LispWorks®

CAPI User's Guide

Edition 3.1

Harlequin Limited

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
England

Contents

Chapter 1
Introduction to the Common API

1.1 Introduction

Differences between Common Lisp implementations are usually minor, and are no longer a problem when porting source code from one implementation to another. However, this is not yet the case for user interfaces, as no real standard in this area has emerged. Moreover, most of these de-facto standards are provided only for X Windows, thus alienating other popular window systems such as Microsoft Windows or the Apple Macintosh Window System.

As there are currently implementations of LispWorks for X Windows and for MS Windows 3, a scheme was needed to allow the creation of portable window applications. A possible solution was to re-implement the complete toolkit running under X Windows (see Toolkit, CLX, CLUE) for MS Windows. Following this path leads to an unsatisfactory implementation which is usually too distanced from the native widget set, introducing some unnecessary overheads and yet another Look and Feel. (The Look and Feel is constant amongst LispWorks applications, but this itself may be foreign to the usual Look and Feel of the target platform's applications).

Harlequin has adopted a completely different strategy, which favours the use of native widget sets at the expense of the consistency of LispWorks applications' look and feel. In the long run this solution offers undoubted advantages — for instance, it will enable the delivery of a UNIX developed application onto a PC platform by means of simple cross-compilation and use of LispWorks delivery technology. The Common Application Programmer's Interface (Common API or CAPI) is the LispWorks solution which implements this strategy.

1.2 Basic Entities in the Common API

A Common API application consists of a hierarchy of CAPI objects which contain application data and state information. CAPI objects are made using `make-instance`, but although they are standard CLOS objects, the slots should only be accessed by the documented accessor functions and not by using `slot-value` and related macros. This enables internal state to remain consistent, and the graphical representation of the CAPI object to be updated. Thus there is generally no need to explicitly ask a CAPI object to redraw itself. T o avoid repeated redrawing of the same object, several changes to one or a group of objects can be surrounded by `with-atomic-redisplay` which delays the update until the end of its body .

CAPI objects can be divided into several categories. Subclasses of `simple-pane` will generally have a corresponding window system representation which manages user interaction and re-display work. Each simple-pane has a set of callback functions which are triggered by user changes to its representation.

Subclasses of `pinboard-object` are managed entirely by the CAPI and have no window system representation. They are drawn on a subclass of `pinboard-layout` which handles user interaction and re-display of its `pinboard-object` children. The children of a pinboard-object can also be simple-panes.

Subclasses of `layout` provide no user interaction and are used to group simple-panes and pinboard-objects according to some layout algorithm.

Subclasses of `interface` are used in two ways: as top-level windows or dialog boxes, and as components with a hierarchy .

Subclasses of `item` are used to hold data and callbacks for simple-panes which provide selection from several options (e.g., `list-panel`).

1.3 Interfaces

An interface groups together the panes, layouts and menus which are required by an application. Many of the callbacks associated with simple-panes are passed an interface to identify the application which contains the pane. Interfaces can be defined with the macro `define-interface`, which automatically generates all the code required to make the panes, layouts and menus from their descriptions. Interfaces can be displayed by calling `display` or `display-dialog`, depending upon the user interaction desired.

1.4 Loading the CAPI

The CAPI can be loaded with the form

```
(lw::initialize-capi)
```

1.5 CAPI Examples

There are various code-examples provided with the CAPI. It is intended that this documentation be read in conjunction with these examples. They can be found in the `/usr/lib/lispworks/examples/capi` directory .

Chapter 2
CAPI Classes

2.1 Introduction

This chapter describes the classes provided by the CAPI, and the functions which can be used with them.

2.2 The Class Description Entry

All class descriptions entries in this document are organised according to the template below . If text appears in diamond brackets, this means that there will be, placed in that position in the description, an appropriate piece of text, in the appropriate typeface. For example, in the description of the class `window-thing` the field <**class name**> will contain the text **window-thing**, in Helvetica Bold type. In the field `<initarg>` there might be the text `:x-position`, in Courier type

Any other text in this template will appear literally in a class description.

Since a class may contain no slots, one slot or many slots, the **Slots** area in an entry may be of arbitrary length. This is also true for the **Related Functions** area: there may be none, one, or many functions listed in the area.

<class name> *Class*

Description

<description of class>

Class Precedence List

`<class> . . . <class>`

Slots

`<initarg> <initform> {<reader> | {<reader> (setf ) }`

`<slot type>`

<description of this slot>

Related Functions

`<name>` *<type >*
<*arguments* >

<description>

The first line of the **Slots** area in the template contains the text

`{<reader> | <reader> (setf ) }`

which means that there could either be

- nothing in that position
- a `<reader>` in that position
- a `<reader>` followed by the text (setf) in that position.

The latter means that the writer associated with that reader has the name `(setf reader).`

The `<slot type>` field contains the type of the data in the slot. Note that a `function` may be an fbound symbol.

2.3 CAPI Class Entries

The classes the CAPI provides, and the functions which can be used with those classes, are described below .

capi-object *Class*

Description

The superclass of all CAPI objects

Class Precedence List

`standard-object`

Slots

`:plist nil capi-object-plist` (setf)

`t`

A plist which can be used to store miscellaneous data. The keys should not be symbols in the CAPI package.

element *Class*

Description

The base class for objects within interfaces.

Class Precedence List

`capi-object`

Slots

`:interface nil element-interface` (setf)

`interface`

The innermost interface containing the element.

simple-pane *Class*

Description

The base class for panes.

Class Precedence List

`element capi-object`

Slots

`:font nil simple-pane-font` (setf)

`generic-font`

The font to be used in the pane.

`:background nil simple-pane-background` (setf)
`(or keyword color-alias color-spec)`

The background color of the simple-pane.

`:foreground nil simple-pane-foreground` (setf)
`(or keyword color-alias color-spec)`

The foreground color of the simple-pane.

`:horizontal-scroll nil simple-pane-horizontal-scroll`
`boolean`

Non- `nil` means that the pane will have a horizontal scroll bar .

`:vertical-scroll nil simple-pane-vertical-scroll`
`boolean`

Non- `nil` means that the pane will have a vertical scroll bar .

`:visible-border :default simple-pane-visible-border`
`(member nil t :default)`

Controls whether the pane has a visible border .

- `nil` — no border
- `t` — border
- `:default` — default for current window system

Related Functions

`top-level-interface` *Function*
pane

Returns the top-level interface containing the *pane* .

`top-level-interface-p` *Function*
pane

Returns true if the *pane* is a top-level interface.

`simple-pane-visible-size` *Generic Function*
pane

Returns two values, the width and height of the visible part of the *pane* after taking account of any scroll bars.

titled-pane *Class*

Description

A pane with an option title and message. The title can be above or to the left. The message is always below the pane.

Class Precedence List

`simple-pane element capi-object`

Slots

`:title titled-pane-title` (setf)
`string`

The title string.

`:title-position`
`(member :left :top)`

The position of the title relative to the pane.

`:title-args`
`list`

An alist of attributes for the title. Allowable keys are `:font`, `:background` and `:foreground`.

`:message titled-pane-message` (setf)
`string`

The message string.

`:message-args`
`list`

An alist of attributes for the title. Allowable keys are `:font`, `:background` and `:foreground`.

callbacks *Class*

Description

Some slots which hold various callbacks used by `collection` and its subclasses. The callback functions can be one of the following:

`:redisplay-menu-bar`

`redisplay-menu-bar` is called with the top-level interface. This is useful for the list-panel callbacks when nothing need be done except to enable or disable some menu items.

function or *symbol*

Call the *function* with arguments as specified by `:callback-type`.

list

Apply the head of the list to the tail.

Class Precedence List

`capi-object`

Slots

`:selection-callback nil callbacks-selection-callback` (setf)

`(or list function)`

Called when a single selection is made.

`:extend-callback nil callbacks-extend-callback` (setf)

`(or list function)`

Called when another item becomes selected.

`:retract-callback nil callbacks-retract-callback` (setf)

`(or list function)`

Called when an items becomes de-selected.

`:action-callback nil callbacks-action-callback` (setf)

`(or list function)`

Called when the action gesture (e.g. double-click) is made. The appropriate selection callback will have been called previously .

`:callback-type nil callbacks-callback-type` (setf)

```
(member :data :data-interface :interface-data :item
        :item-interface :interface-item :interface :full nil
        :none)
```

Specifies the format of the arguments passed to the callback function. This is useful for avoiding having to use many intermediate functions to translate the callback into calls to functions which exist in the application anyway .

Value	**Arguments**
`:data`	`(<item-data>)`
`:data-interface`	`(<item-data> <interface>)`
`:interface-data`	`(<interface> <item-data>)`
`:item`	`(<item>)`
`:item-interface`	`(<item> <interface>)`
`:interface-item`	`(<interface> <item>)`
`:interface`	`(<interface>)`
`:full`	`(<item-data> <item> <interface>)`
`nil or :none`	`()`

`<item-data>` is replaced by the item-data if the item is of type `capi:item`, otherwise by the item itself.

`<item>` means the item itself.

`<interface>` is replaced by the interface of the element.

Related Functions

`attach-interface-for-callback` *Function*
element interface

Sets the interface used in callbacks for element. Normally this is the nearest surrounding interface.

---

item *Class*

Description

A component of a choice or menu which has its own callbacks and data.

Class Precedence List

`callbacks capi-object`

Slots

`:data nil item-data` (setf)
`t`

The data contained in the item. When an item is placed in a choice, the choice is re-displayed when the item data changes.

`:selected nil item-selected` (setf)

`boolean`

The selection state of the item. When an item is placed in a choice, its selection slot is updated when the choice-selection changes, and the choice-selection is updated when `(setf item-selected)` is called.

`:print-function nil item-print-function` (setf)

`(function (t) string)`

The print-function is called to generate a string from the item data. If the item has no print-function, the print-function (if any) of the collection which contains the item is used instead. Otherwise `princ-to-string` is used.

Related Functions

`itemp` *Generic Function*

object

Returns true if and only if `(typep object 'item)` is true.

---

title-pane *Class*

Description

Single line of static text.

Class Precedence List

`simple-pane element capi-object`

Slots

`:text "" title-pane-text` (setf)

`string`

The text.

---

display-pane *Class*

Description

Multiple lines of static text.

Class Precedence List

`titled-pane simple-pane element capi-object`

Slots

`:text "" display-pane-text` (setf)

`(or string list)`

Either the text as a string or a list of strings one for each line.

---

text-input-pane *Class*

Description

Single line of editable text.

Class Precedence List

`titled-pane simple-pane element capi-object`

Slots

`:text "" text-input-pane-text` (setf)

`string`

The text in the editable field.

`:caret-position (length text) text-input-pane-caret-position` (setf)

`integer`

The position of the caret in the text (0 at the left).

`:enabled t text-input-pane-enabled` (setf)

`boolean`

Controls whether the text-input-pane responds to keyboard and mouse input.

`:completion-function nil text-input-pane-completion-function` (setf)

`(or null (function (text-input-pane string) string))`

A function called when the user makes the 'complete' gesture.

`:callback nil text-input-pane-callback` (setf)

`(or null function)`

A function called when the user presses `ENTER` or `RETURN`.

`:callback-type :data-interface`

see `:change-callback nil text-input-pane-change-callback` (setf)

`(or null (function (text-input-pane &key (new-value string)`
`(new-caret-pos integer)) t))`

A function called when a change is made by the user . When the caret is moved, `:new-caret-pos` will be passed. Otherwise both `:new-value` and `:new-caret-pos` will be passed.

`:before-change-callback nil text-input-pane-before-change-callback` (setf)

`(or null (function (text-input-pane &key (new-value string)`
`(new-caret-pos integer)) boolean))`

A function called to validate a change made by the user . When the caret is moved, `:new-caret-pos` will be passed. Otherwise both `:new-value` and `:new-caret-pos` will be passed. The change will take place only if the function returns non- `nil`.

---

collection *Class*

Description

An abstract class which provides a grouping of similar items and a means of accessing them.

Class Precedence List

`callbacks capi-object`

Slots

`:items nil collection-items` (setf)

`t`

Some representation of the items. When the `items-get-function` is `svref`, this value is coerced to a be `simple-vector`. Whatever representation is chosen, there must be a unique way of extracting an item given its index using the `items-get-function`.

`:items-count-function 'length collection-items-count-function`

`(function (t) integer)`

A function which returns 1 plus the maximum index for the items.

`:items-get-function 'svref collection-items-get-function`

`(function (t integer) t)`

A function which, given the items and an index, should return the indexed item.

`:item-map-function 'map-items-vector collection-items-map-function`

`(function (collection (function (t) t) &optional boolean) t)`

A function which calls its second argument on each of the items in `vector`. If the third argument is non- `nil`, the results of the function calls should be returned as a list. The default is the following function:

```
(defun map-items-vector (vector function collect-
results-p)
   (loop for item across vector
         for result = (funcall function item)
         when collect-results-p collect result))
```

`:print-function 'princ-to-string collection-print-function` (setf)

`(function (t) string)`

A function which, given an item, should return its textual representation. The `print-function` may be called once, or more than once, for each item.

`:test-function 'eq collection-test-function` (setf)

`(function (t t) boolean)`

A function which returns non- `nil` if two items are equal.

Related Functions

`count-collection-items` *Generic Function*
collection `&optional` *i-rep*

Return the number of items in the *collection* by calling the `items-count-function`. If *i-rep* is non- `nil`, the value is used in place of the item from the *collection* .

`get-collection-item` *Generic Function*
collection index

Return the *index* 'th item from the *collection* by calling the `items-get-function` of the collection.

`map-collection-items` *Generic Function*
collection function `&optional` *collect-results-p*

Calls the function on each item in the *collection* by calling the `items-map-function`. If *collect-results-p* is non- `nil`, the results of each call are returned as a list.

`search-for-item` *Generic Function*
collection item

Returns the index of *item* in the *collection*, using the `collection-test-function` to determine equality. Returns `nil` if no match is found.

`find-string-in-collection` *Generic Function*
collection string `&optional` *set*

Returns the next item whose printed representation matches *string*. If *set* is non- `nil`, the choice-selection is set to this item. The search is started from the previous search point. If the choice-selection is set, the next search will start from the first selected item.

`collection-search` *Generic Function*
collection `&optional` *set*

Prompt the user for a string and call *find-string-in-collection* with this string.

---

choice *Class*

Description

An abstract subclass of `collection`, which provides selected item(s) and a mode of interaction when the selection is changed.

Class Precedence List

`collection callbacks capi-object`

Slots

`:interaction :single-selection choice-interaction`

```
(member :single-selection :multiple-selection
        :extended-selection)
```

Controls how the selection changes when one of the items becomes selected or de-selected (usually by the user clicking with the mouse).

- `:single-selection` means that all other items become de-selected when a new selection if made. The selection-callback will be called when a selection is made. The retract-callback will be called when an item is de-selected (if the window-system has a gesture for this).

- `:multiple-selection` means that items can be toggled independently of each other . The extend-callback will be called when a item becomes selected. The retract-callback will be called when an item becomes de-selected.

- `:extended-selection` is a combination of `:single-selection` and `:multiple-selection`, in that there are several gestures, one of which selects a single item while others extend or reduce the current selection in some way . The selection-callback will be called when a single item becomes selected. The extend-callback and retract-callback will be called as for `:multiple-selection`.

`:selection nil choice-selection` (setf)
`(or integer list)`

The indices of the currently-selected items. For `:single-selection`, this is the index itself. For other types of selection, this can be a list of indices.

Related Functions

`choice-selected-items` *Generic Function*
choice

Return the currently selected items in the choice.

`(setf choice-selected-items)` *Generic Function*
items choice

Sets the currently selected *items* in the *choice*. The collection-test function is used to match members of the list *items* with the items of the choice.

---

list-panel *Class*

Description

A subclass of `choice`, which is displayed as text, one item per line.

Class Precedence List

`choice collection titled-pane simple-pane element callbacks capi-object`

button *Class*

Description

An abstract class which provides button functionality for `check-button` and `push-button`.

Class Precedence List

`item callbacks capi-object`

Slots

`:enabled t button-enabled` (setf)

`boolean`

Non-`nil` when the button is enabled. A disabled button does not respond to user input.

check-button *Class*

Description

A button which can be toggled on or off independently of other buttons.

Class Precedence List

`button simple-pane element item callbacks capi-object`

push-button *Class*

Description

A button which can be pressed.

Class Precedence List

`button simple-pane element item callbacks capi-object`

button-group *Class*

Description

An abstract class which positions buttons in a layout.

Class Precedence List

`titled-pane simple-pane element capi-object`

Slots

`:layout-class 'row-layout button-group-layout-class`
`(or class symbol)`

The class of the layout which contains the buttons.

`:layout-args nil button-group-layout-args`
`list`

The extra arguments given to the layout.

`:callbacks nil button-group-callbacks`
`list`

The callbacks associated with each item, the first being for item 0, etc.

Related Functions

`set-button-panel-enabled-items` *Generic Function*
button-group `&key` *enable disable set* `(test 'equal)`

Set the enabled state of the items in the button-group. If *set* is true, then *enable* is ignored and all items are enabled except those in the *disable* list. If *set* is `nil`, *disable* is ignored and all items are disabled except those in the *enable* list. If *set* not given, the items in the *enable* list are enabled and the items in the *disable* list are disabled. If an item is in both lists, it is enabled.

A button is in a list when the data of the button matches one of the items in the list. A match is defined as a non-`nil` return-value from the *test* function. The default test function is `equal`.

---

button-panel *Class*

Description

A subclass of `button-group` and `choice` which displays its items as check-buttons or radio-buttons, according to the interaction slot.

Class Precedence List

`button-group titled-pane simple-pane element choice collection callbacks capi-object`

push-button-panel *Class*

Description

A subclass of `button-group` and `collection` which displays its items as push-buttons.

Class Precedence List

`button-group titled-pane simple-pane element collection callbacks capi-object`

imenu-object *Class*

Description

The superclass of all menu objects. This class provides functionality for menu callbacks and enabling.

Class Precedence List

`callbacks capi-object`

Slots

`:enabled-function nil`

`(function (interface) boolean)`

The enabled-function is called with the menu-interface, after the popup-callback. It should return non-`nil` if the menu object is enabled. This initarg should not be used with `enabled-slot`.

`:enabled-slot nil`

`symbol`

This initarg is a shorthand for

```
:enabled-function #'(lambda (interface) (slot-value
 interface enabled-slot))
```

`:callback nil`

`function`

The callback initarg is placed in the selection-callback, extend-callback and retract-callback slots, unless these are supplied explicitly.

`:callback-data-function nil`

`(function () t)`

A function which is called to supply the data argument for the callbacks. If this is not given, the contents of the data slot of the selected item is used.

---

titled-menu-object *Class*

Description

A subclass of `menu-object`, with a title. `menu` and `menu-item` are subclasses of this class.

Class Precedence List

`menu-object callbacks capi-object`

Slots

`:title nil menu-title` (setf)

`(or string keyword)`

The title of the menu, as a string or keyword resource name. If there is a title-function, this slot is updated after the function has been called.

`:title-function nil menu-title-function` (setf)

`(function (interface) (or string keyword))`

The title-function is called after the popup-callback. It should return a title suitable for passing to `(setf menu-title)`.

---

menu *Class*

Description

A group of objects of the following classes:

- `menu-item` — the parts of a menu with which the user interacts
- `menu-component` — visual and functional grouping of menu-items and sub-menus
- `menu` — a sub-menu

Class Precedence List

`element titled-menu-object menu-object callbacks capi-object`

Slots

`:items nil menu-items` (setf)

`list`

The items in the menu. When this slot is set or initialised, any object which is not of class `menu-object` is made into a `menu-item` with the given object as its data.

`:popup-callback nil menu-popup-callback` (setf)
`(function (menu) t)`

The popup-callback is called just before the menu is raised. It can be used to change, enable or disable the items. The selection of menu-components can also be changed.

---

menu-component *Class*

Description

A subclass of `choice` and `menu-object` which is used to group menu-items and submenus either visually or by functionality . The items of a menu-component are surrounded by separators.

Class Precedence List

`choice collection menu-object callbacks capi-object`

Slots

`:interaction :no-selection choice-interaction`
`(member :single-selection :multiple-selection :no-selection)`

Controls the visible appearance and user input-model of its items.

- `nil` is used to provide separators. The visual appearance and input-model of the items is unchanged.
- `:single-selection` has an input-model as for single-selection choice and displays its items with check-marks.
- `:multiple-selection` has an input-model as for multiple-selection choice, and displays its items with check-marks.

---

menu-item *Class*

Description

An individual item in a menu. The text displayed in the menu item will be the contents of title slot if not `nil`, otherwise the result of applying the `print-function` to the data.

Callbacks are made in response to a user gesture on a menu-item. The callback type, function and data-function (see **callbacks** on) are found by looking for a non- `nil` value in first the menu-item, then the menu-component (if any) and finally the menu. This allows a whole menu to have, for example, callback-type `:data` without having to specify this in each item. Some items could override this by having their callback-data slot non- `nil` if needed.

Class Precedence List

`item titled-menu-object menu-object callbacks capi-object`

Slots

none

---

output-pane *Class*

Description

A pane which provides graphics output and user -interaction. `output-pane` supports many graphics-port functions, such as `gp:draw-rectangle`. See Appendix A for details of the graphics port functions which can be used with `output-pane`.

Class Precedence List

`titled-pane simple-pane element capi-object`

Slots

`:display-callback nil output-pane-display-callback` (setf)

```
(or null (function (output-pane integer integer integer
                    integer)
                   t))
```

A function to be called to redraw part of the output-pane. The arguments are the output-pane and the *x*, *y*, width and height of the region to be drawn.

`:scroll-callback nil output-pane-scroll-callback` (setf)

`(or null (function (output-pane integer integer) t))`

A function to be called if the user moves the scroll-bars. The arguments are the output-pane and the (*x, y*) co-ordinates of the top-left-hand corner of the visible part of an output pane within the output pane. This callback is for information only — scrolling and redrawing any exposed areas will be handled by calls to the display-callback.

```
:input-model nil output-pane-input-model
list
```

A list of input-model specifications (see Chapter 4, *Input Models*).

layout *Class*

Description

The superclass of all layouts.

Class Precedence List

`titled-pane simple-pane element capi-object`

Slots

```
:default nil default-layout
boolean
```

A flag to mark the default layout for an interface.

`:description nil layout-description` (setf)
`t`

The description of the layout children. Each layout defines the format of this slot. Setting the slot causes the layout to interpret the new description and position the new children. The parent layout will be enlarged if necessary .

simple-layout *Class*

Description

A layout with a single child. The child is resized to fit the layout. The size of the layout is constrained by the size-constraints of the child. The description is the single child.

Class Precedence List

`layout titled-pane simple-pane element capi-object`

grid-layout *Class*

Description

A layout which positions its children on a two-dimensional grid. The row and column sizes are constrained by the constraints of their contents (e.g., the minimum width of a column is the maximum of the minimum widths of the children in the column). The size of the layout is constrained by the constraints of the rows and columns. The description is either a two-dimensional array or a list in row-major order . In the latter case, one of `:columns` or `:rows` must be given as initargs to set the dimensions.

The elements of `x-ratios` control the width of each column relative to the others. `nil` means that the column will be fixed at its minimum width. Otherwise, the width will be `(round (*` *width* *ratio* `)` *ratio-sum* `)`, where *ratio-sum* is the sum of the non- `nil` element of `x-ratios` and *ratio* is the element of `x-ratios` corresponding to the column. If this width does not fit the constraints on the column width, the column is fixed at its contained width, *ratio* is removed and the calculation is performed again. If `x-ratios` has fewer elements than the number of columns, 1 is used for each missing ratio. This `x-ratios nil` causes all the columns to be of the same width.

The elements of `y-ratios` control the heights of the rows as for `x-ratios`.

Class Precedence List

`layout titled-pane simple-pane element capi-object`

Slots

`:rows nil`

`integer`

The number of rows in the grid. Only needed when the description is in a list.

`:columns nil`

`integer`

The number of columns in the grid. Only needed when the description is in a list.

`:x-ratios nil layout-x-ratios` (setf)

`list`

The ratios between columns (see above).

`:y-ratios nil layout-y-ratios` (setf)

`list`

The ratios between columns (see above).

`:x-gap` `3` `layout-x-gap` (setf)
`integer`

The gap between columns in the grid.

`:y-gap` `3` `layout-y-gap` (setf)
`integer`

The gap between rows in the grid.

`:x-adjust` `:left` `layout-x-adjust` (setf)
`(member :left :center :right)`

The adjustment applied to the *x*-position of each grid cell if the cell width is constrained to be less than the column width

- `:left` — left-justify the cell in the column
- `:center` — position the cell in the center of the column
- `:right` — right-justify the cell in the column

`:y-adjust` `:left` `layout-y-adjust` (setf)
`(member :top :center :bottom)`

The adjustment applied to the *y*-position of each grid cell if the cell height is constrained to be less than the row height.

- `:top` — top-justify the cell in the row
- `:center` — position the cell in the center of the row
- `:bottom` — bottom-justify the cell in the row

row-layout *Class*

Description

A subclass of `grid-layout` with one column. The description is a list of the cells in the row . The initargs `:ratios`, `:adjust` and `:gap` are transformed into `:x-ratios`, `:y-adjust` and `:x-gap` respectively .

Class Precedence List

`grid-layout layout titled-pane simple-pane element capi-object`

column-layout *Class*

Description

A subclass of `grid-layout` with one row. The description is a list of the cells in the column. The initargs `:ratios`, `:adjust` and `:gap` are transformed into `:x-ratios`, `:y-adjust` and `:x-gap` respectively.

Class Precedence List

`grid-layout layout titled-pane simple-pane element capi-object`

form-layout *Class*

Description

A subclass of `layout` displaying a column of elements with titles to the left. The left-hand edges of the objects are aligned. The size of the layout does not affect position or size of its children.

Class Precedence List

`layout titled-pane simple-pane element capi-object`

Slots

`:vertical-gap 2 form-vertical-gap` (setf)
`integer`

The gap between rows in the form.

`:vertical-adjust :center form-vertical-adjust` (setf)
`(member :top :center :bottom)`

The adjustment made to the positions of the title and the element on each row if one is smaller than the other.

- `:top` — align the tops
- `:center` — center the smaller in the size of the larger
- `:bottom` — align the bottoms

`:title-gap 5 form-title-gap` (setf)
`integer`

The gap between the title column and the element column.

`:title-adjust :left form-title-adjust` (setf)
`(member :left :center :right)`

The adjustment made to the horizontal position of each title.

- `:left` — align on the left
- `:center` — center within the largest width
- `:right` — align on the right

switchable-layout *Class*

Description

A subclass of `simple-layout` which displays one of several children. The layout retains the children. The description is a list of the children. The first child will be visible when the layout is first created.

Class Precedence List

`simple-layout layout titled-pane simple-pane element capi-object`

Slots

`:description nil switchable-layout-switchable-elements list`

The children of the `switchable-layout`. The first child is the visible one.

Related Functions

`switchable-layout-visible-child` *switchable-layout* — *Generic Function*

Returns the visible child in the *switchable-layout*.

`(setf switchable-layout-visible-child)` *child switchable-layout* — *Generic Function*

Sets the visible child in the *switchable-layout*. If *child* is not already one of the children, it is added to the list of switchable-children.

pinboard-layout *Class*

Description

A subclass of `layout` and `output-pane`. The children can be simple-panes, pinboard-objects or layouts. The children are not moved by the pinboard, and are fixed at their minimum size, regardless of the size of the pinboard. The pinboard-objects are automatically redrawn as required. The description is a list of the children. In addition to the normal hints, `:x` and `:y` can be given to position the children on the pinboard. If the pinboard has scroll-bars, it will resize its inner dimensions to allow all the children to be visible.

Class Precedence List

`output-pane layout titled-pane simple-pane element capi-object`

`redraw-pinboard-layout` *Function*
pinboard-layout x y width height

Causes the objects within the given rectangle (positioned and bounded by virtue of the *x*, *y* co-ordinates and the *width* and *height* arguments) of the *pinboard-layout* to be redrawn. The drawing will occur under the control of the window system.

`pinboard-object-at-position` *Generic Function*
pinboard-layout x y

Returns the pinboard-object located at the given *x*, *y* co-ordinates. Objects whose `activep` slot is false are not considered.

`set-scroll-range` *Function*
pane width height

Set the range over which the scroll-bars apply.

`set-scroll-position` *Function*
pane x y

Move the scroll bars of the pane to place the given *x*, *y* co-ordinates in the top left-hand corner of the visible window.

---

pinboard-object *Class*

Description

An object can be placed on a pinboard-layout but have no window-system functionality. Pinboard-objects are redrawn entirely by the CAPI.

Class Precedence List

`capi-object`

Slots

`:activep nil pinboard-object-activep` (setf)

`boolean`

Non- `nil` means than `pinboard-object-at-position` will find the pinboard-object.

Related Functions

`over-pinboard-object-p` *Generic Function*

pinboard-object x y

Return true if the *x*, *y* co-ordinates are within the boundary of the *pinboard-object* .

`pinboard-object-overlap-p` *Generic Function*

pinboard-object x0 y0 x1 y1

Return true if the rectangle bounded by the supplied co-ordinates overlaps the boundary of the pinboard-object.

`highlight-pinboard-object` *Generic Function*

pinboard-layout pinboard-object

Draws a highlight on a *pinboard-object* . The default method inverts a rectangle around the object. The method for line-pinboard-object inverts a parallelogram.

`unhighlight-pinboard-object` *Generic Function*

pinboard-layout pinboard-object

Removes a highlight on a *pinboard-object* . The default method calls highlight-pinboard-object.

`redraw-pinboard-object` *Generic Function*

`pinboard-object`

Causes the given *pinboard-object* , and any which overlap it, to be redrawn.

Defining New Pinboard Objects

Methods should be defined for the following generic functions if the default method is inappropriate.

- `pinboard-object-overlap-p`
- `over-pinboard-object-p`
- `highlight-pinboard-object`
- `unhighlight-pinboard-object`
- `calculate-constraints` (see Chapter 6, *Writing a Subclass of Layout*)

`draw-pinboard-object` *Generic Function*
pinboard-layout pinboard-object

This method should draw the *pinboard-object* on its output-pane. The co-ordinates and size of the object should be read from its geometry information (see Chapter 6, *Writing a Subclass of Layout*).

---

item-pinboard-object *Class*

Description

A subclass of `pinboard-object` and `item`. The result of calling the print-function on the item-data is drawn in the given font. The minimum size of the object is determined by the size of the text.

Class Precedence List

`pinboard-object item callbacks capi-object`

Slots

`:font nil item-pinboard-object-font` (setf)
`generic-font`

The font used to display the text.

---

icon-pinboard-object *Class*

Description

A subclass of `pinboard-object` which displays an icon.

Class Precedence List

`pinboard-object capi-object`

Slots

`:pixmap nil icon-pinboard-object-pixmap` (setf)
`generic-image`

The image to be displayed.

titled-icon-pinboard-object *Class*

Description

A subclass of `item-pinboard-object` and `icon-pinboard-object` which displays an icon with some text.

Class Precedence List

`icon-pinboard-object item-pinboard-object pinboard-object item callbacks capi-object`

Slots

`:title-x nil titled-icon-pinboard-object-title-x` (setf)
`(or null integer)`

The *x*-offset of the title from the left side of the pinboard-object. If this is `nil`, the text is centered horizontally .

`:title-y nil titled-icon-pinboard-object-title-y` (setf)
`(or null integer)`

The *y*-offset of the title from the top of the pinboard-object. If this is `nil`, the text is placed below the icon.

line-pinboard-object *Class*

Description

A subclass of `pinboard-object` which displays as a line drawn from (*x*, *y*) to (*x* + *width* , *y* + *height*).

Class Precedence List

`pinboard-object capi-object`

drawn-pinboard-object *Class*

Description

A subclass of `pinboard-object` which is drawn by a given function.

Class Precedence List

`pinboard-object capi-object`

Slots

`:display-callback nil pinboard-object-display-callback` (setf)

```
(function (output-pane drawn-pinboard-object
           integer integer integer integer)
          t)
```

A function called to draw the pinboard-object. The six arguments are:

- an *output-pane* to draw on
- the *drawn-pinboard-object* itself
- *x* and *y* co-ordinates of the object on the pinboard-layout
- the width and height of the object

---

editor-pane *Class*

Description

An Emacs-like editor .

Class Precedence List

`output-pane titled-pane simple-pane element capi-object`

Slots

`:enabled t editor-pane-enabled` (setf)

`boolean`

When non- `nil` the editor -pane accepts input. Otherwise it acts much like a display-pane with extra programmer functionally , such as paragraph filling, etc.

`:fixed-fill nil editor-pane-fixed-fill` (setf)

`boolean`

When non- `nil`, the editor will refill each paragraph when the window changes width.

Related Functions

`pane-string` *Generic Function*
editor-pane

Return the contents of the editor -pane.

`modify-editor-pane-buffer` *Generic Function*
editor-pane `&key` *contents flag fill fixed-fill*

Modify the editor -pane according to the keyword arguments. *contents* (if non-nil) supplies a new string to place in the buffer . *flag*, if given, sets the buffer -flag. *fill*, if non-`nil`, causes the editor to fill each paragraph in the buffer . *fixed-fill* sets the value of the slot fixed-fill. If it is non-nil, *fixed-fill* causes the editor to fill each paragraph in the buffer .

---

interactive-stream *Class*

Description

An `editor-pane` with stream-based I/O functionality .

Class Precedence List

`editor-pane output-pane titled-pane simple-pane element capi-object`

Slots

`:top-level-function 'capi-top-level-function interactive-stream-top-level-function`

`(function (interface interactive-stream stream) t)`

The function called to process input from the interactive-stream. The first argument is the interface containing the interactive-stream. The second argument is the interactive-stream itself. The third argument is the I/O stream. The function should read from the I/O stream to activate the interactive-stream. The default function runs a Lisp Listener top-loop.

---

listener-pane *Class*

Description

A Lisp Listener . This is a subclass of `interactive-stream` which reads and executes Lisp forms.

Class Precedence List

`interactive-stream` `editor-pane` `output-pane` `titled-pane` `simple-pane` `element` `capi-object`

---

collector-pane *Class*

Description

A pane which displays the output sent to a character stream.

Class Precedence List

`editor-pane` `output-pane` `titled-pane` `simple-pane` `element` `capi-object`

Slots

`:stream nil collector-pane-stream`

`(or null string)`

The Common Lisp output stream associated with the collector -pane. A new stream is created, and connected to a buffer named by the `:buffer-name` initarg.

`:buffer-name nil`

`string`

The name of the editor buffer which should contain the output.

Related Functions

`map-typeout` *Function*

pane `&rest` *args*

Makes a collector -pane the visible child of a switchable-layout. The switchable-layout is found by looking up the parent hierarchy , starting from *pane* .

The switchable-layout should have one or two children. If it has one child, a new collector -pane is made using *args* as the initargs with `:buffer-name` defaulting to `"Background Output"` and `:background` defaulting to `:medium-yellow`. If it has two children, the second is used as the collector . The collector -pane is returned.

`unmap-typeout` *Function*
collector-pane

This should be called with a collector -pane returned by `map-typeout`. The collector -pane is replaced by the original child of the switchable-layout, the collector -pane becoming the second child.

`with-random-typeout` *Macro*
`(stream` *pane* `) &body` *body*

Within the *body*, *stream* is bound to a collector -pane-stream associated with *pane* . The collector -pane is automatically mapped and unmapped around the body . If the body exits normally , the typeout is not unmapped until `SPACE` is pressed or the mouse is clicked.

---

interface *Class*

Description

A subclass of `simple-pane` which groups panes in a layout. Interfaces can be embedded within other layouts or can be displayed as top-level windows. See `define-interface` in Chapter 3, .

Class Precedence List

`simple-pane element capi-object`

Slots

`:title nil interface-title` (setf)
`string`

The title of the window representing the interface.

`:layout nil pane-layout` (setf)
`(or layout symbol)`

The top-level layout of the interface. This can be set to a layout or the name of a slot in the interface which contains a layout.

`:menu-bar-items nil interface-menu-bar-items`
`list`

A list of menus to be displayed on the menu-bar of the interface.

`:local-menu nil interface-local-menu`
`menu`

A window-system specific menu.

Related Functions

`display` *Function*
interface `&key` *screen*

Displays a top level *interface* on the given *screen*. If *screen* is not specified and the interface has no screen, the default screen of the current window-system is used.

`destroy` *Generic Function*
interface

Removes a top level *interface* from the screen.

`redisplay-menu-bar` *Function*
interface

Re-displays the menu-bar of an *interface*. This is useful if the `enabled` slot has been changed in one of the top menus on the menu-bar, causing the title of the item to become grayed or un-grayed.

`hide-interface` *Function*
interface

Hide the top-level *interface* from the screen. Typically this means iconifying the top-level window.

`show-interface` *Function*
interface

Undo the affects of `hide-interface` for the top-level *interface*.

`raise-interface` *Function*
interface

Raise the top-level *interface* above all other windows on the screen.

`lower-interface` *Function*
interface

Lower the top-level *interface* below all other windows on the screen.

`display-dialog` *Function*
interface `&key` *screen* `(`*grab-mouse* `t)`

Displays a top-level *interface* as a dialog box. *screen* is used as for `display`. When *grab-mouse* is true, the input focus is constrained to the dialog box.

`abort-dialog` *Function*
`&optional` *value*

Aborts the current dialog box, causing `display-dialog` to return `nil`. *value* is ignored. This can be called from a callback of one of the panes in the dialog.

`exit-dialog` *Function*
`&optional` *value*

Exits the current dialog, causing `display-dialog` to return two values, *value* and `t`. This can be called from a callback of one of the panes in the dialog.

---

screen *Class*

Description

A class which represents the connection to the user . Screens are made automatically and can be queried for various properties.

Class Precedence List

`capi-object`

Related Functions

`screen-width` *Generic Function*
screen

Returns the width of the *screen* in pixels.

`screen-height` *Generic Function*
screen

Returns the height of the *screen* in pixels.

`screen-depth` *Generic Function*
screen

Returns the number of color planes in the *screen* .

`screen-width-in-millimeters` *Generic Function*
screen

Returns the width of the *screen* in millimeters.

`screen-height-in-millimeters` *Generic Function*
screen

Returns the height of the *screen* in millimeters.

`ensure-interface-screen` *Function*
interface `&key` *screen*

Ensures that the top level interface will be displayed on the given *screen* if `display` is called later without a `:screen` argument. After calling `ensure-interface-screen`, fonts and colors can be queried from the elements in the interface. If `screen` is `nil`, the default screen is used. Returns the screen which will be used.

2.4 Miscellaneous Macros and Functions

Below, some miscellaneous macros and functions related to the CAPI are described.

Miscellaneous

Descriptions

`with-atomic-redisplay` *Macro*
`&body` *body*

Some functions, amongst them slot writers, modify state. When the visual appearance of the simple-pane depends upon this state, it will be updated immediately. In order to remove double re-display, or to delay re-display of several panes until some common point, the macro `with-atomic-redisplay` can be used. All window-system changes will then be performed upon exit from the *body*.

`define-menu` *Macro*
function-name (*self*) *title* *menu-body* `&rest` *menu-options*

Defines a function called *function-name* with a single argument (*self*) which will make a menu with *title*, *menu-body* and *menu-options* of the same form as `:menus` in `define-interface`.

`display-message` *Function*
format-string `&rest` *format-args*

Display a message on the default screen, using a window-system built-in dialog box.

`display-message-on-screen` *Function*
screen format-string `&rest` *format-args*

Display a message on the given *screen*, using a window-system built-in dialog box.

2.5 Built-in Dialog Boxes

The CAPI defines some useful dialog boxes. The built-in Confirmer dialog and its related dialog boxes are discussed below.

2.5.1 The Confirmer

The Confirmer is a general purpose dialog box, which can contain a given pane with a `push-button-panel`. By default, there are two buttons, labelled *Ok* and *Cancel*.

`popup-confirmer` *Function*
pane title `&key` *geometry*
value-function
(*exit-function* `'exit-confirmer`)
ok-function
ok-check
(*ok-button* `"Ok"`)
(*cancel-button* `"Cancel"`)
buttons
callbacks
(*callback-type* `:data`)
screen

Pop-up a Confirmer entitled *title* containing *pane* at the top and some buttons beneath it.

The various arguments are described below:

- *pane* — the pane to be displayed (this can be `nil` if no pane is required).
- *title* — the dialog box's title.
- *geometry* — geometry hints passed to the `make-instance` call for the Confirmer.
- *value-function* — the function which is called with the pane as a single argument to compute the value passed to the *ok-function*.
- *exit-function* — the callback of the 'ok' button. The default `exit-confirmer` calls the *value-function*, *ok-check* and *ok-function* before

calling `exit-dialog` if the *ok-check* returns true.

- *ok-function* — the function which converts the result of the *value-function* into the return value of `popup-confirmer`.
- *ok-check* — the function which ensures that the result of the *value-function* is valid. If this returns `nil`, the *ok-function* is not called and the Confirmer remains on the screen.
- *ok-button* — the label of the 'ok' button (the default is *Ok*).
- *cancel-button* — the label of the 'cancel' button (the default is *Cancel*).
- *buttons* — a list of additional items to be placed on buttons at the bottom of the Confirmer. *ok-button* and *cancel-button* are `cons`-ed onto the start of this list.
- *callbacks* — a list of callback functions for each item in *buttons*.
- *callback-type* — the callback-type for the button panel. See **callbacks** on page 2-8.
- *screen* — the screen upon which to display the Confirmer dialog box. see `display` on page 2-33.

`exit-confirmer` *Function*
`&rest` *args*

Attempt to exit from the Confirmer, calling the *value-function*, *ok-check* and *ok-function* before calling `exit-dialog` if the *ok-check* returns true. The arguments *args* are ignored.

`prompt-for-string` *Function*
title `&rest` *popup-args* `&key` *value-function*
(*min-width* `10`)
initial-value
initial-value-to-string-function
caret-position
completion-function

Pop-up a Confirmer containing a `text-input-pane`. Returns the result of the *value-function* applied to the string in the text-input-pane.

The various arguments are described below.

- *title* — the dialog box's title.

- value-function — the function to convert the text in the text-input-pane to the return value of prompt-for -string (if this is nil, the text is returned).
- min-width — the minimum width of the text-input-pane in characters.
- initial-value — the initial value of text-input-pane-text, i.e., the initial text in the prompter .
- initial-value-to-string-function — a function to convert the initial-value to a string.
- caret-position — the initial text-input-pane-caret-position.
- completion-function — the text-input-pane-completion-function, used when the user makes the completion gesture in the text field.
- popup-args — extra arguments passed to `popup-confirmer`.

`prompt-for-form` *Function*

title `&rest` *popup-args* `&key (`*evaluate* `t)`
`(`*quotify* *evaluate* `)`
`&allow-other-keys`

Pop-up a Confirmer containing a `text-input-pane`. Returns the contents of the text-input-pane as a form.

The various arguments are described below .

- *title* — the dialog box' s title.
- *evaluate* — when non- `nil`, `prompt-for-form` returns the result of calling `eval` upon the form.
- *quotify* — when non- `nil`, a quote is placed before the *initial-value* unless it is self-evaluating.
- *popup-args* — extra arguments passed to `prompt-for-string`.

`prompt-with-list` *Function*

items title `&rest` *choice-args*
`&key (`*choice-class* `'list-panel)`
`(`*min-height* `6)`
`(`*interaction* `:single-selection)`
`(`*value-function* `'choice-selected-items)`

immediate-exit
popup-args
`&allow-other-keys`

Pop-up a Confirmer containing a subclass of `choice`, the default being `list-panel`.

The various arguments are described below.

- *items* — the items to display in the choice.
- *title* — the dialog box's title.
- *choice-class* — the subclass of `choice` in which to display the *items*.
- *min-height* — the minimum height of the choice, in characters.
- *interaction* — the `choice-interaction`.
- *value-function* — the function used to compute the return value of `prompt-with-list`. If this is omitted, the `choice-selected-items` of the choice are returned.
- *immediate-exit* — when true, the Confirmer will have no buttons and will exit as soon as an item is selected. This is only valid for `:interaction :single-selection`.
- *popup-args* — extra arguments passed to `popup-confirmer`.
- *choice-args* — arguments passed to the `make-instance` for the choice.

`prompt-for-items-from-list` *Function*
items *title* `&rest` *choice-args*

Call `prompt-with-list` with `:interaction :multiple-selection`.

`prompt-for-integer` *Function*
`&key` *lo* *hi* *title*

Pop-up a Confirmer for an integer in between *lo* and *hi* inclusive. The Confirmer is entitled *title*. If *lo* and *hi* are omitted, the is no restriction upon the range of the integer. The integer is returned.

`confirm-yes-or-no` *Function*

(*format-string* `&rest` *format-args*)

Pop-up a Confirmer containing only the buttons 'yes' and 'no' and the message obtained by `(apply 'format nil` *format-string* *format-args* `).` `confirm-yes-or-no` returns true when the 'yes' button is pressed, and false when the 'no' button is pressed.

Chapter 3
Defining an Interface

3.1 Introduction

The macro `define-interface` is used to define subclasses of `interface`. Interfaces defined in this way can be made with `make-instance` which will cause the specified panes, layouts and menus to be constructed automatically.

3.2 Syntax

The syntax of `define-interface` is given by the B.N.F. for *<define-interface>*.

The following notation is used:

A *name* is a symbol which is not evaluated.

A *form* is a Lisp expression which is evaluated in its lexical context. Within a form, the slots of the interface are bound by `with-slots`, and the variable `self` is bound to the actual interface.

This grammar below employs a special economical scheme for expressing instances of the Kleene star -closure of a non-terminal. Where the grammar reads

```
<non-terminal >*
```

This is equivalent to

```
<closure-of-non-terminal > ::= ε
                             | terminal <closure-of-non-terminal >
```

The B.N.F . for the `define-interface` syntax is given below .

```
<define-interface > ::=
 (define-interface <class name > (<super -class name >*)
                   (<defclass-slot-description >*)
                   <interface-option >*)

<interface-option > ::= <defclass-option >
                      | (:panes <pane-description >*)
                      | (:layouts <layout-description >*)
                      | (:menus <menu-description >*)
                      | (:menu-bar <menu-slot name >*)

<pane-description > ::=
       (<slot name > <class name > <interface-slot-option >*)

<layout-description > ::=
        (<slot name > <class name > <layout-description form >
         <interface-slot-option >*)

<slot-option > ::= :reader <reader name >
                 | :writer <writer name >
                 | :accessor <accessor name >

<interface-slot-option > ::= <slot-option >
    | <keyword > (:initarg {<keyword > | <keyspec > <form >})
    | <keyword > <form >
```

<keyword > and <keyspec > following the `:initarg` will be used as part of the description of the lambda-list of the `initialize-interface :after` method for <class *name* >. `initialize-interace` will be called by the

`intialize-instance :after` method for `interface`.

```
<menu-description> ::= <menu> | <menu-component>

<menu> ::= (<slot name> <title form> (<menu-item>*)
           <menu-option>*)

<menu-option> ::= <slot-option> | <menu-item-option>

<menu-component> ::= (:component (<simple-menu-item>*)
                      <menu-component-option>*)

<menu-component-option> ::= <callback-option>
                | :interaction <menu-component-interaction>

<menu-component-interaction> ::= :no-selection
                               | :single-selection
                               | :multiple-selection

<menu-item> ::= <simple-menu-item> | <menu-component>

<simple-menu-item> ::= <string>
                     | <resource-key>
                     | <slot name>
                     | (<title form> <menu-item-option>*)

<menu-item-option> ::= <callback-option>
                     | :enabled-function <enabled-function form>
                     | :enabled-slot <slot name>
                     | :popup-callback <popup-function form>

<callback-option> ::= :callback <form>
                    | :callback-data <form>
                    | :callback-type <callback-type>

<callback-type> ::= :data | :data-interface
                  | :interface-data | :item | :interface
                  | :item-interface | :interface-item
                  | :full | :none | nil
```

The <title *form*> and <enabled-function *form*> are evaluated each time the menu pops-up. For top-level menus, <title *form*> is evaluated when `redisplay-menu-bar` is called. The `:callback-data` <*form*> is evaluated before each callback is performed.

If `:menu-bar` is omitted, all the menus in the `:menus` option will be placed on the menu bar in the order that they are defined.

The top-level layout will be the first one with `default` set, the first layout given or a column of all the panes. For a discussion of `default`, see `layout` (Chapter 2, page 10).

Slot names must be distinct from names of layout classes. See the *The Layout Description* on page 5-1

Chapter 4
Input Models

4.1 Introduction

An input model is used, in conjunction with an output-pane, to provide the user interface for applications which use `output-pane`. The model describes the relationship between mouse and keyboard activities (known as *gestures*) and the actions they should effect. The actions are defined by the programmer as functions.

Abstract commands provide a window-system independent means of associating actions with gestures. The input model is a list, each element being itself a list of a command name or gesture, a callback-function, and (optionally) extra arguments for the callback-function. The callback-function is called with the pane, the translated gesture information and any extra arguments given in the input model.

4.2 Gesture Specification

A gesture is described by the B.N.F. grammar <gesture>. This B.N.F employs the same syntactic economy mechanism for expressing Kleene-closures of non-terminals as did the description in Chapter 3, *Defining an Interface*.

<gesture> ::= <button-gesture> | <motion-gesture>
| <key-gesture> | <character-gesture>

<button-gesture> ::= (<button-name> [<button-action> <modifier>*])

<motion-gesture> ::= `:motion`
| (`:motion` [<button-name> <modifier>*])

<key-gesture> ::= (<key> <key-action> <modifier>*)

<character-gesture> ::= <character> | `:character`

<button-name> ::= `:button-1` | `:button-2` | `:button-3`

<button-action> ::= `:press` | `:release` | `:second-press`

<modifier> ::= `:shift` | `:control` | `:meta`

<key> ::= `:key` | <key-name>

<key-action> ::= `:press` | `:release`

The various non-terminals are explained below:

- <button-gesture> matches the user actions with the mouse buttons.
  - <button-name> — which button to match
  - <button-action> — the direction of motion of the button
  - <modifier>* — the modifier keys held down with the button.

  The gesture information consists of the (x, y) position of the click relative to the top left-hand corner of the pane.
- <motion-gesture> matches mouse movement. `:motion` matches only when no mouse buttons are held down. Otherwise, <button-action> and <modifier>* are given as for <button-gesture>.

  The gesture information consists of the (x, y) position of the pointer relative to the top left-hand corner of the pane.
- <key-gesture> matches user actions with keys on the keyboard
  - <key> — the key to match, or any key if `:key`

- <key-action> — the direction of motion of the key
- <modifier>* — the modifier keys held down with the key.

  The gesture information consists of the (*x*, *y*) position of the pointer when the key was pressed, and the key.

- <character-gesture> matches a character typed by the user. `:character` matches any character. Otherwise, the gesture matches the given Common Lisp extended character, which can be made up from more than one keystroke according to the keyboard. (e.g. `#\Control-x`.)

  The gesture information consists of the (*x*, *y*) position of the pointer when the character was entered, and the character.

4.3 Command Specification

Commands are defined with `define-command`. Each command has a name, a gesture and optionally a translator and a host. Recall from the Introduction that the command name is used in the input-model specification. The gesture describes how the command is invoked by the user. The translator computes the translated gesture information from the gesture information. This translated information is supplied to the callback function (see *Input Models* on page 4-1). The host is the name of the window system for which the gesture will apply (`nil` means all systems). This allows different gestures for each Look and Feel to invoke the same command.

`define-command` *Macro*

name gesture `&key` *translator host*

Defines the command *name* for the given *gesture*, *translator* and *host*. If *translator* is `nil`, no translation is performed. If *host* is `nil`, the command will be used for all hosts.

`invoke-command` *Function*

command pane `&rest` *gesture-information*

Invokes the *command* in the input-model for the given *pane*. The translator is called to process the *gesture-information*.

`invoke-untranslated-command` *Function*

command pane `&rest` *translated-gesture-information*

Invokes the *command* in the input-model for the given *pane*, without calling the translator.

Chapter 5 The Layout

5.1 Introduction

A layout is used to group instances of `simple-pane` and `pinboard-object`. The type of layout specifies the nature of the grouping.

5.2 The Layout Description

Each layout requires a description of its children, usually as a list of child descriptions. Each child description can be one of the following:

- `simple-element` — the element is used as the child. `simple-element` is a common superclass of both `simple-pane` and `pinboard-object`, i.e., objects which can be placed in a layout.
- `symbol` — the contents of the slot named by the symbol in the interface containing the layout.
- `list` — If the head of the list is a symbol naming a subclass of layout, then a layout of that class is made with the `second` of the list as the description and the `cddr` as initialization arguments. Otherwise, the head should be a `simple-element` or `symbol` as above, and the rest of the list will be used as hints for the child.

5.3 Constraints and Hints

The geometry of each layout child is controlled by the layout itself. The layout algorithm takes account of hints provided by the child (from the initargs to `make-instance`) and the layout description (see above). Hints in the layout description override those in the child. The following size hints are recognised by all layouts:

- `:min-width` — the minimum width of the child
- `:max-width` — the maximum width of the child
- `:min-height` — the minimum height of the child
- `:max-height` — the maximum height of the child

The value of the size hint can be one of the following:

- *number* — the numeric value in pixels is used.
- `t` — for `:max-width` and `:max-height` this means than the maximum width/height is the same as the minimum (i.e., the child is fixed at its minimum size).
- `:text-width` — the maximum width in pixels of all the text in the child.
- `:screen-width` — the width of the screen in pixels.
- `(:character` *number* `)`

and

- `(character` *number* `)` — the size in pixels required for *number* characters to be visible in the child. Any border required by the child is automatically added.
- `(:string` *string* `)`

and

- `(string` *string* `)` — the width in pixels required for *string* to be visible in the child. Any border required by the child is automatically added.

Also, the following operators can be used upon hints:

- `(max` *hint* $^+$`)`
- `(min` *hint* $^+$`)`
- `(+` *hint* $^+$`)`
- `(-` *hint* $^+$`)`
- `(*` *hint* $^+$`)`

- `(/` *hint* $^{+}$`)`

In these forms, the function in the head of the list will be applied to the result of interpreting the hints in the tail. The superscript $^{+}$ in the form

`(operator` *hint* $^{+}$`)`

means that there should be one or more *hint*.

- `(apply` *function otherargs* `)` — the result of applying *function* to the child and *otherargs*.

For example, for a `text-input-pane`, the hint

```
:min-width '(+ :text-width (:character 5))
```

will constrain the text-input-pane to be wider than the width of the text in the child plus five character-widths.

`pinboard-layout` also recognises `:x` and `:y`, their values being the (*x*, *y*) position of the child on the pinboard.

5.4 Functions on Hints

`set-hint-table` *Function*
simple-element plist

The hint-table for the element is set to be the *plist*. All existing hints are retained for keys not in the plist.

`set-geometric-hint` *Function*
simple-element key value `&key (`*override* `t)`

Set the hint associated with *key* to *value*. If *override* is `nil`, the value is not changed when there is already a hint for this key.

Chapter 6
Writing a Subclass of Layout

6.1 Introduction

New subclasses of layout can be defined using `define-layout`. This chapter details the process of defining new layout subclasses.

6.2 The Layout Algorithm

The package `capi-layout` exports symbols which are useful in defining a layout. The CAPI layout algorithm works as follows:

Each pane has some initial constraints on its minimum and maximum size.

In pass one, layouts combine the constraints of their children with their own initial constraints. The details of how the constraints are combined vary according to the layout algorithm. See `calculate-constraints` on page 6-3.

In pass two, each layout is given a size in which to fit its children. This size will be between the minimum and maximum sizes calculated in the first pass. The layout sets the size and position of its children according to its layout algorithm and their constraints. See `calculate-layout` on page 6-3.

Finally , the CAPI resizes the children, which causes any sub-layouts to fit their children, and so on.

6.3 Geometry Information

Simple-panes and pinboard-objects have a slot called `geometry-cache` which contains a geometry object holding all the geometry information for the pane. This is set to be a geometry object automatically by `make-instance` and should not be set directly.

Geometry objects have the following fields:

Field	**Description**
`x`	*x*-position of the left-hand edge
`y`	*y*-position of the top
`width`	width in pixels
`height`	height in pixels
`min-width`	minimum width
`min-height`	minimum height
`max-width`	maximum width
`max-height`	maximum height
`ratio`	ratio assigned: each layout uses this as required
`child`	pane or pinboard-object for this geometry object

To make layout algorithms more concise, there are several macros which can be employed. These macros use `symbol-macrolet` to bind the names `%x%`, `%y%`, `%width%`, `%height%`, `%min-width%`, `%min-height%`, `%max-width%`, `%max-height%`, `%ratio%` and `%child%` to the associated field in a geometry object. These symbol-macros can be read or set as required.

6.4 Required Methods

The following generic functions should have methods for each subclass of `layout`.

`interpret-description` *Method*
layout description interface

The *description* should be interpreted as appropriate for the layout. Each child within the description should be parsed using `parse-layout-descriptor` given the specified *layout* and *interface* as arguments. `interpret-description` should return a list of the geometry objects for all the children of the layout.

`calculate-constraints` *Method*
layout

This method should call `get-constraints` on each of its children and combine the results to set its own minimum/maximum width/height. This will be done in pass one when the constraints of the layout may no longer be valid.

`calculate-layout` *Method*
layout x y width height

This method should set the `%x%`, `%y%`, `%width%` and `%height%` of each of its children according to their constraints and to the layout algorithm. The *x* and *y* co-ordinates describe the top left-hand corner, whilst the *width* and *height* describe the size of the layout.

6.5 Utilities for Layout Definition

The following macros, generic functions and functions can be used in defining new layouts.

`define-layout` *Macro*
class-name superclass-names slots `&rest` *class-options*

Defines a new subclass of `layout`. This macro is the same as `defclass` except that *superclass-names* defaults to `(layout)` instead of `(standard-object)`.

`parse-layout-descriptor` *Function*
child-descriptor interface layout

Returns a geometry object parsed from *child-descriptor* which should match one of the cases described in *The Layout Description* on page 5-1. See also `interpret-description` on page 6-2.

`get-constraints` *Function*
pane

Returns the `min-width`, `min-height`, `max-width` and `max-height` of the *pane* as four values. `calculate-constraints` is called if the constraints are invalid.

`with-geometry` *Macro*
pane `&body` *body*

Within the *body*, the % geometry variables are bound to the geometry from the *pane*.

`with-geometry-values` *Macro*

geometry `&body` *body*

Within the *body*, the % geometry variables are bound from the given *geometry* object.

`with-single-child` *Macro*

`&body` *body*

Within the *body*, the % geometry variables are bound to the geometry from the single child in the layout named `self`.

`do-all-children` *Macro*

`&body` *body*

Executes the *body* for each child of the layout named `self`, with the % geometry variables bound to the geometry from the child.

`do-object-children` *Macro*

`&body` *body*

Executes the *body* for each pinboard-object child of the layout named `self` with the % geometry variables are bound to the geometry from the child.

`do-pane-children` *Macro*

`&body` *body*

Executes the *body* for each simple-pane child of the layout named `self` with the % geometry variables are bound to the geometry from the child.

`layout-panes` *Generic Function*

layout

Returns the list of geometry objects which was computed by `interpret-description` when the *layout* was created.

Chapter 7
Generic Fonts

7.1 Introduction

This chapter discusses Generic Fonts in the CAPI. Generic Fonts provide a platform-independent description of fonts.

7.2 Components of a Generic Font

Generic Fonts have the following components:

- `foundry` — the string name of the foundry, e.g., `"adobe"`. This can be omitted, unless there is a conflict in the `family` name.
- `family` — the string name of the font family, e.g., `"courier"`.
- `weight` — the weight of the font, one of `:bold`, `:medium`, `:light`, `:demibold`.
- `slant` — the slant of the font, one of `:roman`, `:italic`, `:oblique`.
- `size` — the size of the font in points (an integer)
- `average-width` — the average width of the font in points (an integer).

7.3 Font-related functions

The following functions concern operations with Generic Fonts.

`make-generic-font` *Function*
`&key` *foundry family weight slant size average-width*

Return a new generic-font object with the given fields. All fields default to `nil` if not specified.

`query-font-foundaries` *Function*
port `&optional` *fonts*

Query the font foundries available for *port*. If given, *fonts* should be a partially-specified generic-font object to be used in the matching. A list of strings is returned.

`query-font-families` *Function*
port `&optional` *fonts*

Query the font families available for *port*. If given, *fonts* should be a partially specified generic-font object to be used in the matching. A list of strings is returned.

`query-font-weights` *Function*
port `&optional` *fonts*

Query the font weights available for *port*. If given, *fonts* should be a partially-specified generic-font object to be used in the matching. A list of keywords, from those in Section 7.2, is returned.

`query-font-slants` *Function*
port `&optional` *fonts*

Query the font slants available for *port*. If given, *fonts* should be a partially-specified generic-font object to be used in the matching. A list of keywords, from those in Section 7.2, is returned.

`query-font-sizes` *Function*
port `&optional` *fonts*

Query the font sizes available for *port*. If given, *fonts* should be a partially-specified generic-font object to be used in the matching. A list of integers is returned.

`query-font-average-widths` *Function*

port `&optional` *fonts*

Query the average font widths available for *port*. If given, *fonts* should be a partially-specified generic-font object to be used in the matching. A list of integers is returned.

`get-font-ascent` *Function*

port `&optional` *font*

Return the font ascent of *font* for the *port*. If *font* is `nil`, the default font for the port is used.

`get-font-descent` *Function*

port `&optional` *font*

Return the font descent of *font* for the *port*. If *font* is `nil`, the default font for the port is used.

`get-char-width` *Function*

port character font

Return the width in pixels of the *character* in the given *font* for the *port*. If *font* is `nil`, the default font for the port is used.

`get-char-ascent` *Function*

port character font

Return the ascent in pixels of the *character* in the given *font* for the *port*. If *font* is `nil`, the default font for the port is used.

`get-char-descent` *Function*

port character font

Return the descent in pixels of the *character* in the given *font* for the *port*. If *font* is `nil`, the default font for the port is used.

`define-font-alias` *Function*
name font

Define *name* to be an alias for *font*. *name* should be a symbol, and *font* should be a generic-font.

`get-font-alias` *Function*
name

Return the font for which *name* is an alias, or `nil` if none.

Chapter 8
Color Models

8.1 Introduction

This chapter discusses the color models available in the CAPI. Three models of common color -specification schemes are provided.

8.2 The Three Models

There three models provided are RGB, HSV and Gray . The function `color-model` returns the model-type of a color specification *color-spec* .

`color-model` *Function*
color-spec

Returns the color -model for the color -spec.

8.2.1 RGB Model

The following functions relate to RGB-model color specifications.

`make-rgb` *Function*
red green blue

Returns a color -spec in the `:rgb` model with components *red* , *green* and *blue* . These values must be floating-point numbers between 0 and 1.

`color-red` *Function*
color-spec

Returns the red component of the *color-spec*. If the color-spec is not in the `:rgb` model, then the component is calculated.

`color-green` *Function*
color-spec

Returns the green component of the *color-spec*. If the color-spec is not in the `:rgb` model, then the component is calculated.

`color-blue` *Function*
color-spec

Returns the blue component of the *color-spec*. If the color-spec is not in the `:rgb` model, then the component is calculated.

8.2.2 HSV Model

The following functions relate to HSV-model color specifications.

`make-hsv` *Function*
hue saturation value

Returns a color-spec in the `:hsv` model with components *hue*, *saturation* and *value*. The value of *hue* must be a floating-point number between 0 and 6. The *saturation* and *value* arguments must be floating-point numbers between 0 and 1.

`color-hue` *Function*
color-spec

Returns the *hue* component of the *color-spec*. If the color-spec is not in the `:hsv` model, then the component is calculated.

`color-saturation` *Function*
color-spec

Returns the *saturation* component of the *color-spec*. If the color-spec is not in the `:hsv` model, then the component is calculated.

`color-value` *Function*
color-spec

Returns the *value* component of the *color-spec*. If the color-spec is not in the `:hsv` model, then the component is calculated.

8.2.3 Gray Model

The following functions relate to Gray-model color specifications.

`make-gray` *Function*
level

Returns a color-spec in the `:gray` model with level *level*, which must be a floating-point number between 0 and 1.

`color-level` *Function*
color-spec

Returns the *level* of the *color-spec*. If the color-spec is not in the `:gray` model, then the average of the red, green and blue components is taken.

8.3 Color-Alias

Color-aliases provide a means associating names with colors, whether themselves other aliases or color-specs. CAPI provides many built-in aliases for commonly-used colors. The function `get-all-color-names` returns all known aliases.

`define-color-alias` *Function*
name color `&optional` (*if-exists* `:replace`)

Defines *name* to be a color alias for *color*, which must be a color-alias or a color-spec. *if-exists* can be one of:

- `:replace` — replaces any existing alias
- `:error` — raises an error if alias is already defined
- `:ignore` — ignores the redefinition of an alias

and is `:replace` by default.

`get-color-alias-translation` *Function*
color-alias

Returns the color-spec associated with *color-alias*.

8.4 Miscellaneous Color Functions

The following perform miscellaneous color-related functions.

`get-color-spec` *Function*
color

Returns the color-spec for *color*, which may be a color-spec or a color-alias.

`convert-color` *Function*
port color `&key` `(`*errorp* `t)`

Returns the window-system representation of *color* on the given *port*. It is more efficient to use the result of this function with graphics port drawing operations than to use the *color* directly.

`get-all-color-names` *Function*
`&optional` *sort*

Returns a list of all known color-aliases. If *sort* is true then the list will be sorted alphabetically.

`get-all-color-spec-names` *Function*
`&optional` *sort*

Returns a list of all aliases which are directly associated with a color-spec. If *sort* is true then the list is sorted alphabetically.

Appendix A
Users Guide §9.16 and output-pane

A.1 Introduction

The current implementation of `output-pane` supports most of the graphics-port graphics-state, drawing, transform and geometry functions described in the *Lispworks Users Guide* section 9.16. The following sections and functions do not apply.

9.16.1

Second and third paragraphs not applicable

9.16.3

Not applicable

9.16.6

Not applicable

9.16.7 'Definition'

Not applicable

9.16.7 'Creating Off Screen Graphics Port'

Not applicable

9.16.7 'Graphics State'

`with-mask` not applicable

`with-graphics-mask` take as mask rectangle only

9.16.7 ‘Drawing Functions’

`draw-image` not applicable

`copy-pixels` not applicable

9.16.7 ‘Port Parameters’

Not applicable

9.16.7 ‘Images’

Not applicable

9.16.7 ‘Pixmaps’

Not applicable

9.16.8 ‘Pixmap Graphics Ports’

Not applicable

9.16.11 ‘Caches’

Not applicable

A.2 New Graphics Ports Functions

The following functions have been added.

`invalidate-rectangle` *Generic Function*
port x y width height

Invalidate the given rectangle of the port. The window-system will cause the visible parts of the rectangle to be redrawn. All graphics state parameters are ignored.

`draw-generic-image` *Generic Function*
port image to-x to-y width height `&optional` (*from-x* `0`)
(*from-y* `0`)

Draw the generic-image *image* to the port *port*. The co-ordinate (*from-x*, *from-y*) of the image is placed at (*to-x*, *to-y*) on the port. The image is clipped to the size given by *width* and *height*.

`load-generic-image` *Generic Function*
port filename

Load the given *filename* as a generic image for the given port. The method used to find the filename in the file-system is implementation-dependent. The generic-image is returned.

Index of Symbols

T

U

W