 10 350 10 350 100 225 150 100 100 100 10)
      :gray-level 1/4
      :convex-p t)
    (graphics:draw-polygon
      '(100 10 350 10 350 100 225 150 100 100 100 10)
      :filled nil
      :convex-p t)
    (graphics:draw-polygon
      '(259 137 259 165 289 165 289 124 259 137)
      :gray-level 3/4
      :convex-p t)
    (graphics:draw-rectangle 200 70 250 10
                             :gray-level 1/2
                             :opaque t)
    (graphics:draw-rectangle 200 70 250 10
                             :filled nil)
    (graphics:draw-rectangle 135 90 175 40
                             :gray-level 1/20
                             :opaque t)
    (graphics:draw-rectangle 135 90 175 40
                             :filled nil)
    (graphics:draw-line 135 65 175 65)
    (graphics:draw-rectangle 275 90 315 40
                             :gray-level 1/20
                             :opaque t)
    (graphics:draw-rectangle 275 90 315 40
                             :filled nil)
    (graphics:draw-line 275 65 315 65)))

(gray-level-example)
```



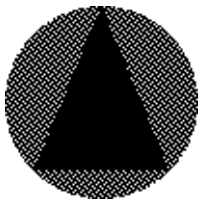
Boolean option specifying whether pixels in the source pattern of a drawing function are cleared (before the graphic is output) or left alone; the default is **t**. **:opaque t** means draw pixels that are off in the background color. The result of **:opaque t** is not influenced by the **:alu** supplied for the pixels that are on. If you draw something over an existing figure using **:opaque t :alu :flip**, the pixels that were originally on are inverted, and pixels that were originally off are cleared (set to the background color).

To see the effect of this option, try calling the following function with **t** and then **nil**.

```
(defun opaque-example (t-or-nil)
  (graphics:with-room-for-graphics (t 200)
    (graphics:draw-circle 200 100 50
                          :pattern tv:75%-gray)
    (graphics:draw-triangle 165 65 235 65 200 150
                            :pattern tv:25%-gray
                            :opaque t-or-nil)))
```

```
(opaque-example t)
```

```
(opaque-example nil)
```



:thickness *Option*

Specifies the thickness, in pixels, of the line or lines drawn by a drawing function. The default is device-dependent: for the screen, it is 0, which specifies the minimum thickness for that device; for the lgp2 and lgp3, it is 1. The thinnest possible thickness is 0, which also specifies that the line is to be positioned roughly. For accurate positioning, specify a thickness of 1.

This option is not operable if the function draws a filled (`:filled t`) figure.

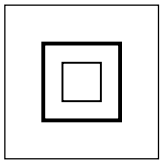
```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-triangle 0 0 50 0 25 25 :filled nil :thickness 2)
 (graphics:draw-triangle 100 0 150 0 125 25 :filled nil :thickness 4)
 (graphics:draw-triangle 200 0 250 0 225 25 :filled nil :thickness 8))
```

:scale-thickness *Option*

Boolean option specifying whether to scale thickness, as well as other linear dimensions, when a scaling factor is specified (via the **:scale** keyword to a drawing function: See the option **:scale**.). The default is **nil**.

This option is not operable if the function draws a filled (`:filled t`) figure.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:with-graphics-translation (t 100 50)
  (graphics:draw-rectangle -10 -10 10 10 :filled nil :thickness 1)
  (graphics:draw-rectangle -10 -10 10 10 :filled nil :thickness 1 :scale 2)
  (graphics:draw-rectangle -10 -10 10 10 :filled nil :thickness 1 :scale 4
                           :scale-thickness nil)))
```



:dashed *Option*

Boolean option specifying whether lines are drawn as a series of dashes by a drawing function; the default is **nil**.

This option is not operable if the function draws a filled (`:filled t`) figure.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-rectangle 0 0 100 100 :filled nil :dashed t))
```

:dash-pattern *Option*

Specifies a sequence, usually a vector, controlling the dash pattern of a drawing function. If no pattern is specified, the default dashes are 10 pixels long and separated by spaces of 10 pixels. The vector must contain an even number of elements or you will get an error.

The following example draws a line as a series of dashes, alternating in length between 16 and 8 pixels, with intervening spaces of 4 pixels. Note that these lengths result from applying a scaling factor of 4, implemented by the **:scale** and **:scale-dashes** keywords.

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-line 25 25 100 25
                      :dashed t :scale 4
                      :scale-thickness nil
                      :scale-dashes t
                      :dash-pattern #(4 1 2 1)))
```


This option is not operable if the **:dashed** option is **nil**.

:scale-dashes *Option*

Boolean option specifying whether to scale dashes when the **:dashed** option to a drawing function is **t** and a scaling factor is specified (via the **:scale** keyword: See the option **:scale**.). The default is **nil**.

For an example: See the option **:dash-pattern**.

This option is not operable if the **:dashed** option is **nil**.

:initial-dash-phase *Option*

Specifies the offset, in pixels, of the start of the first dash from the starting point of the line; the default value is `0`.

This option is not operable if the **:dashed** option to the drawing function is **nil**.

:draw-partial-dashes *Option*

Boolean option specifying whether a partial dash is drawn at the end of a dashed line so that it reaches its specified end-point. The default is **t**: dashes are drawn with the specified numbers of pixels on and off until the endpoint is reached, at which point drawing stops wherever in the pattern you happen to be.

If you specify **nil** for this option, the drawing routine will adjust the spacing of the dashes so that the lines ends on a "dash." In the simple case — that is, with only a single pair of numbers in the dash pattern — a dash is a solid line of (on) pixels, so both ends of such a line are drawn. For example, try these:

```
(graphics:with-room-for-graphics (t 10)
  (graphics:draw-line 0 3 200 3 :dashed t :dash-pattern #(20 15)
                              :draw-partial-dashes t)
  (graphics:draw-line 0 -3 200 -3 :dashed t
                              :dash-pattern #(20 15) :draw-partial-dashes nil)
  (graphics:draw-line 200 -3 200 3))



(graphics:with-room-for-graphics (t 250)
  (let ((zoom 5))
    (dolist (partial '(t nil))
      (graphics:with-graphics-translation (t 0 (if partial (* 25 zoom) 0))
        (dotimes (i 20)
          (let ((y (* (- 19 i) zoom)))
            (graphics:draw-line 0 y (* i 4 zoom) y
                                :dashed T
                                :dash-pattern #(20 15)
                                :draw-partial-dashes partial)
            (graphics:draw-line 0 (- y 1) (* i 4 zoom) (- y 1))))))))
```

For more complicated dash patterns, a dash is considered to be a solid line somewhere in the pattern: you will have to experiment to determine the exact result of using the option.

This option is not operable if the **:dashed** option to the drawing function is **nil**.

Some hardcopy devices, most notably PostScript printers, cannot adjust the spacing of the dashes; that is, they will draw partial dashes even if you specify **:draw-partial-dashes nil**.

:line-end-shape *Option*

Specifies the shape for the ends of lines drawn by a drawing function, one of **:butt**, **:square**, **:round**, or **:no-end-point**; the default is **:butt**.

To see the differences among end shapes, try evaluating the following:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-line 200 40 200 140
                              :line-end-shape :butt
                              :thickness 20)
  (graphics:draw-line 240 40 240 140
                              :line-end-shape :no-end-point
                              :thickness 20)
  (graphics:draw-line 280 40 280 140
                              :line-end-shape :square
                              :thickness 20)
  (graphics:draw-line 320 40 320 140
                              :line-end-shape :round
                              :thickness 20)
  (graphics:draw-line 190 30 330 30)
  (graphics:draw-line 190 150 330 150))
```

The vertical lines on the left have **:butt**ed ends; they do not extend beyond the y-coordinates (`40` and `140`) given for the line. The lines on the right have **:square**d and **:round**ed ends. These lines extend, because of the **:thickness** parameter (`20`), 10 pixels above and below the top and bottom line coordinates.

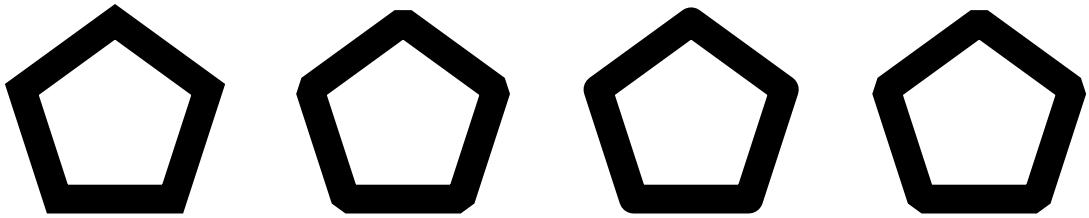
This option is not operable if the function draws a filled (`:filled t`) figure.

:line-joint-shape *Option*

Specifies the shape of joints between line segments of closed, unfilled figures, when the **:thickness** option to a drawing function is greater than `1`. The possible shapes are **:miter**, **:bevel**, **:round**, and **:none**; the default is **:miter**.

The following example draws four pentagons illustrating the differences among, from left to right, the **:miter**, **:bevel**, **:round**, and **:none** joint shapes:

```
(graphics:with-room-for-graphics (t 200)
  (graphics:draw-regular-polygon 40 40 100 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :miter)
  (graphics:draw-regular-polygon 190 40 250 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :bevel)
  (graphics:draw-regular-polygon 340 40 400 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :round)
  (graphics:draw-regular-polygon 490 40 550 40 5
                                 :filled nil
                                 :thickness 15
                                 :line-joint-shape :none))
```



This option is not operable if the function draws a filled (`:filled t`) figure.

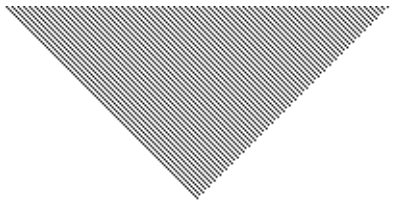
Some hardcopy devices may not support all line joint shapes. Notably, PostScript printers treat **:none** as **:bevel**.

:mask *Option*

Specifies a bitmap of points affected by the drawing operation; the default is **nil**, that is, no bitmap.

Use this option to mask out portions of the output graphic that you do not want displayed. For example, the following form outputs a rectangle, but does so through a triangular mask, so that only a triangle gets displayed:

```
(graphics:with-room-for-graphics (t 200)
  (multiple-value-bind (bitmap x y)
       (graphics:with-output-to-bitmap ()
         (graphics:draw-triangle 100 50 200 150 300 50))
    (graphics:draw-rectangle 0 200 400 0
                             :gray-level .33
                             :mask bitmap :mask-x x :mask-y (+ 100 y))))
```



:mask-x *Option*

Specifies the *x*-coordinate, with respect to the local graphics origin, of the lower lefthand corner of the position of the rectangular bitmap region specified by the **:mask** option.

:mask-y *Option*

Specifies the *y*-coordinate, with respect to the local graphics origin, of the lower lefthand corner of the position of the rectangular bitmap region specified by the **:mask** option.

:return-presentation *Option*

Boolean option specifying whether a drawing function should return the newly created graphic as a presentation object; the default is **nil**.

Use this option when you wish to manipulate the output of a single drawing function as a presentation. If you want to manipulate the collective output of a series of drawing functions as a single presentation, use the **graphics:with-output-as-graphics-presentation** macro instead.

Some facilities are provided for handling graphic presentations, in particular, **graphics:erase-graphics-presentation** and **graphics:replacing-graphics-presentation**. The following example uses the former in conjunction with the **:return-presentation** keyword:

```
(graphics:with-room-for-graphics (t 100)
 (graphics:with-graphics-scale (t 50)
  (let ((presentations
         (list
          (graphics:draw-regular-polygon .75 .5 1.25 .5 4
            :gray-level .25 :return-presentation t)
          (graphics:draw-regular-polygon .25 0 1.25 0 3
            :gray-level .75 :return-presentation t)
          (graphics:draw-ellipse 1.5 .5 .4 .4
            :gray-level .5 :return-presentation t))))
    (sleep 2)
    (graphics:erase-graphics-presentation (first presentations)))))
```

Coordinate-System Keyword Options

:rotation *Option*

Specifies the rotation of the graphic in plus or minus radians; the default is `0`. The axis of rotation is the local origin (0, 0).

In the following example, the origin is first established at the lower, left corner of the graphic display area created by **graphics:with-room-for-graphics**, then translated 300 pixels to the right and 15 pixels up by the **graphics:with-graphics-translation** macro. In this coordinate system, the arrow is rotated counter-clockwise about the origin at an offset of 10 pixels.

```
(defun rotating-arrow ()
  (graphics:with-room-for-graphics (t 225)
    (graphics:with-graphics-translation (t 300 15)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5)
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .125))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .25))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .375))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .5))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .625))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .75))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation (* pi .875))
      (sleep .2)
      (graphics:draw-arrow 10 0 200 0
                           :thickness 5
                           :rotation pi))))

(rotating-arrow)
```

:scale *Option*

Specifies a number applied as a scaling factor to the x and y parameters of a drawing function; the default is 1.

:scale does not affect the line thickness parameter. To do so, use the **:scale-thickness** keyword.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-circle 1 1 1 :scale 20))
```

:scale-x *Option*

Specifies a number applied as a scaling factor to the x parameters of a drawing function; the default is 1.

```
(graphics:with-room-for-graphics (t 100)
```

```
 (graphics:draw-circle 50 50 20 :scale-x 4))
```

:scale-y *Option*

Specifies a number applied as a scaling factor to the y parameters of a drawing function; the default is 1.

```
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-circle 100 100 50 :scale-y 1/2))
```

:transform *Option*

Specifies a list of six elements in the graphics transformation matrix applied to the local coordinate system used for a drawing function. The element order is *a*, *b*, *c*, *d*, *x*, and *y* (see below). The default is **nil**, resulting in no transformation.

Arbitrary transformation of coordinates is effected by multiplication of coordinate vectors by a transformation matrix. For coordinates in two-dimensional space, a 3 x 3 transformation matrix is used,

```
a   b   0

c   d   0

x   y   1
```

of which the elements in the third column are constant. Thus, six elements, effectively control the transformation as follows:

- **Scaling** in the x and y dimensions is controlled by elements *a* and *d*, respectively. Values of 1 for these elements result in no scaling.

- **Translation** in the x and y dimensions is controlled by elements x and y, respectively. Values of `0` for these elements result in no translation.

- **Rotation** about the origin is controlled by elements a, b, c, and d. Counter-clockwise rotation by an angle α is effected by $a = \cos\alpha$, $b = \sin\alpha$, $c = -\sin\alpha$, and $d = \cos\alpha$. A value of `0` for b and c results in no rotation ($\alpha = 0$).

Example:

```
;;; This example scales the size of the rectangle by a
;;; factor of 4 and translates its position by 100 in
;;; the x direction and 50 in the y direction.

(graphics:with-room-for-graphics (t 200)
  (graphics:draw-rectangle 0 20 20 0
                           :transform '(4 0 0 4 100 50)))
```

:translation *Option*

Specifies x and y offsets relative to the local origin (0, 0) used for a drawing function. The offsets are specified as a list, $(x\ y)$. The offset units depend on the output device; if it is the terminal screen, the units are pixels.

Using this option has the effect of moving the local origin to the new position specified by x and y. The default is **nil**, no translation.

```
;;; Contrast this with giving 50 50 as the center, which would have caused it to
;;; be rotated.
(graphics:with-room-for-graphics (t 100)
 (graphics:draw-ellipse 0 0 40 20 :translation '(50 50) :rotation (/ pi 4)))
```