LispWorks®

CAPI User's Guide

Edition 3.1

Harlequin Limited

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
England

This manual is copyright © 1992 Harlequin Limited. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of Harlequin Limited.

LispW orks ® is a registered trademark of Harlequin Limited.

LispW orks Copyright © 1987, 1988, 1989, 1990, 1991 1992 Harlequin Limited.

CLUE and CLX Copyright © 1987, 1988, 1989, 1990, 1991 T exas Instruments Incorporated.

Clipper is a registered trademark of Intergraph Corporation.

SP ARC is a registered trademark of Sun Microsystems Inc.

Unix and Open Look are trademarks of A T&T.

Motif is a trademark of Open Software Foundation, Inc.

The X W indow System is a trademark of Massachusetts Institute of T echnology .

Apple® and Macintosh® are registered trademarks of Apple Computer, Incorporated.

Microsoft, MS and MS-DOS are registered trademarks of the Microsoft Corporation.

IBM PC is a trademark of International Business Machines

The code for walker .lisp and compute-combination-points is excerpted with permission from PCL, Copyright © 1985, 1986, 1987, 1988 Xerox Corporation.

FrameMaker is a registered trademark of Frame T echnology Corporation.

CLX and CLUE bear the following copyright notice, which applies to the parts of LispW orks © derived therefrom:

TEXAS INSTRUMENTS INCORPORA TED
P.O. BOX 149149, MS 2151
AUSTIN, TEXAS 78714-9149

Copyright © 1987, 1988, 1989, 1990, 1991 T exas Instruments Incorporated.

Permission is granted to any individual or institution to use, modify and distribute this software, provided that this complete copyright and permission notice is maintained, intact, in all copies and documentation.

Texas Instruments Incorporated provides this software “as is” without express or implied warranty .

The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispW orks ® derived therefrom:

Copyright © 1989 by the Massachusetts Institute of T echnology , Cambridge, Massachusetts.
Permission to use, copy , modify , and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T . not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T . makes no representation about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty .
M.I.T . DISCLAIMS ALL W ARRANTIES WITH REGARD TO THIS SOFTW ARE, INCLUDING ALL IMPLIED W ARRANTIES OF MERCHANT ABILITY AND FITNESS. IN NO EVENT SHALL M.I.T . BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGAES OR ANY DAMAGES WHA TSOEVER RESUL TING FRON LOSS OF USE, DA TA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT , NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFT-WARE.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Harlequin reserves the right to alter the specification of the LispW orks ® product without notice.

Contents

1 Introduction to the Common API

1.1 Introduction 1-1
1.2 Basic Entities in the Common API 1-2
1.3 Interfaces 1-2
1.4 Loading the CAPI 1-3
1.5 CAPI Examples 1-3

2 CAPI Classes

2.1 Introduction 2-1
2.2 The Class Description Entry 2-1
2.3 CAPI Class Entries 2-2
2.4 Miscellaneous Macros and Functions 2-35
2.5 Built-in Dialog Boxes 2-36
2.5.1 The Confirmer 2-36

3 Defining an Interface

3.1 Introduction 3-1
3.2 Syntax 3-1

4 Input Models

4.1 Introduction 4-1

4.2 Gesture Specification 4-1
4.3 Command Specification 4-3

5 The Layout

5.1 Introduction 5-1
5.2 The Layout Description 5-1
5.3 Constraints and Hints 5-1
5.4 Functions on Hints 5-3

6 Writing a Subclass of Layout

6.1 Introduction 6-1
6.2 The Layout Algorithm 6-1
6.3 Geometry Information 6-2
6.4 Required Methods 6-2
6.5 Utilities for Layout Definition 6-3

7 Generic Fonts

7.1 Introduction 7-1
7.2 Components of a Generic Font 7-1
7.3 Font-related functions 7-1

8 Color Models

8.1 Introduction 8-1
8.2 The Three Models 8-1
8.2.1 RGB Model 8-1
8.2.2 HSV Model 8-2
8.2.3 Gray Model 8-3
8.3 Color-Alias 8-3
8.4 Miscellaneous Color Functions 8-4

9 Index of Symbols

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. To 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)`